GOSPEL — Providing OCaml with a Formal Specification Language

Arthur Charguéraud[1,2], Jean-Christophe Filliâtre[3,1], Cláudio Lourenço[3,1], and Mário Pereira[4]

[1] Inria
[2] Université de Strasbourg, CNRS, ICube
[3] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[4] NOVA LINCS & DI, FCT, Universidade Nova de Lisboa, Portugal

Abstract. This paper introduces GOSPEL, a behavioral specification language for OCaml. It is designed to enable modular verification of data structures and algorithms. GOSPEL is a contract-based, strongly typed language, with a formal semantics defined by means of translation into Separation Logic. Compared with writing specifications directly in Separation Logic, GOSPEL provides a high-level syntax that greatly improves conciseness and makes it accessible to programmers with no familiarity with Separation Logic. Although GOSPEL has been developed for specifying OCaml code, we believe that many aspects of its design could apply to other programming languages. This paper presents the design and semantics of GOSPEL, and reports on its application for the development of a formally verified library of general-purpose OCaml data structures.

1 Introduction

Functional programming languages are particularly suited for producing formally verified code. For example, the formally verified C compiler CompCert [26] is written in the applicative subset common to OCaml and Coq [35]. As another example, the verified microkernel seL4 [21] features components that are written and verified in Haskell, and then translated into C. The main reason for this adequacy is that most functional language constructs directly map to logical counterparts. In Coq, purely functional programs may be directly viewed as logical definitions. Thus, writing specifications for a purely functional program simply amounts to stating a lemma relating input and output values.

Functional programming is not, however, limited to purely applicative programming. The use of effectful features such as arrays and mutable records is necessary to implement efficient data structures and algorithms. For example, OCaml allows writing clean and concise code for functional and imperative data structures and algorithms. The OCaml language (excluding its object-oriented features) provides a straightforward semantics for its constructs that facilitates the verification process, compared with other languages that pervasively use

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

more complex features such as dynamic dispatch or inheritance. Thus, programming in an effectful functional language such as OCaml can be an interesting route to producing verified, relatively efficient code.

Unlike purely functional code that may be mapped directly to logical definitions, effectful code needs additional infrastructure to write specifications. Indeed, one needs means of describing the scope and the nature of side effects. For each function, it is necessary to specify what part of the mutable state it may access, modify, create, and destroy. Prior work has proposed Separation Logic [32] for reasoning about imperative programs, including those featuring nontrivial manipulations of the mutable state. Although it is very expressive, Separation Logic suffers from two downsides that, we believe, limit its wide adoption. First, Separation Logic specifications are fairly verbose in practice. Second, its standard presentation often appears fairly technical.

In this work, we present GOSPEL, a specification language for OCaml interfaces whose semantics is defined in terms of Separation Logic. Compared with Separation Logic, GOSPEL greatly improves conciseness and accessibility. The GOSPEL acronym stands for "Generic Ocaml SPEcification Language". In particular, the word "Generic" underlines the fact that this specification language is not tied to a specific verification tool, but rather intended to be used for different purposes, such as verification, testing, or even informal documentation.

GOSPEL fits in the tradition of other behavioral specification languages [15] such as SPARK [3], JML [23], or ACSL [1]. In contrast with these languages, GOSPEL features permissions as in Separation Logic. Unlike other tools based on Separation Logic, such as VeriFast [17] or Viper [27], GOSPEL implicitly associates permissions with data types, thereby significantly improving concision.

The contributions of this paper are as follows. First, we introduce GOSPEL through examples (Sec. 2). Second, we propose a formal semantics by means of a translation into Separation Logic (Sec. 3). Third, we report on an implementation of GOSPEL and its application to the verification of an OCaml library (Sec. 4). We finish by discussing related (Sec. 5) and future work (Sec. 6).

2 An Overview of GOSPEL

2.1 Basic Operations on a Mutable Queue

We first present a GOSPEL specification for a mutable queue data structure. This specification covers operations exposed by an OCaml interface for mutable queues, independently of any specific implementation (which could be based, *e.g.*, on doubly-linked lists, ring buffer, etc.). In OCaml, an abstract interface is described in an `.mli` file. Within such a file, the GOSPEL specifications appear in comments that begin with the `@` symbol. Such comments are ignored by the OCaml compiler, but can be parsed and processed by a verification tool.

The abstract data type `'a t` represents a parameterized queue storing elements of type `'a`. To begin with, we provide a `mutable model` annotation for this type, to associate a *model field* called `view` with every value of type `'a t`.

```
type 't t
(*@ mutable model view: 't seq *)
```

For a given queue `q`, the projection `q.view` describes the mathematical sequence of elements stored in the queue. The model field `view` has type `'a seq`, which corresponds to the type of purely applicative sequences (*i.e.*, logical sequences). This type `'a seq` is defined in the GOSPEL standard library.

