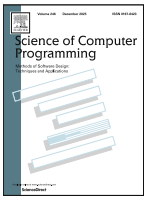
Science of Computer Programming 253 (2026) 103507

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico



Original Software Publication

Practical deductive verification of OCaml programs: The Cameleer tool

Mário Pereira

NOVA LINCS, NOVA School of Science and Technology, Lisbon, Portugal

ARTICLE INFO

Keywords:
OCaml
Cameleer
Gospel
Why3
Deductive verification

ABSTRACT

Despite all the tremendous recent advances in deductive software verification, this has seldom been applied to the verification of programs written in a functional language. In this paper, we give an overview of Cameleer, a verification tool that directly targets OCaml code and is built with a clear focus on proof automation. We leverage on Gospel (Generic OCaml SPEcification Language) to provide formal specification to OCaml implementations. We use a running example to illustrate the specification and verification process employed by Cameleer. Finally, we highlight the crucial impact of Cameleer in recent research and development projects.

Code metadata (Mandatory).

Nr.	Code metadata description	Please fill in this column
C1	Current code version	0.1 dev
C2	Permanent link to code/repository used for this code version	https://github.com/ocaml-gospel/cameleer/
C3	Permanent link to Reproducible Capsule	https://doi.org/10.5281/zenodo.12588707
C4	Legal Code License	MIT License
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	OCaml
C7	Compilation requirements, operating environments and dependencies	OCaml compiler, Gospel package, Why3 framework
C8	If available, link to developer documentation/manual	https://github.com/ocaml-gospel/cameleer/blob/master/README.md
C9	Support email for questions	mjp.pereira@fct.unl.pt

1. Introduction

We present Cameleer, an automated deductive verification tool for OCaml code. It fills a gap in the OCaml community, by providing programmers with a tool to directly specify and verify their implementations, with a clear focus on proof automation. We use Gospel [1] to attach rigorous, yet readable, behavioral specification to OCaml code. Gospel follows the tradition of behavioral interface specification language [2], but we extend it in the scope of the Cameleer project to cope with OCaml implementations (*e.g.*, provide loop invariants, termination measures, or intermediate assertions). So far, we have successfully applied Cameleer to verify more than 1k lines of OCaml code, showing the tool scales well in practice.

E-mail address: mjp.pereira@fct.unl.pt

https://doi.org/10.1016/j.scico.2026.103507
Received 10 October 2025; Received in revised form 25 February 2026; Accepted 3 May 2026
Available online 18 May 2026

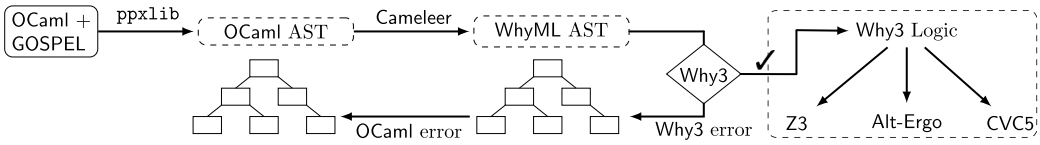


Fig. 1. Cameleer architecture and verification pipeline, taken from Pereira and Ravara [3].

In this paper, we give an overview of the Cameleer architecture and use a running example to showcase its specification and verification capabilities. Finally, we provide a list of relevant projects, where Cameleer has been used and extended for research purposes. Cameleer was initially presented at the CAV'21 conference [3] and was recently used as the vehicle for a tutorial at the FM'24 symposium [4]. Finally, a publicly available artifact [5] contains the latest developments on the Cameleer tool.

2. Software description

A typical use of the Cameleer tool proceeds as follows: (i) given an OCaml implementation, the user must specify it using the Gospel specification language; (ii) Cameleer takes as input annotated-OCaml code and automatically generates an equivalent WhyML program, the specification and verification language of the Why3 framework [6]; (iii) the Why3 *Verification Conditions Generator* (VCGen) produces a set of *Verification Conditions* (VCs) for the translated WhyML program. Discharging all generated VCs implies the original OCaml code adheres to its Gospel specification. Fig. 1 presents the main Cameleer verification workflow. Throughout this pipeline, the user must only provide the OCaml implementation, together with its Gospel specification (full-lined boxes). Every other element is automatically generated (dash-lined boxes). This means the user never manipulates, or even knows about the generated WhyML program. However, the user might interact with the Why3 graphical user interface (IDE) [7] to select provers, apply transformations to VCs (*e.g.*, splitting), or even inspect the proof goal. The latter is presented as a set of hypotheses and a formula to be proved, written in the purely-logical fragment of the WhyML language.

