A Toolchain for Verified OCaml Programs

Jean-Christophe Filliâtre[1,2], Léon Gondelman[3],
Andrei Paskevich[1,2], and Mário Pereira[1,2]

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] Inria Saclay – Île-de-France, Orsay, F-91893
[3] Radboud University Nijmegen

Abstract. This paper presents a methodology to verify OCaml programs with respect to behavioral specifications, using the Why3 tool. First, a formal specification is given in the form of an OCaml module signature extended with type invariants and function contracts, in the spirit of JML. Second, an implementation is written in the programming language of Why3 and then verified with respect to the specification. Finally, an OCaml program is obtained by an automated translation. This methodology is illustrated on several idiomatic OCaml programs. We discuss some of the challenges, such as proving the absence of arithmetic overflows, checking preconditions at runtime, and verifying stateful higher-order functions.

1 Introduction

Development of formally verified programs can be done in various ways. Perhaps, the most widespread approach consists in augmenting an existing mainstream programming language with specification annotations (contracts, invariants, etc.) and proving the conformance of the code to the specification, possibly passing through an intermediate language. Examples include VeriFast and KeY for Java, Frama-C and VCC (via Boogie) for C, GNATprove (via Why3) for Ada/SPARK. One challenge presented by this approach is that we have to encode a significant fragment of a real-life programming language, which was not designed with verification in mind, into a suitable program logic. Designing such an encoding is a non-trivial task in itself, and, what is worse, it may result in rather complex verification conditions, difficult for both automated and interactive proof.

Alternatively, one can proceed in the opposite direction: develop formally verified code in a dedicated verification language/environment and then translate it to an existing programming language, producing a correct-by-construction program. One can cite PVS, Coq, B, F*, Dafny, and Why3 as examples of this approach. It works well for self-containing programs, such as CompCert, but is less suitable when the verified code is supposed to be integrated into a larger

This research was partly supported by the Portuguese Foundation for Sciences and Technology (grant FCT-SFRH/BD/99432/2014) and by the French National Research Organization (project VOCAL ANR-15-CE25-008).

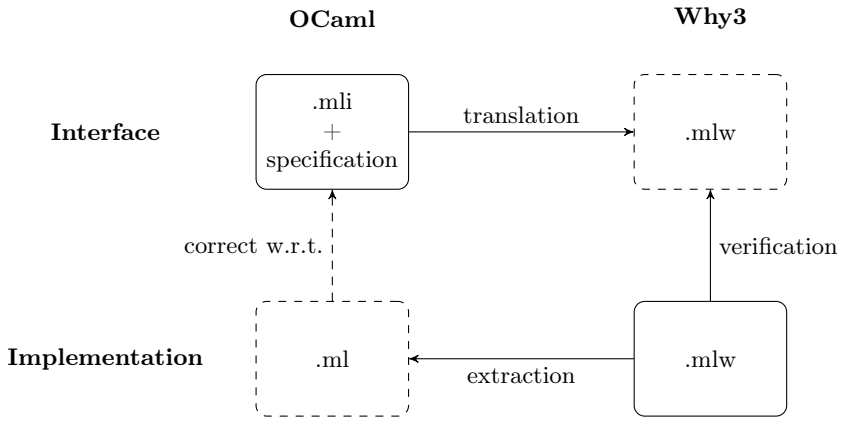

Fig. 1. Methodology diagram.

development. We cannot expect the original source code, developed in a specific verification framework, to be accessible to a common programmer — and the automatically generated code is typically a clobbered mess.

In this paper, we propose a way to reconcile the two approaches, avoiding the aforementioned disadvantages. We choose OCaml as our target language and Why3 as the verification tool. While we believe that the same approach can be applied to other languages and verification frameworks, OCaml has a number of features that facilitate our task. The separate compilation in OCaml is organized around the notions of an *interface* (declarations of types and functions, usually collected in `.mli` files) and an *implementation* (definitions of types and functions, collected in corresponding `.ml` files).

Our approach to producing verified OCaml code consists in splitting verification and implementation process into several steps. Given an OCaml `.mli` interface file, we start by annotating declarations with specifications such as function contracts (pre- and postconditions), type invariants, etc. For example, a function precondition may specify that its integer argument is non-negative. Given an annotated `.mli` file, we then generate automatically a corresponding Why3 input file, in which all annotations are translated into WhyML, the specification and programming language of Why3. The next step is to provide a verified Why3 implementation of the declared operations. This means that, in addition to implementing and verifying a WhyML program, we also establish its correctness with respect to the specifications given in the `.mli` file. Finally, Why3 automatically translates the verified implementation into a `.ml` file, producing a correct-by-construction OCaml program.

