# RUNTIME VERIFICATION OF OCAML PROGRAMS

CLÉMENT PASCUTTO

21st July 2023 — Version 1

SUPERVISORS:
Jean-Christophe Filliâtre
Thomas Gazagnaire

LOCATION:
Laboratoire Méthodes Formelles (Gif-sur-Yvette, France)
Tarides (Paris, France)

TIME FRAME:
2020–2023

Fini, c'est fini, ça va finir, ça va peut-être finir.

(Un temps)

Les grains s'ajoutent aux grains, un à un, et un jour, soudain, c'est un tas, un petit tas, l'impossible tas.

---

*Fin de partie*
Samuel Beckett

Personne n'a craqué ! Personne n'a craqué !

Et nous sommes là, aujourd'hui, devant vous.

---

Olivier Dussopt

## ABSTRACT

Formal verification methods, in particular when it comes to deductive verification, bring strong guarantees about the correctness of software systems. However, they require a high degree of expertise and tremendous development time. These pitfalls sometimes jeopardize their application in industrial-grade software, almost always preventing scaling to complex systems. In that respect, dynamic (read: runtime) verification allows for a more gradual approach. While the user still expresses specifications in a formal, precise language, one checks the correctness of the implementation *via* automatic testing at runtime rather than proofs. It narrows the required expertise to the specification design and the interpretation of test results.

These observations also apply to the OCaml programming language community. Despite the suitability of the language for formal methods, broad adoption still seems out of reach for tools that produce specified or verified code. Moreover, such tools must account for details of the language: its type system, memory representation, garbage collector, and functional idioms.

In this work, we propose runtime verification techniques for OCaml code that apply to preexisting codebases and engineers' workflows. In particular, we briefly introduce Gospel, an accessible yet expressive specification language for OCaml. We describe Ortac, an automated runtime assertion checker for OCaml with a modular interface that allows for multiple usage *scenarii* (fuzzing, monitoring, tests). Ortac aims to support a non-trivial subset of OCaml (*e. g.* functors, exceptions, effects). It uses typing information to produce efficient verifications (*e. g.* narrowing the copies, handling arbitrary precision integers, partially verifying type invariants). Lastly, we elaborate on memory optimizations for verifying postconditions referencing the prestate. They consist of specification transformations, generalized to apply to other languages, that have been proven correct using the Coq proof assistant.

This work opens a way for an automated verification ecosystem that would be unintrusive and suitable for the developers' needs in the OCaml community.

## RÉSUMÉ

Les outils de vérification formelle, en particulier dans le domaine de la vérification déductive, apportent des garanties statiques fortes de correction des systèmes logiciels, mais nécessitent un haut degré d'expertise et des durées de développement considérables. Ces obstacles compromettent parfois leur mise en place dans un contexte industriel, et presque toujours leur passage à l'échelle dans des systèmes complexes. Dans ce contexte, la vérification dynamique (comprendre : à l'exécution) permet une approche plus graduelle. Alors que les spécifications sont toujours exprimées en termes logiques précis, on s'assure de la correction de l'implémentation par des tests automatiques à mesure de son exécution, plutôt que par des preuves. L'expertise nécessaire est alors restreinte à la conception de spécifications et l'interprétation des résultats de test.

La communauté du langage de programmation OCaml n'échappe pas à ce constat. Malgré le fait que le langage semble propice à la mise en place de méthodes formelles, aucun outil ne paraît connaître une adoption large pour la production de code OCaml spécifié ou vérifié. De surcroît, pour un outil prétendant répondre à cette question, il faut également prendre en compte les spécificités du langage, notamment les interactions avec le typage statique, l'influence de la représentation mémoire et du ramasse-miettes ou les idiomes liés à la programmation fonctionnelle.

Dans ce travail, on propose des techniques de vérification dynamique de code OCaml applicables à des bases de code préexistantes et intégrables aux flux de travail des ingénieurs logiciels qui les maintiennent. En particulier, on présente brièvement Gospel, un langage de spécification accessible mais expressif pour OCaml. On décrit Ortac, un outil de vérification dynamique pour OCaml entièrement automatisé dont l'interface modulaire permet son utilisation dans une grande variété de *scenarii* (*fuzzing*, *monitoring*, test). Il entend supporter un sous-ensemble non trivial d'OCaml (*e. g.* foncteurs, exceptions, effets) avec l'appui du typage et dans un souci d'efficacité des vérifications effectuées (*e. g.* limitation des copies, gestion des entiers de précision arbitraire, vérification partielle des invariants de types). Enfin, on développe une famille d'optimisations de la mémoire pour la vérification de post-conditions faisant référence au pré-état. Elles prennent la forme de transformations de spécifications, généralisées pour être applicables dans d'autres langages, et prouvées correctes avec l'assistant de preuves Coq.

Le travail entrepris permet d'envisager un écosystème de vérification automatisé, peu intrusif et adapté aux besoins des développeurs et développeuses de la communauté OCaml.

# PUBLICATIONS

Some ideas and figures presented in this thesis have appeared previously in the following publications:


[1] Jean-Christophe Filliâtre and Clément Pascutto. 'Optimizing Prestate Copies in Runtime Verification of Function Postconditions'. In: *Runtime Verification*. Ed. by Thao Dang 0001 and Volker Stolz. Vol. 13498. Tbilisi, Georgia: Springer International Publishing, Sept. 2022, pp. 85–104. DOI: 10.1007/978-3-031-17196-3_5. URL: https://hal.inria.fr/hal-03690675.

[2] Jean-Christophe Filliâtre and Clément Pascutto. 'Ortac: Runtime Assertion Checking for OCaml (Tool Paper)'. In: *Runtime Verification*. Ed. by Lu Feng 0001 and Dana Fisman. Vol. 12974. Los Angeles, CA, United States: Springer International Publishing, Oct. 2021, pp. 244–253. DOI: 10.1007/978-3-030-88494-9_13. URL: https://hal.inria.fr/hal-03252901/document.

[3] Nicolas Osborne and Clément Pascutto. 'Leveraging Formal Specifications to Generate Fuzzing Suites'. In: *OCaml Users and Developers Workshop, co-located with the 26th ACM SIGPLAN International Conference on Functional Programming*. Virtual, United States, Aug. 2021. URL: https://hal.inria.fr/hal-03328646.

# ACKNOWLEDGMENTS

# REMERCIEMENTS

# CONTENTS

## FIGURES



## DEFINITIONS AND THEOREMS

## LISTINGS

## ACRONYMS

RAC Runtime Assertion Checking

DV Deductive Verification

GC Garbage Collector

CLI Command Line Interface

# Part I

# INTRODUCTION

# 1

# INTRODUCTION

In the beginning God created the programs. Now the programs were formless and empty, darkness and bugs were over the surface of the deep, and the Spirit of God was hovering over the waters.

And God said, 'Let there be correctness', and there was correctness. God saw that correctness was good, and they separated the programs from the bugs.

THE END.

Well, not quite.

Bugs still lingered, multiplying with each stride. Undeterred, humans pressed on, seeking perfection through the harmonious fusion of formal methods and runtime checks. The journey continues, for in this quest, they find the ever-unfolding promise of bug-free software. The end? Nay, a new beginning.

## 1.1 SPECIFICATIONS AND PROOFS

### 1.1.1 *What Should Programs Do?*

Correctness only makes sense with respect to specifications: they set the expectations in the sentence 'what should this program do?'. Software systems are often—almost always—developed without complete, formal specifications. Instead, the development constraints are expressed in plain natural language by a manager, a client, a little voice in our head, or, at best, a written design document. Regardless of the level of details provided, they rely on our human understanding of these spoken constraints, where ambiguities and cultural influence—and therefore misunderstandings—thrive.

Hence we need dedicated *formal* specification languages based on mathematical notations and logic and designed to describe program behaviour with precise semantics. Specifying programs remains a complex task in many cases. It requires a deep understanding of the logic system the specification language implements and the specified program.

### 1.1.2 *Do Programs Do What They Should Do?*

Along with formal specifications, formal methods were developed to *reason* about programs and *characterize* their behaviour to ultimately *prove* that they comply with their specification. They aim at statically—*i. e.* by analyzing the program only, not executing it—predicting program outputs and effects. The results they provide hold for all possible executions of the program. The details of these techniques are off the topic of this thesis. However, we may cite a few significant domains available for program verification.

WEAKEST PRECONDITION CALCULUS. In this method, we compute the weakest precondition that ensures a desired postcondition—*e. g.* the one in the specification—holds after a program is executed. We check that this precondition indeed holds. It allows for compositional reasoning and modularity in verifying the correctness of individual program statements or fragments. WHY3, VERIFAST, DAFNY, or the WP plugin from FRAMA-C are notable tools that rely on this technique.

THEOREM PROVING. Theorem proving involves using mathematical logic closer to Mathematics usage—users write definitions, theorems, proofs—to prove the correctness of software or hardware systems. It requires constructing formal proofs based on axioms and rules of inference. Interactive theorem provers like COQ and ISABELLE are popular tools for this domain. The verified C compiler COMPCERT is a particularly noticeable success story of these methods.

MODEL CHECKING. Model checking is a formal verification technique that exhaustively explores all possible states of a system—or abstract states—to check whether specific properties hold true or if certain conditions are met. It is handy for finite-state systems and concurrent systems. Tools like SPIN and NUSMV are commonly used for model checking.

ABSTRACT INTERPRETATION. Abstract interpretation is a static analysis technique that approximates the behaviour of a program using abstract domains. They are suitable for automated proofs, at the cost of often weaker logic systems. The ASTRÉE and PAGAI tools are examples of abstract interpretation tools.

### 1.1.3 [*Proving*] *Software* [*Correctness*] *is Hard*

While it offers significant advantages regarding reliability and confidence, formal verification is notoriously challenging and resource-intensive. Several factors contribute to the difficulty of proving software using formal verification.

SOFTWARE SIZE AND COMPLEXITY. Deployed software systems consist of millions of lines of code and intricate interactions between numerous components—often written by different authors.

LIMITED AUTOMATION; EXTENSIVE EXPERTISE. While some aspects of formal verification can be automated, such as syntax checking and simple property verification, more complex proofs often require human intervention and manual effort. The process of constructing formal proofs can be labour-intensive and require specialized skills.

INTERACTION WITH THE PROGRAM ENVIRONMENT. Real-world software systems interact with external environments, including user inputs, network communication, and hardware devices. Verifying the correctness of software in such dynamic environments can be particularly challenging as it requires modelling and specifying these too.

UNDECIDABLE PROBLEMS. Last but not least, many problems in software verification are undecidable. The most (in?)famous one is perhaps the halting problem—does this program terminate?—but many deduction systems implemented in specification languages are also undecidable, *e.g.* first-order logic with quantifiers.

## 1.2 TOWARDS RUNTIME ASSERTION CHECKING.

Runtime Assertion Checking (RAC) is a set of dynamic verification techniques that verify that some properties—assertions—about the program hold *at runtime* during the execution of the program. They aim at detecting and reporting misbehaviours as soon as they arise in the execution and the development process and, hopefully, before they cascade into catastrophic events.

LEVEL 0: `assert`. The concept of assertions in programming can be traced back to the 1960s when they were used informally as comments or sanity checks in code. Developers manually insert checks to ensure certain Boolean conditions are met during program execution, aiding debugging and error detection.

DESIGN BY CONTRACT™. The development of formal methods in the 1970s and 1980s brought a more systematic approach to software verification. Bertrand Meyer popularized assertions as a formal means of specifying program behaviour (as opposed to individual statements at specific points in the program) in the 1980s with Eiffel and its Design by Contract™. It advocates using preconditions, postconditions, and class invariants *within* the programming language itself as programmatic documentation. The program Boolean expressions are meant to be verified at runtime during the development phase. Design by Con-

*Funnily enough (?), Design by Contract™ has been a registered trademark of Eiffel Software since 2004 [34].*

tract™ also provides methods and workflows to guide developers into using this new technique that mixes tests and formal specifications.

BROADER ADOPTION. Since then, the approach has been adopted to develop specification languages and their runtime assertion checkers for most mainstream programming languages. The support and verification of these contracts still take various shapes and forms: some are part of the language itself (*e.g.* SPARK for ADA), others are special comments meant to be processed by external tools (*e.g.* JML for JAVA or E-ACSL for C), and some are libraries that let developers *program* the contracts from within the language (*e.g.* DECORATOR CONTRACTS for JAVASCRIPT) before they are executed.

### 1.2.1 *A More Practicable Technique*

Of course, runtime assertion checks offer weaker correctness guarantees: they check *individual* program executions rather than proving the correctness of *all* program executions. They, however, overcome some of the weaknesses of static verifications and are generally easier to apply.

EASE OF USE. With the correct tooling, integrating runtime assertion checks into the code does not require additional expertise other than interpreting the results.

INCREMENTAL APPROACH. Unlike static verification, which often requires the entire program to be formally verified—because the individual parts influence the general outcome of the program—runtime assertion checking allows developers to apply verification selectively. They can specify their programs partially and focus on specific parts of the code that are most critical or prone to errors, making it a more incremental and manageable process.

SYSTEMATIC COUNTEREXAMPLES. Runtime assertion checking aids debugging by identifying errors, pinpointing their locations in the code, and providing failing examples. The failed assertions provide valuable information about the program's state at the time of the error, facilitating efficient bug diagnosis and resolution.

FLEXIBILITY. Assertions can be used for various purposes, such as checking preconditions or postconditions of functions, validating data invariants, or verifying correct behaviour in specific scenarios. This flexibility allows developers to tailor the assertions to suit their specific verification needs.

APPLICABILITY TO LARGE CODEBASES. Runtime assertion checking can be applied to large, existing codebases without significant changes to the overall development process, which makes it a practical option for improving the reliability of legacy software systems.

### 1.2.2 *Some Challenges Ahead*

Is it to say that runtime assertion checking is easy? After all, we *just* have to test the specifications, right? Well, not really.

If the ultimate goal is to provide tools that are easy *to use*, designing them comes with challenges. Some are verification and semantics challenges: we need to bridge the semantics of the programming language and the one of the specification language; others are engineering challenges: the runtime checks should integrate seamlessly into the developers' workflows; others are performance challenges: checking assertions at runtime introduce additional computations that affect the program performance.

## 1.3 THE GOSPEL PROJECT

In 2018, the goal of the Vocal project was to develop VOCAL[39], a formally verified library containing useful data structures like the ones of a standard library (*e.g.* lists, arrays, queues, hash tables) written in (and for) the OCAML programming language. As we mentioned previously, formal verification means correction with respect to a formal specification. However, there was no formal specification language for OCAML.

The key contribution of the project is the design of a specification language for OCAML: GOSPEL. From the beginning, a key design point is that the specification language should remain independent of the tools that will later be used to prove that the programs comply with them.

*We show examples of GOSPEL specifications in chapter 2.*

For instance, it aims at interfacing with CFML—which implements a separation logic and targets pointer-based data structure—, or the WHY3 platform—with its high degree of automation using off-the-shelf SMT solvers[10].

No runtime assertion checking in sight, though! In truth, although GOSPEL did not target a specific verification tool, it was mainly designed with Deductive Verification (DV) in mind, and its specifications are not necessarily executable.

## 1.4 CONTRIBUTIONS

In this thesis, we show some of the challenges that arise when it comes to executing these specifications. We propose techniques and methods for the runtime assertion checking of OCAML code based on the GOS-

PEL language. We implemented them in a tool named ORTAC (Ocaml RunTime Assertion Checking) [33]. Rather than trying to bring developers to prove their software, we believe RAC is a great means to bring formal methods to the developers with a tool that can be useful to them.

### 1.4.1 *Gospel: a Specification Language for OCaml*

In chapter 2, we briefly introduce GOSPEL through three examples. We do not dive into specifics about the language—this would be out of our topic. Instead, we give a sense of the basics to understand why runtime assertion checking is challenging.

In parallel with the work presented in this thesis was a significant development and maintenance of the GOSPEL type-checker and test cases to ensure that the language and its implementation stay aligned with its core principles and evolve along with the uncovered new needs. The initial code base was significantly simplified and was brought up to speed with the common practices of PPX—OCAML preprocessors—development. It provides a library that multiple tools build upon to process the specifications and an executable that perform simple sanity checks over the specifications. Finally, the code is tested with hundreds of test cases that ensure regressions do not occur in future developments.

GOSPEL is an open-source project available at `https://github.com/ocaml-gospel/gospel`.

### 1.4.2 *Ortac: a Runtime Assertion Checking Tool for OCaml and Gospel*

In chapter 3, we present ORTAC, a tool that consumes OCAML interfaces augmented with GOSPEL specifications and generates code that checks the function contracts and type invariants at runtime and reports the specification violations as soon as they occur. It produces wrappers around the functions and types of the instrumented module. It returns a module with the same interface to the user, only augmented with runtime verifications.

The design of ORTAC is guided by several principles: $(a)$ it aims at being fully automated—although configurable—and should not require extra expertise beyond the interpretation of the results; $(b)$ it should be clear at all times what verifications are made by the instrumented code, so the results are easy to interpret; $(c)$ it should be unintrusive to the developers' workflows and integrate properly into the compilation chain; $(d)$ it must not break any abstraction barriers—provided by the module system or the memory representation of OCAML—that are present in the instrumented code. RAC by essence has a performance cost since it introduces extra computations. However, we also set it as a secondary goal to limit this cost to a minimum.

On top of an executable tool, Ortac also provides a library—with the same name—for the translation of Gospel specifications to OCaml programs. It lets developers extend its behaviour *via* plugins. We describe a plugin that turns Ortac into a full-fledged fuzzer for OCaml interfaces that include the boilerplate code necessary to interface with `afl-fuzz` and automatically generates test cases.

Since Gospel is not a language dedicated to runtime assertion checking, some specifications contain parts that *cannot* be checked at runtime without more information, such as unbounded quantifiers or uninterpreted logical symbols. Other specifications *seem hard* to execute—*e. g.* because the specifications refer to values that do not exist or are not accessible in the program but only exist in the logic domain—but one can find workarounds to verify them regardless. Ortac identifies and translates the executable subset of these specifications. It reports non-executable parts to the user but does not block the rest of the instrumentation. It allows incremental specifications improvements over time and partial proofs by other means (*e. g.* DV).

In chapter 6, we highlight some challenging aspects of translating Gospel to OCaml and techniques we implemented to tackle them.

Gospel (rightfully) abstracts away from implementation details in its standard library. For instance, it provides polymorphic containers like sets or a polymorphic equality predicate. However, these 'details' need to be resolved when one wants to execute specifications, and implementing these is not a straightforward task in pure OCaml. We show how Ortac uses the static typing information of OCaml to generate primitives over OCaml values that allow us to implement these complex structures. We also describe cases where the type information allows us to generate more efficient code than using their generic, polymorphic counterparts.

Sometimes, the performance improvements come from something other than implementing the verifications themselves but from the structure of the wrappers. We show how Ortac avoids some computations by skipping useless verifications (*e. g.* type invariants) when they are not needed to ensure the correctness of the program.

Finally, we also show that the complexity of verifying Gospel specifications by instrumenting OCaml code sometimes involves somewhat unexpected combinatorics that has to be dealt with carefully, for instance, when dealing with exceptions. Indeed, exceptional behaviours are also part of the specifications and must be checked. However, we also must consider that the code Ortac generates may itself raise exceptions.

Ortac is an open-source project available online at `https://github.com/ocaml-gospel/ortac`.

### 1.4.3 *A Formalized Subset of Gospel to Reason About Memory*

In chapter 4, we formalize imperative programs along with a subset of our specification language to provide a basis for proofs of specification manipulations. The formalization is done within the Coq theorem prover.

The goal of this language is to be 'simple, but not too simple': it is simple on the surface, and its semantics is easy to manipulate, but it does not hide the details of the memory representation of values.

It features three primitive types—mathematical integers and their arithmetics, mutable homogeneous arrays, and immutable heterogeneous tuples— along with primitives to manipulate them in the specifications. It also features the `old` primitive in Gospel to refer to prestate values in function postconditions. These structures allow us to model complex behaviour and memory patterns *e. g.* involving aliasing, cyclic values, value mutations, or immutable values.