3. Illustrative example – most frequent element in a sorted array

Let us consider the following programming challenge: *write a function that returns the most frequent value in an array*. This is a rather simple exercise. One might immediately consider using an auxiliary data structure, for instance an hash table, to build an histogram for the number of occurrences of each element. Hence, solve the above challenge in linear time and space. Now, let us add an extra piece to our puzzle: *what if the array is sorted?* In that case, one can actually solve it in linear time and constant (extra) space.

In this section, we use an OCaml implementation of the most frequent value in a sorted array challenge as our running example to illustrate the use of the Cameleer tool. We first explain how we provide a Gospel specification to an OCaml program, and then how to use Cameleer to conduct the proof that the code actually adheres to its specification. For the sake of presentation, we consider only a simpler variant of the challenge, where we restrict the input array to contain only integer values. Fig. 2 presents the complete OCaml implementation and Gospel specification of the most_frequent function. The online development [8] provides a solution that is generic on the type of elements in the array.

Specification elements. Gospel elements are enclosed within OCaml comments of the form `(*@ …*)`. Lines 1 to 4 introduce two specification-only symbols. First, a *predicate*, is_sorted, that captures what it means for an array to be sorted in natural order. Then, a *logical function*, numof_eq, which stands for the number of elements in an array a, within range `[l;u)`, that are equal to a value v. For the sake of space, we keep numof_eq as an abstract function. The complete definition of numof_eq, as well as auxiliary lemmas on its use, can be found online [8].

Predicate and logical functions are written in first-order logic. Note the universal quantification, in predicate is_sorted, over two variables, i and j, of type integer. The Gospelinteger type stands for the type of mathematical, ideal integers. It should not be confused with int, the OCaml type of machine integers. Finally, one can use OCaml functions within logical definitions. This is the case of Array.length and the access to an element in the array, using the .() notation.

Function contract. The behavioral specification for function most_frequent is provided as a Gospel annotation, attached after the function definition, lines 21–26. The first line gives a name to the returned value, r here. Next, we provide a list of preconditions and postconditions via the requires and ensures clauses, respectively. The preconditions of most_frequent state that the array must be non-empty and it should be sorted (note the use of the is_sorted predicate).

The postcondition of most_frequent states two essential properties: on one hand, the returned value is indeed an element of the array (*i.e.*, its number of occurrences is strictly greater than zero); on the other hand, for any element x (whose type is inferred by the Gospel type-checker) its number of occurrences is never greater than the number of occurrences of r, the returned element. Together, these two formulae fully capture the notion of being the most frequent value within an array.

Loop invariant. To conclude the specification of the OCaml implementation, one must provide a suitable *loop invariant*. The first condition of the invariant, line 11, states that reference c stores the number of occurrences for the current candidate to most frequent

```
1  (*@ predicate is_sorted (a: int array) = forall i j: integer.
2        0 <= i <= j < Array.length a -> a.(i) <= a.(j) *)
3
4  (*@ function numof_eq (a : 'a array) (v : 'a) (l u: integer) : integer *)
5
6  let most_frequent (a: int array) : int =
7    let c = ref 1 in
8    let r = ref a.(0) in
9    let m = ref 1 in
10   for i = 1 to Array.length a - 1 do
11     (*@ invariant !c = numof_eq a a.(i-1) 0 i
12         invariant !m = numof_eq a !r 0 i
13         invariant forall x. numof_eq a x 0 i <= !m *)
14     if a.(i) = a.(i-1) then begin
15       incr c;
16       if !c > !m then begin m := !c; r := a.(i) end
17     end else
18       c := 1
19   done;
20   !r
21 (*@ r = most_frequent a
22     requires Array.length a > 0
23     requires is_sorted a
24     ensures  numof_eq a r 0 (Array.length a) > 0
25     ensures  forall x. numof_eq a x 0 (Array.length a) <=
26                        numof_eq a r 0 (Array.length a) *)
```

Fig. 2. Most frequent element in a sorted array, implementation and specification.

element in the array. We use numof_eq to capture the number of occurrences of the candidate a.(i-1) up until the current iteration, *i.e.*, in range $[0, i)$ of the array a.