An overview of our methodology is given in Fig. 1. In the diagram, the rows correspond to different levels of abstraction (interface vs. implementation) and the columns correspond to environments (OCaml vs. Why3). The solid rectangles represent the user-written files, namely the annotated OCaml interface and the

WhyML implementation, and the dashed rectangles represent the automatically generated files, namely the WhyML interface and the OCaml implementation. In the next section, we explain this workflow in detail using the example of an insertion sort implementation. Then, Sec. 3 highlights some of the challenging aspects in verifying OCaml code, and shows how we address them. We conclude with related work and perspectives.

Source files for all the examples used in that paper are available from a web appendix at `http://why3.lri.fr/tacas-2018/`.

2 Example: Binary Sort

Let us use binary insertion sort as a working example. This algorithm is a variant of insertion sort with a linearithmic number of comparisons. To this end, it uses a binary search routine to find the insertion place. We start with an idiomatic `bsort.mli` file declaring the following two functions:

```
val bisectr : ('a -> 'a -> int) -> 'a array -> int -> int -> 'a -> int
(** "bisectr cmp a fromi toi v" searches the rightmost insertion
    point for v in a[fromi..toi[ *)

val bsort   : ('a -> 'a -> int) -> 'a array -> int -> int -> unit
(** "bsort cmp a fromi toi" sorts a[fromi..toi[ using comparison cmp *)
```

As usual in OCaml libraries, the behavior of these functions is described informally, in natural language. Our next step is to provide a formal specification.

2.1 Specification

Our annotations must be both mathematically precise and sufficiently simple so that users not familiar with formal methods could understand them. As in other behavioral specification languages, such as JML [16], our annotations are introduced in regular OCaml comments of the form `(*@ ... *)`. This way, annotations are ignored by the OCaml compiler. We annotate the `bisectr` function as follows:

```
val bisectr: ('a -> 'a -> int) -> 'a array -> int -> int -> 'a -> int
(*@ r = bisectr cmp a fromi toi v
      requires Order.is_preorder cmp
      checks   0 <= fromi <= toi <= Array.length a
      requires forall i j. fromi <= i <= j < toi -> cmp a.(i) a.(j) <= 0
      ensures  fromi <= r <= toi
      ensures  forall i. fromi <= i < r   -> cmp a.(i) v <= 0
      ensures  forall i. r     <= i < toi -> cmp a.(i) v >  0 *)
```

The first line introduces names for the function arguments and for the returned value, so that we can refer to them in the function contract. The keywords `requires` and `checks` introduce preconditions. Here, the first precondition requires `cmp` to be a preorder, using a higher-order predicate `is_preorder` from a

specification library. The second precondition requires `fromi` and `toi` to define a valid segment of array `a`, and the third precondition requires this segment to be sorted. Then, three postconditions are introduced using the keyword `ensures`. The first one bounds the returned value, and the last two express that values on the left side (resp. right side) of `r` are not greater (resp. greater) than `v`.

The two keywords `checks` and `requires` make a distinction between preconditions that are checked at runtime and those that are not. Indeed, function `bisectr` could be called from some unverified piece of code, where its preconditions are not met. For instance, `fromi` and `toi` arguments could be negative integers. Instead of letting `bisectr` access array `a` out of bounds, we prefer it to be defensive and to deny execution right away. Concretely, an unsatisfied `checks` precondition results in an `Invalid_argument` exception being raised, as is customary in OCaml libraries. On the other hand, the first precondition, namely that `cmp` is a preorder, cannot be checked at runtime and thus is introduced with `requires`. As for the third precondition, namely that all elements between indices `fromi` and `toi` are ordered, it could be checked at runtime. However, this would incur linear cost, beyond the logarithmic cost of the binary search itself.

The situation is different when function `bisectr` is called from some *verified* piece of code. In that case, we do not want to pay the price of checking unnecessarily preconditions at runtime. For that purpose, the `.mli` file may declare a second function, `unsafe_bisectr`, that has the same type and the same contract as `bisectr`[4], but no runtime checks. Providing a defensive and an unsafe version of the same function is already a usual practice in existing OCaml libraries, see *e.g.*, functions `Array.get` and `Array.unsafe_get` from the OCaml standard library.

