A Toolchain to Produce Verified OCaml Libraries

Jean-Christophe Filliâtre, Léon Gondelman, Cláudio Lourenço, Andrei Paskevich, Mário Pereira, Simão Melo de Sousa, Aymeric Walch

A Toolchain to Produce Verified OCaml Libraries

Jean-Christophe Filliâtre[1], Léon Gondelman[2], Cláudio Lourenço[1], Andrei Paskevich[1], Mário Pereira[3], Simão Melo de Sousa[4], and Aymeric Walch[5]

[1] Université Paris-Saclay, CNRS, Inria, LRI, 91405, Orsay, France
[2] Department of Computer Science, Aarhus University, Denmark
[3] NOVA-LINCS, FCT-Univ. Nova de Lisboa, Portugal
[4] NOVA-LINCS, Univ. Beira Interior, Portugal
[5] École Normale Supérieure de Lyon, France

Abstract. In this paper, we present a methodology to produce verified OCaml libraries, using the GOSPEL specification language and the Why3 program verification tool. First, a formal behavioral specification of the library is written in OCaml/GOSPEL, in the form of an OCaml module signature extended with type invariants and function contracts. Second, an implementation is written in WhyML, the programming language of Why3, and then verified with respect to the GOSPEL specification. Finally, WhyML code is automatically translated into OCaml source code by Why3. Our methodology is illustrated with two examples: first, a small binary search function; then, a union-find data structure that is part of a larger OCaml verified library.

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 [17] and KeY [2] for Java, Frama-C [19] and VCC (via Boogie) [12,3] for C, GNATprove (via Why3) for Ada/SPARK [6,16]. This approach can end up being quite challenging, since real-life programming languages, not designed with verification in mind, have to be encoded into a suitable program logic. Such an encoding is a non-trivial task, and it may result in rather complex verification conditions, that are difficult to discharge by both automated and interactive provers.

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 [22], Coq [28], B [1], F* [27], Dafny [20], and Why3 [13] as examples of this approach. It works well for self-contained programs, such as

This research was partly supported by the French National Research Organization (project VOCAL ANR-15-CE25-008).

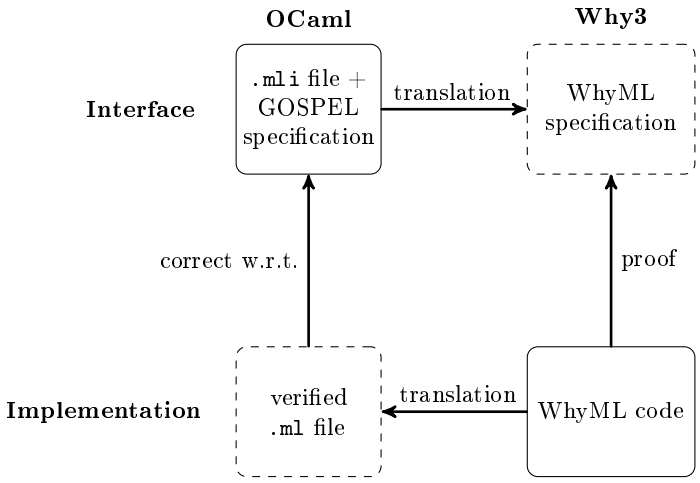

Fig. 1. The workflow.

CompCert [21], but is less suitable when the verified code is supposed to be integrated into a larger 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 both of the aforementioned disadvantages. Our work takes place in the setting of a larger project, named VOCaL (for Verified OCaml Library) [9], whose ambition is to provide a mechanically verified library of efficient general-purpose data structures and algorithms, written in the OCaml language. One of the main lines of work in the VOCaL project is the design of GOSPEL [8], a behavioral specification language for OCaml, similar to what JML is for Java [5], or ACSL for C [4]. The VOCaL project also combines the use of three verification tools, namely Coq, CFML [7], and Why3. This paper focuses on the last.