References r and m store, respectively, the most frequent element up until the current iteration and the maximum number of occurrences so far. The second invariant condition, line 12, states precisely the relation between the values stored in r and m.

The last condition, line 13, captures the fact that m stores the maximum number of occurrences so far. For any element x, it must be the case that its number of occurrences is less than or equal to m.

Proof effort. When the specification and implementation of most_frequent are given as input to Cameleer, the tool automatically translates it into an equivalent WhyML program. This is then fed to the Why3 VCGen, which generates a total of 19 VCs. These range from *safety* properties, namely array access within bounds, to *functional correctness*, namely invariant initialization and preservation, as well as postcondition derivation. Termination is assured by construction, since we use a for loop. All of the generated VCs are automatically discharged, using a combination of the Alt-Ergo and Z3 SMT solvers.

The complete implementation and specification of most_frequent, available online [8], also provides a *functorial*-based solution for this challenge. Using a functor, we abstract away the type of elements within the array, only requiring that these are equipped with a total pre-order relation. This is achieved by attaching Gospel specification to the functor argument. Cameleer is able to deal with functors, even if these are not primitive in Why3 [9].

Verification Failure. Let us consider a scenario where one would have specified the following, incorrect, postcondition for function most_frequent:

```
let most_frequent (a: int array) : int = ...
(*@ r = most_frequent a
    ...
    ensures forall x. numof_eq a x 0 (Array.length a) <
                      numof_eq a r 0 (Array.length a) *)
```

The difference with respects to the specification in Fig. 2 is that the last postcondition now states that, for any element x, its number of occurrences in a is *strictly* less than the number of occurrences of the result r. Such a postcondition is incorrect since it does not even take into account the occurrences of r itself within a.

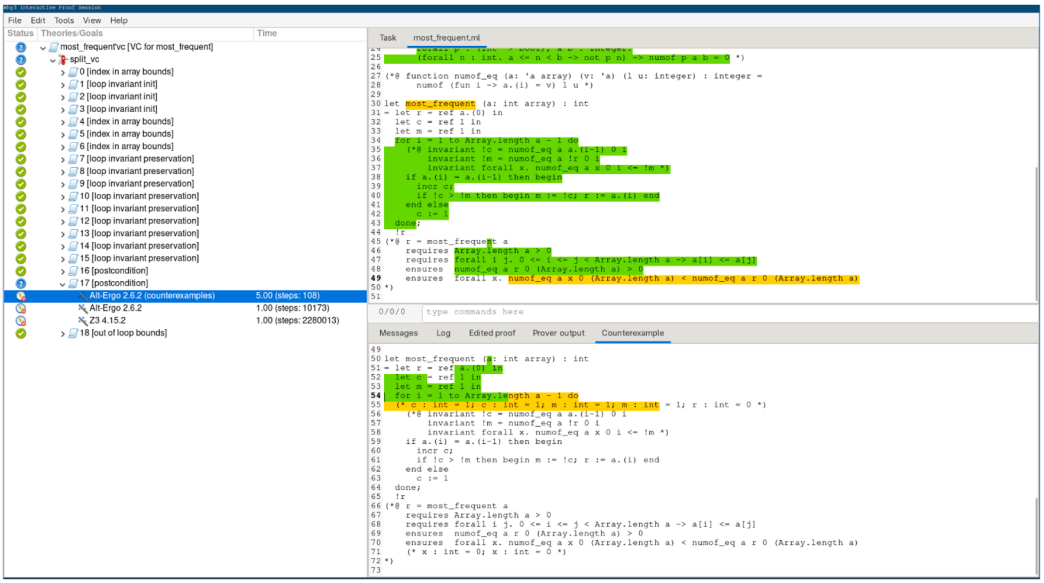


Fig. 3. Counterexample generation for an incorrect postcondition.

Fig. 3 shows the Why3 graphical user interface, resulting from feeding the new specification of function most_frequent to the Cameleer-Why3 pipeline. On left-hand side, the Why3 IDE features a node for each generated VC of function most_frequent. A green button on the left of a node means that a solver was able to discharge the corresponding VC. Hence, the only VC one is not able to prove, in this example, is that the new postcondition holds. Both the Alt-Ergo and Z3 solvers timeout after 1 second.