We annotate function `bsort` in a similar way:

```
val bsort: ('a -> 'a -> int) -> 'a array -> int -> int -> unit
(*@ bsort cmp a fromi toi
      requires Order.is_preorder cmp
      checks   0 <= fromi <= toi <= Array.length a
      modifies a
      ensures  forall i j. fromi <= i <= j < toi -> cmp a.(i) a.(j) <= 0
      ensures  ArrayPermut.permut_sub (old a) a fromi toi *)
```

A significant difference with respect to the previous contract is that `bsort` function changes the contents of `a`, which is stated using the keyword `modifies`. The second postcondition states that we have only permuted the elements between `fromi` and `toi`, leaving the rest of the array unchanged. To do so, we invoke the predicate `permut_sub` from a specification library, using notation `old a` to refer to the contents of `a` before the function call.

[4] In practice, we do not have to duplicate the specification. Implicitly, function `unsafe_bisectr` has the same specification as function `bisectr`, any `checks` being replaced by `requires`.

2.2 Why3 Specification

We have extended Why3 to take annotated `.mli` files as input and to translate the specification they contain into Why3 declarations[5]. In our working example, the contract of each OCaml function is turned into a WhyML contract in a rather straightforward way. For instance, we get the following declaration for `bisectr`:

```
val bisectr (cmp: 'a -> 'a -> int63)
            (a: array 'a) (fromi toi: int63) (v: 'a) : int63
  requires { Order.is_preorder cmp }
  raises   { Invalid_argument ->
               not (0 <= to_int fromi <= to_int toi <= to_int (length a)) }
  requires { forall i j. to_int fromi <= i <= j < to_int toi ->
               to_int (cmp a[i] a[j]) <= 0 }
  ensures  { to_int fromi <= to_int result <= to_int toi }
  ensures  ...
```

Note that the type of arguments `fromi` and `toi`, as well as the return type of `cmp`, is `int63`, which is the Why3 model of OCaml's type `int`. Here, we assume a 64-bit architecture[6]. The pre- and postconditions, however, are considered to be only using mathematical integers. For instance, when we write `fromi <= toi` in the `bsort.mli` file, it is translated into a comparison between two mathematical integers, namely the values of the two machine integers `fromi` and `toi`. These values are obtained with the Why3 function `to_int`, which maps a value of type `int63` to its mathematical value, resulting in the translation `to_int fromi <= to_int toi`. In `.mli` files, constants such as 0 and operations such as `<=` are always understood as referring to mathematical integers.

One can notice that the precondition introduced with `checks` turns into a `raises` clause. This accounts for exception `Invalid_argument` being raised whenever this precondition is not met. Whenever a `checks` clause is present, we also produce a similar function contract for a second function with a precondition instead of an exceptional clause, that is,

```
val unsafe_bisectr (cmp: 'a -> 'a -> int63)
                   (a: array 'a) (fromi toi: int63) (v: 'a) : int63
  requires { Order.is_preorder cmp }
  requires { 0 <= to_int fromi <= to_int toi <= to_int (length a) }
  ...
```

Whenever such a precondition can be proved, it is better calling this function rather than `bisectr`, in order to avoid a runtime check. An example will be our own function `bsort`. As explained in the previous section, it is up to the user to declare function `unsafe_bisectr` may in the `.mli` file.

[5] Why3 provides a plug-in mechanism for parsers and translators for new formats.
[6] On a 64-bit architecture, OCaml's integers are 63 bits long, one bit being reserved for the garbage collector.

2.3 Verified Implementation

The most exciting part of the verification process is to provide an actual implementation and to show its correctness with respect to the stated formal specification. Here, we have to implement the four functions `unsafe_bisectr`, `bisectr`, `unsafe_bsort`, and `bsort`. Let us start with `unsafe_bisectr`. We implement it as a recursive function:

```
let rec unsafe_bisectr (cmp: 'a -> 'a -> int63)
                       (a: array 'a) (fromi toi: int63) (v: 'a) : int63
  ...
  variant  { toi - fromi }
= if fromi >= toi then toi
  else begin
    let mid = fromi + (toi - fromi) / 2:int63 in
    let c = cmp a[mid] v in
    if c <= 0:int63 then unsafe_bisectr cmp a (mid + 1:int63) toi v
                    else unsafe_bisectr cmp a fromi           mid v
  end
```

We do not show here the function contract, which is exactly the same as in the interface. We only add a `variant` clause to prove the termination. The code looks for the rightmost insertion place for value `v`, as required by the function contract. The code is operating on 63-bit integers, hence the use of notation `0:int63` instead of the mathematical integer 0. The proof of this function is straightforward, all verification conditions (VC) being discharged automatically by SMT solvers. Note that these VCs include safety (array access within bounds), termination, and respect of the function contract, as well as absence of arithmetic overflows. In this case, the latter is easy to prove[7], as we have $0 \leq$ `fromi` $<$ `toi` $\leq$ `length a` $\leq$ `max_int`. In other circumstances, proving the absence of arithmetic overflows can be quite subtle; this is discussed later in Sec. 3.1.