The `view` field is tagged `mutable` to account for the fact that the sequence of elements stored in a queue may change over time. In general, a given OCaml type may feature several model fields, each being mutable or not. For example, a fixed-capacity mutable queue would typically feature an immutable model field describing its maximum capacity, in addition to the mutable model field describing the sequence of its elements.

Let us now declare and specify the operation that pushes an element of type `'a` to the front of a queue of type `'a t`. We first write the OCaml type, then the GOSPEL specification.

```
val push: 'a -> 'a t -> unit
(*@ push v q
    modifies q
    ensures  q.view = v :: old q.view *)
```

The GOSPEL specification first names the two arguments with `v` and `q`. Next, it indicates that `q` might be mutated during a call to `push` using the `modifies` clause. Last, it features an `ensures` clause describing the postcondition. In this case, it asserts that the updated sequence of elements in the queue (`q.view`) consists of the sequence of elements before the call (`old q.view`), extended with the new element `v` added at the front. We choose arbitrarily to model the queue with insertion at the front of the sequence and removal at the end of it.

Here, the type of `q` features a single mutable model field, thus the clause `modifies q` is equivalent to `modifies q.view`. In the case of a type featuring several model fields, the `modifies` clause may include only a subset of the fields, capturing the fact that the fields which are not mentioned remain unchanged.

We next present three more functions from the interface of mutable queues to illustrate other features of GOSPEL. The function `pop` extracts an element from the back of a nonempty queue. This function includes a `requires` clause, to express the precondition asserting that the queue must be nonempty. Note that the first line of the GOSPEL specification assigns the name `v` to the return value, so that it may be referred to in the postcondition.

```
val pop: 'a t -> 'a
(*@ v = pop q
    requires q.view <> empty
    modifies q
    ensures  old q.view = q.view ++ v :: nil *)
```

The next function, `is_empty`, tests whether a queue is empty. This function does not mutate the queue, as reflected by the absence of a `modifies` clause.

```
val is_empty: 'a t -> bool
(*@ b = is_empty q
    ensures b <-> q.view = empty *)
```

The function `create` below, returns a fresh queue data structure, with empty contents. It is specified as follows.

```
val create: unit -> 'a t
(*@ q = create ()
    ensures q.view = empty *)
```

The fact that the function returns a queue distinct from any previously-allocated queue is implicit because the type `'a t` has been declared with a mutable model field. This design choice is motivated by the fact that writing a function that returns a non-fresh, mutable data structure is considered bad practice in OCaml.

2.2 Destructive and Nondestructive Operations

We next explain how to specify functions that involve more than one mutable value, by presenting three concatenation functions for mutable queues. The first function, called `in_place_concat`, receives two (distinct) queues as arguments. It migrates the contents of the first queue to the front of the second queue, then clears the contents of the first queue.

```
val in_place_concat: 'a t -> 'a t -> unit
(*@ concat q1 q2
    modifies q1, q2
    ensures  q1.view = empty
    ensures  q2.view = old q1.view ++ old q2.view *)
```

The clause `modifies q1, q2` asserts that both queues are updated. The first `ensures` clause describes the new state of `q1` as the empty sequence. The second `ensures` clause describes the new state of `q2` as the result of the concatenation of the two original sequences. The queues `q1` and `q2` are implicitly required to be separated, that is, not aliased. This implicit assumption is another deliberate design choice of GOSPEL. Only arguments that are read-only may be aliased.

The next function, specified below, is similar to `in_place_concat`, with the difference that it destroys the queue `q1` instead of emptying it. In other words, after the call, `q1` cannot be used anymore. To describe the loss of the queue `q1`, we replace `modifies q1` with the clause `consumes q1`, as shown below.

```
val in_place_destructive_concat: 'a t -> 'a t -> unit
(*@ concat q1 q2
    consumes q1  modifies q2
    ensures  q2.view = old q1.view ++ old q2.view *)
```

Note that the `ensures` clause may only refer to `old q1.view`, but not to `q1.view`, since there is no "valid new state" for `q1`. Note also that an implementation of `in_place_destructive_concat` is allowed to performed arbitrary side effects on `q1`, which gets discarded after the call.

The third function, called `nondestructive_concat`, takes two queues as read-only arguments, and produces a fresh queue with the concatenation of the contents of the two input queues. It is specified as follows.

```
val nondestructive_concat: 'a t -> 'a t -> 'a t
(*@ q3 = concat q1 q2
    ensures q3.view = q1.view ++ q2.view *)
```

The absence of a `modifies` clause implicitly asserts that the arguments are read-only. When arguments are read-only, it is safe to alias them. For example, a call of the form `non_destructive_concat q q` is allowed.

2.3 Higher-Order Functions

In OCaml, iterations over containers are typically implemented using a higher-order function. For example, `map f q` produces a fresh queue whose elements are the pointwise applications of the function `f` to the elements from the queue `q`. Although Separation Logic does support the general case where the function `f` may perform arbitrary side effects [5,6], we cover in this paper only the simpler case where `f` is a pure function. In this case, we specify `map` as follows:

```
val map: ('a -> 'b) -> 'a t -> 'b t
(*@ r = map f q
    ensures length r.view = length q.view
    ensures forall i. 0 <= i < length q.view ->
                      r.view[x] = f q.view[i] *)
```

We leave the generalization to the general case of an effectful `f` to future work.

