Cameleer: a Deductive Verification Tool for OCaml*

Mário Pereira and António Ravara

NOVA LINCS, Nova School of Science and Technology, Portugal
{mjp.pereira, aravara}@fct.unl.pt

Abstract. We present Cameleer, an automated deductive verification tool for OCaml. We leverage on the recently proposed GOSPEL (Generic OCaml SPEcification Language) to attach rigorous, yet readable, behavioral specification to OCaml code. The formally-specified program is fed to our toolchain, which translates it into an equivalent one in WhyML, the programming and specification language of the Why3 verification framework. We report on successful case studies conducted in Cameleer.

Keywords: Deductive Software Verification · OCaml · Why3 · GOSPEL

1 Introduction

Over the past decades, we have witnessed a tremendous development in the field of deductive software verification [11], the practice of turning the correctness of code into a mathematical statement and then prove it. Interactive proof assistants have evolved from obscure and mysterious tools into *de facto* standards for proving industrial-size projects. On the other end of the spectrum, the so-called *SMT revolution* and the development of reusable intermediate verification infrastructures contributed decisively to the development of practical automated deductive verifiers.

Despite all the advances in deductive verification and proof automation, little attention has been given to the family of *functional languages* [27]. Let us consider, for instance, the OCaml language. It is well suited for verification, given its well-defined semantics, clear syntax, and state-of-the-art type system. Yet, the community still lacks an easy to use framework for the specification and verification of OCaml code. The working programmers must either re-implement their code in a proof-aware language (and then rely on code extraction), or they must turn themselves into interactive frameworks. Cameleer fills the gap, being a tool for the deductive verification of programs written in OCaml, with a clear focus on proof automation. Cameleer uses the recently proposed GOSPEL [5], a specification language for OCaml. We advocate here the vision of the *specifying*

* This work is partly supported by the HORIZON 2020 Cameleer project (Marie Skłodowska-Curie grant agreement ID:897873) and NOVA LINCS (Ref. UIDB/04516/2020)

programmer: the person who writes the code should also be able to naturally provide suitable specification. GOSPEL terms are written in a subset of the OCaml language, which makes them more appealing to the regular programmer. Moreover, we believe specification and implementation should co-exist and evolve together, which is exactly the approach followed in Cameleer.

Cameleer takes as input a GOSPEL-annotated OCaml program and translates it into an equivalent counterpart in WhyML, the programming and specification language of the Why3 framework [16]. Why3 is a toolset for the deductive verification of software, clearly oriented towards automated proof. A distinctive feature of Why3 is that it interfaces with several different off-the-shelf theorem provers, namely SMT solvers.

Contributions. To the best of our knowledge, Cameleer is the first deductive verification tool for annotated OCaml programs. It handles a realistic subset of the language, and its interaction with the Why3 verification framework greatly increases proof automation. Our set of case studies successfully verified with the Cameleer tool constitutes, by itself, an important contribution towards building a comprehensive body of verified OCaml codebases. Finally, it is worth noting that the original presentation of GOSPEL was limited to the specification of interface files. In the scope of this work, we have extended it to include implementation primitives, such as loop invariants and ghost code (*i.e.*, code that has no computational purpose and is used only to ease specification and proof effort) evolving GOSPEL from an interface specification language into a more mature proof language.

2 Illustrative Example – Binary Search

Higher-order implementation. Fig. 1 presents an implementation of binary search, where the comparison function, `cmp`, is given as an argument to the main function. For the sake of readability, we give the type of arguments and return value of function `binary_search`, but these can be inferred by the OCaml compiler.