Imperative programs in this formalization are agnostic of the language they are written in, and the specification language only has constructs common to most specification languages, making it reusable in other settings.

### 1.4.4 *Optimizations for an Efficient Capture of Prestates in Postconditions*

In most behavioural specification languages for imperative languages, function postconditions may refer to the prestate of the function, typically using some `old`, `at`, or `pre` operator. Gospel is no exception and provides an `old` operator that does just that. For instance, a function postcondition `!x = old !x + 1` states that the function call increments the variable `x`.

In order to perform runtime verification, we need to evaluate terms and predicates—such as the term `old !x` above—after function calls. However, the prestate, which `old` refers to, does not exist anymore at this point in the program: mutable state can be modified—in function parameters or even global state—and the structure of the memory may be different under the action of the Garbage Collector (GC)—in particular, some values may not be accessible anymore. Consequently, the code instrumentation must record any value required to evaluate the predicates involving `old`.

In chapter 5, we consider the problem of efficiently capturing these prestates in Ortac. We propose specification transformations that let us copy a sound subset of the memory and ultimately let us produce correct and well-typed instrumented code. We also propose a last transformation that optimizes the runtime verification cost of logical assertions containing `old` by reducing the subset of the memory one must copy to compute these checks. We rely on the language formalization to prove that these transformations are sound and improve the

performance of the instrumented programs. We show the efficiency of this method with benchmarks that confirm significant improvements in memory usage and CPU time.

### CONCLUSION

Runtime assertion checking is paving the path towards a more formal approach to programming. By inserting dynamic checks into the code, developers can actively detect errors and violations during program execution, boosting confidence in software correctness. This pragmatic technique complements formal methods and type-checking, ensuring rapid bug identification and enhancing code reliability, all while preserving the agility needed for modern software development.

GOSPEL and ORTAC provide a framework to better structure the tests developers already write by incorporating precise semantics and documentation into their libraries. These automated tests are unintrusive and easily deployable and are the first step for a smooth transition towards verified software.

# 2

# GOSPEL: A FORMAL SPECIFICATION LANGUAGE FOR OCAML

In this chapter, we introduce the Gospel specification language and the process of specifying OCaml interfaces with it. We do not aim to provide a language specification or a user manual. Instead, we present three examples that involve interesting features of Gospel with respect to the runtime verification capabilities of Ortac and its limitations. A more detailed and up-to-date language specification is available online at https://ocaml-gospel.github.io/gospel, along with a user manual and some more examples of use cases.

## 2.1 EXAMPLE 1: FIBONACCI NUMBERS

In this first example, we aim to specify a simple function that computes the $n$th Fibonacci number. Recall that Fibonacci numbers are defined as follows:

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2}
\end{aligned}
$$

Here is a tail-recursive implementation of this function in OCaml.

```
let fib n =
 let rec aux n b a =
   if n ⩽ 0 then a
   else aux (n - 1) (a + b) b in
 aux n 1 0
```

Its signature is simple: the function takes an integer and returns an integer.

```
val fib: int -> int
(** [fib n] is the [n]th Fibonacci number. *)
```

Its functional correctness, however, is not trivial (in particular because of the inner function `aux`). Therefore, writing a specification for this function and attaching it to its interface makes sense. We write

GOSPEL specifications in special comments starting with the character @. They have an attachment mechanism similar to OCAML documentation comments (starting with the character *). Documentation comments and GOSPEL specification comments can (and should!) coexist.

In most cases, the function specification consists of two parts: (*a*) a specification *header*, which lets us name the function arguments and return value so we can mention them in the rest of the specification; (*b*) a specification *body*, which contains a number of *clauses* that specify the behaviour of the function.

### 2.1.1 *Specifying Allowed Input Values*

In the case of `fib`, a minimal specification can already contribute to the documentation, which implicitly assumes that $n$ should be positive, but does not specify the function behaviour in that case or whether this is considered a valid call.

There are many ways of specifying this condition on the input. We show three of them in this example. We show a fourth one based on exceptions in the following example in section 2.2.

FORBIDDEN VALUES. A first way of specifying `fib` is to mention that one should *never* call it with a negative argument. We can express this precondition with a `requires` clause.

```
val fib: int -> int
(*@ r = fib n
    requires 0 <= n *)
```

In that case, the behaviour of `fib` is unspecified when $n$ is negative. It can return an arbitrary integer, raise an exception, crash the program with a segmentation fault, or even not terminate at all.

A MORE DEFENSIVE VERSION. One could also consider a more defensive version of `fib` that checks that the input is valid before doing anything and raises an exception to the user otherwise:

```
let fib n =
 let rec aux n b a =
   if n = 0 then a
   else aux (n - 1) (a + b) b in
 if n < 0 then invalid_arg "argument must be positive";
 aux n 1 0
```

This time, the function is in charge of checking the precondition instead of the caller. Although the previous specification is still valid, GOSPEL also lets us specify this idiomatic pattern with a `checks` precondition instead of a `requires` clause:

```
val fib: int -> int
(*@ r = fib n
    checks 0 <= n *)
```

*It is up to each proof or test tool to determine if this is considered a valid call.*

The `checks` clauses express that a client may call `fib` with a negative input, but the function raises an `Invalid_argument` exception from the standard library in that case.

SIGNIFICANT DEFAULT VALUE. Our initial implementation does return a value even if $n$ is negative (it returns 0 in that case). A third possible specification would be to expose this default value for the client to use if necessary. In this scenario, passing a negative integer to `fib` is a valid call, and the caller is guaranteed to get 0 as a return. Therefore, we need not express a precondition but a postcondition instead. We introduce postconditions clauses with the `ensures` keyword.

*Arguably, this would be a poor design choice in this case.*

```
val fib: int -> int
(*@ r = fib n
    ensures n < 0 -> r = 0 *)
```

This postcondition uses the implication operator $\rightarrow$ to state that the result is null if the argument is negative.

### 2.1.2 *Specifying the Result*

Now that we specified the behaviour when $n < 0$, let us ensure that when $n$ is non-negative, `fib n` indeed returns the $n$th Fibonacci number. Let us assume that $n$ is non-negative in this section to avoid unnecessary repetitions in the specifications.

INTRODUCING FIBONACCI NUMBERS TO GOSPEL. Since GOSPEL has no *a priori* knowledge of Fibonacci numbers, we may define it in a GOSPEL logic `function` that follows the mathematical definition. The syntax for defining this function is close to the one of OCAML, which makes it easy to read and write for OCAML developers.

```
(*@ function rec fibonacci (n: integer) : integer =
    match n with
    | 0 -> 0
    | 1 -> 1
    | i -> fibonacci (i - 1) + fibonacci (i - 2) *)
```

*The GOSPEL logic is total, meaning one must prove that this definition is well-founded. We do not discuss the specifics of this process here.*

We may now use this definition and state that `fib` indeed returns a value that corresponds to this definition when its argument is non-negative:

```
val fib: int -> int
(*@ r = fib n
    checks 0 <= n
    ensures r = fibonacci n *)
```

MATHEMATICAL INTEGERS AND MACHINE INTEGERS. Note that the definition of `fibonacci` involves values of type `integer`. GOSPEL features both the OCAML `int` type for its 63 (or 31) bits integers and a logic type `integer` for mathematical arbitrary-precision integers. It defaults to arbitrary-precision integers for most operations. In particular, the operations + and - in the definition are defined in the GOSPEL standard library over the type `integer`. In order to keep the specifications readable, GOSPEL also provides a coercion mechanism that implicitly promotes machine integers to mathematical integers in many cases.

EXPRESSING OVERFLOWS. We have established that in our specification, the function `fibonacci` computes the $n$th Fibonacci number according to the mathematical definition. In this context, we cannot hope the function `fib` (or any other implementation, for that matter) to comply with its specification since machine `int` are bounded. However, Fibonacci numbers and mathematical `integers` are not. In fact, `fib n` only is the $n$th Fibonacci number when the $n$th Fibonacci number does not exceed `Int.max_int`, which occurs for $n = 91$ on 64 bits machines and $n = 47$ on 32 bits machines. Therefore, a correct specification of `fib`'s postcondition (on 64 bits machines) would be:

```
val fib: int -> int
(*@ r = fib n
    checks 0 <= n
    ensures n <= 91 -> r = fibonacci n *)
```

We can even use `fibonacci` as a source of truth to detect the overflow independently of the architecture:

```
val fib: int -> int
(*@ r = fib n
    checks 0 <= n
    ensures fibonacci n <= Int.max_int ->
            r = fibonacci n *)
```

### 2.1.3 *Rearranging the Clauses and Wrapping Up*

Once again, this last specification is partial: it does not specify the value of $r$ when an overflow occurs, even though that would be a valid call (there is no precondition in this specification). We can use one of the methods presented in section 2.1.1 to change it into a precondition if we wish to defer this constraint to the client:

```
val fib: int -> int
(*@ r = fib n
    checks 0 <= n
    requires fibonacci n <= Int.max_int
    ensures r = fibonacci n *)
```

In that case, we do not need to repeat the precondition in the postcondition: the preconditions are always assumed in the postconditions. A fully specified interface for `fib` is presented in listing 2.1.

```
(* This function is a specification helper for fib. *)
(*@ function rec fibonacci (n: integer) : integer =
    match n with
    | 0 -> 0
    | 1 -> 1
    | i -> fibonacci (i - 1) + fibonacci (i - 2) *)

val fib : int -> int
(** [fib n] is the [n]th Fibonacci number. *)
(*@ r = fib n
    requires fibonacci n <= Int.max_int
    checks 0 <= n
    ensures r = fibonacci n *)
```

Listing 2.1: Specified Fibonacci function.

## 2.2 EXAMPLE 2: MUTABLE QUEUES

In this second example, we specify a polymorphic mutable container: a queue (*a.k.a.* a FIFO). Specifically, let us consider the interface of the `Queue` presented in listing 2.2, borrowed from the OCaml standard library.

The main challenge when specifying this interface with behavioural function contracts is that the operations over the structure modify (or create) queues. However, the type of queues and their contents is abstract and not visible in the interface. There are two main ways of overcoming this issue using Gospel: *models* and *pure* projection functions.

### 2.2.1 *Specifying Abstract Types Using Models*

Models allow us to attach Gospel types to an OCaml type to describe that type's values in specifications further.

#### 2.2.1.1 *The Type* `'a t`

To enable reasoning about the elements of a queue, we attach a *model* to its type declaration:

```
type 'a t
(*@ model { mutable elements: 'a seq } *)
```

```
type 'a t
(** The type of queues containing elements of
    type ['a]. *)

exception Empty
(** Raised when {!Queue.pop_exn} is applied to
    an empty queue. *)

val create: unit -> 'a t
(** Return a new queue, initially empty. *)

val push: 'a -> 'a t -> unit
(** [push x q] adds the element [x] at the end
    of the queue [q]. *)

val unsafe_pop: 'a t -> 'a
(** [pop_exn q] removes and returns the first
    element in non-empty queue [q]. *)

val pop: 'a t -> 'a
(** [pop_exn q] removes and returns the first
    element in non-empty queue [q]. *)

 val pop_exn: 'a t -> 'a
(** [pop_exn q] removes and returns the first
    element in queue [q], or raises {!Empty}
    if the queue is empty. *)
```

Listing 2.2: The `Queue` module interface.

*GOSPEL annotations provide extra insight and are also relevant for documentation: the mutability of the type `'a t` cannot be deduced from its OCaml declaration alone.*

The model `elements` represents the mathematical sequence of elements stored in the queue. The type `'a seq` is the type of logic sequences defined in the GOSPEL standard library. It is defined using GOSPEL comments and is usable for specifications only (it does not exist as an OCaml type). The `mutable` keyword states that the `elements` model can change over time. Models only exist in specifications to represent abstract types or add more information to exposed types. They do not infringe on the abstraction barrier or expose implementation details.

#### 2.2.1.2 *Creating Queues*

*It is worth mentioning that the specification implicitly assumes `q` to be disjoint from every previously allocated queue.*

The first function features the creation of a queue. Its declaration and specification are as follows:

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

Like for the function `fib`, the first line of the specification is the header: it names the argument and return value of `create` in the context of this specification. The newly created queue has no elements: its `elements` model equals the `empty` sequence, as stated by the postcondition.

#### 2.2.1.3 *Pusing Into the Queue*

Let us now declare and specify a `push` operation for these queues:

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

In this case, there is no need to name the output since it is of type `unit`. The `modifies` clause states that the function `push` may mutate the contents of `q`. Finally, the `ensures` clause introduces a postcondition that describes the model `elements` of `q` after a call to `push`: the new `elements` extends the old value of `elements` with the value `v` at the front. We use the keyword `old` to refer to the value of an expression (here, `q.elements`) in the pre-state, *i. e.* before the function call.

#### 2.2.1.4 *Various Flavours of* `pop`

Let us now move to the functions that remove and return the first element of a queue and illustrate three ways of handling assumptions from the client in GOSPEL specifications.

EXCEPTIONAL VERSION. In this version, `pop_exn` raises an `Empty` exception if its argument is an empty queue. We specify this behaviour as follows:

```
val pop_exn: 'a t -> 'a
(*@ v = pop_exn q
    modifies q
    ensures old q.elements = q.elements ++ (singleton v)
    raises Empty -> q.elements = old q.elements = empty *)
```

We have two postconditions:

- The first one, introduced with `ensures`, states the post-condition that holds whenever the function `pop` returns a value `v`.
- The second one, introduced by `raises`, states the exceptional post-condition that holds whenever the call raises the exception `Empty`.

Similarly to the `push` case, the clause `modifies` indicates that this function call may mutate `q`. Note that this also applies to the exceptional case, which explains why we have stated that `q` is both empty and not modified in that case.

UNSAFE VERSION. Now, let us consider an unsafe variant of `pop` that should only be called on a non-empty queue, leaving the responsibility of that property to the client code. The function does not raise `Empty` but expects a non-empty argument. We can thus add the following precondition to the contract using the keyword `requires`:

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

DEFENSIVE VERSION. Instead of assuming the caller guarantees the precondition, we can adopt a more defensive approach where `pop` raises `Invalid_argument` whenever an empty queue is provided. As stated in the last section, GOSPEL provides a way to declare such a behavior, using `checks` instead of `requires`:

```
val unsafe_pop: 'a t -> 'a
(*@ v = unsafe_pop q
    checks q.elements q <> empty
    modifies q
    ensures old q.elements = q.elements ++ (singleton v) *)
```

The `checks` keyword means that the function itself checks the precondition `q.elements <> empty` and raises `Invalid_argument` whenever it does not hold. Note that `q.elements` is just a logical model and may not exist in the implementation. However, the function checks a property that results in `q.elements` not being empty.

*Remark* 1. The `checks` and `raises` clauses are similar, yet they present a major difference: `checks` states that if the queue is empty, then the function raises an exception , whereas `raises` states that *if* an exception is raised, *then* the queue was originally empty.

The interface is now fully specified and reproduced in listing 2.3. However, in the context of runtime verification, there is a potential for trouble in this specification. Indeed, models are purely logic structures that do not exist in the implementation. Therefore, checking conditions on these structures not only requires a translation of the predicates from GOSPEL to OCAML but also requires the ability to construct and maintain the model ourselves, which is generally not possible. While there might be solutions to overcome this issue (see chapter 7 for some insight about the support of models), there is also a way to transform the specification to eliminate models overall.

### 2.2.2 *Specifying Abstract Types Using Pure Projections*

The type for queues is abstract, so we used a model type to represent its contents and specify it. Another solution exists if we have—or can write and expose—a projection function in the interface. For instance, suppose the OCaml interface for `Queue` also provides a function `elements`:

*Exposing such a function is not unusual in OCaml, and an equivalent one even exists in some of the standard library modules (e.g. `elements` in `Set.S`).*

```
val elements : 'a t -> 'a list
(** [elements q] is the list of elements contained
    in [q]. *)
```

Unfortunately (or rather, fortunately), not all functions can be used in Gospel specifications; only *pure* ones. In our context, pure functions are functions that (*a*) do not perform any observable writing effect on mutable data; (*b*) do not raise exceptions; (*c*) always terminate. If these conditions are met by our implementation of `elements` (they hopefully are), we can mark the function as pure in its Gospel contract:

*The Gospel type-checker (or Ortac) does* not *check whether the function is pure.*

```
val elements : 'a t -> 'a list
(*@ pure *)
```

We may now use this function in the specification and replace the model references `q.elements` by functions calls `elements q`, and the `Sequence` functions by the corresponding ones in `List`, for instance:

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

*Remark* 2. When using calling `List` functions in Gospel contracts, we in fact call pure functions located in the OCaml standard library. In fact, Gospel embeds a (partially) specified version of the OCaml standard library where functions are marked as pure when they comply with the above conditions, *e.g.* the `( :: )` function.

The model `elements` is no longer helpful in the type specification. However, removing the type specification altogether is impossible since it also carries its mutability information. By default, types with no specifications are immutable; and the clauses `modifies q` would then be invalid. When a type is mutable but carries no model, one can add an `ephemeral` clause instead:

```
type 'a t
(*@ ephemeral *)
```

The resulting specification is presented in listing 2.4. By reducing the amount of models, we improve our chances of being able to execute the specification. However, as shown in the following chapters, this is not always sufficient (or even necessary).

## 2.3 EXAMPLE 3: UNION-FIND

In this third example, we specify an interface for a *union-find* data structure. Recall that a union-find data structure (sometimes also called a *disjoint-set* data structure) is a data structure that stores a partition of a set. Notable use cases are the Kruskal minimum spanning forest algorithm, congruence closure algorithm for decision procedures in SMT solvers, or register allocation in compilers.

This section shows a restricted version of union-find that stores partitions of sets of the form $\mathbb{N}_n$. A more general version is presented in the GOSPEL documentation [38]. However, its specification is not executable by ORTAC as it uses advanced features of GOSPEL (*e.g.* models, ghost values and ghost arguments). In this section, we only use pure projections in the specification. Here is the OCAML interface we want to specify:

```
type t
(** The type of union-find.
    It stores a partition of {0, ..., n-1}, where n is
    provided in [create]. *)

val size : t -> int
(** [size t] is the number of elements in [t], i.e. the
    argument provided to [create] during its creation. *)

val num_classes : t -> int
(** [num_classes t] is the number of subsets contained
    in [t]. *)

val create : int -> t
(** [create n] is a fresh union-find of size [n]
    representing {{0}, ..., {n-1}}. *)

val find : t -> int -> int
(** [find t i] is the representative of [i] in [t]. *)

val union : t -> int -> int -> unit
(** [union t i j] merges the subsets containing [i] and
    [j] in [t]. *)
```

### 2.3.1 *Specifying Effects*

Most of the time, the simplest specification (and therefore an excellent way to start writing one) consists of the set of effects provided in the interface: which types are mutable, what functions have modifying effects, what are the possible exceptions, etc.

#### 2.3.1.1 *Effects on the Type* `t`

First, look at the type of union-find `t`. Although we do not expose models, we can specify that this type is mutable (in particular, `union` modify it in place). Similarly to the type of queues in the previous section, we can state this using the `ephemeral` keyword.

```
type t
(*@ ephemeral *)
```

Since the type is mutable, we can also specify the functions that modify it: in this interface, only `union` does:

*Arguably, if `find` performs path compression in the implementation, it also modifies its argument. However, it does it in a non-observable way.*

```
val union : t -> int -> int -> unit
(*@ union t i j
    modifies t *)
```

#### 2.3.1.2 *Exceptions and Pure Functions*

Our functions do not raise exceptions, except for `Invalid_argument`, when invalid integers are passed (integers that are outside the domain of the union-find, *i. e.* not in $\mathbb{N}_n$), but we will come to that later. On the other hand, the functions `size`, `find`, and `num_classes` are pure, so we can mark them and use them in the rest of the specification.