Once we have implemented `unsafe_bisectr`, we can use it to implement the defensive function `bisectr`. It amounts to first checking the precondition, raising an exception when it is not met, and otherwise calling `unsafe_bisectr`.

```
let bisectr (cmp: 'a -> 'a -> int63)
            (a: array 'a) (fromi toi: int63) (v: 'a) : int63
  ...
= if not (zero <= fromi <= toi <= length a) then
    raise Invalid_argument;
  unsafe_bisectr cmp a fromi toi v
```

The proof is simple, as the test is mimicking the formula from the specification. Note, however, that the code is operating on 63-bit integers, while the specification is written using the values of these machine integers mapped to mathematical integers. We proceed in a similar way with functions `unsafe_bsort` and `bsort`. The implementation of the former calls `unsafe_bisectr`, since we can statically verify its precondition.

[7] We still need to be careful in computing the mid-point [5].

2.4 Specification Inclusion

Now that we have a verified implementation, we need to show that it conforms to the specification given in the file `bsort.mli`. We do so with the help of WhyML instruction `clone` for module refinement. This instruction takes a module M and a substitution σ as arguments, and produces a new module $\sigma(M)$. In the process, it generates verification conditions to ensure that the substitution is valid. In particular, it checks for specification inclusion for function contracts, *i.e.*, that the preconditions of a function f in M imply the preconditions of $\sigma(f)$, if any, (contract contravariance) and that the postconditions of $\sigma(f)$ imply the postconditions of f (contract covariance). In our case, the substitution simply maps the four functions from `bsort.mli` to the four verified implementations from the previous section.

```
clone bsort.Sig with
  val bisectr        = bisectr,
  val unsafe_bisectr = unsafe_bisectr,
  val bsort          = bsort,
  val unsafe_bsort   = unsafe_bsort
```

Here, `bsort.Sig` is the module automatically obtained from file `bsort.mli` as described in Sec. 2.2. All VCs are straightforwardly proved, since implementations have exactly the same contracts as in `bsort.Sig`.

2.5 Extraction to OCaml

Why3 provides an extraction mechanism that automatically translates a WhyML implementation to OCaml. This amounts to erasing ghost code and logical annotations, translating WhyML constructs to their OCaml counterparts, and mapping symbols from Why3 standard library to those of OCaml standard library. The latter part is controlled using a text file, called a *driver*. Below we show a fragment of that file that maps the Why3 module `Array63` to the OCaml module `Array`.

```
module Array63
  syntax type array  "%1 array"
  syntax val  ([])   "Array.unsafe_get %1 %2"
  syntax val  ([]<-) "Array.unsafe_set %1 %2 %3"
  ...
end
```

The OCaml code is given as a string, where `%n` introduces a placeholder for the n-th argument of a WhyML symbol. In this case, the polymorphic type `array` from module `Array63` is replaced with the predefined OCaml type `array`. Similarly, functions `([])` and `([]<-)` are replaced by `Array.unsafe_get a i` and `Array.unsafe_set a i v`, respectively. Note that we are using here OCaml's `unsafe_` array operations that do not perform any runtime check, as we are translating code that has been verified.

In our running example, OCaml code for functions `bisectr` and `bsort` is extracted into a file `bsort.ml`. This file conforms to the interface file `bsort.mli` we started with, from the OCaml compiler's point of view. We have actually established a much stronger property, namely that `bsort.ml` conforms to the formal specification in `bsort.mli`. This closes the diagram in Fig. 1.

3 Methodology at Work

We now illustrate our toolchain on various examples. Each one exemplifies a particular issue in the process of verifying OCaml code.

3.1 Different Flavors of Machine Arithmetic

Using machine arithmetic is efficient and avoids the need for external libraries implementing arbitrary-precision integers. From a proof perspective, however, the price to pay is to prove the absence of arithmetic overflows. In the previous section, for instance, we proved the absence of arithmetic overflows for the binary sort implementation. It was rather easy, as all integers were bound by the array length. In this section, we explain how to deal with machine arithmetic in more subtle situations.