The function contract is given after its definition as a GOSPEL annotation, written within comments of the form `(*@ ... *)`. The first line names the returned value. Next, the first precondition establishes that the `cmp` is a total pre order following the OCaml convention: if `x` is smaller than `y`, then `cmp x y < 0`; if `x` is greater than `y`, then `cmp x y > 0`; finally, `cmp x y = 0` if `x` and `y` are equal values[1]. It is worth noting that GOSPEL, hence Cameleer, assumes `cmp` to be a pure function (*i.e.*, a function without any form of side-effects). The second precondition requires the array to be sorted according to the `cmp` relation. Finally, the last two clauses capture the possible outcomes of execution: the regular postcondition (`ensures` clause) states the returned index is within the bounds of `a` and its value is equal to `v`; the exceptional postcondition (`raises`) states that whenever exception `Not_found` is raised, there is no such index within bounds whose value is equal to `v`. As usual in deductive verification, the presence of

[1] For the sake of space, we omit the definition of predicate `is_total_pre_order`.

```
let binary_search (cmp: 'a -> 'a -> int) (a: 'a array) (v: 'a) : int =
  let l = ref 0 in
  let u = ref (length a - 1) in
  let exception Found of int in
  try while !l <= !u do
      (*@ variant   !u - !l *)
      (*@ invariant 0 <= !l && !u < length a *)
      (*@ invariant forall i. 0 <= i < length a -> cmp a.(i) v = 0 ->
            !l <= i <= !u *)
      let m = !l + (!u - !l) / 2 in
      let c = cmp a.(m) v in
      if c < 0 then l := m + 1
      else if c > 0 then u := m - 1
      else raise (Found m)
    done;
    raise Not_found
  with Found i -> i
(*@ i = binary_search cmp a v
      requires is_total_pre_order cmp
      requires forall i j. 0 <= i <= j < length a -> cmp a.(i) a.(j) <= 0
      ensures  0 <= i < length a && compare a.(i) v = 0
      raises   Not_found -> forall i. 0 <= i < length a -> cmp a.(i) v <> 0 *)
```

Fig. 1. Binary search implemented as a higher-order function.

the `while` loop requires one to supply a loop invariant. Here, it boils down to the two `invariant` clauses, which state the limits of the search space are always within the bounds of `a` and that for every index `i` for which `a.(i)` is equal to `v`, then `i` must be within the limits of the current search space. We also provide a decreasing measure (`variant`) in order to prove loop termination.

Assuming file `binary_search.ml` contains the program of Fig. 1, starting a proof with Cameleer is as easy as typing `cameleer binary_search.ml` in a terminal. Users are immediately presented with the Why3 IDE, where they can conduct the proof. Twelve verification conditions are generated for `binary_search`: two for loop invariant initialization, four loop invariant preservation (two for each branch of `if..then..else`), two for safety (check division by zero and index in array bounds), two for loop termination (one for each branch), and finally one for each postcondition. All of these are easily discharged by SMT solvers.

Functor-based implementation. The implementation in Fig. 2 depicts (the skeleton of) an alternative implementation of the binary search routine. Instead of passing the comparison function as an argument of `binary_search`, here the functor `Make` takes as argument a module of type `OrderedType`, which provides a monomorphic comparison function over a type `t`. This is the same approach found in the OCaml standard library, namely in the `Set` and `Map` modules. The

```
module type OrderedType = sig
  type t

  val[@logic] cmp: t -> t -> int
  (*@ axiom total_pre_order: is_total_pre_order cmp *)
end

module Make (Ord: OrderedType) = struct
  let binary_search a v =
    ...
    try while !l <= !u do
        ...
        let c = Ord.cmp a.(m) in
        ...
    (*@ i = binary_search a v ... *)
end
```

Fig. 2. Binary search implemented as a functor.

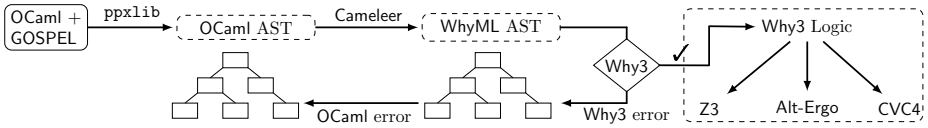

Fig. 3. Cameleer verification workflow.