2.4 Ghost Variables

Ghost arguments and ghost return values may be used to specify a function. In GOSPEL syntax, ghost entities appear within square brackets in a function prototype. Consider the example below of a function that computes the largest power of two no greater than a given integer `n`. The ghost return value `k` is a convenient means of specifying that `r` is a power of two.

```
val power_2_below: int -> int
(*@ r, [k: integer] = power_2_below n
    requires n >= 1
    ensures  r = power 2 k && r <= n < 2 * r *)
```

In GOSPEL, the type of a ghost variable must be provided. Here, `k` is declared with type `integer`, which denotes the GOSPEL type of mathematical integers.

2.5 Non-Visible Side Effects

We next discuss operations that may modify the internal state of a data structure without modifying the value of the model fields. Simply pretending that the

operation does not modify the structure would be unsound, as it would suggest that the structure is read-only.

As a first example, consider a random generator module. Type `rand_state` represents the internal state of a generator (*i.e.*, the current value of the seed). We do not wish to expose the implementation of the state, yet we wish to expose in the specification the fact that there exists an internal state. To achieve this, we associate with the type `rand_state` a mutable model field of type `unit`.

```
type rand_state
(*@ mutable model internal: unit *)
```

The function `random_init` takes as argument a seed and produces a fresh random generator state of type `rand_state`. This function needs no specification.

```
val random_init: int -> rand_state
```

The function `random_int` takes as argument a state `s` and an integer `m`, and returns a pseudo-random integer smaller than `m`. This function performs a side effect on the state `s`, hence the clause `modifies s`.

```
val random_int: rand_state -> int -> int
(*@ n = random_int s m
    requires m > 0  modifies s  ensures  0 <= n < m *)
```

Similarly to the mutable queue example, here the clause `modifies s` is equivalent to `modifies s.internal`. Even though there is only one possible value of type `unit` for this model field, the fact that it is declared in the `modifies` clause is important because it specifies that internal side effects may be performed. If no `modifies` clause were provided, the argument `s` would be implicitly assumed to be read-only: no side effects would be allowed, even internally.

As a second and more challenging example, consider a union-find data structure that maintains disjoint sets using a pointer-based representation of a reverse forest. Each element of a union-find is represented as a value of type `elem`, an abstract data type. Operations on a union-find instance perform path compression, hence they modify the internal state of the structure, even when the exposed logical state remains unchanged. To account for the fact that an operation performed on a given element does not alter only this element, but potentially all the elements in the union-find instance, we do not associate any mutable model field to the type `elem`. Instead, we introduce a ghost type named `uf_instance`, meant to describe the state of all the elements in the union-find instance.

The ghost type `uf_instance` features three mutable model fields: a domain `dom` describing the set of elements in the current instance; a logical map `rep` that binds each element to its representative; and a model field called `internal` of type `unit`. The latter is used to describe the internal side effects performed by operations such as `find` which exposes no visible side effect.

We impose several well-formedness invariants on these fields. These invariants must hold for any value of type `uf_instance`, before and after any call to a function from the interface. The GOSPEL specification is as follows.

```
type elem
```

```
(*@ type uf_instance *)
(*@ mutable model dom: elem set *)
(*@ mutable model rep: elem -> elem *)
(*@ mutable model internal: unit *)
(*@ invariant forall x. mem x dom -> mem (rep x) dom *)
(*@ invariant forall x. mem x dom -> rep (rep x) = rep x *)
```

The function `equiv` below takes as arguments a ghost value `uf` of type `uf_instance` and two elements of type `elem` from the domain of the instance. It tests whether the two elements belong to the same class. The `modifies uf.internal` clause indicates that side effects may be performed on the internal state.

```
val equiv: elem -> elem -> bool
(*@ b = equiv [uf: uf_instance] e1 e2
    requires mem e1 uf.dom && mem e2 uf.dom
    modifies uf.internal
    ensures  b <-> uf.rep e1 = uf.rep e2 *)
```

The ghost function `create_instance`, specified below, enables creating a fresh and empty instance of union-find.

```
(*@ val create_instance: unit -> uf_instance *)
(*@ uf = create_instance ()
    ensures uf.dom = {} *)
```

The function `make` populates a given instance with a fresh element. It updates the union-find instance to reflect the extension of its domain with the new element.

```
val make: unit -> elem
(*@ e = make [uf: uf_instance] ()
    modifies uf
    ensures  not (mem e (old uf.dom))
    ensures  uf.dom = union (old uf.dom) (singleton e)
    ensures  uf.rep = update (old uf.rep) e e *)
```

The full union-find interface may be found in the VOCaL library—see Sec. 4.