### 2.3.2 *Preconditions and Bound Validity*

In the functions `find` and `union`, the integer arguments represent integers in the set the union-find represents. Therefore, some sort of bound checking is necessary.

```
val find : t -> int -> int
(*@ j = find t i
    requires 0 <= i < size t
    pure *)

val union : t -> int -> int -> unit
(*@ union t i j
    checks 0 <= i < size t
    checks 0 <= j < size t
    modifies t *)
```

Note that the same predicate is repeated three times. Although it is pretty simple, one can easily imagine the problems it would cause with more complex properties. We can define a logic *predicate* to avoid repetitions:

```
(*@ predicate valid (t: t) (i: int) = 0 < i <= size t *)
```

```
val find : t -> int -> int
(*@ j = find t i
    requires valid t i
    pure *)

val union : t -> int -> int -> unit
(*@ union t i j
    checks valid t i
    checks valid t j
    modifies t *)
```

Similarly, `create` only makes sense for positive sizes. We can state it using a `requires` or `checks` clause:

```
val create : int -> t
(*@ t = create n
    checks n > 0 *)
```

### 2.3.3 *Capturing the* `union` *Semantics in Postconditions*

There are a few postconditions that we can specify using our pure functions:

- (line 6) the representatives of the unioned elements are now the same;
- (line 7) when $k$ is an integer in the union-find, then
  - if it was in the class of $i$ or $j$, then it is now in the class of $i$ (and $j$),
  - otherwise, its class remains unchanged;
- (line 13) the number of classes is not more than before the union;
- (lines 14–15) if the two elements were in different classes, then the number of classes was decremented.

```
val union : t -> int -> int -> unit
(*@ union t i j
    modifies t
    ensures find t i = find t j
    ensures forall k:int.
              valid t k ->
              if (find (old t) k = find (old t) i
               || find (old t) k = find (old t) j)
              then find t k = find t i
              else find t k = find (old t) k
    ensures num_classes t <= num_classes (old t)
    ensures find (old t) i <> find (old t) j ->
            num_classes t = num_classes (old t) - 1 *)
```

We can also extend the specifications for `find`, `num_classes`, and `create`. The full specification is displayed in listing 2.5.

## RELATED WORK

Specification languages are not new. We may identify dozens of them, but they all have different goals and constraints. For instance, some have been designed for runtime assertion checking and are, therefore, executable, while others focus on deductive verification and allow more expressivity. More generally, the destination of the language is critical to the design decisions. GOSPEL is agnostic of its usage; it is meant to be usable for both DV and RAC. A second aspect is whether specifications are meant to be entirely discharged by automated tools, which may impose a particular presentation style for specifications.

EIFFEL [31] is the first programming language to embed behavioural contract-based specifications. It is an object-oriented language that provides class invariants, methods preconditions and postconditions, and loop invariants, all embedded in the programming language. It is designed for runtime assertion checking: its assertions and instructions are written in the same (obviously executable) language. Contract violations are reported back to them when the user enables their monitoring.

JML [7, 11, 25] is a behavioral specification language for JAVA which is also executable. It is suitable for both runtime assertion checking and deductive verification, *e. g. via* the OPENJML [15] project.

SPEC# [3] extends the C# programming language with support for function contracts. ASML [4, 5], then CODE CONTRACTS [2], implement similar yet less intrusive approaches for the .NET framework.

SPARK [8, 30] also integrates program specifications into its host language, ADA.

The FRAMA-C [17] framework for the C language also provides a specification language: ACSL [6]. The specifications are not necessarily executable and were initially designed for deductive verification. However, the E-ACSL plugin [18, 42] aims at identifying and translating an executable subset of ACSL for runtime assertion checking.

Another consideration on the design of specification languages is how they treat the *frame problem* and how they describe the *separation* of arguments and the *freshness* of return values. Specifications languages such as SPARK, JML, or ACSL require explicit freshness assertions. In GOSPEL, however, accessibility predicates, disjointness and freshness assertions are always implicit and cannot be specified at this point, although there is active work in that direction.

Some verification tools like VIPER [32], WHY3 [20], and DAFNY [27] also come with their own programming language on top of their specification language. GOSPEL, on the other hand, applies to a general-purpose programming language rather than one dedicated to proof

and verification. While they share some features, GOSPEL must remain unintrusive and integrate into its host language. This approach can add additional constraints; for instance, GOSPEL is not more expressive than OCAML when it comes to memory abstraction and cannot mention memory locations explicitly.

### CONCLUSION

In these examples, we showed how GOSPEL lets us incrementally specify a simple function interface by adding more details and precisions as the development continues. All the intermediary specifications make sense to GOSPEL, even if they are partial, which helps to make its learning curve more gradual. The specification style may also vary depending on the use of the module, *i. e.* depending on the client to verify constraints, adopting a defensive style, or a mix of both.

In contrast, GOSPEL intends to lightly and incrementally introduce ideas taken from formal methods into the OCaml community. For instance, GOSPEL may be used in large projects to specify and verify some critical core components while leaving other components unverified.

```
(*@ open Sequence *)

type 'a t
(*@ mutable model elements: 'a sequence *)

exception Empty

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

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

val unsafe_pop: 'a t -> 'a
(*@ v = unsafe_pop q
    requires q.elements <> empty
    modifies q
    ensures old q.elements = q.elements ++ (singleton v) *)

val pop_exn: 'a t -> 'a
(*@ v = pop_exn q
    modifies q
    ensures old q.elements = q.elements ++ (singleton v)
    raises Empty -> q.elements = old q.elements = empty *)

val pop: 'a t -> 'a
(*@ v = pop q
    checks q.elements <> empty
    modifies q
    ensures old q.elements = q.elements ++ (singleton v) *)
```

Listing 2.3: Queues specified with models.

```
type 'a t
(*@ ephemeral *)

exception Empty

val elements: 'a t -> 'a list
(*@ pure *)

val create: unit -> 'a t
(*@ q = create ()
    ensures elements q = [] *)

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

val unsafe_pop: 'a t -> 'a
(*@ v = unsafe_pop q
    requires elements q <> []
    modifies q
    ensures old elements q = (elements q) @ [v] *)

val pop_exn: 'a t -> 'a
(*@ v = pop_exn q
    modifies q
    ensures old elements q = (elements q) @ [v]
    raises Empty -> elements q = old (elements q) = [] *)

val pop: 'a t -> 'a
(*@ v = pop q
    checks elements q <> []
    modifies q
    ensures old elements q = (elements q) @ [v] *)
```

Listing 2.4: Queues specified with pure projections.

```
type t
(*@ ephemeral *)

val size : t -> int
(*@ pure *)

(*@ predicate valid (t: t) (i: int) = 0 < i <= size t *)

val find : t -> int -> int
(*@ j = find t i
    requires 0 <= i < size t
    pure
    ensures 0 <= j < size t *)

val num_classes : t -> int
(*@ c = num_classes t
    pure
    ensures c <= size t *)

val create : int -> t
(*@ t = create n
    checks n > 0
    ensures size t = n
    ensures forall i:int. 0 <= i < n -> find t i = i
    ensures num_classes t = n *)

val union : t -> int -> int -> unit
(*@ union t i j
    checks 0 <= i < size t
    checks 0 <= j < size t
    modifies t
    ensures find t i = find t j
    ensures forall k:int. 0 <= k < size t ->
                          if (find (old t) k = find (old t) i
                           || find (old t) k = find (old t) j)
                          then find t k = find t i
                          else find t k = find (old t) k
    ensures num_classes t <= num_classes (old t)
    ensures find (old t) i <> find (old t) j
            -> num_classes t = num_classes (old t) - 1 *)
```

Listing 2.5: Specified union-find interface.

Part II

# ORTAC: A RUNTIME ASSERTION CHECKER FOR OCAML

# 3

## ORTAC: HANDLING THE TOOL

In this chapter, we consider some OCaml module interfaces specified with Gospel and show what Ortac does and how to apply it. Let us consider a variant of the `Fibonacci` module presented in listing 2.1 where the interface now exposes *two* functions: (*a*) `fib` is the same as presented previously; (*b*) `fib_all` returns *all* the Fibonacci numbers up to its argument in an array. We present its interface in listing 3.1.

```
(* This function is a specification helper. *)
(*@ function rec fibonacci (n: integer) : integer =
    match n with
    | 0 -> 0
    | 1 -> 1
    | i -> fibonacci (i - 1) + fibonacci (i - 2) *)

val fib : int -> int
(** [fib n] is the [n]th Fibonacci number. *)
(*@ r = fib n
    requires fibonacci n <= Int.max_int
    checks 0 <= n
    ensures r = fibonacci n *)

val fib_all : int -> int array
(** [fib_all n] is an array containing the [n+1] first
    Fibonacci numbers. *)
(*@ a = fib_all n
    requires fibonacci n <= Int.max_int
    checks 0 <= n
    ensures Array.length a = n + 1
    ensures forall i, 0 <= i <= n -> a.(i) = fibonacci i
```

Listing 3.1: The interface of `Fibonacci` augmented with `fib_all`.

Internally, `fib` is implemented by calling `fib_all` and returning the last value, as shown in listing 3.2.

If we aim to verify these specifications *at runtime*, we need a full program that we can *execute*. Let us write a simple client for the `Fibonacci`

*Of course, `fib`'s memory complexity is suboptimal, but this is not relevant to the discussion in this chapter.*

```
let fib_all n =
  let fm2 = ref 0 in
  let fm1 = ref 1 in
  Array.init (n + 1) (function
    | 0 -> 0
    | 1 -> 1
    | i ->
      let fi = !fm1 + !fm2 in
      fm2 := !fm1;
      fm1 := fi;
      fi)

let fib n = (fib_all n).(n)
```

Listing 3.2: The implementation of `Fibonacci` with `fib_all`.

module: it reads an integer on the command line, passes it to `fib`, and prints out the result.

```
let () =
  Sys.argv.(1)
  ▷ int_of_string
  ▷ Fibonacci.fib
  ▷ Printf.printf "%d\n"
```

The structure of our program is depicted in figure 3.1.

*Remark* 3. We chose this interactive client for educational purposes, but the contents of the client do not matter for ORTAC. Although clients may have different verification needs (see sections 3.1.2 and 3.2 for more insight about this), the tool can apply to arbitrary clients *e. g.* other libraries, servers, or unikernels.

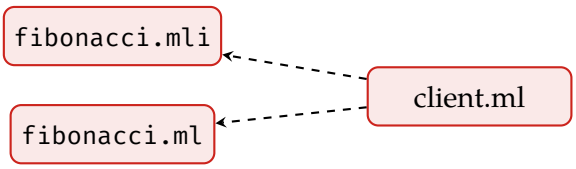



Figure 3.1: Structure of the `fib` program.

Once the build system is correctly configured, we may compile and execute it:

```
$ dune build
$ ./fib 10
55
```

Our program is now ready for instrumentation using the `ortac` executable tool, the main entry point of this work. It is intended for

specification writers who want to instrument their code as described earlier. We show how to use it and interpret its results in section 3.1 On the other hand, ORTAC also comes as a library intended for tool developers who want to extend ORTAC or write new tools based on GOSPEL. We show examples of these use-cases in section 3.2.

## 3.1 THE ORTAC INSTRUMENTATION TOOL

When using ORTAC, the instrumented program has the structure depicted in figure 3.2. The `ortac` tool reads interface files annotated with GOSPEL specifications (*e.g.* `fibonacci.mli`) and produces corresponding OCAML code that checks them in an implementation file (`fibonacci_rac.ml`). It does not modify the original implementation of the modules (`fibonacci.ml`). Instead, it creates wrappers around the functions exposed in the initial modules that verify the specification clauses. For instance, the wrappers verify the preconditions, call the original functions, and check the postconditions. The newly formed modules (`fibonacci_rac.ml`) have the same interface as the original (`fibonacci.mli`) but overrides the exposed values with the instrumented ones. Their interface file (`fibonacci_rac.mli`) is obtained by a simple copy. In order to make the verifications and have proper error reporting, the generated code depends on `ortac-runtime`, a lightweight library provided with ORTAC. It contains various helpers implementing the GOSPEL standard library or handling errors, for example.

*The generated module may contain more values than the original one, but these are only needed for the verifications and need not be exposed.*

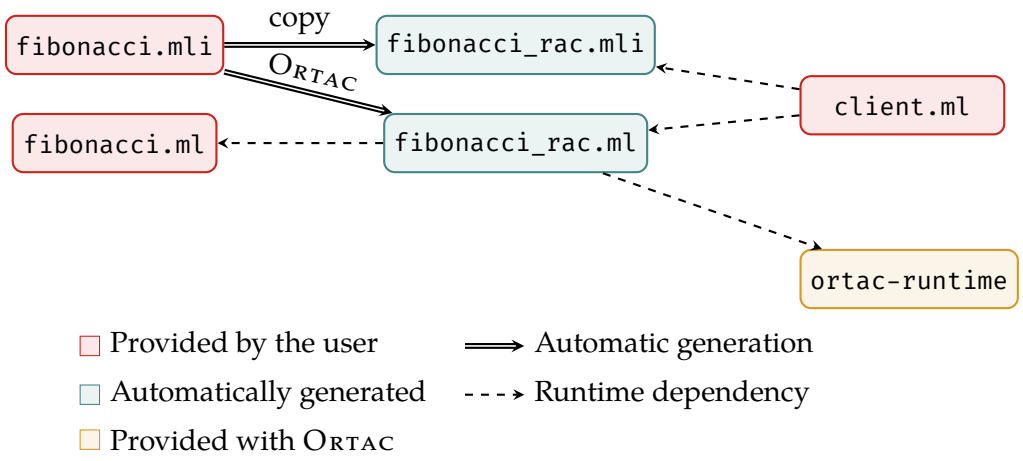



Figure 3.2: Structure of the instrumented `fib` program.

*Remark* 4. ORTAC never reads the implementation files at any point (either to get additional information or to modify implementations) and always respects the abstraction barrier of the interfaces of the modules.

Since ORTAC creates wrappers instead of modifying the existing code, it does not check the specifications of the calls internal to the module.

The figure 3.3 shows the call paths in our example. Note how the `fib` instrumented function never calls the instrumented `fib_all` function: it always calls the uninstrumented one.

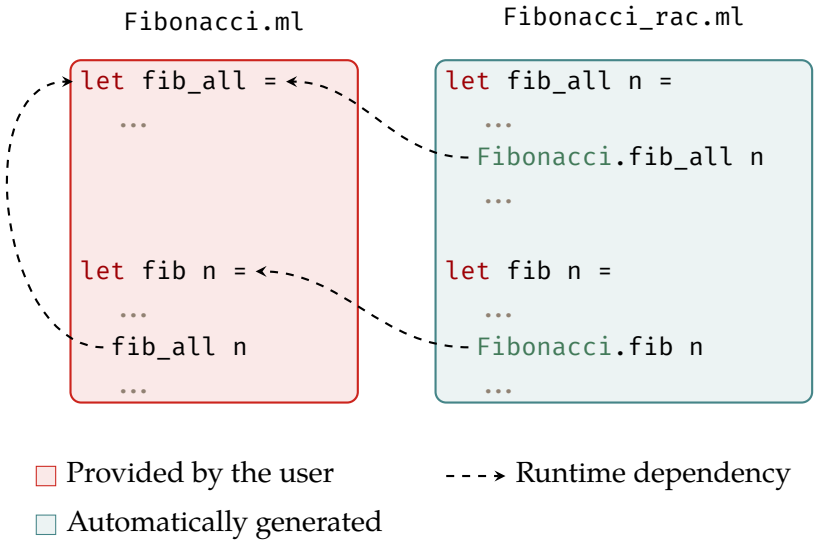



Figure 3.3: Internal calls in the `Fibonacci` module.

This behaviour aligns with how interfaces conceptually fit the code structure in OCaml: internal calls do not 'go through' the interface. For instance, (*a*) one can call internal functions without exposing them at all; (*b*) even when exposed, their type might be different internally than the one declared in the interface.

A NOTE ON VERIFYING INTERNAL CALLS. Verifying the internals call would require reading and modifying (or creating a modified copy) the implementation files to replace the calls with calls to instrumented versions. Implementing this would be challenging.

First, it is not a simple syntactic operation since one can use the same symbol for different values: typing is necessary for name resolution. Currently, Ortac does not need typed implementation files since it does not read them. On the technical front, implementing this is challenging. On the one hand, using the outcome of the OCaml compiler is not straightforward as it requires parsing the compilation artefacts and plugging Ortac in the middle of the compilation chain (rather than a preprocessor right now). On the other hand, re-implementing OCaml typing independently (or extracting it from the compiler sources) is daunting and hard to maintain.

Regardless of the technical aspects of implementing this, checking the internal calls only makes sense if the internal type is compatible with the specification type. In that case, Ortac would have to ignore *some* of the internal calls, which we believe hurt the clarity of its results.

> **Example 1.** Consider the following interface:
>
> ```
> val id : int -> int
> (*@ y = id x
>     ensures x + 1 = y + 1 *)
> ```
>
> implemented with `let id x = x`. One can call this function `id` internally on any type (in particular, not integers), but its specification assumes integer values. For such internal calls, there is no meaningful specification to check.

### 3.1.1 *Usage*

The tool expects a positional argument: the filename of an OCaml interface file (a `.mli` file) annotated with Gospel specifications. It outputs the instrumented code (a `.ml` file) on the standard output or in a file provided with the `-o` argument.

```
$ ortac fibonacci.mli > fibonacci_rac.ml
$ ortac fibonacci.mli -o fibonacci_rac.ml
```

STATIC CHECKS FAILURES. At this stage, Ortac (*a*) *assumes* that the interface file and its corresponding implementation file (if any) are accepted by the OCaml compiler; (*b*) *checks* that the specifications are well-formed and reports the same errors as `gospel check` otherwise; (*c*) *checks* that the specifications are executable and emits warnings otherwise. For instance, when fed with the interface of queues specified with models (see listing 2.3), Ortac emits a series of warnings since these models are not supported.

```
$ ortac queue_models.mli -o queues_models_rac.ml
File "queue_models.mli", line 4, characters 19-39:
Warning: the model elements attached to the type t is not
supported.
File "queue_models.mli", line 10, characters 13-30:
Warning: the predicate q.elements = empty references the
model elements, which is not supported. It will not be
checked.
File "queue_models.mli", line 15, characters 13-48:
Warning: the predicate q.elements = cons v (old
q.elements) references the model elements, which is not
supported. It will not be checked.
...
```

Even if the specification is not fully executable by Ortac, it still instruments the code to check as much as possible. For instance, although it cannot check any clause involving `elements`, it will still check for unexpected exceptions raised by the functions.

ORTAC does not create the corresponding interface file itself. Since the instrumented code has the same signature, users may do it trivially by copying the original one.

```
$ cp fibonacci.mli fibonacci_rac.mli
```

*In some cases where types are exposed, users may need to add type aliases manually signal that the types in the original and the instrumented modules are the same.*

Now that we have an instrumented version of the `Fibonacci` module, we may reuse our client and replace it with the `Fibonacci_rac` module to check the specifications at runtime:

```
let () =
  Sys.argv.(1)
  |> int_of_string
  |> Fibonacci_rac.fib
  |> Printf.printf "%d\n"
```

The project now has the structure presented in figure 3.2. We can build it with no modification to the build system parameters.

```
$ dune build
```

When the specifications are satisfied, the instrumented code provides the same result as the original one: The verifications, however, are not cost-free: one may notice a performance impact when using the instrumented module.

```
$ ./fib 10
55
```

However, when a specification is unsatisfied, the instrumented code stops the program and reports the violations to the user. Here are a few examples of error messages provided by the instrumented code.

PRECONDITION AND POSTCONDITION VIOLATION. Let us start with a faulty call to `fib`, *i. e.* a call that violates the function's precondition. If we pick a large enough argument $n$, then the $n$th Fibonacci number exceeds `Int.max_int`. ORTAC implements GOSPEL integers with arbitrary precision integers using the `zarith` library, which lets it detect integer overflows:

```
$ ./fib 4242
File "fibonacci.mli", lines 8-13, characters 0-30:
 Runtime error when executing fib 4242:
 - the precondition
     fibonacci 4242 ⩽ Int.max_int
   did not hold.
```

The error message provides the following information: (*a*) the location of the specification that was violated; (*b*) the faulty call, with its arguments when ORTAC inferred a way to print them (see chapter 6 for more details); (*c*) the nature of the failures—here, a predicate did not hold—; (*d*) the faulty clauses—there is only one here, but all the failures are reported when multiple clauses are violated.