`@logic` attribute instructs Cameleer that `cmp` is both a programming and logical function. This is what allows us to provide the axiom about the behavior of `cmp`.

Other than the call to `Ord.cmp`, the implementation and specification of `binary_search` does not change, hence we omit it here. When fed into Cameleer, the functorial implementation generates the *exact same* twelve verification conditions as the higher-order counterpart, all of them easily discharged as well. Thus, the use of a functor does not impose any verification burden, showing the flexibility of Cameleer to handle different idiomatic OCaml programming styles.

3 Implementation

Cameleer workflow. Fig. 3 depicts the verification workflow of the Cameleer tool. We use the GOSPEL toolchain[2], in order to parse and manipulate (via the `ppxlib` library) the abstract syntax tree of the GOSPEL-annotated OCaml program. A dedicated parser and type-checker (extended to handle implementation features) treat GOSPEL special comments and attach the generated specification to nodes in the OCaml AST. Cameleer translates the decorated AST into an equivalent WhyML representation, which is then fed to Why3. The Why3 type-and-effect

[2] `https://github.com/ocaml-gospel/gospel`

system might reject the input program, in which case the reported error is propagated back to the level of the original OCaml code. Otherwise, if the translated program fits Why3 requirements, the underlying VCGen computes a set of verification conditions that can then be discharged by different solvers. Throughout all this pipeline, the user only has to write the OCaml code and GOSPEL specification (represented in Fig. 3 as a full-lined box), while every other element is automatically generated (dash-lined boxes). The user never needs to manipulate or even care about the generated WhyML program. In short, the Cameleer user intervenes in the beginning and in the end of the process, *i.e.*, in the initial specifying phase and in the last step, helping Why3 to close the proof. Our development effort currently amounts to 1.8K non-blank lines of OCaml code.

Translation into WhyML. The core of Cameleer is a translation from GOSPEL-annotated OCaml code into WhyML. In order to guide our implementation effort, we have defined such a translation as a set of inductive inference rules between the source and target languages [26]. Here, rather than focusing on more fundamental aspects, we give a brief overview of how the translation works in practice.

OCaml and WhyML are both dialects of the ML-family, sharing many syntactic and semantics traits. Hence, translation of OCaml expressions and declarations into WhyML is rather straightforward: GOSPEL annotations are readily translated into WhyML specification, while supported OCaml programming constructions (including ghost code) are easily mapped into semantically-equivalent WhyML constructions. Consider, for instance the following piece of OCaml code:

```
type 'a non_empty_list = { self: 'a list }
(*@ invariant self <> [] *)

let[@ghost] hd (l: 'a non_empty_list) = match l with
  | [] -> assert false
  | x :: _ -> x
(*@ r = hd l
      ensures match l with
              | [] -> false
              | x :: _ -> r = x *)
```

For such case, Cameleer generates the following WhyML program:

```
type non_empty_list 'a = { self: list 'a }
invariant { self <> Nil }

let ghost hd (l: non_empty_list 'a)
  returns { r -> match l with
                 | Nil -> false
                 | Cons x _ -> x = r end }
= match l with
  | Nil -> absurd
  | Cons x _ -> x end
```

Other than the small syntactic differences, the generated WhyML program is identically to the original OCaml one. In particular, the `@ghost` annotation generates a ghost function in WhyML, while the `assert false` expression (which is treated in a special way by the OCaml type-checker) is translated into the `absurd` construction, with the same semantics. Supplied annotations, in this case post-condition and type invariant, are readily mapped into equivalent specification.