Our approach to producing verified OCaml code consists in splitting the verification and implementation process into several steps. The workflow is given in Fig. 1: solid rectangles represent user-written files, and dashed rectangles represent automatically generated files. First, we start with a GOSPEL specification file: an OCaml `.mli` interface file where OCaml declarations are augmented with specification annotations, such as function contracts (pre- and postconditions) and type invariants. GOSPEL annotations are written as OCaml comments, and thus ignored by the OCaml compiler. Our framework parses and type checks this file and automatically generates a corresponding Why3 input file, in which all annotations are translated into WhyML, the specification and programming language of Why3. Second, we provide a verified WhyML 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 specifi-

cations given in the `.mli` file. Finally, the Why3 tool automatically translates the verified WhyML implementation into a correct-by-construction OCaml program.

In the following sections, we explain this workflow in detail using the examples of a binary search function (Sec. 2 and 3) and of a union-find library (Sec. 4). Section 5 gives an overview of the other OCaml modules verified with Why3 in the VOCaL project. Source files for all the OCaml modules mentioned in this paper are available from `https://vocal.lri.fr/`.

2 Why3 and WhyML

Why3 is a tool for deductive program verification [16]. It uses its own programming and specification language, called WhyML, which is largely inspired by OCaml syntax. The theoretical foundation of Why3 is the weakest-precondition calculus and a custom type system with regions to handle mutable heap-allocated data [14]. As a programming language, WhyML can be seen as a subset of OCaml with support for exceptions, algebraic types, type polymorphism, and restricted higher-order (side-effect-free functional arguments). The program annotations (function contracts, loop invariants, assertions, etc.) are written in a rich logical language that reuses the data types of programs and features pattern matching, recursive and inductive definitions, as well as higher-order functions. An important feature of WhyML is so-called *ghost code* which is a part of program code that serves exclusively to facilitate specification and proof, and cannot influence the actual computations [15]. For example, complex data structures in WhyML would often feature ghost fields that contain the logical model of the structure. This allows us to specify the program functions that manipulate such a data type in terms of a simple mathematical model, without referring to the details of the implementation.

Let us illustrate the use of Why3 on the simple example of a binary search function. It can be specified and implemented in WhyML as follows:

```
let rec binary_search (cmp: 'a -> 'a -> int63)
                      (a: array 'a) (lo hi: int63) (v: 'a) : int63
  requires { is_pre_order cmp }
  requires { 0 <= lo <= hi <= length a }
  requires { forall i j. lo <= i <= j < hi -> cmp a[i] a[j] <= 0 }
  ensures  { lo <= result < hi /\ cmp a[result] v = 0 }
  raises   { Not_found ->
               forall i. lo <= i < hi -> cmp a[i] v <> 0 }
  variant  { hi - lo }
=
  if lo >= hi then raise Not_found;
  let mid = lo + (hi - lo) / 2 in
  let c = cmp a[mid] v in
  if      c < 0 then binary_search cmp a (mid + 1) hi v
  else if c > 0 then binary_search cmp a lo       mid v
  else mid
```

The first two lines show the type signature of the function `binary_search`. The functional parameter `cmp` must be a total stateless and effect-free function, so that we can use it in specification annotations. Type `int63` denotes the 63-bit signed integers and is distinguished from type `int` that represents unbounded mathematical integers. The values of range types like `int63` are implicitly coerced to `int` inside specification annotations.

After the type signature comes the function contract. The clauses `requires` correspond to the preconditions: the functional parameter `cmp` is required to implement a total pre-order (the sign of the return value indicates the result of the comparison); the range `lo..hi` must lie inside the array bounds; the array must be ordered within that range. The clause `ensures` describes the postcondition associated to a normal (*i.e.*, non-exceptional) termination: the return value of type `int63`, named `result`, must be a valid array index where the sought value `v` is stored. The clause `raises` describes the exceptional postcondition: if `binary_search` raises exception `Not_found`, then `v` does not occur in the array between `lo` and `hi`. Finally, the last clause, `variant`, is not part of the contract, but helps Why3 to prove the termination: with each recursive call, the value of the variant must strictly decrease with respect to some well-founded order.