*When ORTAC cannot print the arguments, it uses the names present in the specification.*

*Remark* 5. Unfortunately, there is no way of providing the location of the *caller* instead of the callee in the error message without modifying the client code. It is, however, still possible to retrieve this information by enabling OCaml's backtrace printing.

UNEXPECTED AND INVALID EXCEPTIONS. Recall that functions cannot raise exceptions unless specified otherwise with a `raises` clause. Ortac catches unexpected exceptions and reports them to the user. For instance, if our function `fib` was to raise a `Failure` exception, it would be reported as follows:

```
$ ./fib 0
File "fibonacci.mli", lines 8-13, characters 0-30:
 Runtime error when executing fib 0:
 - the call raised an unlisted exception:
     Failure.
```

When an exceptional postcondition is associated with the exception, the instrumented code also checks it. If the exception is correct with respect to the specification, it is re-raised as if the code was not instrumented.

FAILURE TO CHECK A SPECIFICATION. Users may encounter reports related to the behaviour of Ortac-instrumented code when it fails to execute the specification, *i.e.* when its execution itself raises an exception. When that happens, the failure is reported to the user, but it does not stop the execution of the program. Indeed, the instrumented code could not verify the clause, so we cannot deduce anything from it at this point: it may or may not hold. In other words, it is treated as a warning.

```
$ ./fib 424242
File "fibonacci.mli", lines 8-13, characters 0-30:
 Warnings when executing fib 424242:
 - the evaluation of the precondition
     fibonacci 424242 ⩽ Int.max_int
   raised an exception:
     Stack_overflow.
   It could not be checked.
 - the evaluation of the postcondition
     3202736562209518488 = fibonacci 424242
   raised an exception:
     Stack_overflow.
   It could not be checked.
3202736562209518488
```

The exceptions raised by the verifications themselves are discussed in more detail in section 6.3.1.

*Remark* 6. The definition provided for `fibonacci` has terrible performances. It is fine: specifications should be clear, simple and as close as possible to the properties they translate. In ORTAC, we try not to encourage users to care about the efficiency of the definitions. Instead, we automatically generate memoized versions of user-provided recursive functions. Therefore, the execution of `fib 100` terminates *instantly*.

*The automatic memoization can be disabled with the* `--no-memo` *option.*

DEFENSIVE STRATEGY NOT ENFORCED. When the specification has some defensive preconditions (`checks` clauses), then the instrumentation checks that the function indeed raises `Invalid_argument` if such a precondition does not hold:

```
$ ./fib -10
File "fibonacci.mli", lines 8-13, characters 0-30:
 Runtime error when executing fib -10:
 - the evaluation of the precondition
     fibonacci 424242 ⩽ Int.max_int
   raised an exception:
     Stack_overflow.
   It could not be checked.
 - the checks precondition
     0 ⩽ -10
   did not hold.
   The function should have raised Invalid_argument.
```

The first message is a warning concerning the `requires` precondition, as we discussed in the last paragraph: when executed with negative input, it indeed overflows the stack. The second message is about the `checks` precondition violation.

The instrumentation also checks for false positives and reports if the function incorrectly raises `Invalid_argument`. For instance, if `fib` was to raise `Invalid_argument` when provided the input 0, the user would get the following message:

```
$ ./fib 0
File "fibonacci.mli", lines 8-13, characters 0-30:
 Runtime error when executing fib 0:
 - the call raised Invalid_argument,
   but none of the checks preconditions
     - 0 ⩽ 0
   were violated.
```

You can read more about the instrumentation of defensive preconditions in section 6.3.2.2.

### 3.1.2 *Checking Mode*

By default, the instrumentation verifies as much of the specification as ORTAC is able to translate. This is usually a good default when de-

veloping software with GOSPEL and ORTAC. However, there are cases where some verifications are better disabled depending on the performance requirements or the use of the instrumented modules. Therefore, ORTAC users may choose amongst multiple verification modes corresponding to different instrumentation levels. The mode is given using the `--mode` Command Line Interface (CLI) argument, or `mode` option in `.ortac`.

```
$ ortac --mode=<mode>
```

NO CHECKS. The mode `nop` is trivial: it does not perform any verification at all. The 'instrumented' module re-exposes the original module without modifications. Although the relevance of this mode seems questionable, there are two reasons why this mode is available: (*a*) it lets the user control the executability of their specification, as ORTAC still issues warnings; (*b*) from a development workflow perspective, the developer can keep referring to the `*_rac` modules in the client code, regardless of whether they want the verifications enabled or not, rather than having to change these references in the code.

EXCEPTIONS ONLY. The mode `exceptions` only monitors the exceptions raised by the instrumented functions. Optionally (the mode is then `exceptions-cond`), it can check that the associated conditions hold when they exist. It is a relatively lightweight instrumentation, and results are generally easy to interpret. These verifications help diagnose implementations using exceptions since the OCAML typechecker brings few static guarantees regarding exceptions. Not only do exceptions not appear in the function's types, but they can also escape the scope where they are defined, even if when are not declared in the module interfaces.

*See* [28] *for an attempt at including exceptions in OCaml types.*

*Remark* 7. When the verification of the associated conditions is enabled, it would be naive to consider that 'it only costs something when the function raises an exception'. Indeed, we show in chapter 5 and section 6.3.2.2 how the `old` primitives or the `checks` clauses will trigger (potentially costly) computations regardless.

PRECONDITIONS ONLY. In the mode `requires`, ORTAC only monitors the functions preconditions in `requires` clauses and ignores the rest. It is particularly interesting for library developers, as the correction obligation created by these conditions relies on the caller, *i. e.* the library client. Once their libraries are appropriately tested (or proved correct using another tool!), developers can release their libraries with those preconditions monitored and ensure that the users correctly use them.

POSTCONDITIONS AND INVARIANTS. On the other hand, one can also enable the monitoring of the obligations created for the callee, *i. e.* the instrumented module, with the mode `ensures`. It turns all the tests on:

preconditions, postconditions, exceptions, type invariants at functions entry *and* exit, etc.

TOGGLING THE VERIFICATIONS. The user decides the level of checks statically when they invoke the `ortac` executable. The mode defines the kind of instrumentation that ORTAC generates and the verifications embedded in the final executable. However, regardless of the level of verifications, users can toggle these verifications dynamically (*a*) before running each function, `ortac-runtime` reads the `ORTAC_DISABLE` environment variable and turns off the tests when it is set to a truthy value; (*b*) at any point during the execution, users may toggle the verifications by sending the `SIGUSR1` POSIX signal to the instrumented process to enable the tests, or `SIGUSR2` to disable them. When using this feature, the verifications are not simply silenced; they are fully disabled, meaning they do not cost any resources besides checking the environment variable.

Dynamically toggling the tests or monitoring for specific runs or specific parts of the execution is interesting, particularly in regard with the performance-guarantees tradeoff. As mentioned earlier, although ORTAC has some optimizations implemented, the cost of verifications can still be high, and having all the verifications running at all times may not be feasible.

For instance, it is often interesting to run functional tests (with verifications enabled) and performance tests (with verifications disabled) during the same feedback loop. In that case, setting the environment variable is enough to change the context instead of recompiling the executable. Another possible use case is when only some operations are critical and should be monitored, but they are not segregated in the codebase and cannot be instrumented separately. One can then toggle the tests before and after this specific operation from the client using an environment variable or an external process using the signals.

## 3.2 USING THE ORTAC LIBRARY: OTHER FRONTENDS

Thus far, we presented the default instrumentation provided by the `ortac` executable, which stops the program by raising an exception and displays a (hopefully helpful) error message as soon as a specification violation is detected. Although it is sometimes possible to catch this `Ortac_runtime.Error` exception and silence the error message to handle it differently, we find this default behaviour is too specialized and prevents other exciting use cases. Therefore, ORTAC also provides a library that lets developers customize the instrumentation to implement different verification policies. These alternative instrumentations (we call them *frontends*) can then be compiled into plugins for the `ortac` executable. The user provides them to `ortac` through a CLI argument or an option of the same name in `.ortac`.

```
$ ortac --frontend=<frontend>
```

### 3.2.1 *Automated Testing, a.k.a. Fuzzing*

Using RAC for testing during the development phase of the software is its most obvious use case. ORTAC automates the assertion checking. However, the developer is still responsible for organizing the testing suites and writing relevant test cases that expose potential misbehaviours or trigger edge cases. Organizing the tests often consists in writing a decent amount of 'boilerplate code' using a test framework, and finding edge cases is a tedious task.

*It is easier to find edge cases when the implementation is known or when a bug is already detected, but testing is much more about* discovering *bugs.*

The goal of the `monolith` frontend is to fully automate these extra steps in the context of fuzz testing (or fuzzing), so developers only have to write down the properties that the implementation should satisfy, and the rest is automated. The frontend presented in this section is extracted from joint work with Nicolas Osborne, whom I supervised during his internship.

FUZZING AND AFL-FUZZ. Fuzz testing is a group of techniques aiming at automating the test case elaboration phase. It consists in feeding the program under test with randomly generated data and observing crashes. In this frontend, we use the `afl++` [21] fuzzer to generate this data. It is a grey-box fuzzer: it requires instrumented compiled code (the `+afl` compiler variant provides such instrumentation) to maximize the exploration of the universe of possible executions. It is mutation-based: it initially generates random data, then mutates it to explore different execution paths at each iteration and try to find buggy ones.

`afl++` *is the successor—and a fork—of the* `afl-fuzz` *fuzzer*[44].

MONOLITH. The fuzzer feeds the program under test with the bytes it generates and observes program crashes. Some 'glue' is needed to interface it to the OCAML libraries under tests: they require more—or differently—structured data and are not designed to stop the program (tests are!). The MONOLITH[36, 37] library is a model-based testing framework that does just that. Other testing frameworks interface OCAML tests with the AFL generator, such as CROWBAR [19]. We did not identify any obstacle to writing a similar frontend for CROWBAR in place of MONOLITH. MONOLITH requires as input two modules with the same signature, a 'reference' and a 'candidate', along with a dynamic representation of that signature. It processes the `afl` bytes to generate an execution scenario, *i. e.* a list of function calls and along with their arguments that are possibly chained. It then executes this scenario for both the reference and the candidate and crashes when it sees discrepancies so that the fuzzer can see and report the failure.

The `monolith` frontend for ORTAC leverages the typing information to generate the dynamic representations of the signature. It then uses

the original implementation as the candidate and the wrapper generated by `ortac` as the reference, containing the expected behaviour information. The wrapper has to be slightly modified though: since `monolith` will feed random data to the tested interface, some of this data will likely not comply with the preconditions. Precondition violations, therefore, should *not* be reported as errors. Instead, they are reported to `monolith` (and in turn to `afl`) as 'bad input', letting them know that the corresponding execution path is not relevant and should be ignored in the subsequent tests.

WHAT IS TESTED. In summary, this frontend tests that the instrumented and original codes behave the same, which is the case when the program is correct. It reports discrepancies between the two; three kinds of differences may appear:

- The original module terminates normally (*i. e.* not with an exception), but the instrumented module raises an exception. In that case, the instrumented module caught a specification violation and reported it: if ORTAC is correct (we hope it is!), we likely found a bug in our code.
- The original module raises an exception, but the instrumented module exits with a value. This case should not happen and shows that the code generated by ORTAC is incorrect.
- The original module and the instrumented one both return values, but they are different. Again, the ORTAC instrumentation is faulty in that case.

These last two cases should never occur in released versions of ORTAC. It was, however, useful to test ORTAC itself during its development stages.

The frontend generates a full program ready to be executed as a standalone executable—in that case, it will use fully random data—or *via* the `fuzzer` for better results.

```
# random mode
$ ./main

# fuzzing mode with afl
$ afl++ -i inputs/ -o outputs/ -- ./main
```

In both cases, Monolith provides inputs to the annotated functions and reports errors in the directory `outputs/crashes` as replayable *scenarii*. The user can replay the scenario by passing the corresponding file name to the generated program as an argument to get more information about misbehaviour. This way, the user has access to—and can replay—the failure scenario and all the errors reported by `ortac` on these specific inputs, highlighting the broken specifications.

```
$ ./main outputs/crashes/<filename>
File "fibonacci.mli", lines 8-13, characters 0-30:
 Runtime error when executing fib 4242:
 - the precondition
     fibonacci 4242 ⩽ Int.max_int
   did not hold.
```

LIMITATIONS. The experiments conducted so far have been underwhelming. Fuzzing is famously hard to predict: it relies on the fuzzer's ability to analyze the program structure correctly and generate the relevant test cases. Mutation-based fuzzers generally perform [29] mildly at this task. We identify three reasons why fuzzing is made even harder in the case of this ORTAC frontend:

- The instrumentation we generate can be oversized and is branching by nature, which grows the space of execution paths very quickly.
- The cost of this instrumentation is sometimes important, which reduces the number of execution cycles `afl` can perform in a given timeframe and thus reduces its ability to test data mutations.
- When the specification contains preconditions, it can be tricky (or impossible) for the pseudo-random generators to produce compliant data, and most of the executions triggered by `afl` end up being marked as non-significant. One solution would be to let the user provide their own (specification-compliant) generators manually rather than using entirely random ones, but this is not currently available in ORTAC.

More recently, Nicolas Osborne and Samuel Hym have been developing another automated testing frontend for ORTAC named QCHECK-STM. It relies on QCHECK [16], a property-based testing framework inspired by HASKELL'S QUICKCHECK [13].

### 3.2.2 *Monitoring*

Another interesting derived use-case of the instrumentations is the use of RAC to monitor long-running applications (*e. g.* servers) and provide reports on their execution.

On top of reporting specification violations, it reports the ordinary events of the execution, particularly function calls, their arguments and return values, and successful verifications. When a specification appears violated, it logs the events (with a different logging level) rather than stopping the program.

*One could write a very similar frontend to aggregate statistics about the execution and expose them in a PROMETHEUS server, for instance.*

These reports are meant to be aggregated in a file for further inspection after the end of the execution (or the failing verifications). Users

can easily customize the log format by programming it in the client code.

The results provided by this frontend should be treated with caution. When multiple violations are reported, they cannot be considered independently: once the program fails once, the subsequent execution as a whole is faulty to the specification.

### RELATED WORK

**WRAPPERS AND INTERNAL CALLS VERIFICATIONS.** In section 3.1, We discussed why ORTAC does not verify internal function calls. However, many existing tools [14, 30, 41] do provide this feature. The main difference lies in the specification and host languages and their design choices. The OCAML language has a strong concept of abstraction associated with the interfaces, which is also applied by GOSPEL (and therefore ORTAC) by specifying the *interfaces* rather than the *implementations*. Instead, ACSL, JML or SPARK specifications are much closer to the implementation, both by their locations—they are located in the implementation—and their constructs—some even feature loop invariants, local assertions, or code locations. For these tools, instrumenting the original code is a much more reasonable option that also gives them the ability to check finer properties if they need to (*e.g.* type invariants inside function bodies or internal calls)

**PARTIAL VERIFICATIONS.** The idea of providing different levels of instrumentation (see section 3.1.2) for different use-cases or stages of the development already existed in the Design by Contract™ approach presented through and implemented by EIFFEL [31]. The RAC tool provided by OPENJML provides options to turn off some checks but based on different criteria [14]. For instance, one relies on OPENJML being able to distinguish internal calls from external ones and turns off the verifications for internal calls. E-ACSL provides a script that generates both an instrumented and un-instrumented version but does not allow more granularity [41]. In any case, none of these tools can dynamically turn the verifications on or off at runtime.

**MONITORING MODE.** Both E-ACSL and OPENJML have a `keep-going` option that lets the program run even if a violation occurs, but they do not provide a complete trace log with function calls and successful verifications.

# 4

# SETTING UP THE WORKBENCH: MICROSPEL, A TOY LANGUAGE

In the following chapters, we will dive into the transformations that Ortac operates to correctly identify the executable parts of a Gospel specification and produce OCaml code that checks them at runtime.

Ortac aims at covering as much of the OCaml language as possible and relies on external libraries, *e.g.* `ppxlib` to manipulate the OCaml AST. Therefore, completely formalizing and proving it would be a daunting task with questionable relevance. We also hope that this work can apply to other runtime assertion checkers for other languages, and thus prefer to simplify our approach by modeling Ortac's behaviour using a more abstract language.

*Gospel, Ortac, and OCaml itself are rapidly moving targets that are expected to keep evolving in the upcoming years.*

In this chapter, we introduce Microspel, a simple specification language attached to black-box programs to model the behaviour of imperative interfaces annotated with logic contracts. We will later use this language to formalize and prove the instrumentation that Ortac produces. We believe it is generic enough to enable detailed reasoning about the semantics and memory models of OCaml and Gospel. We use it in the following chapters to describe some of the instrumentation techniques implemented in Ortac in a way that is generalisable to other programming languages where similar issues arise.

## 4.1 PROGRAMS

Let us set up the context for Microspel specifications. We want to consider programs that:

- can read data and perform side-effects on their environment;
- can have non-deterministic behaviour;
- do not expose their implementation, and only allow observing the inputs and outputs of its executions;

### 4.1.1 *Program Values and Program States*

Programs operate *via* effects (*i.e.* reading or writing) on *program states* $S$ during their execution. This section defines program values and states.

#### 4.1.1.1 *Program Values*

The *program values* $v$ handled by our language can be either integers $n$ or addresses $a$.

$$\begin{array}{rcll} v & ::= & n & \textit{integer} \\ & | & a & \textit{address} \end{array}$$

The addresses point to locations in a memory that depends on the program state.

#### 4.1.1.2 *Program States*

The program states are the execution environment of the program. They consist of variable bindings that associate variables to values (function $V$), and memory bindings that associate memory addresses to sequences of values that represent arrays or tuples (function $M$).

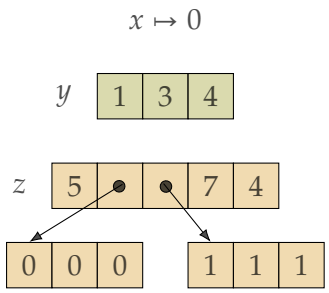
$$\begin{array}{rcl} V & ::= & x \mapsto v \\ M & ::= & a \mapsto [v, \dots, v] \\ S & ::= & V \times M \end{array}$$

*One could consider that these functions are total, but return garbage outside their domain. This distinction is not relevant to us, as we will ensure memory access is always safe.*

These functions are partial (not all variables and addresses are bound in a program state); we note $dom(V)$ (*resp.* $dom(M)$) their domain, *i. e.* the set of variables (*resp.* addresses) where they are defined.

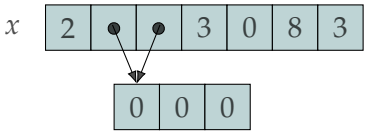
*Notation* 1. For the sake of conciseness, we always assume the notation $S = (V, M)$ in the rest of this thesis, which means $V$ (*resp.* $V'$, *resp.* $V_1$) is the variable function associated to the state $S$ (*resp.* $S'$, *resp.* $S_1$).

> **Example 2** (Simple program state)**.** The state $S_0$ contains three variables: $(a)$ the variable $x$ is bound to an integer which value is 0; $(b)$ the variable $y$ is bound to an integer array of size 3; $(c)$ the variable $z$ is bound to a tuple of size 5 containing both integers and integer arrays.
>
> $$V_0 : \begin{array}{l} x \mapsto 0 \\ y \mapsto a_y \\ z \mapsto a_z \end{array} \qquad M_0 : \begin{array}{l} a_y \mapsto [1, 3, 4] \\ a_z \mapsto [5, a_0, a_1, 7, 4] \\ a_0 \mapsto [0, 0, 0] \\ a_1 \mapsto [1, 1, 1] \end{array}$$
>
> In the graphical representation, we ignore addresses, and we replace them with arrow pointers, as the addresses themselves are irrelevant to the understanding of the state structure.