The translation of the OCaml module language is more interesting and involved. A WhyML program is a list of modules, a module is a list of top-level declarations, and declarations can be organized within *scopes*, the WhyML unit for namespaces management. However, there is no dedicated syntax for functors on the Why3 side. These are represented, instead, as modules containing only abstract symbols [17]. Thus, when translating OCaml functors into WhyML, we need to be more creative. If we consider, for instance, the `Make` functor from Fig. 2, Cameleer will generate the following WhyML program:

```
scope Make
  scope Ord
    type t

    val function cmp t t : int
    axiom total_pre_order: is_total_pre_order cmp
  end

  let binary_search a v = ...
end
```

The functor argument `Ord` is encoded as a nested scope inside `Make`. This means the `binary_search` implementation can access any symbol from the `Ord` namespace, via name qualification (*e.g.*, `Ord.t` and `Ord.cmp`).

Interaction with Why3. One distinguishing feature of the Why3 architecture is that it can be extended to accommodate new front-end languages [32, Chap. 4]. Building on the devised OCaml to WhyML translation scheme, we use the Why3 API to build an in-memory representation of the WhyML program. We also register OCaml as an admissible input language for Why3, which amounts to instructing Why3 to recognize `.ml` files as a valid input format and triggering our translation in such case. Following this integration, we can use any Why3 tool, out of the box, to process a `.ml` file. We are currently using the `extract` and `session` tools: the latter to gather statistics about number of generated verification conditions and proof time; the former to erase ghost code.

Limitations of using Why3. The WhyML specification sub-language and GOSPEL are similar. Moreover, they share some fundamental principles, namely the arguments of functions are not aliased by construction and each data structure carries an implicit representation predicate. However, one can use GOSPEL to formally specify OCaml programs which cannot be translated into WhyML. This

is evident when it comes to recursive mutable data structures. Consider, for instance, the `cell` type from the `Queue` module of the OCaml standard library[3]:

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

As we attempt to translate such data type, Why3 emits the following error:

```
This field has non-pure type, it cannot be used in a recursive
type definition
```

Recursive mutable data types are beyond the scope of Why3's type-and-effect discipline [14], since these can introduce arbitrary memory aliasing which breaks the *bounded-mutability* principle of Why3 (*i.e.*, all aliases must be statically-known). The solution would be to resort to an axiomatic memory model of OCaml in Why3, or to employ a richer program logic, *e.g.*, Separation Logic [28] or Implicit Dynamic Frames [31]. We describe such an extension as future work (Section 6).

4 Evaluation

In order to assess the usability and performance of Cameleer, we have put together a test suite of over 1000 lines of OCaml code. The reported case studies are all automatically verified. To build our gallery of verified programs we used a combination of Alt-Ergo 2.4.0, CVC4 1.8, and Z3 4.8.6. Fig. 4 summarizes important metrics about our verified case studies: the number of generated verification conditions for each example; the total lines of OCaml code, GOSPEL specification, and lines of ghost (these are also included in the number of OCaml LOC), respectively; the time it takes (in seconds) to replay a proof; and finally, if the proof is immediately discharged, *i.e.*, no extra user effort is required other than writing down suitable specification.

Our test bed includes OCaml implementations issued from realistic and massively used programming libraries: the `List.fold_left` iterator and `Stack` module from the OCaml standard library; the Leftist Heap implementation from `ocaml-containers`[4]; finally, the applicative `Queue` module from `OCamlgraph`[5]. We have used Cameleer to verify programs of different nature. These include: numerical programs (*e.g.*, binary multiplication and fast exponentiation); sorting and searching (*e.g.*, binary search and insertion sort); logical algorithms (conversion of a propositional formula into conjunctive normal form); array scanning (finding duplicate values in an array of integers); small-step iterators; data structures implemented as functors (*e.g.*, Pairing Heaps and Binary Search Trees); historical algorithms (checking a large routine by Turing, Boyer-Moore's majority algorithm, FIND by Hoare, and binary tree same fringe); examples in Rustain Leino's forthcoming textbook "Program Proofs"; and higher-order implementations (height of a binary tree computed in CPS). Both small-step iterators and