Let us take a *zipper* for lists as an example. A zipper for a list is a data structure to navigate back and forth within this list, in a purely applicative way, to perform local insertions and deletions [13]. We start by declaring the zipper type as an abstract type, as follows:

```
type 'a t
(*@ field view : 'a seq  *)
(*@ field idx  : integer *)
(*@ invariant 0 <= idx <= length view *)
```

The first two lines introduce a logical model of type `t`, using two ghost fields. The first field, `view`, models the list contents, using a type of mathematical sequences from the Why3 standard library; the second field, `idx`, represents the position in that sequence on which the zipper is currently placed. This index has type `integer`, the type of mathematical integers. A type invariant states that the zipper is always placed within the limits of the list. Since type `t` is an abstract data type, the specification of any zipper operation is stated in terms of the two fields `view` and `idx` only. For instance, the insertion operation is specified as follows:

```
val insert: 'a -> 'a t -> 'a t
(*@ r = insert x z
      ensures r.view = snoc z.view[.. z.idx] x ++ z.view[z.idx ..]
      ensures r.idx  = z.idx + 1 *)
```

Here, `(++)`, `snoc`, and `([ .. ])` are logical operations over sequences: operation `(++)` concatenates two sequences; `snoc` appends an element to the end of

a sequence; operation `s[..j]` extracts the sub-sequence of `s` up to index `j` excluded; and operation `s[i..]` extracts the sub-sequence from index `i` included. Consequently, the specification above expresses that `x` is inserted at position `z.idx` in the returned zipper `r`, the index being placed after `x`.

The zipper for lists is implemented as a pair of lists, one being the list of elements on the left side of the index, in reversed order, and the other being the list of elements on the right side of the index. This way, we can conveniently perform local operations on both sides of the index. In addition, we store the total length of the list, in order to get it in constant time. We thus end up with the following WhyML implementation for type `t`:

```
type t 'a = {
        left : list 'a;  (* the left part of the list, reversed *)
        right: list 'a;  (* the right part of the list *)
        len  : int63;    (* the total length of the list *)
  ghost view : seq 'a;
  ghost idx  : int;
}
```

Note that the length is stored as a machine integer. Intuitively, this is safe, as we will never build a list with 2^{62} elements. Yet, when we have to increment the length, we have to prove the absence of overflow. Consider for instance the implementation of `insert`:

```
let insert (x: 'a) (z: t 'a) : t 'a
= { ... len = z.len + one; ... }
```

Among the VCs for `insert`, we have to prove $\texttt{z.len} + 1 \leq 2^{62} - 1$. Without any further information on `z.len`, we cannot prove it. One obvious solution would be to add this inequality as a precondition for `insert`. We argue that such a solution is not satisfactory. First, this extra precondition would fatally pollute the proof of any client of `insert`. Second, there are situations where it is not even practical to fulfill such a precondition.

We adopt here the solution proposed by Clochard *et al.* [9]. The idea is as follows: Unless we expect our program to have century-long runs, we can rest assured that a counter that only grows by one at a time is, for all intents and purposes, safe from overflow. To materialize this meta-argument, it suffices to introduce a new type of integers, called *Peano integers*, with limited arithmetic operations. In Why3, we introduce a new library `Peano`, with a type `t` for Peano integers, a zero constant `zero`, and a successor function `succ`. We forgo the non-overflow precondition for `Peano.succ`, as there is no other way of producing a Peano integer than by starting at zero and incrementing it one by one, and so the 2^{62} limit will not be reached in any real-life situation. In our example, we give the field `len` type `Peano.t`, instead of `int63`, and increment it with `Peano.succ z.len`, instead of `z.len + 1`. Since `Peano.succ` has no precondition, we can prove function `insert` without any change to its specification.

When it comes to extract OCaml code, type `Peano.t` is translated into OCaml's type `int`, and `Peano.succ` into a machine addition. This is done by extending our OCaml driver with the following:

```
module Peano
  syntax type t    "int"
  syntax val  succ "%1 + 1"
end
```

Of course, Peano integers can not be used for values whose growth is not limited by the execution time of the program. In this case, our meta-argument cannot be applied and the absence of overflows must be proved explicitly.

3.2 Higher-order Effectful Functions

OCaml being a functional programming language, features functions as first-class values. We already encountered examples of higher-order functions in Sec. 2: a comparison function `cmp` passed as an argument to functions `bisectr` and `bsort`. In our Why3 proof, we implicitly assumed function `cmp` to be pure. Since we require `cmp` to implement a preorder, it would be really difficult to give it a specification and to use it if `cmp` depended on the state or, worse, had side effects.