The code of `binary_search` is a rather idiomatic OCaml code. When we run Why3 on this program, verification conditions are generated to prove the following properties:

- all function calls respect the preconditions of the callee (this includes showing that all operations over bounded integers do not produce overflows and that all array accesses are made within bounds);
- all recursive function calls make the value of the variant decrease;
- the postconditions are met both for normal and exceptional termination.

Why3 uses multiple automated and interactive provers (including Alt-Ergo, CVC4, Z3, E, Coq, and Isabelle) to discharge the proof obligations. Moreover, the user can apply various *transformations*, such as goal splitting, case analysis, and definition unfolding, during a Why3 interactive verification session in order to simplify proof tasks before sending them to the background provers. For example, the above implementation of binary search is proved automatically by CVC4 in a matter of seconds after one application of goal splitting.

Why3 supports automated translation of WhyML code to OCaml. An important part of this translation is removal of ghost code and ghost data. Once ghost code and specification annotations are eliminated, Why3 produces an OCaml source file. The program symbols and types which are not implemented inside WhyML code (and are axiomatized instead) are translated to their OCaml counterparts via a so-called *driver*: a text file that maps WhyML symbols to fragments of OCaml code. Drivers are part of the trusted base of Why3: an incorrect translation would result in a program whose behaviour is different from that of the verified WhyML code and will not necessarily respect the contract. In the example above, the types `int63` and `array`, as well as standard operations over them, are mapped to the corresponding OCaml types and operations. The resulting OCaml code is practically identical to the WhyML source.

3 GOSPEL and its Translation to WhyML

The specification language GOSPEL [8] was developed in the context of the VOCaL project. It extends the syntax of OCaml `.mli` interface files with behavioral specifications, written as specially formatted OCaml comments. GOSPEL is not tied to any particular verification tool. Instead, the methodology of VOCaL considers GOSPEL as a common frontend specification language for different verification tools, that either verify OCaml code directly (*e.g.*, CFML) or can produce correct-by-construction OCaml code (*e.g.*, Why3 or Coq).

Here is the GOSPEL specification for our `binary_search` function:

```
val binary_search:
  ('a -> 'a -> int) -> 'a array -> int -> int -> 'a -> int
(** Search for value [v] in array [a], between indices [lo]
    inclusive and [hi] exclusive, using comparison function [cmp].
    Returns an index where [v] occurs, or raises [Not_found]
    if no such index exists. *)
(*@ result = binary_search cmp a lo hi v
    requires is_pre_order cmp
    checks   0 <= lo <= hi <= Array.length a
    requires forall i j. lo <= i <= j < hi -> cmp a.(i) a.(j) <= 0
    ensures  lo <= result < hi && cmp a.(result) v = 0
    raises   Not_found ->
               forall i. lo <= i < hi -> cmp a.(i) v <> 0 *)
```

The file begins with a standard OCaml function declaration, together with an informal `ocamldoc` comment. Then follows the formal specification, enclosed in a special comment starting with '`(*@`'. The first line of the specification gives names to the function parameters and to its return value. Wherever possible, GOSPEL uses OCaml syntax for primitive operations (*e.g.*, Boolean connectives and array access).

A notable difference with the WhyML specification from the previous section is clause `checks`, which describes a precondition that is expected to be checked during execution. When such a precondition is violated, the OCaml implementation must raise the OCaml built-in exception `Invalid_argument`. Contrary to JML or SPARK, GOSPEL annotations are not meant to be executable: they may contain unbounded quantifiers, abstract logical functions, etc. It is up to the verified OCaml implementation to fulfill the provided contract, including the runtime checks.

Although the semantics of GOSPEL is given in terms of Separation Logic [8], the specification language itself is kept simple, without explicit Separation Logic operators. Instead, GOSPEL adopts a number of reasonable conventions, such as separation of function parameters and return values, and full ownership transmission from the caller to the callee and back. This is done intentionally, in order to make the language accessible to a larger audience of OCaml programmers. Incidentally, this also simplifies translation from GOSPEL to WhyML, which adheres to the same conventions [14].