Note that states may also contain aliased values, *i.e.* distinct pointers values that point to the same address in memory, as shown in the following example.

**Example 3** (States with aliases)**.** The state $S_1$ binds a single variable $x$ to a tuple containing aliases.




*Notation* 2. For the sake of clarity, we will only show states using their graphical representation in the following from now on.

STATE INCLUSION. Let us define a partial order over program states for inclusion. It is useful to later describe the semantics of MICROSPEL specifications. We say that $S_0$ is included in $S_1$ whenever all the bindings in $S_0$ are also in $S_1$.

**Definition 1** (State inclusion)**.** Let $S_0$ and $S_1$ be two states. $S_0$ is included in $S_1$, and we note $S_0 \sqsubseteq S_1$, when:

(*a*) $V_0$ is included in $V_1$ (we note $V_0 \sqsubseteq_V V_1$ ): for all variables $x$ in $dom(V_0)$, we have $x \in dom(V_1)$ and $V_0(x) = V_1(x)$;

(*b*) $M_0$ is included in $M_1$ (we note $M_0 \sqsubseteq_M) M_1$): for all addresses $a$ in $dom(M_0)$, we have $a \in dom(M_1)$ and $M_0(a) = M_1(a)$.

WELL-FORMED STATES. Not all states—and this is especially relevant for initial states—are valid states for evaluating programs. A program state is well formed when its variables and memory do not contain dangling addresses. In other words, all addresses point to well-defined memory locations.

**Definition 2** (Well-formed program states)**.** A program state $(V, M)$ is well formed when:

(*a*) for all variable $x$ such that $V(x) = a$, we have $a \in dom(M)$;

(*b*) for all address $a$ such that $M(a) = [v_0, v_1, \ldots, v_{n-1}]$, and for all $v_i$ that is an address, we have $v_i \in dom(M)$.

We note $wf(S)$ when $S$ is well formed. The notation $wf(\_)$ will also be used for other good formation definitions, but the context should always make the disambiguation immediate.

*There are always ways to trick the type system, e.g. using the `Obj` module or some `unsafe_*` functions for the standard library, or even C bindings.*

This definition is consistent with the guarantees provided by the OCaml type system, which ensures that all pointers in memory are initialized before being accessed.

### 4.1.2 *Programs*

A *program* $p$ is a relation between two states, noted $\overset{p}{\rightsquigarrow}$. An execution of $p$ is noted $S \overset{p}{\rightsquigarrow} S'$. In this context, we refer to $S$ as the *pre-state* of the execution and $S'$ as the *post-state* of the execution.

Programs implementations details are not relevant in our model, and this definition lets us abstract away from it. We ensure that our model language does not let us inspect the program implementation, since Ortac also has this design constraint.

> **Example 4** (Counter increment)**.** The program $p_{incr}$ increments the value contained in the variable $x$.
>
> $$S \overset{p_{incr}}{\rightsquigarrow} S' \implies \exists n.\, V(x) = n \wedge V'(x) = n + 1$$
>
> The following shows a sample execution of $p_{incr}$:
>
> $$x \mapsto 3 \qquad \overset{p_{incr}}{\rightsquigarrow} \qquad x \mapsto 4$$

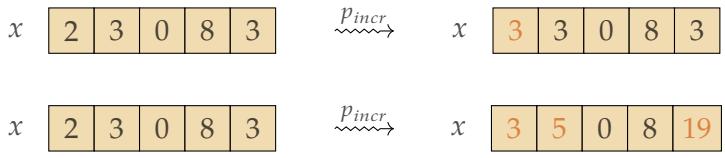
> **Example 5** (Array sorting)**.** The program $p_{sort}$ sorts the array bound to the variable $x$ in-place according to an *is_sorted* predicate.
>
> $$S \overset{p_{sort}}{\rightsquigarrow} S' \implies \exists a.\quad V(x) = V'(x) = a \wedge \\ is\_sorted(M'(a)) \wedge \\ is\_permutation(M'(a), M(a))$$

Note that defining programs as relations between states—rather than functions from states to states—does not prevent a program from associating multiple post-states to the same pre-state. Therefore, this semantics accounts for possibly non-deterministic programs.

**Example 6** (Greater array)**.** The program $p_{greater}$ replaces the cells in the integer array $x$ with greater values.

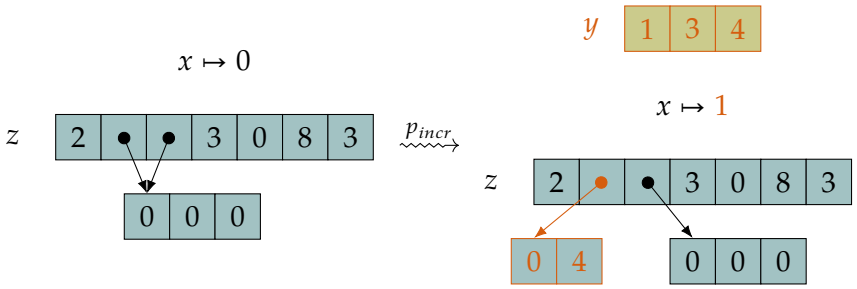
$$S \overset{p_{greater}}{\rightsquigarrow} S' \implies \exists a, n_0, n'_0, \dots, n_{n-1}, n'_{n-1}.\ V(x) = V'(x) = a \wedge M(a) = [n_0, \dots, n_{n-1}] \wedge M'(a) = [n'_0, \dots, n'_{n-1}] \wedge \forall i \in \mathbb{N}_n.\ n_i \leq n'_i$$

The two following executions share the same initial state and have different post-states, but are both valid.




*Remark* 8. The examples 4 to 6 define classes of programs, rather than single programs, as their definitions do not specify the program effects outside of $x$.

For instance, the following execution is also valid for a program $p_{incr}$. Besides modifying $x$, it creates variable a variable $y$ and modifies $z$'s contents:




Note that $p_{incr}$ may or may not modify the contents of $z$, and it can even remove or add new bindings in the program state, as it does in this example.

WELL-FORMED PROGRAMS. Programs are well-formed, when they maintain the good formation of states.

**Definition 3** (Well-formed programs)**.** A program $p$ is well formed (we write $wf(p)$) when, for all states $S$ and $S'$ such that $wf(S)$ and $S \overset{p}{\rightsquigarrow} S'$, we have $wf(S')$.

## 4.2 PROGRAM SPECIFICATIONS

A *program specification sp* is made of four parts: (*a*) variable declarations that define the program domain; (*b*) a subset of these variables that the program may modify during its execution; (*c*) additional identifiers introduced by copying the result of *terms* evaluation before the program starts; (*d*) a *predicate* that the program must satisfy after its execution, *i. e.* a post-condition.

*The predicate P may use identifiers from both a and c.*

$$
\begin{aligned}
sp \;::=\;& \texttt{domain}\ x\texttt{:}\tau\texttt{,}\ x\texttt{:}\tau\texttt{,}\ \ldots\texttt{,}\ x\texttt{:}\tau\texttt{;}\\
& \texttt{modifies}\ x\texttt{,}\ x\texttt{,}\ \ldots\texttt{,}\ x\texttt{;}\\
& \texttt{let}\ x\texttt{,}\ x\texttt{,}\ \ldots\texttt{,}\ x\ \texttt{=}\ \texttt{copy}\ \texttt{(}t\texttt{,}\ t\texttt{,}\ \ldots\texttt{,}\ t\texttt{)}\ \texttt{in}\\
& \texttt{ensures}\ P
\end{aligned}
$$

MICROSPEL does not differentiate program arguments, return values, or global variables; all of these are part of the program's domain and are treated the same.

In the following, we describe in more details the components of these specifications.

### 4.2.1 *Types*

The types appearing in the domain specification (a) reflect those of the data manipulated by the program, *i. e.* they can be an integer, a homogeneous array—*i. e.* containing a single type of values—or a heterogeneous tuple.

$$
\begin{aligned}
\tau \;::=\;& \texttt{int} && \textit{integer}\\
\mid\;& \tau\ \texttt{array} && \textit{array}\\
\mid\;& \tau\ \texttt{*}\ \ldots\ \texttt{*}\ \tau && \textit{tuple}
\end{aligned}
$$

### 4.2.2 *Terms*

The purpose of *terms* is to read program states in the logic space. They can appear in the post-condition (d) to express computations about the program states, or in the additional bindings (c) to save computations before the program execution.

Terms can be integer literals, variables, basic arithmetic, and array and tuple accessors. They also feature the specific primitive `old` that refers to the pre-state value of a term. Terms only feature read-only

*Similarly to OCAML, this language does not allow any direct manipulation of memory addresses in its terms.*

constructs, and modifying program states from the specifications is not allowed.

| | | | |
|---|---|---|---|
| $t$ | ::= | $n$ | *integer literal* |
| | \| | $x$ | *variable* |
| | \| | $t$ `+` $t$ \| $t$ `-` $t$ | *basic arithmetic* |
| | \| | $t$`[`$t$`]` | *array getter* |
| | \| | `length` $t$ | *array length* |
| | \| | $t$`.`$n$ | *tuple getter* |
| | \| | `old` $t$ | *prestate reference* |

### 4.2.3 *Predicates*

The purpose of predicates is to express properties about the program states. They are the predicate of the first-order logic with bounded quantifications over integers. They provide Boolean constants, logic negation, logic conjunction and disjunction, bounded existential and universal quantifiers over integers, as well as an equality predicate over terms.

| | | | |
|---|---|---|---|
| $P$ | ::= | `true` \| `false` | *constants* |
| | \| | `not` $p$ | *negation* |
| | \| | $t$ `==` $t$ | *equality* |
| | \| | $P$ `/\` $P$ | *conjunction* |
| | \| | $P$ `\/` $P$ | *disjunction* |
| | \| | `forall` $x$`,` $t$ `<=` $x$ `<` $t$ `->` $P$ | *universal* |
| | \| | `exists` $x$`,` $t$ `<=` $x$ `<` $t$ `/\` $P$ | *existential* |

*We use the equality symbol `==` in specifications to differentiate from the mathematical equality $=$ we use in formulas. The meaning of `==` is described in section 4.3.3.*

### 4.2.4 *Well-formed Specifications*

For specifications to be well formed, their components must comply with basic typing rules.

WELL-TYPED TERMS. Terms—whether they appear in copies or in predicates—should be well typed. We introduce a typing judgment $\Gamma \vdash t : \tau$ meaning that $t$ has type $\tau$ in the typing environment $\Gamma$, which associates variables to types. The inference rules for this judgment are standard and should follow intuition; they are available in figure 4.1.

These rules make the typing relation a deterministic relation over terms: there is at most one type associated to a term in a given typing environment.

**Lemma 1** (Terms typing is deterministic). *Let $\Gamma$ be a typing environment, $t$ a term, and $\tau$ and $\tau'$ two types.*
*If $\Gamma \vdash t : \tau$ and $\Gamma \vdash t : \tau'$, then $\tau = \tau'$.*

$$\frac{}{\Gamma \vdash n : \texttt{int}} \text{ (Ty-Int)} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \text{ (Ty-Var)}$$

$$\frac{\Gamma \vdash t : \tau_1 \times \tau_2 \times \ldots \times \tau_n \qquad 0 \leq i < n}{\Gamma \vdash t.i : \tau_i} \text{ (Ty-Pi)}$$

$$\frac{\Gamma \vdash t_1 : \tau \texttt{ array} \qquad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1[t_2] : \tau} \text{ (Ty-Get)}$$

$$\frac{\Gamma \vdash t : \tau \texttt{ array}}{\Gamma \vdash \texttt{length } t : \texttt{int}} \text{ (Ty-Length)} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \texttt{old } t : \tau} \text{ (Ty-Old)}$$

Figure 4.1: Typing rules for terms.

*Proof.* By induction over $t$. □

WELL-TYPED PREDICATES. Predicates also have to be well typed, and similar rules apply to them, which essentially express that all their sub-terms are also well typed. We note $\Gamma \vdash P$ to denote that $P$ is well typed in the environment $\Gamma$. The rules for deciding $\Gamma \vdash P$ are shown in figure 4.2.

WELL-FORMED SPECIFICATIONS. We say that a specification is well formed when $(a)$ the variables in the `domain` declaration are all distinct; $(b)$ the `modifies` identifiers are a subset of the domain identifiers; $(c)$ there are as many identifiers as terms in the `copy` declaration; $(d)$ all the `copy` terms are well typed in the environment formed by the `domain`; $(e)$ the `ensures` predicate is well typed in the environment formed by the `domain` and the variables introduced by the `copy` declaration.

**Definition 4** (Well-formed specifications)**.** A specification

```
domain x_0: τ_0, ..., x_{d-1}: τ_{d-1};
modifies y_0, ..., y_{m-1};
let z_0, ..., z_{c-1} = copy (t_0, ..., t_{c'-1}) in
ensures P
```

is well formed when:

$(a)$ for all distinct integers $i$ and $j$, we have $x_i \neq x_j$;

$(b)$ for all integer $i$, there exists an integer $j$ such that $y_i = x_j$;

$(c)$ we have $c = c'$;

$(d)$ for all integer $i$, there exists a type $\tau_{t_i}$ such that

$$(x_0 : \tau_0), \ldots, (x_{d-1} : \tau_{d-1}) \vdash t_i : \tau_{t_i}$$

$$\frac{}{\Gamma \vdash \texttt{true}} \text{ (Ty-True)} \qquad \frac{}{\Gamma \vdash \texttt{false}} \text{ (Ty-False)}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash \texttt{not } P} \text{ (Ty-Not)} \qquad \frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \ \wedge \ P_2} \text{ (Ty-And)}$$

$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \ \vee \ P_2} \text{ (Ty-Or)}$$

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 \ = \ t_2} \text{ (Ty-Equal)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{int} \qquad \Gamma \vdash t_2 : \texttt{int} \qquad \Gamma, i \mapsto \texttt{int} \vdash P}{\Gamma \vdash \texttt{forall } i, \ t_1 \ \leqslant \ i \ < \ t_2 \ \texttt{->} \ P} \text{ (Ty-Forall)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{int} \qquad \Gamma \vdash t_2 : \texttt{int} \qquad \Gamma, i \mapsto \texttt{int} \vdash P}{\Gamma \vdash \texttt{exists } i, \ t_1 \ \leqslant \ i \ < \ t_2 \ \wedge \ P} \text{ (Ty-Exists)}$$

Figure 4.2: Typing rules for predicates.

$(e)$ we have

$$(x_0 : \tau_0), \dots, (x_{d-1} : \tau_{d-1}), (z_0 : \tau_{t_0}), \dots, (z_{c-1} : \tau_{t_{c-1}}) \vdash P$$

When a specification $sp$ is well formed, we note $wf(sp)$.

## 4.3 PROGRAM CORRECTNESS

In this section, we consider well-formed states, programs, and specifications, and seek to define program correctness with respect to a specification, *i. e.* what it means for a program to 'meet its specification'. We first need to introduce the values handled by the specification and define the semantics for terms and predicates, then we can wrap it up together to provide a meaning for the specifications.

### 4.3.1 *Logic Values*

Our semantics for specifications separates the program space—which the program itself manipulates—and the logic space—which the specification considers. Terms and predicates manipulate *logic values lv*, which are agnostic of the memory. They represent self-contained val-

ues rather than program values made of unresolved addresses. Logic values can be integers, or arrays or tuples of logic values.

*At this stage, we cannot differentiate arrays from tuples; typing provides this information.*

$$\begin{array}{rcll} lv & ::= & n & \textit{integer} \\ & | & [lv, \dots, lv] & \textit{array or tuple} \end{array}$$

FROM PROGRAM VALUES TO LOGIC VALUES. The specification (hence logic values) can manipulate the result of program computations via variables (hence program values). Therefore, we need to establish the rules for resolving program values into logic values. We note $M, v \rightsquigarrow lv$ when the program value $v$ *resolves* to the logic value $lv$ in the memory $M$. Unsurprisingly, this resolution is defined by recursively browsing the value and querying the memory whenever an address is encountered:

$$\frac{}{M, n \rightsquigarrow n} \quad \text{(R-INT)}$$

$$\frac{M(a) = [v_0, \dots, v_{n-1]} \qquad \forall i \in \mathbb{N}_n . M, v_i \rightsquigarrow lv_i}{M, a \rightsquigarrow [lv_0, \dots, lv_{n-1}]} \quad \text{(R-ADDR)}$$

This relation is deterministic, meaning that a program value resolves to at most one logic value in a given memory.

**Lemma 2** ($\rightsquigarrow$ is deterministic)**.** *Let $M$ be a memory, $v$ a program value, and $lv$ and $lv'$ two logic values.*
*If $M, v \rightsquigarrow lv$ and $M, v \rightsquigarrow lv'$, then $lv = lv'$.*

```
Lemma lv_of_v_deterministic :
  forall M v lv lv',
    lv_of_v M v lv -> lv_of_v M v lv' -> lv = lv'.
```

*Proof.* By induction over the proof of $M, v \rightsquigarrow lv$. □

It is also worth mentioning that in well-formed states, program variables always resolve to a logic value.

**Lemma 3** (Value Resolution in Well-formed States)**.** *Let $S$ be a state, and $x$ a variable.*
*If $wf(S)$ and $V(x) = v$, then there exists a logic value $lv$ such that $M, v \rightsquigarrow lv$.*

*Proof.* By induction over $v$. □

### 4.3.2 *Terms Semantics*

Terms only exist in the specification world, so evaluating them leads to logic values. Because terms contain the `old` primitive, evaluating them not only requires a program state, but *two* program states: one for the pre-state when a sub-term is under `old`, and one for the post-state in

other cases. We note $[\![t]\!]_S^{S'}$ the logical value produced by evaluating of the term $t$ in the pre-state $S$ and post-state $S'$.

While terms are generally evaluated in the post-state, the `old` operator lets you refer to the pre-state. The semantics is expressed by evaluating the term in the pre-state $S$ only, rather than in the couple $S, S'$.

*This rule conveniently makes the `old` primitive idempotent.*

$$[\![\texttt{old } t]\!]_S^{S'} = [\![t]\!]_S^S$$

Evaluating variables—which typically exist in the program space—requires to query the variable bindings to get the corresponding address, then apply the value resolution to compute the logic value.

$$\frac{V'(x) = v \qquad M', v \rightsquigarrow lv}{[\![x]\!]_S^{S'} = lv} \quad \text{(T-VAR)}$$

The other rules for the judgment $[\![t]\!]_S^{S'} = lv$ are simple and follow the intuition.

$$\frac{}{[\![n]\!]_S^{S'} = n} \quad \text{(T-INT)}$$

$$\frac{[\![t_1]\!]_S^{S'} = n_1 \qquad [\![t_2]\!]_S^{S'} = n_2 \qquad n = n_1 + n_2}{[\![t_1 \ + \ t_2]\!]_S^{S'} = n} \quad \text{(T-PLUS)}$$

$$\frac{[\![t_1]\!]_S^{S'} = n_1 \qquad [\![t_2]\!]_S^{S'} = n_2 \qquad n = n_1 - n_2}{[\![t_1 \ - \ t_2]\!]_S^{S'} = n} \quad \text{(T-MINUS)}$$

$$\frac{[\![t_1]\!]_S^{S'} = [lv_0, \ldots, lv_{n-1}] \qquad [\![t_2]\!]_S^{S'} = n_0 \qquad 0 \leq n_0 < n}{[\![t_1[t_2]]\!]_S^{S'} = lv_{n_0}} \quad \text{(T-GET)}$$

$$\frac{[\![t]\!]_S^{S'} = [lv_0, \ldots, lv_{n-1}]}{[\![\texttt{length } t]\!]_S^{S'} = n} \quad \text{(T-LENGTH)}$$

$$\frac{[\![t]\!]_S^{S'} = [lv_0, \ldots, lv_{n-1}] \qquad 0 \leq i < n}{[\![t.i]\!]_S^{S'} = lv_i} \quad \text{(T-PI)}$$

The evaluation of terms is also deterministic: if a term evaluates to two logic values in the same couple of states, then these values are in fact equal.