However, it is not always acceptable to make such an assumption. A typical example of a stateful higher-order function is an iterator over the elements of a collection. In OCaml, it is idiomatic to provide, for some abstract type `t`, a function such as

```
val iter: (elt -> unit) -> t -> unit
```

A call to `iter f c` then applies function `f` sequentially to each element of collection `c`[8]. Obviously, this only makes sense when function `f` performs some side effects. Giving a complete specification to this function would require a more elaborated specification logic to account for function effects, including abrupt termination if an exception is raised in the middle of the iteration. This would lead to a rather unreadable contract for the OCaml programmer, breaking our principle of a simple specification language.

Instead, we claim that the best specification for an `iter` function is an operationally equivalent program with a clear and transparent meaning to any OCaml programmer. Let us illustrate our point of view on the example of the `iter` function over arrays. We propose the following specification:

```
val iter : ('a -> unit) -> 'a array -> unit
(*@ iter f a
      equivalent "for i = 0 to Array.length a - 1 do f a.(i) done" *)
```

The keyword `equivalent` introduces a program with the same operational behavior as `iter`. In this case, this is almost identical to the implementation of `Array.iter`[9]. Let us consider now the more general case of some abstract collection. We still can give `iter` a specification as follows:

[8] Here the term "collection" is taken broadly, meaning it does not necessarily refer to some data structure.

[9] The code of `Array.iter` uses `Array.unsafe_get`, as it is safe here.

```
val iter : ('a -> unit) -> 'a t -> unit
(*@ iter f c
      equivalent "List.iter f (elements c)" *)
```

Function `elements` returns the list of the elements in the order they are traversed. In particular, we move the problem of specifying the iteration order to the specification of `elements`. A way to do this was recently proposed by two of the authors [12]. It consists of characterizing the returned list with two predicates, one to identify valid prefixes and one to identify complete lists. Combined with the semantics of `List.iter`, familiar to any OCaml programmer, this fully specifies the behavior of function `iter`.

As such, `List.iter f (elements c)` is likely to be an inefficient program, that builds an intermediate list unnecessarily. This makes it unsuitable to be used as the actual implementation of the `iter` function in most cases. As suggested by the `equivalent` keyword, we need a mechanism for proving the equivalence between the program given as a specification and the actual implementation. Currently, we are not doing so, and we simply copy-paste the `equivalent` clause into the generated `.ml` file. This is not satisfactory, however, since we are not even proving the safety of that piece of code. In the future, we intend to prove the equivalence of the two programs, *e.g.*, along the lines of Barthe et al. [3].

3.3 Pointer-based Mutable Data Structures

As mentioned in the previous section, OCaml is a language with imperative features. This includes arrays and records with mutable fields. Since the version 4.03 of the compiler[10], it is also possible to declare some components of algebraic data types as mutable. For instance, one can declare a type for mutable singly-linked lists as follows:

```
type 'a cell =
  | Nil
  | Cons of { content: 'a; mutable next: 'a cell }
```

Compared to a similar data type in C or Java using the `null` pointer, this solution still has the benefits of an algebraic data type: the type system of OCaml ensures that one cannot use a `Nil` value to access the fields `content` or `next`.

We would like to be able to verify OCaml programs manipulating such types, *e.g.*, the mutable queues from the OCaml standard library. Unfortunately, a type definition such as `cell` is not possible in WhyML. The reason is that Why3 uses a type and effect discipline to track all aliases statically [11], and mutable lists are beyond the scope of such a static analysis. The solution is to resort to an explicit memory model, that is a set of types for pointers and memory together with operations to allocate, read, and write memory. This is not different from the solution adopted in tools using Why3 as an intermediate language, *e.g.*, Frama-C [10]. Contrary to such tools, though, we make a model specific for the type `cell` and we keep using other Why3 types as usual. We start by introducing an

[10] `https://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec257`

uninterpreted type in WhyML, together with a constant `nil` for the constructor `Nil`:

```
type cell 'a
val constant nil : cell 'a
```

The syntax `val constant` indicates that the symbol `nil` can be used in both specifications and programs. Then we model the memory using the following record type:

```
type mem 'a = {
          content: cell 'a -> option 'a;
  mutable    next: cell 'a -> option (cell 'a);
} invariant { forall l. contents l = None <-> next l = None }
```

Fields `content` and `next` mimic the two fields of constructor `Cons`. They are introduced as functions from memory locations of type `cell` to optional values. The allocated cells are those for which `next` and `content` return a value different from `None`. To operate on this memory model, we introduce several Why3 functions: `mk_cell` for allocation, `get_content` and `get_next` for read access, and `set_next` for write access. For instance, `set_next` is declared as follows:

```
val set_next (ghost mem: mem 'a) (l1 l2: cell 'a)  : unit
  requires { l1 <> nil }
  requires { mem.next l1 <> None }
  writes   { mem.next }
  ensures  { mem.next = Map.set (old mem.next) l1 (Some l2) }
```

It receives the memory as a first argument. This argument is ghost, as the memory model is not intended to be extracted to OCaml. The preconditions require `l1` to be different from `nil` and to be allocated. The postcondition states how the memory is updated.