To debug such a proof attempt, we can ask specific solvers for a *counterexample*. Fig. 3 depicts the use of Alt-Ergo counterexample generation mechanism, which can be achieved by typing the string `alt-ergo-ce` in the interactive command bar (right-hand side, middle of the window). After a few seconds, the solver returns a candidate counterexample, shown under the `Counterexample` tab. This is basically an assignment for the manipulated references: on line 55, it shows the values stored after exiting the loop; on line 71, it chooses 0 to instantiate x in the postcondition. By assigning 0 to r, this counterexample thus exposes a possible scenario where the value returned by most_frequent does not adhere to the newly supplied specification.

4. Impact

Other than the goal of providing a robust, mostly-automated verification tool for the OCaml community, Cameleer has successfully served as a vehicle to conduct applied and fundamental research on deductive verification of functional programs. In particular, Cameleer is used and extended in the scope of several Master and PhD theses. In this section, we highlight the most recent research projects where Cameleer played a crucial role.

Specification and verification of algebraic effects. Version 5 of the OCaml compiler, best known as *multicore OCaml*, was an important milestone in the development of this programming language. As the name suggests, it brought multi-core features to OCaml: shared-memory parallelism [10] and direct-style concurrency. The latter is achieved by the so-called *algebraic effects* [11].

Algebraic effects are akin to exceptions, with the difference that effects expose, in the nearest handler, a continuation that allows one to resume the program execution where the effect was performed. Since Cameleer supports reasoning about exceptions, we took this as our starting point to develop a framework to automatically reason about effects [12]. We devised an embedding of effects and handlers in WhyML, based on exceptions and defunctionalization [13], and extended Cameleer to automatically translate OCaml programs into this embedding. Finally, following the work of de Vilhena and Pottier [14], we extended the Gospel language with a notion of *protocol*, which allows us to reason about the interaction between the effect and its handler. We put together a set of verified non-trivial uses of algebraic effects, including a co-routine based implementation of the Koda-Ruskey algorithm [15].

Reasoning about higher-order iteration. The use of higher-order iterators is ubiquitous in every functional programming language. In OCaml, for instance, one can find several instances of fold or iter-like functions in the standard library. However, reasoning about higher-order, effectful iteration is known to be challenging for automated verification tools.

We extended Gospel with new specification clauses targeting iteration only, as well as implemented in Cameleer a verification methodology to automatically prove clients of higher-order iterators [16]. Following the work by Filliâtre and Pereira [17], we propose to specify iteration only by means of two logical relations: what are the valid sub-sequences of enumerated elements, and when does the iteration halt. Finally, from a verification perspective, we translate uses of higher-order iterators into equivalent

cursors-based iteration. We have successfully applied our methodology to the formal verification of several modules of OCamlGraph, a widely-used library where the many forms of graph iterations are always expressed via higher-order iterators.

Verification of OCaml heap-dependent programs. With the advent of Separation Logic [18], many verification tools emerged to support reasoning about programs that deal with dynamic memory. In particular, we have witnessed the extraordinary development of automated verification tools [19–21], whose underlying logic is a variant of Separation Logic.

Given its imperative layer, one can use OCaml to implement heap-dependent data structures and programs. These are natural targets for verification using Cameleer. However, Why3 does not support reasoning about arbitrary use of pointers, mainly due to its bounded type-and-region system [22].

In order to circumvent Why3 limitations, we introduce a new backend to Cameleer [23]. Instead of relying only on a translation to WhyML, we provide the tool with the ability to also translate Gospel-annotated OCaml programs into Viper, a verification language supporting Implicit Dynamic Frames [24], a variant of Separation Logic amenable to proof automation. We used the Cameleer-Viper pipeline to verify several heap-dependent OCaml implementations, in particular the Queue module from the standard library.

5. Conclusions and future work

Cameleer is tool for the automated verification of OCaml programs. It accepts as input a Gospel-annotated OCaml implementation, and translates it into an equivalent WhyML program. Cameleer avoids the need to re-write entire OCaml code bases, just for the sake of verification, as it would be the case if one used directly a tool like Why3. In this paper, we used a running example to illustrate the use of Cameleer to prove that an OCaml function adheres to its logical specification. Finally, we presented projects that highlight the impact of Cameleer, not only as a usable verification tool, but also as a vehicle to conduct research on formal verification.