We have extended Why3 with a new input format for GOSPEL. The type and function declarations are translated to corresponding WhyML declarations. A special treatment is provided for `checks` clauses: for each function `foo` whose contract contains a clause `checks` ϕ, Why3 produces the declarations of two WhyML functions, `foo` and `unsafe_foo`, where the former contains an exceptional postcondition `raises { Invalid_argument ->` $\neg\phi$ `}` and the latter contains a precondition `requires {` ϕ `}`. The latter function is deemed unsafe because it trusts the caller to respect the precondition ϕ, whereas the former implements a defensive runtime check. A similar practice already exists in OCaml, *e.g.*, `Array.unsafe_get`. Unsafe functions are perfectly safe when called from verified code, since precondition must be satisfied, and provide a better performance.

Why3 can generate proof obligations to establish that a given WhyML implementation conforms to a given WhyML specification translated from GOSPEL. In our case, we can show that our implementation of binary search from Sec. 2 corresponds to the declaration of function `unsafe_binary_search` (where the `checks` clause is translated as a WhyML precondition).

4 Example: Union-Find

We now describe a more complex example, taken from the VOCaL library. This is an OCaml module implementing a union-find data structure, with the following API (borrowed from [10]):

```
type 'a elem                              (* type of the elements *)
val make: 'a -> 'a elem                   (* a singleton class    *)
val find: 'a elem -> 'a elem              (* the representative   *)
val eq: 'a elem -> 'a elem -> bool        (* in the same class?   *)
val union: 'a elem -> 'a elem -> unit     (* merge two classes    *)
```

In this API, a value of type `'a` is attached to each equivalence class. Our actual implementation includes the access and update functions to manipulate this value. For the sake of brevity, we do not discuss this functionality in the paper.

Specification. We start with a GOSPEL specification. In order to give a specification to the functions above, we need a logical representation of the global state of the union-find data structure. This logical representation takes the form of a set of all elements, together with a function selecting a canonical element in each equivalence class:

```
(*@ type 'a uf
      mutable model dom: 'a elem set
      mutable model rep: 'a elem -> 'a elem
      invariant forall x. mem x dom -> rep (rep x) = rep x
      invariant forall x. mem x dom -> mem (rep x) dom      *)
```

Notice that type `uf` is declared inside a GOSPEL annotation and not as an OCaml type. Consequently, it will only be available for specification or as a type

of ghost parameters. The fields `dom` and `rep` are declared `mutable` to reflect the possible changes in the state of the union-find structure. The two invariants ensure that the set `dom` is indeed partitioned by the relation "to have the same canonical representative given by `rep`".

We are now in position to provide a specification to each OCaml function above. Let us use `make` and `find` as examples.

```
val make: 'a -> 'a elem
(*@ e = make [uf: 'a uf] v
      modifies uf
      ensures  not (mem e (dom (old uf)))
      ensures  dom uf = add e (dom (old uf))
      ensures  rep uf = (rep (old uf))[e <- e] *)

val find: 'a elem -> 'a elem
(*@ r = find [uf: 'a uf] e
      requires mem e (dom uf)
      modifies uf
      ensures  dom uf = dom (old uf)
      ensures  rep uf = rep (old uf)
      ensures  r = rep uf e          *)
```

For the purpose of the specification, `make` and `find` receive an extra parameter `uf` of type `'a uf`. Square brackets identify it as a ghost parameter. The `modifies` clause in the function contract accounts for the modification of the union-find data structure (caused by path compression in the case of `find`). The term `old uf` refers to the state of the structure at the beginning of the function call.

Verified Implementation. The next step is to implement and verify the union-find data structure. The OCaml implementation we target is based on the following data types:

```
type 'a content = Link of 'a elem | Root of int * 'a
and  'a elem    = 'a content ref
```

Each element is a mutable reference which can be in one of two states: either it is a canonical element (`Root`), with a rank of type `int` and a value of type `'a`; or it points (`Link`) to another element in the same equivalence class.