We are now in position to implement and verify a Why3 program operating on this memory model. For instance, a function computing the length of a list would look as follows:

```
let length (ghost mem: mem 'a) (l: cell 'a) : Peano.t = ...
```

One could add a precondition stating that `l` is a `Nil`-terminated list. Note the use of Peano integers from Sec. 3.1 to avoid proving the absence of overflow in such a program.

Once the proof is done, the final step is to extract OCaml code from the Why3 program. To this end, we make use of a custom driver that maps the elements of our memory model to OCaml types and operations. In our case, this driver looks like

```
syntax type cell        "%1 SinglyLL.cell"
syntax val  (==)        "%1 == %2"
syntax val  nil         "SinglyLL.Nil"
syntax val  get_content "SinglyLL.get_content %1"
syntax val  get_next    "SinglyLL.get_next %1"
syntax val  set_next    "SinglyLL.set_next %1 %2"
syntax val  mk_cell     "SinglyLL.Cons { content = %1; next = %2 }"
```

where `SinglyLL` is an OCaml module containing the definition of type `'a cell` and auxiliary functions to safely access fields. For instance, function `set_next` is defined as follows:

```
let set_next l1 l2 = match l1 with
  | Nil    -> invalid_arg "set_next"
  | Cons c -> c.next <- l2
```

Note that the memory model is no more an argument of `set_next`, having been erased by the extraction because of its ghost status.

3.4 Functors

Why3 features a module system somewhat different from that of OCaml. A Why3 file can be divided into several top-level modules, introduced using the keyword `module`; a module namespace can be further divided into several sub-namespaces, introduced using the keyword `scope`. A major difference w.r.t. OCaml is that a module cannot contain sub-modules, but only sub-namespaces. Another distinguishing feature of the Why3 module system is that it allows uninterpreted and defined symbols to appear in the same namespace.

Functors are used in OCaml to implement modules parameterized by other modules. A typical example is a data structure that requires its elements to be equipped with an order relation, *e.g.*, a priority queue. The corresponding `.mli` file looks like (with logical annotations omitted)

```
module Make(X: sig
  type elt
  val compare: elt -> elt -> int
end) : sig
  type t
  val empty: t
  val add: X.elt -> t -> t
  ...
end
```

Our tool translates this into a Why3 module `Make`, with a sub-namespace `X`. The next step is to provide a Why3 implementation with identical namespaces.

```
module Make
  scope X
    type elt
    val compare elt elt : int63
  end
  type t = ...X.elt...
  let empty = ...
  let add x h = ...
end
```

The last step is to extract this Why3 module to OCaml. By default, the extraction will fail on some uninterpreted symbol that is not defined in the driver,

such as type `elt` or function `compare` here. However, we make an exception for a namespace containing only uninterpreted symbols, such as namespace `X` here. In that case, it is turned into a functor argument. With our example, we thus get extracted OCaml code with the expected structure, that is

```
module Make(X: sig ... end) = struct ... end
```

Note that extraction would fail if we mix defined and uninterpreted symbols within the same namespace, as there is no counterpart for that in OCaml.

3.5 Experimental Evaluation

We have used our approach to verify several OCaml modules.

module	loc	los	#VCs		
`PairingHeap`	42	34	36	persistent priority queues	(Sec. 3.4)
`ZipperList`	58	48	62	zipper data structure for lists	(Sec. 3.1)
`Arrays`	54	30	78	binary search and binary sort	(Sec. 2)
`Queue`	70	49	70	mutable queues	(Sec. 3.3)
`Vector`	150	129	148	resizable arrays	

All VCs are discharged using SMT solvers.

4 Related Work

The verified C compiler CompCert [18] and the static analyzer Verasco [14] are two notable large-scale examples of verified OCaml programs. Both are implemented in the Coq proof assistant and translated to OCaml afterwards using Coq extraction mechanism [19]. It is worth pointing out that Coq has a mechanism to replace some symbols by OCaml code at extraction time, in a way very similar to our driver mechanism.

