A Toolchain to Produce Correct-by-Construction OCaml Programs

Jean-Christophe Filliâtre, Léon Gondelman, Andrei Paskevich, Mário Pereira, Simão Melo de Sousa

To cite this version:

Jean-Christophe Filliâtre, Léon Gondelman, Andrei Paskevich, Mário Pereira, Simão Melo de Sousa. A Toolchain to Produce Correct-by-Construction OCaml Programs. 2018. <hal-01783851>

HAL Id: hal-01783851
https://hal.inria.fr/hal-01783851

Submitted on 2 May 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Toolchain to Produce Correct-by-Construction OCaml Programs

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

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] Inria Saclay – Île-de-France, Orsay, F-91893
[3] Radboud University Nijmegen, The Netherlands
[4] Lab. Informática, Sistemas e Paralelismo, Univ. Beira Interior, Portugal

Abstract. This paper presents a methodology to get correct-by-construction OCaml programs using the Why3 tool. First, a formal behavioral specification is given in the form of an OCaml module signature extended with type invariants and function contracts, in the spirit of JML. Second, an implementation is written in the programming language of Why3 and then verified with respect to the specification. Finally, an OCaml program is obtained by an automated translation. Our methodology is illustrated with the proof of a union-find library. Several other proofs of data structures and algorithms are included in the companion artifact to this paper.

1 Introduction

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

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

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

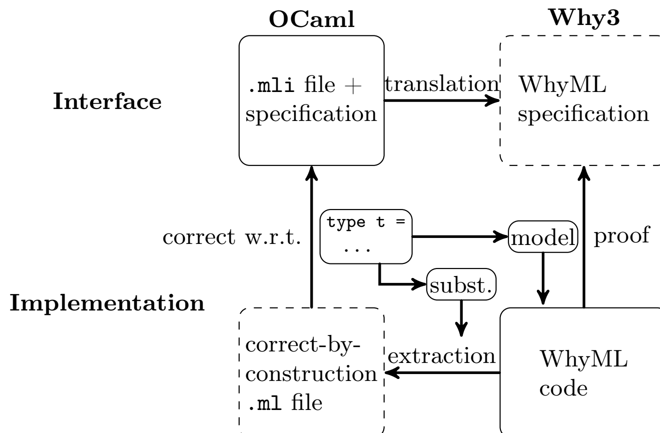

Fig. 1. Methodology diagram.

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 the aforementioned disadvantages. Our work takes place in the setting of a larger project, named VOCaL [5], 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 a specification language for OCaml, similar to what JML is for Java [2], or ACSL for C [1]. The VOCaL project combines the use of three verification tools, namely Coq, CFML, and Why3. This paper focuses on the latter.

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

An overview of our methodology is given in Fig. 1. The solid rectangles represent the user-written files, and the dashed rectangles represent the automatically generated files. Whenever an OCaml type cannot be mapped directly to a WhyML type, due to use of mutable data beyond the reach of WhyML's type system [7], a custom memory model is built for this type. When it comes to

translation of WhyML to OCaml, we return to the original OCaml type using a consistent substitution file. This is illustrated in the central part of the diagram.

In the next section, we explain this workflow in detail using the example of a union-find library. Source files for this example, as well as several other examples shortly described in Sec. 3, are contained in the companion artifact.

2 Example: Union-Find

To illustrate our methodology, let us consider the verification of a union-find library. We reuse the OCaml API from Charguéraud and Pottier's proof [6]:

```
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 any equivalence class. Our 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 formal specification. It is added to the OCaml interface above using special comments starting with '`(*@`'. Before we can give a specification to any of these functions, we need to be able to refer to the universe of all the elements we are considering, which is not materialized in this API. To this end, we first introduce a *ghost* type `uf`:

```
(*@ 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      *)
```

Being declared inside a specification comment, this type is not visible to the OCaml compiler. This is a mutable, abstract data type, whose contents is modeled through a set `dom` and a function `rep`. The two invariants ensure that the set `dom` is indeed partitioned by the relation "to have the same value by `rep`".

We are now in position to provide a specification to each OCaml function. Let us use `find` as an example.

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

The first line names the parameter `e` and the returned value `r`, so that we can refer to them in the function contract. For the purpose of the specification, `find` receives an extra parameter `uf` of type `'a uf`. Square brackets identify it as a ghost parameter. The function contract is then given using standard `requires`, `modifies`, and `ensures` clauses. Here, `modifies` accounts for the internal modification of the union-find data structure caused by path compression.

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 either a representative element (`Root`), containing the rank of type `int` and a value of type `'a`, or a pointer (`Link`) to another element in the same equivalence class.

Unfortunately, this type definition cannot be used as is in WhyML. The reason is that Why3 uses a type-and-effect discipline to track aliases statically [7], and recursive mutable types are beyond the scope of Why3's static analysis. The solution is to resort to an explicit memory model, that is a set of types for pointers and memory together with the operations to allocate, read, and write memory. In this case, we translate the OCaml types above into the following Why3 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 type to represent locations of OCaml's heap-allocated references of type `ref`. This type comes with another type to model the heap contents, that is

