Deductive Verification of Realistic OCaml Code*

Carlos Pinto[1], Mário Pereira[2], and Simão Melo de Sousa[1]

[1] NOVA LINCS, Universidade da Beira Interior, Portugal
[2] NOVA LINCS, Nova School of Science and Technology, Portugal

Introduction. Cameleer [6] is a deductive verification platform for OCaml programs, actively developed over the last year, built on top of the Why3 framework [4]. It leverages on the recently proposed GOSPEL [1] (*G*eneric *O*Caml *SPE*cification *L*anguage), which allows programmers to attach rigorous, yet readable, behavioral specification to OCaml code. GOSPEL has been already used in a framework for extracting correct-by-construction OCaml implementations with respect to provided, annotated interfaces [2]. With Cameleer, we have pushed this approach further, by centering the verification effort directly on the OCaml implementation. We believe Cameleer has now reached enough maturity to tackle the verification of realistic OCaml code. This presentation proposal reports this claim, namely on our experience of using the Cameleer tool to provide a complete proof of the `Set` module from the OCaml standard library[3].

Deductive verification framework. The workflow of the verification toolchain we report is depicted as follows:

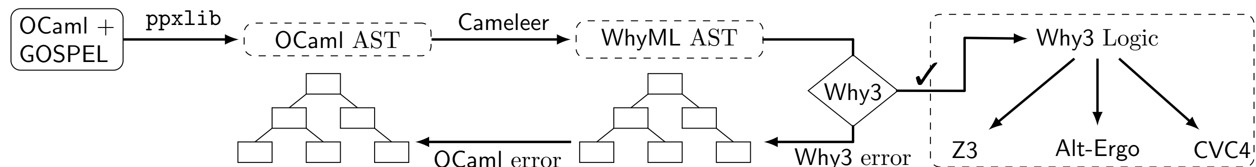

Fig. 1. Cameleer verification workflow.

Cameleer uses the GOSPEL toolchain, in order to parse and manipulate (via the `ppxlib` library) the abstract syntax tree of GOSPEL-annotated OCaml programs. Cameleer translates the decorated AST into an equivalent WhyML representation, which is then fed to Why3. The user never needs to manipulate 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.

Module `Set`. In order to assess the maturity of the Cameleer tool, several OCaml modules implementing data structures and algorithms have been verified. We focus on this presentation on the verification of the `Set` module of the standard library. This is an interesting module, since it is a realistic piece of OCaml code, widely-used, and presenting subtle design choices.

The following, is an excerpt of the functorial implementation of `Set`:

```
module type OrderedType = sig
  type t

  val[@logic] compare : t -> t -> int
  (*@ axiom is_pre_order: is_pre_order compare *)
end
```

* 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)

[3] https://ocaml.org/api/Set.html

```
module Make(Ord: OrderedType) = struct

  let bal l v r =
    ...
  (*@ res = bal l v r
        requires avl l && avl r
        requires abs (height l - height r) <= 3
        requires forall y. mem y l -> Ord.compare y v < 0
        requires forall x. mem x r -> Ord.compare v x < 0
        requires forall x y. mem x l -> mem y r -> Ord.compare x y < 0
        ensures  avl res
        ensures  forall w. w <> v -> occ w res = occ w l + occ w r
        ensures  occ v res = 1
        ensures      max (height l) (height r) <= height res <=
                 1 + max (height l) (height r)
        ensures  abs (height l - height r) <= 2 -> res = create l v r *)
```

The rebalancing function `bal` is here presented together with a suitable GOSPEL contract. Cameleer and GOSPEL allows the programmer to define logical definitions, for instance to describe what is a well-balanced binary search tree, according to the balance criteria used in the `Set` module.

Further details will be provided during the presentation itself. In a few words, the `bal` function takes as arguments two balanced trees, `l` and `r`, a new element `v` and builds a new balanced tree containing `v` and all the elements from `l` and `r`. For such a function it is required for `l` and `r` to be indeed balanced trees (this is captured by the recursive predicate `avl`). We use the `occ` recursive logical function to count the number of occurrences of a certain element inside a tree. Such a predicate allows us to write very simple and elegant contracts, such as all the elements distinct from `v` maintain their occurrence in the resulting tree, while the occurrence of `v` is exactly one. The other specification elements have the usual semantics. It is worth mentioning that the informal comments provided in the `set.ml` source file are very clear and can be, almost directly, translated into GOSPEL specification.

Conclusion. Our proof of a subset of the operations from the `Set` module is publicly available[4]. It amounts to 122 lines of code and 117 lines of GOSPEL specification. Cameleer automatically feeds to Why3 a translation of the selected `Set` functions, which generates a total of 521 Verification Conditions (VCs). The entire proof can be replayed in an average time of 66 seconds, measured on a laptop equipped with macOS Catalina, with 8 GB of Ram and 2 CPU cores (2.6 GHz).

The proof effort was conducted mainly by an undergraduate student, with no previous experience on formal verification. This was intentional, in order to assess our claim on the maturity and ease of use of the Cameleer toolchain. We honestly believe our experience is an important contribution towards the effort of building a corpus of realistic, formally verified OCaml modules. Regarding previous verification efforts of AVL data structures, in particular the work by Filliâtre and Letouzey [3], our work clearly improves on proof automation. We believe this is an important aspect, since automated verification approaches are more likely to be adopted by regular programmers, who are not necessarily proof experts. Moreover, the Cameleer approaches precludes the need of OCaml code extraction (which is the approach followed by Coq and Why3, for instance), and for programmers to write entire code bases for the sake of verification.

As future work, we intend to specify and prove higher-order functions from module `Set`. As a first step, we will tackle the verification of higher-order computation where is reasonable to assume the argument function is effect-free (*e.g.*, `map` and `fold`). Finally, a more challenging approach is the proof of higher-order functions with effects, for instance `iter`. Currently, neither Why3 or GOSPEL support effectful higher-order computation. We plan to follow the approach by Filliâtre and Pereira [5] for the modular treatment of iteration structures, and extend the core Cameleer to cope with the proposed translation of `iter`-like routines.

[4] `https://github.com/ocaml-gospel/cameleer/blob/testing_carlos/examples/Set.ml`

References

1. Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. GOSPEL - Providing OCaml with a Formal Specification Language. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *23rd Internation Symposium on Formal Methods (FM)*, volume 11800 of *LNCS*, pages 484–501. Springer, 2019.
2. Jean-Christophe Filliâtre, Léon Gondelman, Andrei Paskevich, Mário Pereira, and Simão Melo de Sousa. A toolchain to Produce Correct-by-Construction OCaml Programs. Technical report, 2018. artifact: `https://www.lri.fr/~mpereira/correct_ocaml.ova`.
3. Jean-Christophe Filliâtre and Pierre Letouzey. Functors for proofs and programs. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2004.
4. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - Where Programs Meet Provers. In Matthias Felleisen and Philippa Gardner, editors, *22nd European Symposium on Programming, (ESOP)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
5. Jean-Christophe Filliâtre and Mário Pereira. A Modular Way to Reason About Iteration. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *8th NASA Formal Methods Symposium (NFM)*, volume 9690 of *LNCS*, pages 322–336. Springer, 2016.
6. Mário Pereira and António Ravara. Cameleer: a Deductive Verification Tool for OCaml. *32 ème Journées Francophones des Langages Applicatifs*, 2021.