3 Semantics

In this section, we provide a formal semantics for GOSPEL, by means of a translation into Separation Logic. First, we describe the source and the target languages of this translation. Then, we illustrate the translation using functions from Sec. 2. Finally, we present the general translation scheme.

3.1 General Form of GOSPEL Specifications

In the following, we say that a type is *represented* if its type declaration features one or more mutable model fields. By extension, we say that an argument of

a function is represented if its type is represented. Otherwise, we say that it is *non-represented*. The terminology reflects the fact that such arguments get, or do not get, represented as predicates in Separation Logic.

A GOSPEL specification consists of the prototype of a function, followed with a list of clauses. The prototype indicates the name of the function, of its arguments, and of its return values, including ghost arguments and ghost return values within square brackets. The list of clauses include one or several of each `requires`, `modifies`, `consumes`, and `ensures`.

The `requires` clause consists of a proposition that may refer to the model fields of represented arguments and to the names of the non-represented arguments, whether they are ghost or not. The `consumes` clause consists of a list of represented arguments. The `modifies` clause consists of a list of mutable model fields associated with represented arguments. These arguments must not already appear in the `consumes` clause. If the name of an argument appears in the `modifies` clause without a projection, it is interpreted as the list of mutable model fields associated with that arguments. The `ensures` clauses consists of a proposition that may refer to the same entities as the `requires` clause, minus the arguments listed in the `consumes` clause, plus the "`old` versions" of the fields listed in the `modifies` clause. If several `requires` (or `ensures`) clauses appear, they can be grouped using a conjunction. If several `consumes` (or `modifies`) clause appear, they can be grouped by appending their contents.

3.2 Basics of Separation Logic

A heap predicate, written H, is a predicate over the mutable state. If `Heap` denotes the type of states, and `Prop` denotes the type of logical propositions, then a heap predicate H has type `Heap` $\rightarrow$ `Prop`.

We write $^{\mathsf{HOARE}}\{H\}\,t\,\{\lambda x.H'\}$ to denote a Hoare triple [16] for a program term t, with precondition H and postcondition H', where x binds a name for the result produced by t. In total correctness, the interpretation of such a triple is: "if the predicate H holds in the input state, then the evaluation of t terminates and produces a value x for which the predicate H' holds in the output state".

In Hoare logic, H and H' describe the whole input and output states. In contrast, Separation Logic allows specifying only the fragment of the state that is relevant to the execution of the program. For this purpose, Separation Logic introduces the *star* operator: $H_1 \star H_2$ is a predicate that holds of a state that can be decomposed in two disjoint parts, one satisfying H_1 and another satisfying H_2.

A Separation Logic triple is written $^{\mathsf{SL}}\{H\}\,t\,\{\lambda x.H'\}$. Such a triple is equivalent to the proposition: $\forall H''.\ ^{\mathsf{HOARE}}\{H \star H''\}\,t\,\{\lambda x.H' \star H''\}$. This equivalence captures the property that a Separation Logic triple is a specification that remains valid in any extension of the input heap over which the program t operates, with the guarantee that the evaluation of t does not alter values from this extension. This property is reflected by the *frame rule*:

$$\frac{^{\mathsf{SL}}\{H\}\,t\,\{\lambda x.\ H'\}}{^{\mathsf{SL}}\{H \star H''\}\,t\,\{\lambda x.\ H' \star H''\}}\ \textsc{frame}$$

Three other Separation Logic operators are useful for the purpose of this paper. The construct $[P]$ lifts a pure proposition P of type `Prop` into a predicate of type `Heap → Prop`. The construct $\exists x.\, H$ denotes existential quantification over heap predicates. The construct $\mathsf{RO}(H)$ denotes a read-only version of the predicate H. Read-only predicates are provided by an extension of Separation Logic [8] that features the following read-only-frame rule:

$$\frac{\mathsf{SL}\{H \star \mathsf{RO}(H'')\}\, t\, \{\lambda x.\, H'\}}{\mathsf{SL}\{H \star H''\}\, t\, \{\lambda x.\, H' \star H''\}}\ \text{RO-FRAME}$$

We employ read-only predicates in our translation from GOSPEL to Separation Logic. All that the reader needs to know is that read-only predicates are duplicatable at will; that they may be discarded at any time (*i.e.*, they are not linear, but affine); and that they may appear only in preconditions, not in postconditions.

3.3 Example Translations of Mutable Queue Specifications

Before presenting the general translation scheme from GOSPEL to Separation Logic, we first provide concrete instances of the translation for some queue operations from Sec. 2. Recall that a queue has a single mutable model field of type `'a seq`. As a consequence, we introduce a *representation predicate* `R` of type `loc → 'a seq → Heap → Prop`. Here, `loc` denotes the type of pointers in Separation Logic. Concretely, given a pointer `q` and a sequence `L`, the heap predicate `R q L` captures the piece of state and invariants involved in the memory representation of a mutable queue at address `q` with contents `L`.