This type definition cannot be used as is in WhyML, which does not support recursive mutable types. The solution in Why3 is to resort to an explicit memory model, that is a set of types and operations to model the heap, pointers, allocation, and memory access. We translate the OCaml types above into the following WhyML types

```
type loc_ref 'a
type content 'a = Link (elem 'a) | Root int63 'a
with elem    'a = loc_ref (content 'a)
```

where `loc_ref 'a` is an abstract immutable type to represent locations of OCaml's heap-allocated references of type `ref`. The contents of the heap is modeled with another WhyML type

```
type mem_ref 'b = private {mutable refs: loc_ref 'b -> option 'b}
```

where non-allocated locations are mapped to `None`, and each allocated location is mapped to `Some` c for some value c of type `'b`.

Instead of modeling a single global heap, we adopt an approach of "small heaps", *i.e.*, local chunks of memory, which are passed as ghost arguments to heap-manipulating functions [23, Chapter 5]. For instance, a reference is updated using the following function:

```
val set_ref (ghost mem: mem_ref 'b) (l: loc_ref 'b) (c: 'b): unit
  requires { mem.refs l <> None }
  writes   { mem }
  ensures  { mem.refs = (old mem.refs)[l <- Some c] }
```

Once this memory model is built, we can implement and verify the union-find data structure. In particular, we have to implement the data type `uf`. It is a record data type that contains, in addition to the fields `dom` and `rep`, the contents of the memory:

```
type uf 'a = { memo: mem_ref (content 'a); ... }
```

As declared in the interface, all union-find functions receive a ghost parameter of type `uf` and then exploit it to perform read/write operations on memory:

```
let rec find (ghost uf: uf 'a) (x: elem 'a) : elem 'a
= match get_ref uf.memo x with
  | Root _ _ -> x
  | Link y   -> let rx = find uf y in
                set_ref uf.memo x (Link rx); rx end
```

Here, the call to `set_ref` accounts for path compression. Once we have implemented all operations, we prove that they conform to the GOSPEL specification written in the `.mli` file and translated to WhyML by our tool.

Translation to OCaml. The last step consists in translating WhyML to OCaml. We extend the standard driver of Why3 with a custom driver file for our memory model, as follows:

```
module UnionFind.Mem
  syntax type loc_ref "%1 ref"
  syntax val  set_ref "%1 := %2"
  ...
```

We do not provide a translation for type `mem_ref`, since it is only used for ghost parameters. For the same reason, function `set_ref` only receives two parameters in the translated code. Such a file must be written with care, as it is clearly part of the trusted base. In particular, we trust OCaml references to have the same semantics as the one described in our memory model.

module	spec	code	#VCs	
`UnionFind`	71	176	92	union-find
`Vector`	142	285	63	resizable arrays
`PriorityQueue`	56	290	219	mutable priority queues
`PairingHeap`	43	244	66	persistent priority queues
`ZipperList`	65	150	54	zipper data structure for lists
`Arrays`	43	126	104	*e.g.*, binary search, binary sort
`Mjrty`	11	35	37	Boyer&Moore's majority
`RingBuffer`	44	94	61	circular arrays
`CountingSort`	19	80	128	array counting sort

Fig. 2. Verified OCaml Modules.

5 The VOCaL Project: The State of the Library

We have used our approach to verify several other OCaml modules, listed in Fig. 2. For each OCaml module, column "spec" shows the number of lines in the `.mli` file and column "code" shows the number of lines in the WhyML implementation and proof. Column "#VCs" shows the total number of verification conditions. All of them were discharged automatically using the combined effort of Alt-Ergo, CVC4, and Z3.

These examples involve many aspects not described in this paper, due to lack of space. We describe some of them, briefly:

- One module, `PriorityQueue`, implements mutable priority queues on top of another module, `Vector`, which implements resizable arrays. The proof is performed in a modular way: the Why3 proof of `PriorityQueue` only makes use of the GOSPEL specification for `Vector`, but not of its implementation.
- Modules `PriorityQueue` and `PairingHeap` are OCaml functors, *i.e.*, modules parameterized by a module. This is the idiomatic way in OCaml to provide types and functions as parameters (here, a type of elements equipped with a comparison function). From GOSPEL's point of view, there is no difference between the specification of a parameter module and that of a toplevel module. From Why3's point of view, there is no difference between verifying a module B that uses another module A or that is parameterized with a module A. The main difference lies in the translation from Why3 to OCaml, which must produce an actual OCaml functor.
- The verified module `PriorityQueue` has been integrated into Why3 source code. It is not used in the trusted part of Why3, but only in some heuristic algorithm that matches former proof attempts with new verification conditions. In this way, there is no circularity in the proof of `PriorityQueue`.
- Two of the modules involve arithmetic computations for which it is not obvious to prove the absence of arithmetic overflow (the rank in `UnionFind` and a list length in `ZipperList`). We use a Why3 library providing a protected integer type with a restricted set of operations to solve that issue [11].

The GOSPEL specification, OCaml code, and Why3 proof for all these modules is available from `https://vocal.lri.fr/`.

6 Related Work and Conclusion

Related Work. The verified C compiler CompCert [21] and the static analyzer Verasco [18] 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 [26]. It is worth pointing out that Coq has a mechanism to replace certain symbols by OCaml code at extraction time, in a way very similar to our driver substitution mechanism.

The CFML tool [7] 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 [25] and verification of the correctness and amortized complexity of a union-find library [10]. Contrary to the CFML proof, ours is fully automatic but we only treat functional correctness and not the complexity bounds.

Surprisingly, program verification has seldom been applied to libraries of significant size. A remarkable exception is the verification of the EiffelBase2 containers library [24], performed with the AutoProof system [29]. It is our purpose to continue using and improving our methodology to grow our verified library to a size comparable to that of EiffelBase2. However, we do not focus specifically on the verification of containers, but also on general-purpose algorithms, *e.g.*, our union-find implementation.

Conclusion. We proposed a new workflow to produce correct-by-construction OCaml programs. It builds upon the existing tool Why3, with the addition of the following contributions: a specification language for OCaml, called GOSPEL; a tool to translate it to WhyML; a technique to build memory models for mutable recursive OCaml types; an enhanced extraction mechanism for Why3, with support for OCaml functors; a practical validation with the proof of nine non-trivial OCaml modules.

Acknowledgments. We are grateful to the anonymous reviewers of a previous submission for their helpful comments and suggestions. We also thank all the members of the VOCaL project for fruitful discussions.

References

1. Jean-Raymond Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.
2. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.
3. Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.
4. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.
5. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.
6. Bernard Carré and Jonathan Garnsworthy. SPARK—an annotated Ada subset for safety-critical programming. In *Proceedings of the conference on TRI-ADA'90*, TRI-Ada'90, pages 392–402, New York, NY, USA, 1990. ACM Press.
7. 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.
8. Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. GOSPEL — providing OCaml with a formal specification language. In Annabelle McIver and Maurice ter Beek, editors, *FM 2019 23rd International Symposium on Formal Methods*, Porto, Portugal, October 2019.
9. Arthur Charguéraud, Jean-Christophe Filliâtre, Mário Pereira, and François Pottier. VOCAL – A Verified OCaml Library. ML Family Workshop, September 2017.
10. 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*, 62(3):331–365, March 2019.
11. 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)*, volume 9593 of *Lecture Notes in Computer Science*, pages 94–109, San Francisco, California, USA, July 2015. Springer.
12. Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009.
13. Jean-Christophe Filliâtre. Why3 — where programs meet provers. In *KeY Symposium 2017*, Rastatt, Germany, October 2017. Invited talk.

14. 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`.
15. Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.
16. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.
17. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.
18. 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.
19. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.
20. 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.
21. Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
22. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga Springs, NY, June 1992. Springer.
23. Mário Pereira. *Tools and Techniques for the Verification of Modular Stateful Code*. Thèse de doctorat, Université Paris-Saclay, December 2018.
24. Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. *Formal Asp. Comput.*, 30(5):495–523, 2018.
25. 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.
26. Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.
27. Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.
28. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.9*, 2019. `http://coq.inria.fr`.
29. Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. Autoproof: Auto-active functional verification of object-oriented programs. In *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer, 2015.