*This justifies the notation $[\![t]\!]_S^{S'} = lv$.*

**Lemma 4** ($[\![t]\!]_S^{S'}$ is deterministic)**.** *Let $S$ and $S$ be two states, $t$ a term, and $lv$ and $lv'$ two logic values.*
*If $[\![t]\!]_S^{S'} = lv$ and $[\![t]\!]_S^{S'} = lv'$, then $lv = lv'$*

*Proof.* By induction over the proof of $[\![t]\!]_S^{S'} = lv$. □

### 4.3.3 *Predicate Semantics*

The semantics of predicates naturally derives from the semantics of the terms. We note $S, S' \vDash P$ when $P$ holds in pre-state $S$ and post-state $S'$. Unsurprisingly, most of the definition follow the rules of first-order logic.

$$
\begin{array}{lrcl}
S, S' \vDash & \texttt{true} & ::= & \top \\
S, S' \vDash & \texttt{false} & ::= & \bot \\
S, S' \vDash & P_1 \ \wedge \ P_2 & ::= & S, S' \vDash P_1 \quad \wedge \quad S, S' \vDash P_2 \\
S, S' \vDash & P_1 \ \vee \ P_2 & ::= & S, S' \vDash P_1 \quad \vee \quad S, S' \vDash P_2 \\
S, S' \vDash & \texttt{not}\ P & ::= & \neg (S, S' \vDash P)
\end{array}
$$

Let us pause on the definition of the equality predicate. In fact, the logic domain of predicates and terms is not aware of addresses at all; we reason directly on the contents of the memory instead of their location. Comparing arrays $a$ and $b$ with the predicate $a = b$ should compare the contents of the arrays (recursively if necessary), rather than their addresses in memory. Hence, we compare logical values rather than program values.

$$S, S' \vDash \quad t_1 = t_2 \quad ::= \quad [\![t_1]\!]_S^{S'} = [\![t_2]\!]_S^{S'}$$

Finally, the quantifiers require to lexically substitute the quantified variable in the predicate before evaluating it. The standard capture-avoiding substitution of $x$ by $t$ in $P$ is noted $P[x \leftarrow t]$. The semantics of the `forall` and `exists` constructs are is defined as follows.

$$
\begin{aligned}
S, S' \vDash \texttt{forall}\ x,\ t_1 \leqslant x < t_2 \ \texttt{->}\ P ::= \\
\exists n_1, n_2.\ & [\![t_1]\!]_S^{S'} = n_1 \wedge \\
& [\![t_2]\!]_S^{S'} = n_2 \wedge \\
& \forall j.\, n_1 \leq j < n_2 \implies S, S' \vDash P[x \leftarrow j]
\end{aligned}
$$

$$
\begin{aligned}
S, S' \vDash \texttt{exists}\ x,\ t_1 \leqslant x < t_2 \ \wedge\ P ::= \\
\exists n_1, n_2.\ & [\![t_1]\!]_S^{S'} = n_1 \wedge \\
& [\![t_2]\!]_S^{S'} = n_2 \wedge \\
& \exists j.\, n_1 \leq j < n_2 \wedge S, S' \vDash P[x \leftarrow j]
\end{aligned}
$$

Although it is not structural over $P$, this recursive definition is well-founded, because the substitution of a variable by an integer does not affect the size of the predicate.

*This property make it possible for* Ortac *to execute and verify those predicates at runtime. This will be discussed in the following chapters.*

Thanks to the bounds over the quantified variables, the statement $S, S' \vDash P$ is always decidable.

**Lemma 5** ($S, S' \vDash P$ is decidable)**.** *Let $S$ and $S'$ be two states, and $P$ a predicate.*

*Then $S, S' \vDash P$ is decidable.*

*Proof.* By induction over P. □

### 4.3.4 *Program Correctness*

Now that we defined the semantics of terms and can decide whether a predicate holds, we can finally give a meaning to 'the program $p$ is correct with respect to the specification $sp$'.

#### 4.3.4.1 *Effects Correctness*

The second condition for a program to meet its specification concerns its `modifies` variables: if the program can modify the contents of a variable at any depth (*e. g.* if the corresponding logic value changes), then that variable must appear in the `modifies` list. Variables outside the `modifies` variables are left unchanged by any execution of the program.

**Definition 5** (Effects correctness)**.** Let $p$ be a program and $sp$ a specification. The program $p$'s effects are correct with respect to $sp$ when:

- for all states $S$, $S'$ such that $S \overset{p}{\rightsquigarrow} S'$,
- for all variables $x \notin \{y_0, \dots, y_m\}$,

we have

$$S(x) = lv \iff S'(x) = lv$$

Note that this definition is about *logic values*, so they apply to variable contents in their full depth. Hence, when two variables share some memory space that the program may modify, they should *both* appear in the `modifies` list.

#### 4.3.4.2 *Post-condition Correctness*

Finally, the third condition concerns the `ensures` post-condition, with its associated `copy`. The program is correct with respect to this part of the specification when the predicate holds for every execution, when the pre-state and post-state have been extended with the copied terms.

**Definition 6** (Post-condition correctness)**.** Let $p$ be a program and $sp$ a specification, with the following post-condition:

`let` $y_0$`, ...,` $y_{c-1}$ `= copy (`$t_0$`, ...,` $t_{c-1}$`) in ensures` $P$

We say that $p$ is correct with respect to $sp$'s post-condition when,

- for all states $S$ and $S'$ such that $S \overset{p}{\rightsquigarrow} S'$,
- for all state $Sc$ such that $S \sqsubseteq Sc$ and $\forall y_i.\ [\![y_i]\!]^{Sc}_{Sc} = [\![t_i]\!]^{S}_{S}$,
- for all state $Sc'$ such that $S' \sqsubseteq Sc'$ and $\forall y_i.\ [\![y_i]\!]^{Sc'}_{Sc'} = [\![t_i]\!]^{S}_{S}$,

we have $Sc, Sc' \models P$.

The state $Sc$ (*resp.* $Sc'$) is a superset of $S$ (*resp.* $S'$), augmented with the variables $y_i$ bound to values that resolve to the same logic values as the terms $t_i$ in the pre-state.

Note that these extended states always exist, as long as there are enough free addresses available in $S'$ for new bindings. We discuss the perspective of dealing with a 'full memory' (and `Out_of_memory` exceptions) in more details in chapter 6.

*Remark* 9 (About `copy` redundancy). The construction introducing the new identifiers $y_i$ bound to the pre-state computations of the terms $t_i$ is in fact redundant as the same result can be achieved by at least two other means.

First, it is equivalent to using `old`$(t_i)$ in the predicates directly (see the rule T-OLD). However, both differ by their nature, as `old` is generally a logic primitive, not accessible in most programming languages, whereas `copy` is a feature of imperative programming language, not available to specifications as they are generally pure.

Likewise, this state extension could be part of the program execution in the first place, and the variables $y_i$ added to the program domain, rather than considered only when verifying the predicate.

MICROSPEL aims at transitioning from the specifications to the (verifying) programs, which explains some of its redundancy. In the next chapters, we will consider specifications without `copy`, and will transform them to specifications without `old`, then will translate the result into a program.

### CONCLUSION

This chapter introduced a formalization of imperative programs and their MICROSPEL specifications. We described what it means for a program to be correct with respect to a specification.

The following chapter uses the definitions introduced previously to describe specification transformations that allow ORTAC to efficiently decide the predicate's validity with only the computing capabilities available to the OCaml programmer (*e. g.* no access to previous states, no access to the addresses, no `old` primitive).

The attentive reader may have noticed that MICROSPEL uses a `copy` keyword, but no copying is involved at any point in the definitions presented in this chapter. In a real-world context where imperative program alter the state during the execution, accessing 'the value of a term in the pre-state' is only possible in the pre-state, which has been destroyed. Therefore, copying some memory chunks corresponding to these terms may be the only way to actually implement this state extension, which is mandatory for checking that the predicate holds.

# 5

# EFFICIENT PRESTATE CAPTURES IN FUNCTION POSTCONDITIONS

In this chapter, we use MICROSPEL as a model to show how ORTAC instruments OCAML code when its GOSPEL specification contains postconditions that involve the `old` operator. Indeed, as we explained in chapter 2 (for GOSPEL) and formalized in chapter 4 (for MICROSPEL), the terms appearing under `old` primitives are meant to be evaluated in the prestate of the function before its execution, and the result of this evaluation is used to evaluate a postcondition predicate in a poststate.

In the presence of effects and a transient memory, the program may mutate the memory portions referred to *via* `old`. Since this memory is no longer accessible after the execution of the program, copying is necessary.

EXECUTABILITY CRITERIA. Since OCAML does not provide such a primitive, we must perform an `old` elimination in the specifications before executing them. In other words, we must ensure that $(a)$ the copied terms $t_i$ contain no `old` primitives; $(b)$ the postcondition predicate $P$ contains no `old` primitives.

**Definition 7** (Specification executability criteria)**.** A well-formed specification *sp* is executable when it contains no `old` primitive.

## 5.1 MOTIVATING EXAMPLE

Let us consider the following running example for this chapter.

*This specification initially has no copies, so we omit them in the presentation. Also note that the specification does not say anything about the rest of the array* `a`.

```
domain a: (int array) array;
modifies a;
ensures length a[0] = 2 * length (old (a[0])) ∧
        forall i, 0 ⩽ i < length (old (a[0])) ->
                    a[0][i] = old (a[0][i])
```

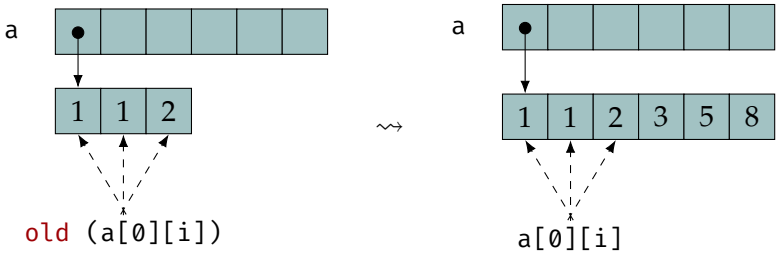
*Notation* 3. We will use the following notations to shorten the two parts of the predicate:

$P_1$ := `length a[0] = 2 * length (old (a[0]))`

$P_2$ := `forall i, 0 ⩽ i < length (old (a[0])) ->`
`a[0][i] = old (a[0][i])`

It specifies a program that takes an array of integer arrays `a` as input, and modifies it in place. It doubles the size of the array in the first cell ($P_1$), but maintains the existing elements at the head of the array ($P_2$).

For instance, the following execution is correct with respect to this specification: it modifies nothing other than the array `a`; the size of `a[0]` is doubled from 3 to 6, and the first three elements (1, 1, and 2) are still the first three elements after the execution.




COPYING IS NECESSARY. When executing this specification, the postcondition should be evaluated in the poststate (*i.e.* in the right-hand side state above). However, some of the data necessary for the computation (*e.g.* `old a[0][i]` or `length (old (a[0]))`) points to memory portions in the prestate (left-hand side). Therefore, copying a prestate version of `a[0]` is necessary before the execution of the program.

REPLACING `old` WITH COPIES IS TOO NAIVE. A first intuition would consist in copying the data necessary to evaluate the terms under `old`, saving their contents into fresh variables, and replacing the terms with these variables, as noted in the previous chapter. However, this can lead to ill-formed specifications. For instance, in the case of $P_2$, copying `a[0][i]` in the prestate does not make sense since `i` does not exist in the prestate; the resulting specification would not pass the typing phase.

## 5.2 EXECUTING PRESTATE CAPTURES

This section presents the transformations we apply to specifications in order to make them executable and acceptable by ORTAC (with respect to definition 7).

### 5.2.1 *Removing Redundant* `old` *Primitives*

This section describes a first sanitation transformation that removes `old` primitives that do not impact the specification semantics. Specifically, we want to remove (*a*) all `old` primitives in the copied terms; (*b*) nested `old` primitives in the postcondition.

*Remark* 10. One could assume or enforce these properties during the typing phase, but being overly restrictive gets in the way of writing intelligible specifications.

Indeed, in the context of GOSPEL, the library developer may willingly add redundant `old` in the postcondition to emphasize a particular aspect of the specification. The primitives may also appear after applying and inlining logic predicates that use them in their definition. Finally, forbidding redundant `old` could get in the way of automatically adding these primitives in GOSPEL or ORTAC, *e. g.* around values that are not mutable or while generating specifications.

The first part of the transformation lets us remove all `old` primitives from a term and its sub-terms, *i. e.* it replaces the sub-terms of the form `old` $t$ with $t$.

**Definition 8** (*remove_old*)**.** We define the transformation *remove_old* as follows when applied to `old` terms:

$$\texttt{old}\ t \ \mapsto\ t$$

In other cases, *remove_old* is a simple recursive traversal of the terms.

$$\begin{aligned}
x &\mapsto x \\
n &\mapsto n \\
t_1\ \texttt{+}\ t_2 &\mapsto \mathit{remove_old}(t_1)\ \texttt{+}\ \mathit{remove_old}(t_2) \\
t_1\ \texttt{-}\ t_2 &\mapsto \mathit{remove_old}(t_1)\ \texttt{-}\ \mathit{remove_old}(t_2) \\
t_1\texttt{[}t_2\texttt{]} &\mapsto (\mathit{remove_old}(t_1))\texttt{[}\mathit{remove_old}(t_2)\texttt{]} \\
\texttt{length}\ t &\mapsto \texttt{length}\ (\mathit{remove_old}(t)) \\
t\texttt{.}n &\mapsto (\mathit{remove_old}(t))\texttt{.}n
\end{aligned}$$

The definition is well-founded since recursive calls apply to structurally decreasing terms.

> **Example 7.**
>
> $$\mathit{remove\_old}(\texttt{2 * length (old (a[0]))}) = \texttt{2 * length a[0]}$$

The second part, *remove_nested_old*, lets us remove *nested* `old` primitives in a term. Just like *remove_old*, it recursively traverses the terms except for terms of the form `old` $t$, where it removes all `old` primitives in $t$.

**Definition 9** (*remove_nested_old* (terms))**.** We define the term transformation *remove_nested_old* as follows for `old` terms:

$$\texttt{old}\ t \ \mapsto\ \texttt{old}\ (\mathit{remove\_old}(t))$$

In other cases, *remove_nested_old* is also a recursive traversal of the terms.

$$
\begin{array}{rcl}
x & \mapsto & x \\
n & \mapsto & n \\
t_1 \texttt{ + } t_2 & \mapsto & \mathit{remove\_nested\_old}(t_1) \texttt{ + } \mathit{remove\_nested\_old}(t_2) \\
t_1 \texttt{ - } t_2 & \mapsto & \mathit{remove\_nested\_old}(t_1) \texttt{ - } \mathit{remove\_nested\_old}(t_2) \\
t_1\texttt{[}t_2\texttt{]} & \mapsto & (\mathit{remove\_nested\_old}(t_1))\texttt{[}\mathit{remove\_nested\_old}(t_2)\texttt{]} \\
\texttt{length } t & \mapsto & \texttt{length } (\mathit{remove\_nested\_old}(t)) \\
t\texttt{.}n & \mapsto & (\mathit{remove\_nested\_old}(t))\texttt{.}n
\end{array}
$$

This definition is also well-founded for the same reason as *remove_old*: it recursively applies to structurally decreasing terms.

> **Example 8.**
>
> *remove_nested_old* `(old (2 * length (old (a[0]))))` =
> `old (2 * length a[0])`

We also generalize the transformation *remove_nested_old* for predicates with a simple morphism that applies the previously defined transformation down to the terms.

> **Example 9.**
>
> *remove_nested_old*
> `(forall i, 0 ⩽ i < n -> a[i] = old ((old a)[i]))` =
> `forall i, 0 ⩽ i < n -> a[i] = old (a[i])`

Finally, the *sanitize* transformation applies to specification in order to remove duplicate `old` operators.

**Definition 10** (*sanitize*)**.** If *sp* is the following specification:

```
domain x_0: τ_0, ..., x_{d-1}: τ_{d-1};
modifies y_0, ..., y_{m-1};
let z_0, ..., z_{c-1} = copy (t_0, ..., t_{c-1}) in
ensures P,
```

then *sanitize*(*sp*) is the following specification:

```
domain x_0: τ_0, ..., x_{d-1}: τ_{d-1};
modifies y_0, ..., y_{m-1};
let z_0, ..., z_{c-1} =
  copy (remove_old(t_0), ..., remove_old(t_{c-1}))
in
ensures remove_nested_old(P).
```

**Example 10.** The following specification

```
domain a: (int array) array;
modifies a;
let x = copy ((old a)[0]) in
ensures length a[0] = 2 * old (length x) ∧
        forall i, 0 ⩽ i < length x ->
                    a[0][i] = old ((old a)[0][i])
```

is sanitized into

```
domain a: (int array) array;
modifies a;
let x = copy (a[0]) in
ensures length a[0] = 2 * old (length x) ∧
        forall i, 0 ⩽ i < length x ->
                    a[0][i] = old a[0][i]
```

*Remark* 11. In the context of OCaml, notice how one could also simplify `old (length x)` into `length x` since `x` is already a prestate variable, and arrays lengths in OCaml cannot be modified. This task is not the role of *sanitize*; we will discuss it in the next section.

CORRECTNESS. This transformation is correct, meaning that (*a*) it does not change the good formation of the specification; (*b*) it does not change the semantics of the specification; (*c*) the copied terms in the result do not contain any `old` and the postcondition of the result does not contain any nested `old`.

**Theorem 6** (*sanitize* preserves the good formation of the specification)**.** *If sp is a specification, we have*

$$wf(sp) \iff wf(sanitize(sp)).$$

```
Theorem sanitize_wf:
  forall sp, wf_spec sp <-> wf_spec (sanitize sp).
```

*Proof.* By induction over terms and predicates. The `old` primitive does not affect typing (see TY-OLD rule). □

**Theorem 7** (*sanitize* maintains program correctness)**.** *If p is a well-formed program and sp is a well-formed specification, we have*

$$p \vDash sp \iff p \vDash sanitize(sp).$$

```
Theorem sanitize_correct:
  forall p sp,
    wf_spec sp ->
    wf_prog p ->
    correct p sp <-> correct p (sanitize sp).
```

*Proof.* By induction over terms and predicates. Concerning nested `old`, as noted in section 4.3.2, the primitive is idempotent, so the inner `old` in nested configurations have no effect on the specification semantics. As for the copied terms, they are always evaluated in the prestate (see section 4.3.4.2), so `old` primitives are also safely removable in this context. □

*Remark* 12. In this theorem, and the upcoming similar ones in this chapter, the direct implication ( $\Longrightarrow$ ) ensures that the transformed specification is not more restrictive than the original one, which would cause ORTAC to trigger *false positives*. Conversely the indirect implication ( $\Longleftarrow$ ) ensures that the transformation did not loosen the constraints of the specification, which would trigger *false negatives*.

**Theorem 8** (*sanitize* postcondition)**.** *If sp is a specification, then*

(*a*) *no copied term in sanitize*(*sp*) *contain a* `old`*;*

(*b*) *the postcondition in sanitize*(*sp*) *contains no nested* `old`*.*

```
Theorem sanitize_post:
  forall sp,
   no_old (post (sanitize sp)) ∧
   Forall no_nested_old (copies (sanitize sp)).
```

*Proof.* By induction over terms and predicates. □

### 5.2.2 *Moving* `old` *Down to Variables*

In order to address the issue of locally bound variables we raised in our example, we propose a second term transformation with two effects: (*a*) it 'moves' the `old` primitives down to the variables; (*b*) it removes some of these `old`.

Indeed, when `old` is applied to a variable introduced by a `forall` or `exists` binder, we claim that removing `old` does not change the semantics of the term. Similarly, we remove `old` primitives around variables introduced by `copy`, with no prejudice to the semantics of the specification. Finally, we also remove `old` around variables that are not declared in the `modifies` variables.