The translations for the specifications of the queue operations `push`, `pop`, `is_empty`, and `create` are shown below.[5] Thereafter, variable `v` has type `'a`, variable `L` type `'a seq`, variable `b` type `bool`, and variable `u` type `unit`.

```
{ (R q L) } push v q { λu. ∃L'. (R q L') ⋆ [L' = v::L] }
{ (R q L) ⋆ [L ≠nil] } pop q { λv. ∃L'. (R q L') ⋆ [L = L'++v::nil] }
{ RO (R q L) } is_empty q { λb. [b = true ↔ L = nil] }
{ [True] } create u { λq. ∃L. (R q L) ⋆ [L = nil] }
```

Observe in particular how the function `is_empty` takes as argument a read-only description of the queue. The corresponding triple implicitly asserts that the queue is returned unmodified in the postcondition.

We next present the translation for the three variants of the concatenation function from Sec. 2.

```
{ (R q1 L1) ⋆ (R q2 L2) }
in_place_concat q1 q2
{ λu. ∃L1' L2'. (R q1 L1') ⋆ (R q2 L2') ⋆ [L1' = nil ∧ L2' = L1++L2] }
```

[5] The triples are obtained by applying our translation scheme; more concise triples may be derived for `push` and `create` by eliminating existential quantifiers.

```
{ (R q1 L1) ⋆ (R q2 L2) }
in_place_destructive_concat q1 q2
{ λu. ∃L2'. (R q2 L2') ⋆ [L2' = L1++L2] }

{ RO (R q1 L1) ⋆ RO (R q2 L2) }
nondestructive_concat q1 q2
{ λq3. ∃L3'. (R q3 L3') ⋆ [L3' = L1++L2] }
```

The first function clears the contents of its first argument, whereas the second function consumes the representation predicate for its first argument. The third function differs in that it takes two read-only arguments.

3.4 General Translation Scheme from GOSPEL to Separation Logic

To keep things concise and readable, we define the general pattern of the translation by considering an example that captures the various possible cases. Without lack of generality, let us assume a type `t` with two mutable model fields called `left` and `right`. Their types are irrelevant to what follows.

```
type t
(*@ mutable model left:  type1 *)
(*@ mutable model right: type2 *)
```

To specify values of type `t` in Separation Logic, we introduce a representation predicate, called `T`, of type `loc → type1 → type2 → Heap → Prop`. Concretely, a heap predicate of the form `T p X Y` describes the memory layout of a structure of type `t` at address `p`, whose left and right model fields are described by `X` and `Y`, respectively. If invariants were attached to the data type `t` (as illustrated for example in Sec. 2.5), predicate `T` would capture those invariants.

Consider now the function `f` specified as follows, for the sake of example.

```
val f: t -> t -> t -> t -> int -> t * t * int
(*@ p5, p6, m, [h: integer] = f p1 p2 p3 p4 n [g: integer]
    requires P
    modifies p1, p2.left  consumes p3
    ensures  Q *)
```

Argument `p1` appears in the `modifies` clause, thus both its model fields may be modified; argument `p2` has only its `left` field modifiable, thus its right model field remains unchanged; argument `p3` is declared in the `consumes` clause, thus it gets lost during the call; argument `p4` is not declared in the `modifies` clause, thus it is read-only.

Additionally, a precondition `P` and a postcondition `Q` are declared. The precondition `P` is a logical proposition that may refer to the `left` and `right` projections of `p1`, `p2`, `p3`, and `p4`, as well as to `n` and `g`. The postcondition `Q` may refer to the same set of variables, minus `p3` (which is consumed), plus the `old` values of the modified model fields (namely, `old p1.left`, `old p1.right`, and `old p2.left`), plus the `left` and `right` projections of the return values `p5` and `p6`, as well as to the return values `m` and `h`.

We translate the specification for the function `f` into the following Separation Logic statement, where the variables `X`i (resp. `Y`i) refer to the values of the `left` (resp. `right`) model fields.

$$
\begin{array}{l}
\forall \texttt{p1 p2 p3 p4 n g X1 Y1 X2 Y2 X3 Y3 X4 Y4},\\
\{\ [\texttt{P}] \star (\texttt{T p1 X1 Y1}) \star (\texttt{T p2 X2 Y2}) \star (\texttt{T p3 X3 Y3}) \star \texttt{RO}\ (\texttt{T p4 X4 Y4})\ \}\\
\texttt{f p1 p2 p3 p4 n}\\
\{\ \lambda(\texttt{p5,p6,m}).\ \exists\, \texttt{h X1' Y1' X2' X5' Y5' X6' Y6'}.\ [\texttt{Q}] \star\\
\qquad\qquad (\texttt{T p1 X1' Y1'}) \star (\texttt{T p2 X2' Y2}) \star (\texttt{T p5 X5' Y5'}) \star (\texttt{T p6 X6' Y6'})\ \}
\end{array}
$$

Observe in particular how the postcondition first binds the return values, then quantifies existentially: (1) the ghost return value `h`, (2) the updated model fields associated with the represented arguments, and (3) the model fields associated with the represented return values. Observe also how the read-only predicate for `p4` appears only in the precondition (as discussed in Sec 3.2).