[3] `https://caml.inria.fr/pub/docs/manual-ocaml/libref/Queue.html`

[4] `https://github.com/c-cube/ocaml-containers/blob/master/src/core/CCHeap.ml`

[5] `https://github.com/backtracking/ocamlgraph/blob/master/src/lib/persistentQueue.ml`

Case study	# VCs	LOC / Spec. / Ghost	Proof time	Immediate
Applicative Queue	23	25 / 17 / 4	1.26	✓
Arithmetic Compiler	258	235 / 44 / 155	16.31	✗
Binary Multiplication	12	10 / 6 / 0	0.69	✓
Binary Search	37	62 / 40 / 0	1.23	✓
Binary Search Trees	31	20 / 26 / 0	1.45	✗
Checking a Large Routine	16	25 / 15 / 0	0.75	✓
CNF Conversion	93	113 / 47 / 14	2.92	✓
Duplicates in an Array	11	10 / 9 / 0	0.63	✓
Ephemeral Queue	44	40 / 29 / 7	1.34	✓
Even-odd Test	6	6 / 8 / 0	0.55	✓
Factorial	8	10 / 9 / 0	0.64	✓
Fast Exponentiation	5	4 / 5 / 0	0.62	✓
Fibonacci	15	16 / 15 / 2	0.64	✓
FIND Algorithm	6	13 / 7 / 0	0.57	✓
Insertion Sort	17	13 / 34 / 0	1.28	✓
Integer Square Root	11	8 / 15 / 0	0.63	✓
Leftist Heap	161	99 / 178 / 11	4.33	✓
Mjrty	25	33 / 12 / 0	2.56	✓
OCaml List.fold_left	28	5 / 21 / 0	0.79	✗
OCaml Stack	22	25 / 27 / 1	0.89	✓
Pairing Heap	70	65 / 101 / 29	2.30	✗
Program Proofs	63	93 / 54 / 24	1.60	✗
Same Fringe	23	22 / 16 / 0	0.78	✓
Small-step Iterators	46	42 / 52 / 2	2.01	✗
Tree Height CPS	4	8 / 8 / 0	0.80	✓
Union Find	67	36 / 29 / 7	6.19	✓

Fig. 4. Summary of the case studies verified with the Cameleer tool.

the `list_fold` function use a modular approach to reason about iteration [18]. Our largest case study to date is a toy compiler from arithmetic expressions to a stack machine, while Union Find features the most involved, but very elegant, specification. The former is inspired by the presentation in Nielsons' textbook [25]; the latter follows recently proposed specification techniques [7, 12] to achieve fully automatic proofs of correctness and termination.

The runtimes shown in Fig. 4 were measured by averaging over ten runs on a Lenovo Thinkpad X1 Carbon 8th Generation, running Linux Mint 20.1, OCaml 4.11.1, and Why3 1.3.3 (developer version). They show that Cameleer can effectively verify realistic OCaml code in a decent amount of time. Following good practices in deductive verification, Cameleer allows the user to write *ghost code* in order to ease proof and specification. The number of lines of ghost code in Fig. 4 stands for ghost fields in record types, ghost functions, and lemma functions. In particular, the arithmetic compiler example uses lemma functions to prove, by induction, results about semantics preservation. Finally, case studies marked with ✗ required some form of manual interaction in the Why3 IDE [9]. These

are very simple proofs by induction (of auxiliary lemmas) and case analysis, in order to better guide SMT solvers.

From our experience developing this gallery of verified programs, we believe the required annotation effort is reasonable, although non-negligible. Some case studies, namely the Heap implementations, feature a considerable amount of lines of GOSPEL specification. However, these are classic definitions (*e.g.*, minimum element) and results (*e.g.*, the root of the Heap is the minimum element), which are easily adapted to any variant of Heap implementation.