*As in* ORTAC*, the* `modifies` *part is considered an hypothesis that we can use to optimize the specification.*

The outcome of this transformation is a specification where only the potentially modified variables available in the prestate appear under `old` primitives.

The first part of this transformation is a function *old_vars* that takes two arguments: a term and the set of locally bound variables $\mathcal{B}$, *i. e.* variables introduced by a `copy` or a quantifier. It returns a new term where it added `old` around all variables, except for those in $\mathcal{B}$.

**Definition 11** (*old_vars*)**.** We define the transformation *old_vars* as follows when applied to variables.

$$x,\ \mathcal{B} \mapsto \begin{cases} x & \text{if } x \in \mathcal{B} \\ \texttt{old}\ x & \text{otherwise} \end{cases}$$

In other cases, *old_vars* recursively traverses the term while maintaining the set $\mathcal{B}$ unchanged (these terms introduce no new variable).

*The definition provided for* $old\_vars(\texttt{old}\ t, \mathcal{B})$ *does not matter as long as old_down remains well-founded since the global transformation defined later in definition 14 will never reach that case.*

$$\begin{aligned}
n,\ \mathcal{B} &\mapsto n \\
t_1\ \texttt{+}\ t_2,\ \mathcal{B} &\mapsto old_vars(t_1, \mathcal{B})\ \texttt{+}\ old_vars(t_2, \mathcal{B}) \\
t_1\ \texttt{-}\ t_2,\ \mathcal{B} &\mapsto old_vars(t_1, \mathcal{B})\ \texttt{-}\ old_vars(t_2, \mathcal{B}) \\
t_1\texttt{[}t_2\texttt{]},\ \mathcal{B} &\mapsto (old_vars(t_1, \mathcal{B}))\texttt{[}old_vars(t_2, \mathcal{B})\texttt{]} \\
\texttt{length}\ t,\ \mathcal{B} &\mapsto \texttt{length}\ (old_vars(t, \mathcal{B})) \\
t_1\texttt{.}n,\ \mathcal{B} &\mapsto (old_vars(t_1, \mathcal{B}))\texttt{.}n \\
\texttt{old}\ t,\ \mathcal{B} &\mapsto old_vars(t, \mathcal{B})
\end{aligned}$$

**Example 11.**

$$old\_vars(\texttt{length a[0]}, \emptyset) \ = \ \texttt{length ((old a)[0])}$$

We may now use this transformation to introduce *old_down*, which 'moves' the `old` primitives down to the variables and removes the unnecessary ones using *old_vars*. It takes the same arguments as *old_vars* and returns a term.

**Definition 12** (*old_down* (terms))**.** We define the function *old_down* as follows for the cases of variables and `old`.

*Unlike in old_vars, variables are left untouched, because they are not located under* `old` *at this stage.*

$$\begin{aligned}
x,\ \mathcal{B} &\mapsto x \\
\texttt{old}\ t,\ \mathcal{B} &\mapsto old_vars(t, \mathcal{B})
\end{aligned}$$

Again, *old_down* recursively traverses the term with the same set $\mathcal{B}$ in other cases.

$$\begin{aligned}
n,\ \mathcal{B} &\mapsto n \\
t_1\ \texttt{+}\ t_2,\ \mathcal{B} &\mapsto old_down(t_1, \mathcal{B})\ \texttt{+}\ old_down(t_2, \mathcal{B}) \\
t_1\ \texttt{-}\ t_2,\ \mathcal{B} &\mapsto old_down(t_1, \mathcal{B})\ \texttt{-}\ old_down(t_2, \mathcal{B}) \\
t_1\texttt{[}t_2\texttt{]},\ \mathcal{B} &\mapsto (old_down(t_1, \mathcal{B}))\texttt{[}old_down(t_2, \mathcal{B})\texttt{]} \\
\texttt{length}\ t,\ \mathcal{B} &\mapsto \texttt{length}\ (old_down(t, \mathcal{B})) \\
t_1\texttt{.}n,\ \mathcal{B} &\mapsto (old_down(t_1, \mathcal{B}))\texttt{.}n
\end{aligned}$$

**Example 12.**

```
old_down(2 * old (length a[0]), ∅) =
                              2 * length ((old a)[0])

old_down(2 * old (length a[0]), {a}) =
                                    2 * length a[0]
```

We also extend *old_down* for predicates and take special care to expand $\mathcal{B}$ with newly bound variables when traversing quantifiers.

**Definition 13** (*old_down* (predicates))**.** The transformation *old_down* is a predicate transformation defined as follows for quantifiers that introduce variables:

$$\texttt{forall } x, t_1 \leqslant x < t_2 \texttt{ -> } P,\ \mathcal{B} \mapsto \texttt{forall } x,\ \mathit{old\_down}(t_1, \mathcal{B}) \leqslant x < \mathit{old\_down}(t_2, \mathcal{B}) \texttt{ -> } \mathit{old\_down}(P, \mathcal{B} \cup \{x\})$$

$$\texttt{exists } x, t_1 \leqslant x < t_2 \wedge P,\ \mathcal{B} \mapsto \texttt{exists } x,\ \mathit{old\_down}(t_1, \mathcal{B}) \leqslant x < \mathit{old\_down}(t_2, \mathcal{B}) \wedge \mathit{old\_down}(P, \mathcal{B} \cup \{x\})$$

In other cases, *old_down* is a simple morphism.

$$\begin{aligned}
\texttt{true},\ \mathcal{B} &\mapsto \texttt{true} \\
\texttt{false},\ \mathcal{B} &\mapsto \texttt{false} \\
\texttt{not } P,\ \mathcal{B} &\mapsto \texttt{not } \mathit{old_down}(P, \mathcal{B}) \\
t_1 = t_2,\ \mathcal{B} &\mapsto \mathit{old_down}(t_1, \mathcal{B}) = \mathit{old_down}(t_2, \mathcal{B}) \\
P_1 \wedge P_2,\ \mathcal{B} &\mapsto \mathit{old_down}(P_1, \mathcal{B}) \wedge \mathit{old_down}(P_2, \mathcal{B}) \\
P_1 \vee P_2,\ \mathcal{B} &\mapsto \mathit{old_down}(P_1, \mathcal{B}) \vee \mathit{old_down}(P_2, \mathcal{B})
\end{aligned}$$

**Example 13.**

```
old_down(P1) = length a[0] =
                 2 * length ((old a)[0])
old_down(P2) = forall i,
                 0 ⩽ i < length ((old a)[0]) ->
                 a[0][i] = (old a)[0][i]
```

Finally, we define *old_down* for specifications. It is a transformation that takes a specification as an argument and returns a specification. It applies *old_down* (for predicates) to the postcondition predicate. The initial set $\mathcal{B}$ is the set of variables introduced by `copy` or declared in `modifies`.

**Definition 14** (*old_down* (specification))**.** If *sp* is the following specification:

```
domain x_0: τ_0, ..., x_{d-1}: τ_{d-1};
modifies y_0, ..., y_{m-1};
let z_0, ..., z_{c-1} = copy (t_0, ..., t_{c-1}) in
ensures P,
```

then *old_down*(*sp*) is the following specification:

```
domain x_0: τ_0, ..., x_{d-1}: τ_{d-1};
modifies y_0, ..., y_{m-1};
let z_0, ..., z_{c-1} = copy (t_0, ..., t_{c-1}) in
ensures old_down(P, {y_0, ..., y_{m-1}, z_0, ..., z_{c-1}}).
```

> **Example 14.** After applying this transformation, our initial specifications becomes
>
> ```
> domain a: (int array) array;
> modifies a;
> ensures length a[0] = 2 * length ((old a)[0]) ∧
>         forall i, 0 ⩽ i < length ((old a)[0]) ->
>                     a[0][i] = (old a)[0][i].
> ```

CORRECTNESS. This transformation is correct, meaning that (*a*) it does not change the good formation of the specification; (*b*) it does not change the semantics of the specification; (*c*) in the resulting specification, if a sub-term is of the form `old t`, then *t* is a variable that belongs to the domain of the specification.

**Theorem 9** (*old_down* preserves the good formation of the specification)**.** *If sp is a specification, we have*

$$wf(sp) \iff wf(old\_down(sp)).$$

```
Lemma wf_old_down:
  forall sp, wf_spec sp <-> wf_spec (old_down sp).
```

*Proof.* Similar to theorem 6: `old` does not affect the typing of terms. □

**Theorem 10** (*old_down* maintains program correctness)**.** *If p is a well-formed program and sp is a well-formed specification, we have*

$$p \vDash sp \iff p \vDash old\_down(sp).$$

```
Lemma old_down_correct:
  forall p sp,
    wf_spec sp ->
    wf_prog p ->
    correct p sp <-> correct p (old_down sp).
```

*Proof.* By induction over the terms and predicates. Moving `old` down to the variables does not affect the semantics (see rules T_Old and T_Var. Moreover, the `old` we chose to remove are either locally introduced, or not modified by the program; in both cases, the `old` primitive is redundant. □

**Theorem 11** (*old_down* postcondition)**.** *If sp is a well-formed specification, and t is a term appearing in old_down(sp), then t is a variable declared in sp's mutable variables.*

```
Theorem old_down_post:
  forall sp t,
    wf_spec sp ->
    subterm (TOld t) sp ->
    exists x,
      In x (modifies sp) ∧ t = TVar x.
```

*Proof.* By induction over terms and predicates. □

### 5.2.3 *Introducing Copies*

In this section, we show the last transformation required to make the specifications executable. The function *collect_old* traverses the terms and collects the `old` sub-terms to replace with fresh variables. It produces the transformed terms, and a vector $\mathcal{V}$ of pairs $(z, t)$ of variables and terms that represent the sub-term under `old` that it replaced and the corresponding variables that replaced it.

*To make the definition easier to read, we present it using an imperative syntax, where collect_old modifies a global vector $\mathcal{V}$. In Coq, we implemented a state monad to achieve a similar presentation.*

**Definition 15** (*collect_old*)**.** We define the function *collect_old* as follows. When it faces a term `old` $t$, *collect_old* creates a fresh variable $x$, adds the pair $(x, t)$ to the vector $\mathcal{V}$, and returns the variable $x$.

$$\texttt{old}\ t \ \mapsto \quad \begin{array}{lll} x & \leftarrow & \mathit{fresh\_var}(); \\ \mathcal{V} & \leftarrow & (x, t) \ :: \ \mathcal{V}; \\ x & & \end{array}$$

For the other cases, *collect_old* recursively traverses the terms.

$$
\begin{array}{rcl}
x & \mapsto & x \\
n & \mapsto & n \\
t_1 \texttt{ + } t_2 & \mapsto & t_1' \leftarrow \mathit{collect\_old}(t_1); \\
 & & t_2' \leftarrow \mathit{collect\_old}(t_2); \\
 & & t_1' \texttt{ + } t_2' \\
t_1 \texttt{ - } t_2 & \mapsto & t_1' \leftarrow \mathit{collect\_old}(t_1); \\
 & & t_2' \leftarrow \mathit{collect\_old}(t_2); \\
 & & t_1' \texttt{ - } t_2' \\
t_1[t_2] & \mapsto & t_1' \leftarrow \mathit{collect\_old}(t_1); \\
 & & t_2' \leftarrow \mathit{collect\_old}(t_2); \\
 & & t_1'[t_2'] \\
\texttt{length } t & \mapsto & t' \leftarrow \mathit{collect\_old}(t); \\
 & & \texttt{length } t' \\
t_1.n & \mapsto & t' \leftarrow \mathit{collect\_old}(t); \\
 & & t'.n
\end{array}
$$

We generalize this transformation for predicates, where it recursively traverses the predicate and applies *collect_old* to every term. Finally, the transformation *introduce_copies* applies *collect_old* to its postcondition, then adds the pairs in $\mathcal{V}$ to its `copy` bindings.

**Definition 16** (*introduce_copies*)**.** If *sp* is the following specification:

```
domain x_0: τ_0, ..., x_{d-1}: τ_{d-1};
modifies y_0, ..., y_{m-1};
let z_0, ..., z_{c-1} = copy (t_0, ..., t_{c-1}) in
ensures P,
```

we apply the function *collect_old* to its postcondition:

$$P' \leftarrow \mathit{collect\_old}(P),$$

*Applying collect_old here creates the predicate $P'$, and also fills the vector $\mathcal{V}$.*

then return the following specification:

```
domain x_0: τ_0, ..., x_{d-1}: τ_{d-1};
modifies y_0, ..., y_{m-1};
let z_0, ..., z_{c-1}, z'_0, ..., z'_s =
  copy (t_0, ..., t_{c-1}, z'_0, ..., z'_s)
in
ensures P',
```

when $\mathcal{V} = \{(z_0', t_0'), \ldots, (z_s', t_s')\}$.

> **Example 15.** Consider the following specification:

```
domain a: (int array) array;
modifies a;
ensures length a[0] = 2 * length ((old a)[0]) ∧
        forall i, 0 ⩽ i < length (old (a[0])) ->
                  a[0][i] = (old a)[0][i].
```

The function *introduce_copies* will produce

```
domain a: (int array) array;
modifies a;
let z_0, z_1, z_2 = copy (a, a[0], a[0][i]) in
ensures length a[0] = 2 * length z_0[0] ∧
        forall i, 0 ⩽ i < length z_1 -> a[0][i] = z_2.
```

CORRECTNESS. Before proving the correctness of this function, note that the properties that hold for *sanitize* and *old_down* do *not* necessarily hold for *introduce_copies*. For instance, transforming a well-formed specification can lead to an ill-formed specification as output.

**Example 16** (*introduce_copies* can lead to ill-formed specification)**.** Our example specification is well formed, as we discussed earlier. However, applying *introduce_copies* leads to the following specification:

```
domain a: (int array) array;
modifies a;
let z_0, z_1, z_2 = copy (a[0], a[0], a[0][i]) in
ensures length a[0] = 2 * length z_0 ∧
        forall i, 0 ⩽ i < length z_1 -> a[0][i] = z_2,
```

which is ill-formed, since one cannot type `a[0][i]` in the environment `a : (int array) array` (because of the unbound variable `i`).

In fact, *introduce_copies* is meant to be executed *after old_down*, and we need this hypothesis to prove its correctness. This transformation is correct, meaning that when fed a well-formed specification that also obeys the properties described in theorem 11, (*a*) it maintains the good formation of the specification; (*b*) it maintains the semantics of the specification; (*c*) there are no more `old` primitives in the resulting specification's postcondition.

**Theorem 12** (*introduce_copies* preserves the good formation of the specifications)**.** *If sp is a specification, we have*

$$wf(old\_down(sp)) \iff wf(introduce\_copies(old\_down(sp))).$$

```
Theorem introduce_copies_wf:
  forall sp,
```

```
    wf_spec (old_down sp) <->
    wf_spec (introduce_copies (old_down sp)).
```

*Proof.* Replacing a term with a variable of the same type does not change the good formation of the specification. □

**Theorem 13** (*introduce_copies* maintains program correctness)**.** *If p is a well-formed program and sp is a well-formed specification, we have*

$$p \vDash sp \iff p \vDash introduce\_copies(sp).$$

```
Theorem introduce_copies_correct:
  forall p sp,
    wf_spec sp ->
    wf_prog p ->
    correct p sp <-> correct p (introduce_copies sp).
```

*Proof.* By applying T-Old and the `copy` semantics described in definition 6. □

**Theorem 14** (*introduce_copies* postcondition)**.** *If sp is a well-formed specification, then for all sub-terms of the form* `old t` *appearing in old_down*(*sp*)*, the term t is a variable declared in sp's domain.*

```
Theorem introduce_copies_post:
  forall sp t,
    wf_spec sp ->
    subterm (TOld t) sp ->
    exists x, In x (domain sp) ∧ t = TVar x.
```

*Proof.* By applying T-Old and the `copy` semantics described in definition 6. □

### 5.2.4 *Wrapping Up: How We Make the Specification Executable*

We may now compose the three transformations we presented to transform arbitrary well-formed specifications into equivalent well-formed specifications that satisfy the executability criteria (definition 7).

We note this final transformation $T_{base}$:

$$T_{base} = introduce\_copies \circ old\_down \circ sanitize$$

*Remark* 13. The order of *sanitize* does not matter for the computation result or the proofs; the only constraint is that *introduce_copies* is fed a specification where the property from theorem 11 holds. However, applying *sanitize* first is convenient since it lets us ignore some cases that are eliminated in our transformation (*e. g.* in *old_down*, as we noted earlier).

*This specification could be simplified by only copying* `a` *once rather than three times. However, this difference is of little importance if the concrete implementation of* `copy` *handles memory sharing properly, which we will show in chapter 6.*

**Example 17.** Our initial example

```
domain a: (int array) array;
modifies a;
ensures length a[0] == 2 * length (old (a[0])) ∧
        forall i, 0 ⩽ i < length (old (a[0])) ->
                    a[0][i] == old (a[0][i])
```

finally becomes

```
domain a: (int array) array;
modifies a;
let a_0, a_1, a_2 = copy (a, a, a) in
ensures length a[0] == 2 * length a_0[0] ∧
        forall i, 0 ⩽ i < length a_1[0] ->
                  a[0][i] == a_2[0][i],
```

which is indeed well formed and does not contain any more `old` operators, therefore is executable.

**Theorem 15** ($T_{base}$ preserves the good formation of the specifications)**.** *If sp is a specifications, we have*

$$wf(sp) \iff wf(T_{base}(sp)).$$

```
Theorem t_base_wf:
  forall sp, wf_spec sp <-> wf_spec (t_base sp).
```

*Proof.* By applying theorems 6, 9 and 12. □

**Theorem 16** ($T_{base}$ maintains program correctness)**.** *If p is a well-formed program and sp is a well-formed specification, we have*

$$p \vDash sp \iff p \vDash T_{base}(sp).$$

```
Theorem t_base_correct:
  forall p sp,
    wf_spec sp ->
    wf_prog p ->
    correct p sp <-> correct p (t_base sp).
```

*Proof.* By applying theorems 7, 10, 11 and 13. □

**Theorem 17** ($T_{base}$ produces executable specifications)**.** *If sp is a specification, then* $T_{base}(sp)$ *does not contain any* `old`*.*

```
Theorem t_base_executable:
  forall sp, wf_spec sp -> executable (t_base sp).
```

*Proof.* By applying theorems 8, 11 and 14. □

## 5.3 REDUCING THE COPIED SPACE

We made sure that the transformation $T_{base}$ is correct, but we did not pay attention to the performance of the execution of the resulting specification just yet. Although we have not yet discussed the execution of the specifications itself, we can already sense with definition 6 that executing a specification will induce a memory cost due to the `copy` part. For instance, the result of $T_{base}$ on our specification presented in section 5.2.4 induces a copy of the full array `a`, whereas only `a[0]` is necessary to decide if the postcondition holds.

*This is the object of the chapter 6, where we will also discuss the actual implementation and cost of the copy.*

In this section, we present a last transformation that investigates *how* we manipulate the `old` variables to helps us reduce the copied space and improve the performance. Indeed, the instrumentation not only results in high runtime verification overhead, but can also change the complexity of the algorithm (see an example in section 5.5.3), threatening its scalability.

### 5.3.1 *The Cost of Copying*

For now, let us approximate the cost of `copy (`$t_1$`, ..., `$t_c$`)` as the size (*i. e.* the number of memory cells) of the memory necessary to fully evaluate the terms $t_1, \dots, t_c$. This is the metric we will try to optimize.

**Definition 17** (Footprint of a copy)**.** Let us note $\mathcal{A}_M(v)$ the set of addresses that one must read to resolve $M, v \twoheadrightarrow lv$ following section 4.3.1. We have

$$\begin{aligned} \mathcal{A}_M(n) &= \emptyset; \\ \mathcal{A}_M(a) &= \{a\} \cup \left( \bigcup_{i=0}^{n-1} \mathcal{A}_M(v_i) \right), \end{aligned}$$

when $M(a) = [v_0, \dots, v_{n-1}]$.