As future work, we anticipate two main avenues. On one hand, we plan to apply and adapt Cameleer to reason about industrial, safety-critical systems, in particular those powered by the Unikernels approach [25]. On the other hand, one must always take into account the fact there is a strong connection between Gospel and Cameleer. As the specification language evolves, so does Cameleer, in order to cope with an even larger family of OCaml programs. This naturally leads to extensions to our WhyML translation but, most importantly, it highlights the need to target multiple intermediate verification languages. As an example, we are currently working on Gospel extensions to provide more fine-grained control over ownership of mutable data structures. As such, we will extend Cameleer to target Separation Logic-based environments: on one hand, continue our work on the Viper translation; on the other hand, to interface with richer frameworks such as CFML [26] or Osiris [27].

CRediT authorship contribution statement

Mário Pereira: Writing – review & editing.

Data availability

None.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

I sincerely thank all my colleagues and students that, over the past years, have advised and encouraged me in developing Cameleer, or even contributed themselves directly to the development of the tool. In alphabetic order of their last name: Daniel Castanho, Arthur Charguéraud, Ion Chirica, Jean-Christophe Filliâtre, Pedro Gasparinho, Charlène Gros, Jorge Sousa Pinto, François Pottier, António Ravara, Tiago Soares, Simão Melo de Sousa.

This work was partly supported by the HORIZON 2020 Cameleer project (Marie Sklodowska-Curie grant agreement ID:897873), Agence Nationale de la Recherche (ANR) grant ANR-22-CE48-0013-01 (GOSPEL) and NOVA LINCS (UID/04516/2025) with the financial support of FCT.IP (https://doi.org/10.54499/UID/04516/2025).

References

[1] A. Charguéraud, J.-C. Filliâtre, C. Lourenço, M. Pereira, GOSPEL-providing OCaml with a formal specification language, in: M.H. ter Beek, A. McIver, J.N. Oliveira (Eds.), Formal Methods – The Next 30 Years, Springer, 2019, pp. 484–501. https://doi.org/10.1007/978-3-030-30942-8_29

[2] J. Hatcliff, G.T. Leavens, K.R.M. Leino, P. Müller, M.J. Parkinson, Behavioral interface specification languages, ACM Comput. Surv. 44 (3) (2012) 16:1–16:58. https://doi.org/10.1145/2187671.2187678

[3] M. Pereira, A. Ravara, Cameleer: a deductive verification tool for OCaml, in: A. Silva, K.R.M. Leino (Eds.), Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, 2021, Proceedings, Part II, 12760 of Lecture Notes in Computer Science, Springer, 2021, pp. 677–689. https://doi.org/10.1007/978-3-030-81688-9_31

[4] M. Pereira, Practical deductive verification of OCaml programs, in: A. Platzer, K.Y. Rozier, M. Pradella, M. Rossi (Eds.), Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9–13, 2024, Proceedings, Part II, 14934 of Lecture Notes in Computer Science, Springer, 2024, pp. 518–542. https://doi.org/10.1007/978-3-031-71177-0_29

[5] M. Pereira, Practical Deductive Verification of OCaml Programs, 2024. https://doi.org/10.5281/zenodo.12588707

[6] J. Filliâtre, A. Paskevich, Why3 - where programs meet provers, in: M. Felleisen, P. Gardner (Eds.), 22nd European Symposium on Programming, (ESOP), 7792 LNCS, Springer, 2013, pp. 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

[7] S. Dailler, C. Marché, Y. Moy, Lightweight interactive proving inside an automatic program verifier, in: 4th Workshop on Formal Integrated Development Environment (F-IDE), 2018.

[8] M. Pereira, Most frequent element in a sorted array, 2025. Formal Proof Development, https://github.com/ocaml-gospel/cameleer/blob/master/examples/most_frequent.ml.

[9] J. Filliâtre, A. Paskevich, Abstraction and genericity in Why3, in: T. Margaria, B. Steffen (Eds.), 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA), 12476 of LNCS, Springer, 2020, pp. 122–142. https://doi.org/10.1007/978-3-030-61362-4_7

[10] K.C. Sivaramakrishnan, S. Dolan, L. White, S. Jaffer, T. Kelly, A. Sahoo, S. Parimala, A. Dhiman, A. Madhavapeddy, Retrofitting parallelism onto OCaml, Proc. ACM Program. Lang. 4 (ICFP) (2020). https://doi.org/10.1145/3408995

[11] K.C. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, A. Madhavapeddy, Retrofitting effect handlers onto OCaml, in: S.N. Freund, E. Yahav (Eds.), PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20–25, 2021, ACM, 2021, pp. 206–221. https://doi.org/10.1145/3453483.3454039

[12] T. Soares, M. Pereira, A framework for the automated verification of algebraic effects and handlers (Extended Version), 2023. https://arxiv.org/abs/2302.01265.