5 Related Work

Automated deductive verification. One can cite Why3, F* [1], Dafny [23], and Viper [24] as successful automated deductive verification tools. Formal proofs are conducted in the proof-aware language of these frameworks, and then executable reliable code can be automatically extracted. In the Cameleer project, we chose to develop a verification tool that accepts as input a program written directly in OCaml, instead of a dedicated proof language. This obviates the need to re-write entire OCaml codebases (*e.g.*, libraries), just for the sake of verification.

Regarding tools that tackle the verification of programs written in mainstream languages, one can cite Frama-C [21] (for the C language), VeriFast [20] (C and Java), Nagini [10] (Python), Leon [22] (Scala), Spec# [3] (C#), and Prusti [2] (Rust). Despite the remarkable case studies verified with these tools, programs written in the these languages can quickly degenerate into a nightmare of pointer manipulation and tricky semantics issues. We argue the OCaml language presents a number of features that make it a better target for formal verification.

Finally, language-based approaches offer an alternative path to the verification of software. Liquid Haskell [34] extends standard Haskell types with Liquid Types [29], a form of refinement types [30], in order to prove properties about realistic Haskell programs [33]. In this approach, verification conditions are generated and discharged during type-checking. This is also its major weakness: in order to remain decidable, the expressiveness of the refinement language is hindered. In Cameleer, the use of GOSPEL allows us to provide rich specification to relevant case studies, while still achieving good proof automation results.

Deductive verification of OCaml programs. Prior to our work, CFML [4] and `coq-of-ocaml` [8] were the only available tools for the deductive verification of OCaml-written code, via translation into the Coq proof language. On one hand, CFML features an embedding of a higher-order Separation Logic in Coq, together with a *characteristic formulae* generator. On the other hand, `coq-of-ocaml` compiles non-mutable OCaml programs to pure Gallina code. These two tools have been successfully applied to the verification of non-trivial case studies, such as the correctness and worst-case amortized complexity bound of cycle detection algorithm [19], as well as part of the Tezos' blockchain protocol[6].However, they

[6] `https://clarus.github.io/coq-of-ocaml/examples/tezos/`

still require a tremendous level of expertise and manual effort from users. Also, no behavioral specification is provided with the OCaml implementation. The user must write specification at the level of the generated code, which breaks our vision that implementation and specification must coexist and evolve together.

The VOCaL project aims at developing a mechanically verified OCaml library [6]. One of the main novelties of this project is the combined use of three different verification tools: Why3, CFML, and Coq. The GOSPEL specification language was developed in the scope of this project, as a tool-agnostic language that could be manipulated by any of the three mentioned frameworks. Up to this point, the three mentioned tools were only using GOSPEL for interface specification, and not as a proof language. We believe the Cameleer approach nicely complements the existing toolchains [13] in the VOCaL ecosystem.

6 Conclusions and Future Work

In this paper we presented Cameleer, a tool for automated deductive verification of OCaml programs, with bounded mutability. We use the recently proposed GOSPEL language, which we also extended in the scope of this work, in order to attach formal specification to an OCaml program. Cameleer fulfills a gap in the OCaml community, by providing programmers with a tool to directly specify and verify their implementations. By departing from the interactive-based approach, we believe Cameleer can be an important step towards bringing more OCaml programmers to include formal methods techniques in their daily routines.

The core of Cameleer is a translation from OCaml annotated code into WhyML. The two languages share many common traits (both in their syntax and semantics), so it makes sense to target this intermediate verification language in the first major iteration of Cameleer. We have successfully applied our tool and approach to the verification of several case studies. These include implementations issued from existing libraries, and scale up to data structures implemented as functors and tricky effectful computations. In the future, we intend to apply Cameleer to the verification of even larger case studies.

What we do not support. Currently, we target a subset of the OCaml language which roughly corresponds to `caml-light`, with basic support for the module language (including functors). Also, WhyML limits effectful computations to the cases where alias is information statically known, which limits our support for higher-order functions and mutable recursive data structures. Adding support for the objective layer of the OCaml language would require a major extension to the GOSPEL language and a redesign of our translation into WhyML. Nonetheless, Why3 has been used in the past to verify Java-written programs [15], so in principle an encoding of OCaml objects in WhyML is possible.