The CFML tool [6] implements another approach to the verification of OCaml programs using Coq. It goes the other way around, turning an OCaml program into a "characteristic formula", that is an expression of its semantics into a higher-order separation logic embedded in Coq. CFML provides Coq tactics to help the user carry out proofs efficiently. Examples of recent applications of CFML include a verified implementation of hash tables [20] and verification of the correctness and amortized complexity of a union-find [8].

One key ingredient in our approach is the ability to refine a Why3 module containing specifications by a Why3 module containing an implementation, with suitable VCs to ensure correctness. This is reminiscent of other systems using a refinement approach. One obvious example is the B-method [1]. A fundamental difference, though, is that we only proceed in one step, where a B machine is typically refined in several steps.

Closer to our work is the integration of module refinement in the Dafny program verifier [17] by Leino and Koenig [15]. Dafny module system does not

make distinction between interface and implementation: the same notion of module is used both to give abstraction and to refine it. When refining a module in Dafny, one may give definitions to the data structures and methods left uninterpreted in the interface module, bring additional declarations, and refine previously given specifications. The main difference between module refinement in Dafny and Why3 concerns mutable data structures. In Dafny, mutable state is encapsulated within a class and dynamic frames are typically used to control side effects. In Why3, mutable data is encapsulated within record types, and it is the type system that controls side effects.

Mixins [2] is another example of a flexible module system that can mix uninterpreted symbols with defined ones. However, contrary to Why3 and Dafny module systems, mixins are designed for programming purposes only.

5 Conclusions and Perspectives

We have presented a toolchain for the verification of OCaml programs. We have used it to verify several state-of-the-art OCaml libraries. In the process, we have addressed several challenging problems, such as proving the absence of arithmetic overflows and coping with higher-order stateful functions, mutable data structures, and functors. It is important to point out that the trusted computing base in our approach is quite large, as it includes Why3, the SMT solvers used in the verification, and the driver files used during the extraction.

Our work is part of a larger project aiming at verifying an OCaml library [7]. There is still a lot to do to accomplish that goal. First, the OCaml specification language sketched in this paper is still under development. Second, we still have to prove the `equivalent` clauses mentioned in Sec. 3.2. One solution would be to integrate relational Hoare logic in Why3 [4,3]. Last, our experiments with mutable data structures presented in Sec. 3.3 require a better tool support from Why3. This could be either a separation logic library, as in KIV [21], or a dynamic frames library.

References

1. Jean-Raymond Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.
2. Davide Ancona and Elena Zucca. An algebraic approach to mixins and modularity. In M. Hanus and M. Rodrìguez Artalejo, editors, *5th Intl. Conf. on Algebraic and Logic Programming*, number 1139 in Lecture Notes in Computer Science, pages 179–193, Berlin, 1996. Springer.
3. Gilles Barthe, Juan Manuel Crespo, and César Kunz. *Relational Verification Using Product Programs*, pages 200–214. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
4. Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 14–25, New York, NY, USA, 2004. ACM.

5. Joshua Bloch. Nearly all binary searches and mergesorts are broken, 2006. http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html.
6. Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, September 2011. ACM.
7. Arthur Charguéraud, Jean-Christophe Filliâtre, Mário Pereira, and François Pottier. VOCAL – A Verified OCaml Library. ML Family Workshop, September 2017.
8. Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, September 2017.
9. Martin Clochard, Jean-Christophe Filliâtre, and Andrei Paskevich. How to avoid proving the absence of integer overflows. In Arie Gurfinkel and Sanjit A. Seshia, editors, *7th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE)*, Lecture Notes in Computer Science, San Francisco, California, USA, July 2015. Springer.
10. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, number 7504 in Lecture Notes in Computer Science, pages 233–247. Springer, 2012.
11. Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A pragmatic type system for deductive verification. Research report, Université Paris Sud, 2016. https://hal.archives-ouvertes.fr/hal-01256434v3.
12. Jean-Christophe Filliâtre and Mário Pereira. A modular way to reason about iteration. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *8th NASA Formal Methods Symposium*, volume 9690 of *Lecture Notes in Computer Science*, Minneapolis, MN, USA, June 2016. Springer.
13. Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.
14. Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–259, Mumbai, India, January 2015. ACM.
15. Jason Koenig and K. Rustan M. Leino. Programming language features for refinement. In John Derrick, Eerke A. Boiten, and Steve Reeves, editors, *Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015.*, volume 209 of *EPTCS*, pages 87–106, 2015.
16. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.
17. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
18. Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
19. Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer, 2003.

20. François Pottier. Verifying a hash table and its iterators in higher-order separation logic. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*, January 2017.
21. W. Reif, G. Schnellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In William McCune, editor, *14th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 69–72, Townsville, North Queensland, Australia, july 1997. Springer.