[13] J.C. Reynolds, Definitional interpreters for higher-order programming languages, in: Proceedings of the ACM Annual Conference - Volume 2, ACM '72, Association for Computing Machinery, New York, NY, USA, 1972, p. 717–740. https://doi.org/10.1145/800194.805852

[14] P.E. de Vilhena, F. Pottier, A separation logic for effect handlers, Proc. ACM Program. Lang. 5 (POPL) (2021). https://doi.org/10.1145/3434314

[15] J. Filliâtre, M. Pereira, Producing all ideals of a forest, formally (Verification Pearl), in: S. Blazy, M. Chechik (Eds.), Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers, 9971 of Lecture Notes in Computer Science, 2016, pp. 46–55. https://doi.org/10.1007/978-3-319-48869-1_4

[16] I. Chirica, M. Pereira, Unfolding iterators: specification and verification of higher-order iterators, in OCaml, 2025. (Accepted for publication at the 20th International Conference on Integrated Formal Methods, iFM 2025), https://arxiv.org/abs/2506.20310.

[17] J. Filliâtre, M. Pereira, A modular way to reason about iteration, in: S. Rayadurgam, O. Tkachuk (Eds.), 8th NASA Formal Methods Symposium (NFM) 9690 of LNCS, Springer, 2016, pp. 322–336. https://doi.org/10.1007/978-3-319-40648-0_24

[18] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS '02, IEEE Computer Society, USA, 2002, p. 55-74.

[19] P. Müller, M. Schwerhoff, A.J. Summers, Viper: a verification infrastructure for permission-based reasoning, in: B. Jobstmann, K.R.M. Leino (Eds.), 17th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), 9583 of LNCS, Springer, 2016, pp. 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[20] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, F. Piessens, VeriFast: a powerful, sound, predictable, fast verifier for C and Java, in: M.G. Bobaru, K. Havelund, G.J. Holzmann, R. Joshi (Eds.), 3rd NASA Formal Methods Symposium 6617 of LNCS, Springer, 2011, pp. 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[21] P. Maksimovic, S. Ayoun, J.F. Santos, P. Gardner, Gillian, Part II: real-world verification for JavaScript and C, in: A. Silva, K.R.M. Leino (Eds.), Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II, 12760 of Lecture Notes in Computer Science, Springer, 2021, pp. 827–850. https://doi.org/10.1007/978-3-030-81688-9_38

[22] J.-C. Filliâtre, L. Gondelman, A. Paskevich, A Pragmatic Type System for Deductive Verification, Research Report, Université Paris Sud, 2016. https://hal.archives-ouvertes.fr/hal-01256434v3.

[23] C. Gros, M. Pereira, Le chameau et le serpent rentrent dans un bar: vérification quasi-automatique de code OCaml en logique de séparation, in: 36es JournÉes Francophones des Langages Applicatifs (JFLA 2025), 2025. https://arxiv.org/abs/2412.14894.

[24] J. Smans, B. Jacobs, F. Piessens, Implicit dynamic frames, ACM Trans. Program. Lang. Syst. 34 (1) (2012) 2:1–2:58. https://doi.org/10.1145/2160910.2160911

[25] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, J. Crowcroft, Unikernels: library operating systems for the cloud, ACM SIGARCH Comput. Archit. News 41 (1) (2013) 461–472.

[26] A. Charguéraud, Characteristic formulae for the verification of imperative programs, in: M.M.T. Chakravarty, Z. Hu, O. Danvy (Eds.), ACM SIGPLAN International Conference on Functional Programming (ICFP), 2011, pp. 418–430. https://doi.org/10.1145/2034773.2034828

[27] R. Seassau, I. Yoon, J.-M. Madiot, F. Pottier, Formal semantics and program logics for a fragment of OCaml, Proc. ACM Program. Lang. 9 (ICFP) (2025). https://doi.org/10.1145/3747509