We do not support some of the more advanced type features in OCaml, namely Generalized Algebraic Data Types (GADTs) and polymorphic variants. One way to support such constructions would to be extend the type system of Why3 itself, which would likely mean a considerable redesign of the WhyML language.

Another possible route is to extend the core of Cameleer with the ability to translate OCaml code into other, richer, verification frameworks.

Interface with Viper and CFML. In order to augment the class of OCaml programs we can treat, we plan on extending Cameleer to target the Viper infrastructure and the CFML tool. On one hand, Viper is an intermediate verification language based on Separation Logic but oriented towards SMT-based software verification, allowing one to automatically verify heap-dependent programs. On the other hand, the CFML tool allows one to verify effectful higher-order programs. We plan on extending the CFML translation engine, in order to take source-code level GOSPEL annotations into account. Since it targets the rich proof language and type system of Coq, it can in principle be extended to reason about GADTs and other advanced OCaml features. Even if it relies on an interactive proof assistant, CFML provides a comprehensive tactics library that eases proof effort.

Our ultimate goal is to grow Cameleer to a verification tool that can simultaneously benefit from the best features of different intermediate verification frameworks. Our motto: we want Cameleer to be able to verify parts of OCaml code using Why3, others with Viper, and some very specific functions with CFML.

References

1. Ahman, D., Hritcu, C., Maillard, K., Martínez, G., Plotkin, G., Protzenko, J., Rastogi, A., Swamy, N.: Dijkstra monads for free. In: 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL). pp. 515–529. ACM (2017). https://doi.org/10.1145/3009837.3009878
2. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust Types for Modular Specification and Verification. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360573
3. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J., Muntean, T. (eds.) Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3362, pp. 49–69. Springer (2004). https://doi.org/10.1007/978-3-540-30569-9_3, `https://doi.org/10.1007/978-3-540-30569-9_3`
4. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 418–430 (2011). https://doi.org/10.1145/2034773.2034828
5. Charguéraud, A., Filliâtre, J., Lourenço, C., Pereira, M.: GOSPEL - Providing OCaml with a Formal Specification Language. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) 23rd Internation Symposium on Formal Methods (FM). LNCS, vol. 11800, pp. 484–501. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_29
6. Charguéraud, A., Filliâtre, J.C., Pereira, M., Pottier, F.: VOCAL – A Verified OCAml Library. In: ML Family Workshop (2017), `https://hal.inria.fr/hal-01561094`