The above example illustrates the general scheme behind our translation. Two other minor aspects are worth mentioning. First, if an argument features an immutable model field, then we treat this field like a mutable model field not declared in the `modifies` clause. Second, for a polymorphic function, we need to quantify the appropriate type variables in the Separation Logic statement.

4 Implementation and Application

We next describe GOSPEL tools and applications: its parser, its type-checker, its mathematical library, its connection with verification tools, and its application to the specification and verification of a general-purpose library of data structures and algorithms.

Parsing and Type-Checking of GOSPEL Specifications. As explained in Sec. 2.1, GOSPEL specifications appear in comments in an OCaml interface file. The GOSPEL parser proceeds in two stages. First, the parser from the OCaml compiler is invoked to parse the structure of the file. It produces a parse tree that features, in particular, type and prototype declarations. The GOSPEL comments are stored as *attributes* to these definitions, with payloads represented as strings.[6] Second, a dedicated GOSPEL parser is used to parse the attributes that correspond to GOSPEL specifications, and to integrate them with the corresponding OCaml declarations. The use of two distinct parsers is a deliberate choice, aimed at making the framework easily maintainable in the face of evolution of either the OCaml syntax or the GOSPEL syntax.

After being parsed, GOSPEL specifications are type checked. We developed a type checker independent from that of the OCaml compiler to handle, *e.g.*, types associated with model fields. Our type checker performs ML-style type inference, allowing the user to quantify variables without providing their types, and to

[6] We patched the parser from the OCaml compiler so as to process comments of the form `(*@ ...*)` as if they were written as OCaml attributes of the form `[@@gospel "..."]`. The OCaml parser already processes documentation comments in this way.

apply polymorphic functions without explicit instantiations. The GOSPEL typechecker verifies in particular the well-formedness of the specification clauses. For example, it checks that only legitimate variables appear in the `requires`, `consumes`, `modifies`, and `ensures` clauses (as explained in Sec. 3.1).

The GOSPEL Library. The purpose of the GOSPEL library is twofold. First, the library provides mathematical theories to be used in specifications, covering unbounded integers, sequences, sets, bags, and maps. For example, a queue is specified using a sequence, a priority queue is specified using a bag, etc. Second, the library provides logical models for built-in OCaml data types, such as machine integers, lists, arrays, and strings. The GOSPEL library takes the form of regular OCaml `.mli` files, containing only GOSPEL declarations. These libraries may contain symbols that are left uninterpreted. For instance, the library currently does not give any definition for what a "set" is. For the moment, it appears more practical to leave a collection of mathematical symbols abstract and to provide, for each verification tool, a mapping from these abstract symbols towards their correponding realization (*e.g.*, in SMT theories or Coq mathematical theories).

Program Verification w.r.t. GOSPEL Specifications. In Sec. 3, we have provided GOSPEL with a formal semantics. The existence of this semantics means that GOSPEL specifications make sense independently of which verification tool is used to carry out the proofs. Thus, for a given program, we are free to use the most suitable verification tool. For example, if the code is purely functional, it makes sense to verify it directly using Coq. If the code features advanced pointer manipulations, then CFML [5,6], with its interactive proofs in Separation Logic, would be the tool of choice. If mutability is limited, then the Why3 tool [13] provides convenient support for automated proofs, by leveraging SMT provers. Thanks to the existence of the common specification language GOSPEL, it is even possible to build modular proofs where different components are verified using distinct tools.

Implementing a verification tool to handle GOSPEL specifications can be achieved in several ways. In the case of Why3, the GOSPEL specification is translated into Why3's specification language; the source code is written in WhyML, proved to satisfy the specification, then extracted into OCaml code.[7] In the case of CFML, the GOSPEL specification is translated into CFML's specification language; the OCaml source to be verified is parsed by CFML and converted into a *characteristic formula* expressed in higher-order logic; one then proves that the characteristic formula entails the desired specification.

Application to the VOCaL Library. A collection of general-purpose data structures and algorithms is an essential ingredient for the successful construction of a large-scale software. When it comes to formal verification, it thus makes sense to start with the verification of such libraries. This observation has motivated efforts in the deductive verification community to verify programming

[7] More details about the Why3 workflow may be found in Pereira's PhD thesis [30].

libraries [11,31]. OCaml is a programming language that lends itself particularly well to formal verification, in particular thanks to its simple semantics. Moreover, OCaml is used to implement several tools whose soundness is critical, *e.g.*, proof assistants [35], static analysis tools [9, 20], SMT solvers [2]. Thus, there would be strong benefits in developing a verified library for OCaml.