```
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`. We also declare primitive operations to manipulate the heap. For instance, a reference is updated using the following:

```
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] }
```

Note that we pass the heap as a ghost argument, instead of declaring a single global variable to model the heap. The reason is two-fold. First, global variables must have monomorphic types. Second, by using "small heaps" passed through the chain of function calls as hidden arguments, we statically enforce separation between the heaps and avoid complicated frame conditions. This is yet another instance of Burstall's "component-as-array" [3]. Soundness of heap manipulation

is guaranteed by the fact that `mem_ref` is a private data type that is updated through abstract functions.

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 specification written in the `.mli` file and translated to WhyML by our tool.

Extraction to OCaml. The last step consists in translating WhyML to OCaml, using Why3's extraction mechanism. This translation is governed by a substitution file, which maps WhyML types (*e.g.*, `int63`) to OCaml types (*e.g.*, `int`). Together with our memory model, we build a consistent substitution file for it, as follows:

```
module UnionFind.Mem
  syntax type loc_ref   "%1 ref"
  syntax function Link "Link %1"
  syntax function Root "Root (%1, %2)"
  syntax val set_ref   "%1 := %2"
  ...
```

The OCaml code is given as a string, where `%n` introduces a placeholder for the n-th argument of a WhyML symbol. Note that the memory model is no more an argument of `set_ref`, having been erased by the extraction because of its ghost status.

3 Experimental Evaluation

We have used our approach to verify several other OCaml modules. These examples illustrate many features not described in this paper, such as preconditions verified at runtime, OCaml functors, higher-order effectful functions, absence of arithmetic overflows, etc. They are all contained in the companion artifact. Figure 2 summarizes the size of these examples, column "spec" showing the number

module	spec	code	#VCs	
`UnionFind`	74	176	135	union-find
`PairingHeap`	41	245	52	persistent priority queues
`ZipperList`	66	180	87	zipper data structure for lists
`Arrays`	37	121	77	binary search and binary sort
`Queue`	54	185	119	mutable queues
`Vector`	149	309	142	resizable arrays
`HashSet`	21	34	12	sets using hash tables
`MergeSort`	12	401	630	in-place mergesort of lists

Fig. 2. Verified OCaml Modules.

of lines in the `.mli` files and the "code" column showing the number of lines in the WhyML implementation and proof. All verification conditions are discharged automatically using SMT solvers.

4 Related Work and Contributions

Related Work. The verified C compiler CompCert [11] and the static analyzer Verasco [8] 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 [12]. 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 substitution mechanism.

The CFML tool [4] 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 [13] and verification of the correctness and amortized complexity of a union-find [6]. Contrary to that proof, ours is fully automatic. However, we do not treat the proof of complexity bounds.

Closer to our work is the integration of module refinement in the Dafny program verifier [10] by Leino and Koenig [9]. Dafny module system does not make distinction between interface and implementation: the same notion of module is used both to give abstraction and to refine it. When refining a module in Dafny, one may give definitions to the data structures and methods left uninterpreted in the interface module, bring additional declarations, and refine previously given specifications. The main difference between module refinement in Dafny and Why3 concerns mutable data structures. In Dafny, mutable state is encapsulated within a class, and dynamic frames are typically used to control side effects. In Why3, mutable data is encapsulated within record types, and it is the type system that controls side effects.

Contributions. We propose a new workflow to produce correct-by-construction OCaml programs. It builds on the existing tool Why3, with the addition of the

following contributions: a specification language for OCaml, *à la* JML; a tool to translate it to WhyML; a systematic way to build memory models for mutable recursive OCaml types; an enhanced extraction mechanism for Why3, featuring a modular translation, up to OCaml functors; a practical validation with the proof of eight non-trivial OCaml modules.

References

1. 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`.
2. 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.
3. Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
4. 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.
5. Arthur Charguéraud, Jean-Christophe Filliâtre, Mário Pereira, and François Pottier. VOCAL – A Verified OCaml Library. ML Family Workshop, September 2017.
6. Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, September 2017.
7. 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`.
8. 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.
9. Jason Koenig and K. Rustan M. Leino. Programming language features for refinement. In John Derrick, Eerke A. Boiten, and Steve Reeves, editors, *Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015.*, volume 209 of *EPTCS*, pages 87–106, 2015.
10. 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.
11. Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
12. Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer, 2003.
13. 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.