7. Charguéraud, A., Pottier, F.: Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. J. Autom. Reason. **62**(3), 331–365 (2019). https://doi.org/10.1007/s10817-017-9431-7
8. Claret, G.: Program in Coq. (Programmer en Coq). Ph.D. thesis, Paris Diderot University, France (2018), `https://tel.archives-ouvertes.fr/tel-01890983`
9. Dailler, S., Marché, C., Moy, Y.: Lightweight interactive proving inside an automatic program verifier. In: 4th Workshop on Formal Integrated Development Environment (F-IDE) (2018)
10. Eilers, M., Müller, P.: Nagini: A Static Verifier for Python. In: Chockler, H., Weissenbacher, G. (eds.) 30th International Conference on Computer Aided Verification (CAV). LNCS, vol. 10981, pp. 596–603. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_33
11. Filliâtre, J.C.: Deductive Software Verification. International Journal on Software Tools for Technology Transfer (STTT) **13**(5), 397–403 (2011). https://doi.org/10.1007/s10009-011-0211-0
12. Filliâtre, J.C.: Simpler Proofs with Decentralized Invariants. Journal of Logical and Algebraic Methods in Programming (2020), `https://hal.inria.fr/hal-02518570`, To appear
13. Filliâtre, J.C., Gondelman, L., Lourenço, C., Paskevich, A., Pereira, M., Melo De Sousa, S., Walch, A.: A Toolchain to Produce Verified OCaml Libraries. Research report, Université Paris-Saclay (2020), `https://hal.archives-ouvertes.fr/hal-01783851`
14. Filliâtre, J.C., Gondelman, L., Paskevich, A.: A pragmatic type system for deductive verification. Research report, Université Paris Sud (2016), `https://hal.archives-ouvertes.fr/hal-01256434v3`
15. Filliâtre, J., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) 19th International Conference on Computer Aided Verification (CAV). LNCS, vol. 4590, pp. 173–177. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_21
16. Filliâtre, J., Paskevich, A.: Why3 - Where Programs Meet Provers. In: Felleisen, M., Gardner, P. (eds.) 22nd European Symposium on Programming, (ESOP). LNCS, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
17. Filliâtre, J., Paskevich, A.: Abstraction and Genericity in Why3. In: Margaria, T., Steffen, B. (eds.) 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA). LNCS, vol. 12476, pp. 122–142. Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_7
18. Filliâtre, J., Pereira, M.: A Modular Way to Reason About Iteration. In: Rayadurgam, S., Tkachuk, O. (eds.) 8th NASA Formal Methods Symposium (NFM). LNCS, vol. 9690, pp. 322–336. Springer (2016). https://doi.org/10.1007/978-3-319-40648-0_24
19. Guéneau, A., Jourdan, J., Charguéraud, A., Pottier, F.: Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, (ITP). LIPIcs, vol. 141, pp. 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.18
20. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) 3rd NASA Formal Methods Symposium. LNCS, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4

21. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A Software Analysis Perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
22. Kuncak, V.: Developing verified software using leon. In: Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9058, pp. 12–15. Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_2, `https://doi.org/10.1007/978-3-319-17524-9_2`
23. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). LNCS, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
24. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) 17th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 9583, pp. 41–62. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2
25. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science, Springer (2007). https://doi.org/10.1007/978-1-84628-692-6
26. Pereira, M., Ravara, A.: Cameleer: a Deductive Verification Tool for OCaml. CoRR (2021), `https://arxiv.org/abs/2104.11050`
27. Régis-Gianas, Y., Pottier, F.: A Hoare Logic for Call-by-Value Functional Programs. In: Audebaud, P., Paulin-Mohring, C. (eds.) 9th International Conference on Mathematics of Program Construction (MPC). LNCS, vol. 5133, pp. 305–335. Springer (2008). https://doi.org/10.1007/978-3-540-70594-9_17
28. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. p. 55–74. LICS '02, IEEE Computer Society, USA (2002)
29. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid Types. In: Gupta, R., Amarasinghe, S.P. (eds.) 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 159–169. ACM (2008). https://doi.org/10.1145/1375581.1375602
30. Rushby, J., Owre, S., Shankar, N.: Subtypes for Specifications: Predicate Subtyping in PVS. IEEE Transactions on Software Engineering **24**(9), 709–720 (1998). https://doi.org/10.1109/32.713327
31. Smans, J., Jacobs, B., Piessens, F.: Implicit Dynamic Frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012). https://doi.org/10.1145/2160910.2160911
32. The Why3 Development Team: The Why3 platform, version 1.3.3. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay (2020), `http://why3.lri.fr/manual.pdf`
33. Vazou, N., Breitner, J., Kunkel, R., Horn, D.V., Hutton, G.: Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). In: Wu, N. (ed.) 11th ACM SIGPLAN International Symposium on Haskell. pp. 132–144. ACM (2018). https://doi.org/10.1145/3242744.3242756
34. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Jones, S.L.P.: Refinement Types for Haskell. In: Jeuring, J., Chakravarty, M.M.T. (eds.) 19th ACM SIGPLAN international conference on Functional programming (ICFP). pp. 269–282. ACM (2014). https://doi.org/10.1145/2628136.2628161