The recent VOCaL project precisely aims at developing a "mechanically *V*erified *OCa*ml *L*ibrary" of efficient general-purpose data structures and algorithms. The public GitHub repository of the project already includes several OCaml modules, such as resizable arrays, priority queues, and union-find.[8] These libraries have been verified using Why3 or CFML. As a contribution of the present work, we provide GOSPEL specifications for all these verified libraries.

The VOCaL library may be looked at in different ways, depending on one's needs. First, one could choose to ignore all the GOSPEL annotations and simply be interested in using VOCaL as a *trustworthy* library of OCaml code. Second, one could be interested in reading GOSPEL annotations from the VOCaL libraries in order to *unambiguously* understand what is the semantics of the operations that it provides. Third, one might be interested in producing a *formally verified* OCaml program, by leveraging the VOCaL libraries. In this case, the user would engage in verification proofs and would reason about interactions with the VOCaL libraries by exploiting their GOSPEL specifications.

5 Related Work

In recent years, a number of behavioral specification languages [15] have been proposed for various state-of-the-art programming languages, such as JML for Java [23] and ACSL for C [1]. The SPARK [3] programming and specification language is a subset of the Ada language targeting verification. Several verification tools, such as VeriFast [17], Viper [27], Why3 [13], and Dafny [24], come with their own specification languages.

Three important aspects influence the design of specification languages. The first aspect is whether specifications are meant to be executable or not. For example, JML and SPARK specifications are executable [22]. ACSL specifications are not executable, but contains an executable subset called E-ACSL [33]. Requiring executable specifications severely constrains expressivity. For this reason, we chose to not impose executable specifications in GOSPEL. A second aspect is whether specifications are meant to be entirely discharged by automated tools. For example, Dafny emits proof obligations for SMT solvers (Z3, in particular). Targeting fully automated proofs may impose a certain presentation style for specifications. GOSPEL is agnostic to the verification tool. Both SMT-based and interactive-proofs-based approaches can be used.

The third aspect of a specification language is how it treats the *frame problem*, and how it describes the *separation* of arguments and the *freshness* of return values. Specifications languages such as SPARK, JML, or ACSL require explicit

[8] https://github.com/vocal-project/vocal

freshness assertions. Dafny [24] exploits Dynamic Frames [19], an approach that is flexible but that leads to relatively verbose specifications [18]. Chalice [25] leverages Implicit Dynamic Frames [34]. This approach, partially inspired by Separation Logic, aims at providing first-order tool support. Its assertions are interpreted with non-separating conjunctions, like in Separation Logic, yet with explicit accessibility predicates. For more details on Dynamic Frames technique, we refer to Kassios' tutorial [18], and to an article by Parkinson and Summers [29] which formally explores the relationship between Implicit Dynamic Frames and Separation Logic. In contrast, GOSPEL is firmly grounded on Separation Logic: accessibility predicates, disjointness and freshness assertions are always implicit.

Why3 [13] is a deductive verification tool with a dedicated programming and specification language called WhyML. A number of aspects of GOSPEL are based on WhyML. There are, however, important differences. A first important difference is that the semantics of GOSPEL is given by means of Separation Logic, whereas WhyML is given a more traditional semantics in terms of weakest-precondition calculus and first-order logic [12]. A second difference is that GOSPEL targets a mainstream programming language used in the development of large and complex software systems. Although WhyML has many features similar to OCaml, it remains a verification-oriented language, with many OCaml features missing. In contrast, GOSPEL intends to introduce, lightly and incrementally, ideas of formal methods into the OCaml community. For instance, GOSPEL may be used in large projects to specify and verify a number of critical core components, while leaving other components unverified.

Compared with writing specifications directly in Separation Logic, the use of GOSPEL significantly improves the practical experience of writing and reading specifications. The example from Sec. 3.4 gives an idea of how much more concise a GOSPEL specification might be relative to its Separation Logic counterpart. We next summarize the key design choices that we have made w.r.t. plain Separation Logic.

In Separation Logic, one has the possibility to introduce several representation predicates for a same type. This possibility may be useful in rare cases for specifying advanced access patterns in complex data structures. In practice, the vast majority of data structures are naturally specified with a unique representation predicate. In GOSPEL, we leave the representation predicates implicit, and instead refer directly to the names of the model fields. (We could add support for multiple representation predicates in the future, while keeping the current behavior as a default.) Furthermore, in GOSPEL, unlike in Separation Logic, we do not need to provide names for the model fields that are not explicitly involved in the specification.

Another difference is that in Separation Logic, permissions (representation predicates) have to be provided even for read accesses. Yet, it would serve little purpose to provide a function with a pointer if not providing at least the corresponding read permission. Thus, we have chosen for GOSPEL a design that assumes implicit read permissions for all the arguments provided to a function. For arguments that require write access, Separation Logic specifications require

to repeat the permission both in the precondition and the postcondition. One exception is in the rare case where an argument is consumed. In GOSPEL, we only require a list of the names of the modified arguments to appear either in the `modifies` clause or in the `consumes` clause. This design avoids repetitions and significantly reduces the clutter.

VeriFast [17] is a verification tool targeting C and Java programs. It features a specification language based on Separation Logic. As recently demonstrated [4], it is possible to encode model fields in VeriFast, although with some overheads. On the contrary, GOSPEL provides builtin support for representation predicates. Thus, it can leverage dedicated features for manipulating and referring to model fields, and indicating which ones may be modified. This design enables important gains in conciseness.

Viper [27] is an intermediate verification language, which features front-ends for several programming languages, including Java, Python, and Rust. Viper's specification language is based on permissions, which are explicitly manipulated both in contracts and in the code. To indicate that a method has access to a field, the specification must include an explicit *accessibility predicate*. Moreover, to distinguish between read and write accesses, Viper relies on *fractional permissions*: only a full permission (*i.e.*, a fraction equal to 1) enables write access. In contrast, GOSPEL design makes read-access permissions implicit for all fields, and write-access permissions are simply listed in the `modifies` clause. Furthermore, GOSPEL design takes advantage of read-only permissions, whose benefits over fractional permissions are discussed in details in the paper that introduces read-only permissions [8] (§1.3 and §5.4).

6 Conclusion and Future Work

We have presented GOSPEL, a behavioral specification language for OCaml. So far, a subset of OCaml was identified for which GOSPEL specifications can be translated to Separation Logic. We expect to extend GOSPEL to a larger subset of OCaml in the future, to support other constructs such as signature constraints (`with type`) and inclusion (`include`). GOSPEL can also be extended in other directions, *e.g.*, to allow specifying the asymptotic cost of each function [7,14,28].

We have developed verification frameworks on top of GOSPEL and successfully applied them to the verification of an algorithms and data structures library. So far, these frameworks are based on Why3 and CFML. It would be interesting to also try and target the Viper ecosystem [27]. One could hope for a straightforward translation from GOSPEL to Viper, which is based on Separation Logic.

There are several other interesting directions for future work for GOSPEL. It could be extended to include invariants for, *e.g.*, loops. It could be exploited for runtime assertion checking, by identifying an executable subset. It could be integrated with a property-based testing framework, for example leveraging the `qcheck` [10] tool that generates random test values satisfying given invariants.

Acknowledgments. We are grateful to X. Leroy, F. Pottier, A. Guéneau, and A. Paskevich for discussions and comments during the preparation of this paper.

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. François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. http://alt-ergo.lri.fr/.
3. 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.
4. Raphaël Cauderlier and Mihaela Sighireanu. A verified implementation of the bounded list container. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 172–189, Cham, 2018. Springer International Publishing.
5. Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris 7, 2010. http://www.charguer aud.org/arthur/research/2010/thesis/.
6. 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.
7. 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.
8. Arthur Charguéraud and François Pottier. Temporary read-only permissions for separation logic. In Hongseok Yang, editor, *Proceedings of the European Symposium on Programming (ESOP 2017)*, Lecture Notes in Computer Science. Springer, April 2017.
9. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *ESOP*, number 3444 in Lecture Notes in Computer Science, pages 21–30, 2005.
10. Simon Cruanes, Rudi Grinberg, Jacques-Pascal Deplaix, and Jan Midtgaard. Qcheck, 2019. https://github.com/c-cube/qcheck.
11. Stijn de Gouw, Frank S de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying openjdk's sort method for generic collections. *Journal of Automated Reasoning*, 2017.
12. Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013. Springer.
13. 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.
14. Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *ESOP 2018 - 27th European Symposium on Programming*, volume 10801 of *Lecture Notes in Computer Science*. Springer, 2018.
15. John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.

16. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.
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. I. T. Kassios. Dynamic frames and automated verification, 2011. Tutorial for the 2nd COST Action IC0701 Training School, Limerick 6/11, Ireland.
19. I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288, May 2011.
20. 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.
21. Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, June 2010.
22. Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In Tiziana Margaria and Bernhard Steffen, editors, *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 9952 of *Lecture Notes in Computer Science*, pages 461–478, Corfu, Greece, October 2016. Springer.
23. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.
24. 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.
25. K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 378–393, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
26. Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
27. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.
28. Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in Iris. In Luis Caires, editor, *European Symposium on Programming (ESOP)*, volume 11423 of *Lecture Notes in Computer Science*, pages 1–27. Springer, April 2019.
29. Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3), 2012.
30. Mário José Parreira Pereira. *Tools and Techniques for the Verification of Modular Stateful Code*. PhD thesis, Université Paris-Saclay, 2018.
31. Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. *Formal Asp. Comput.*, 30(5):495–523, 2018.

32. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.
33. Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper. In *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES'17)*, September 2017.
34. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 — Object-Oriented Programming*, Lecture Notes in Computer Science, pages 148–172. Springer Berlin / Heidelberg, 2009.
35. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.9*, 2019. http://coq.inria.fr.