We also use the notation $\mathcal{A}_S$ for the function that gives the footprint of a term in the state $S$. In the case of variables, we look for the footprint of the value associated to this variable:

$$x \mapsto \mathcal{A}_M(V(x)).$$

*We can avoid a definition for $\mathcal{A}_S$(`old` $t$) when sanitize is applied first, since after sanitation, there are no more `old` in the copied terms.*

The cases for tuple and array projections are defined as follows:

$$\begin{aligned} t_1[t_2] &\mapsto \mathcal{A}_M(v_i), \\ &\quad \text{when } t_1 \text{ evaluates to } [v_0, \dots, v_{n-1}] \text{ and } t_i \text{ evaluates to } i. \\ t.i &\mapsto \mathcal{A}_M(v_i), \\ &\quad \text{when } t \text{ evaluates to } [v_0, \dots, v_{n-1}]. \end{aligned}$$

The terms which evaluation produces integers have an empty footprint.

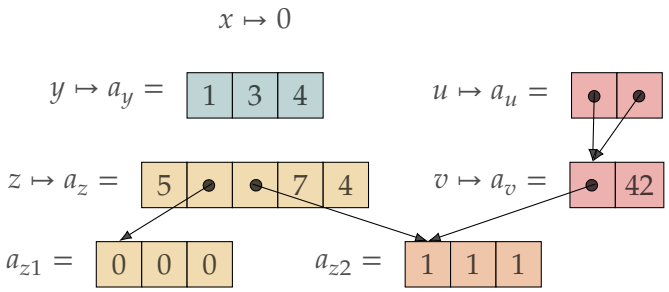
$$\begin{aligned} n &\mapsto \emptyset \\ t_1 \texttt{ + } t_2 &\mapsto \emptyset \\ t_1 \texttt{ - } t_2 &\mapsto \emptyset \\ \texttt{length } t &\mapsto \emptyset \end{aligned}$$

Finally, the memory footprint of the copy in a specification is the union of the footprints of the copied terms:

$$\mathcal{A}_S(sp) = \bigcup_{i=0}^{c-1} \mathcal{A}_S(t_i),$$

when $copies(sp) = [(z_0, t_0), \dots, (z_{c-1}, t_{c-1})]$.

**Example 18.** Consider the following state $S$.




In this state, we have:

$$\begin{aligned}
\mathcal{A}_S(x) &= \emptyset \\
\mathcal{A}_S(y) &= \{a_y\} \\
\mathcal{A}_S(z) &= \{a_z, a_{z1}, a_{z2}\} \\
\mathcal{A}_S(\texttt{1 + u[0].1}) &= \emptyset \\
\mathcal{A}_S(u) &= \{a_u, a_v a_{z2}\}
\end{aligned}$$

We can use this definition to define the cost of a copy as the sum of the sizes of the memory blocks pointed by the addresses in the footprint.

*This definition is* not *equivalent to considering the cost of the copy of a term as the size of its logic value. Indeed, logic values do not take memory sharing into account.*

**Definition 18** (Cost of a copy)**.** Let us note $\mathcal{W}_S(sp)$ the final *cost* of the copies induced by $sp$. We have

$$\mathcal{W}_S(sp) = \sum_{a \in \mathcal{A}_S(sp)} |M(a)|,$$

where $|M(a)| = n$ when $M(a) = [v_0, \dots, v_{n-1}]$.

**Example 19.** In the state we considered above, we have:

$$\begin{aligned}
\mathcal{W}_S(x) &= 0 \\
\mathcal{W}_S(y) &= 3 \\
\mathcal{W}_S(z) &= 5 + 3 + 3 = 11 \\
\mathcal{W}_S(\texttt{1 + u[0].1}) &= 0 \\
\mathcal{W}_S(u) &= 2 + 2 + 3 = 7
\end{aligned}$$

> Note that although $u$ contains two pointers to $v$, the footprint taken by $v$ is only counted once.

### 5.3.2 *Moving* `old` *Upwards*

The goal of the transformation *old_up* is to 'move' the `old` operators upwards in the terms. That will lead to moving as much computations *under* `old` as possible, so the term size is reduced *before* we copy it using *introduce_copies*.

**Definition 19** (*old_up*)**.** We define the term transformation *old_up* as follows. For `old` terms, no transformation is needed:

$$\texttt{old}\ t \ \mapsto\ \texttt{old}\ t$$

In the other cases, *old_up* moves `old` upwards when it can transform all the sub-terms into `old`.

*Note that in the case of variables, we do* not *add an* `old` *primitive.*

$$x \mapsto x$$

$$n \mapsto \texttt{old}\ n$$

$$t_1\ \texttt{+}\ t_2 \mapsto \begin{cases} \texttt{old}\ (t_1' + t_2') & \text{when } \mathit{old\_up}(t_1) = \texttt{old}\ t_1' \\ & \quad \text{and } \mathit{old\_up}(t_2) = \texttt{old}\ t_2' \\ \mathit{old\_up}(t_1)\ \texttt{+}\ \mathit{old\_up}(t_2) & \text{otherwise} \end{cases}$$

$$t_1\ \texttt{-}\ t_2 \mapsto \begin{cases} \texttt{old}\ (t_1' - t_2') & \text{when } \mathit{old\_up}(t_1) = \texttt{old}\ t_1' \\ & \quad \text{and } \mathit{old\_up}(t_2) = \texttt{old}\ t_2' \\ \mathit{old\_up}(t_1)\ \texttt{-}\ \mathit{old\_up}(t_2) & \text{otherwise} \end{cases}$$

$$t_1[t_2] \mapsto \begin{cases} \texttt{old}\ (t_1'[t_2']) & \text{when } \mathit{old\_up}(t_1) = \texttt{old}\ t_1' \\ & \quad \text{and } \mathit{old\_up}(t_2) = \texttt{old}\ t_2' \\ \mathit{old\_up}(t_1)[\mathit{old\_up}(t_2)] & \text{otherwise} \end{cases}$$

$$\texttt{length}\ t \mapsto \begin{cases} \texttt{old}\ (\texttt{length}\ t') & \text{when } \mathit{old\_up}(t) = \texttt{old}\ t' \\ \texttt{length}\ (\mathit{old\_up}(t)) & \text{otherwise} \end{cases}$$

$$t.n \mapsto \begin{cases} \texttt{old}\ (t'.n) & \text{when } \mathit{old\_up}(t) = \texttt{old}\ t' \\ \mathit{old\_up}(t).n & \text{otherwise} \end{cases}$$

This definition is well-founded for the same reason as *remove_old* as it applies to structurally decreasing terms.

*Remark* 14. When applying *old_up*, we may end up with more `old` primitives than before, particularly around constants and operations on constants. For instance the term `1 + x` becomes `old 1 + x`. These `old` are obviously unnecessary, but they $(a)$ have a copy cost of 0 since they are constants; $(b)$ are easy to detect and remove if necessary.

We generalize this transformation to predicates as a simple recursive traversal that applies to terms. We also generalize it to specifications, where we apply *old_up* to the postcondition.

Finally, the ultimate transformation applies the function *old_up* before *introduce_copies*, but after *old_down*:

*The functions old_up and old_down have opposite effects, but they are* not *the inverse function of each other.*

$$T_{opt} = \mathit{introduce\_copies} \circ \mathit{old\_up} \circ \mathit{old\_down} \circ \mathit{sanitize}$$

**Example 20** ($T_{opt}$ examples)**.** The term `1 + (old x)[0]` is transformed into `old (1 + x[0])`.
However, when considering the predicate

```
forall i, 0 ⩽ i < n -> 1 + (old x)[i] == 0,
```

we cannot transform `1 + (old x)[i]` into `old (1 + x[i])` because the variable `i` is not under `old`.
If we apply *old_up* ∘ *old_down* to our example specification. We get

```
domain a: (int array) array;
modifies a;
ensures length a[0] == old (2 * length a[0]) ∧
        forall i, 0 ⩽ i < old (length a[0]) ->
                  a[0][i] == old (a[0])[i]
```

Then, after *introduce_copies*, we obtain the following final specification.

```
domain a: (int array) array;
modifies a;
let z_0, z_1, z_2 =
  copy (2 * length a[0], length a[0], a[0])
in
ensures length a[0] == z_0 ∧
        forall i, 0 ⩽ i < z_1 ->
                  a[0][i] == z_2[i]
```

### 5.3.3 *Correctness and Optimization*

PROOF OF CORRECTNESS. The correctness of this transformation does not make additional hypothesis on the specification. The transformation does not affect the good formation of the specification, nor does it change its semantics.

**Theorem 18** (*old_up* preserves the good formation of the specification)**.** *If sp is a specification, we have*

$$\mathit{wf}(\mathit{sp}) \iff \mathit{wf}(\mathit{old\_up}(\mathit{sp}))$$

*Proof.* This is the same argument as in theorem 9: the `old` operator does not affect the types of the terms. □

**Theorem 19** (*old_up* maintains program correctness)**.** *If $p$ is a well-formed program and $sp$ is a well-formed specification, we have*

$$p \vDash sp \iff p \vDash T_{opt}(sp)$$

*Proof.* By induction on terms and predicates. □

PROOF OF OPTIMIZATION. When moving `old` up before introducing copies, we reduce the cost of copies. In fact, the set of addresses that `copy` needs to read when applying $T_{opt}$ is a subset of the one obtained after applying $T_{base}$ only.

**Theorem 20** (*old_up* reduces the set cost of the copies)**.** *For all specifications $sp$, we have*

$$\mathcal{A}\left(T_{opt}(sp)\right) \subseteq \mathcal{A}\left(T_{base}(sp)\right).$$

*It immediately follows that*

$$\mathcal{W}\left(T_{opt}(sp)\right) \leq \mathcal{W}\left(T_{base}(sp)\right).$$

*Proof.* The function $\mathcal{A}_S$ over terms is monotonically non-increasing. □

ABOUT OPTIMALITY. Although this approach lets us save significant memory space and thus computation time in some cases (*e. g.* see section 5.5), it does not provide the optimal solution to minimizing the copied memory.

Indeed, a trivial example would consist in a specification with the postcondition `old x = old x`. Of course, the equality predicate is reflexive (see the rule P-EQUAL in section 4.3.3), so evaluating (or worse, copying) `x` in this context is not necessary, but will not be avoided by $T_{opt}$. One could imagine combining other methods (*e. g.* symbolic execution, abstract interpretation, or deductive verification) to tweak the transformations further, but these experiments have not been conducted in ORTAC yet.

In the context of GOSPEL, the predicates have the expressiveness of the first-order logic, which is known to be undecidable [12, 43]. Therefore, there might be no way of deciding the optimal copies when considering a predicate of the form `if P then old x else old y`. Instead, ORTAC has to copy both `x` and `y`, while only one of those might be sufficient every time (*e. g.* if $P$ always holds).

## 5.4 CONCRETE IMPLEMENTATION IN ORTAC: EXTENSIONS AND LIMITATIONS

ORTAC applies the principles presented *via* the transformations of this chapter to make GOSPEL specifications that contain `old` executable.

However, Gospel's logic is richer, so the actual implementation in Ortac requires to extend these transformations, and sometimes faces limitations.

### 5.4.1 *Prestate Captures Applies to Predicates Too*

In Microspel, `old` is a primitive that applies to a term and gives a term back. In Gospel however, `old` can also apply to predicates. Therefore, we need to establish new transformation rules so that *old_up* can 'move' `old` up in the predicates too.

> **Example 21.** The case of quantifiers is interesting. Consider for instance the following postcondition predicate:
>
> ```
> forall i, 0 ⩽ i < n -> (old a)[0][i] = 0.
> ```
>
> In Microspel, this predicate is transformed by $T_{opt}$ into
>
> ```
> forall i, 0 ⩽ i < n -> (old (a[0]))[i] = 0,
> ```
>
> which later only requires copying `a[0]` rater than the full array `a`. In Gospel, we can go one step further and propose
>
> ```
> old (forall i, 0 ⩽ i < n -> a[0][i] = 0).
> ```
>
> In that case, the predicate under `old` evaluates to a single Boolean in Ortac, and no copies at all are necessary.

*See chapter 6 for more details about the interpretation of predicates by Ortac.*

In Ortac, *old_up* is slightly different. Let us consider a modified transformation *old_up′* that acts like *old_up* except it adds `old` around quantified variables. The transformation for quantifiers is then as follows:

$$
\texttt{forall i, } t_1 \texttt{ ⩽ i < } t_2 \texttt{ -> } P \;\mapsto\;
\begin{cases}
\texttt{old (forall i, } t_1' \texttt{ ⩽ i < } t_2' \texttt{ -> } P'\texttt{)} \\
\quad \text{when } \mathit{old\_up}'(t_1) = \texttt{old } t_1' \\
\qquad \text{and } \mathit{old\_up}'(t_2) = \texttt{old } t_2'. \\
\qquad \text{and } \mathit{old\_up}'(P) = \texttt{old } P'. \\
\texttt{forall i, } \mathit{old\_up}(t_1) \texttt{ ⩽ i < } \mathit{old\_up}(t_2) \texttt{ -> } \mathit{old\_up}(P) \\
\quad \text{otherwise.}
\end{cases}
$$

In other words, we apply `old` to a quantified predicate if and only if all its sub-terms and sub-predicates except for variables introduced by quantifiers can be put under `old`.

> **Example 22.** If we consider the predicate
>
> ```
> forall i, 0 ⩽ i < old (length m) -> old m[i] = 0,
> ```

> the function *old_up* does not let us go any further, because of the variable `i`. However, when applying the rule above, we get the following predicate:
>
> ```
> old (forall i, 0 ⩽ i < length m -> m[i] == 0),
> ```
>
> which we can compute in the prestate and get a single Boolean to save, rather than the full array `m`.

### 5.4.2 *Some Terms May Require Allocating New Memory*

In MICROSPEL, the terms constructs do not need to allocate memory, and their result is always *smaller* in memory than their arguments. In other words, they are memory projections. When considering GOSPEL's terms however, this is not always the case, because they feature calls to arbitrary functions. In the general case, calls to functions that are declared `pure` can appear in specifications. A natural extension of *old_up* is to transform function calls the following way:

*There are conditions for a function to be usable in a specification. See chapter 2.*

$$\texttt{f (old x)} \quad \mapsto \quad \texttt{old (f x)},$$

which is indeed correct if `f` is pure. These functions, provided by the user and written in OCAML, can allocate memory during their execution, and return values that are larger than their arguments. In that case, transforming `f (old x)` into `old (f x)` in *old_up* actually *increases* the cost of the copies.

Correctly deciding whether to apply this rule or not is again undecidable, because `f` can contain arbitrary code. In ORTAC, we default to always moving `old` upwards, as we observed most of the functions used in specifications are actually projections, or conversions (*i. e.* from one container to another, with equivalent memory complexity).

However, one could imagine a static allocation analysis to decide whether to apply it or not depending on the context, in particular if space complexity specification makes it way into Gospel in the future. We also experimented with a heuristic based on types to try and improve the decision making. For instance, one could approximate that a `string` is smaller than a `string array`. However, most of these approximations are over-simplifications that are easy to contradict, and their use did not show significant improvements in practice.

### 5.4.3 *Branching Leads to Worse CPU Time*

In some cases, some of the branches of the control flow of the specification do not need to be executed, for instance when using a conditional branching. In such situations, applying *old_up* can induce additional CPU cost, even if it preserves the memory optimization property. Let us explore this trade-off through an example.

**Example 23.** Let us consider a predicate of the following form:

```
if P then sorted (old x) else sorted (old y),
```

where `sorted` returns a sorted version of its argument.
In our previous discussion about the optimality of $T_{opt}$, we established that in the general case, we *have to* copy both `x` and `y`.
However, when applying *old_up* to this predicate, we obtain

```
if P then old (sorted x) else old (sorted y),
```

which leads to $(a)$ sorting the original arrays `x` and `y`; $(b)$ copying both the resulting arrays.
Therefore, applying *old_up* not only makes us copy two arrays, but also execute potentially costly operations that is ultimately not necessary for the check: we sort two arrays instead of one.

*This issue arises even without introducing function calls.*

Scenarii like this one show a weakness of this approach when faced to code containing branching, as correctly checking those actually require to execute *all* the branches, even those that will not, in the end, be useful to the check. In Ortac, we resolve this time-memory trade-off by choosing to always apply *old_up*, as our experiments show the typical use-cases benefit from it. We do, however, provide an option to deactivate the optimization if the user knows they might be facing tricky cases.

## 5.5 EXAMPLE AND BENCHMARKS

In this section, we show how Ortac applies these specification transformations to an existing program implemented in OCaml and annotated with Gospel specifications, making them executable and verifiable at runtime. We compare the optimization levels to demonstrate that a simple program and its specification are sufficient for this optimization to be critical to the performance and practicability of the instrumentation in production.

### 5.5.1 *A Maze Generator*

Our stress test program takes an integer $n$ as input and generates a perfect, random maze on a $n^2$ square grid. A maze is perfect when for any pair of points $A$ and $B$, there exists a single path between $A$ and $B$. The algorithm is as follows:

$(a)$ Create a list $\mathcal{W}$ of all the possible walls in the grid.

$(b)$ Create a set of sets of cells $C$ containing the singletons $\{c\}$ for each cell $c$.

$(c)$ For each wall in $\mathcal{W}$, in some random order,

- if the cells that this wall separates belong to distinct sets,
  a) remove the wall from the list $\mathcal{W}$;
  b) join the sets of the formerly separated cells.

The set $\mathcal{C}$ maintain the connected components of the grid, so during each iteration, we remove a wall from $\mathcal{W}$ if and only if it joins otherwise disconnected components. The loop goes through all the walls, so there is only one connected component at the end of the iterations. Therefore, the remaining walls in the list constitute a perfect maze.

A natural data structure implementation for $\mathcal{C}$ is union-find [1]. Our OCaml implementation of the union-find exposes the interface reproduced in listing 5.1, which we instrument using Gospel contracts that we aim to verify at runtime.

*The specifications in these contracts is partial, but sufficient for our example.*

```
type t
(*@ ephemeral *)

val create : int -> t
(*@ uf = create n
    checks n >= 0 *)

val num_classes : t -> int
(*@ pure *)

val find : t -> int -> int
(*@ pure *)

val union : t -> int -> int -> unit
(*@ union uf i j
    modifies uf
    requires 0 <= i < size uf
    requires 0 <= j < size uf
    ensures num_classes uf <= num_classes (old uf)
    ensures find (old uf) i <> find (old uf) j
            -> num_classes uf = num_classes (old uf) - 1
*)
```

Listing 5.1: Union-find module interface.

The type `t` represents an instance of the data structure: a set of sets of cells that we represent with integer identifiers. Our module will be operating in place, so this type is mutable, which is reflected by the Gospel clause `ephemeral`.

The function `num_classes` returns the number of disjoint sets in the data structure and `find` returns the representative element of a set. These functions do not perform any writing effects (*i. e.* they do not modify the union-find structure), do not raise exceptions, and always

*The function `find` may actually perform path compression, but as far as the specification is concerned, this is not observable.*

terminate. Therefore, they are considered `pure` by GOSPEL, and we can use them to specify other functions further.

Finally, the function `union` performs the union of two sets in the structure. We will focus on this function in the rest of this example. Its contract states it can modify the data structure with the `modifies` clause. Because the type of union-find is mutable, and this function potentially modifies it, executing properties that refer to the old structure version will require copies. Therefore, the transformations we proposed in section 5.2 are relevant in this example.

### 5.5.2 *Runtime Verification with ORTAC*

We use ORTAC to generate OCAML code that checks these contracts at runtime. More precisely, the generated implementation performs the following operations:

(*a*) Check the preconditions and fail if they do not hold or raise an exception.

(*b*) Evaluate the terms under `old` operators, and copy their values into fresh variables.

(*c*) Call the function `union` and fail if it raises an exception.

(*d*) Replace the terms precomputed in step 2 with their value in the postconditions, execute them, and then fail if they do not hold or raise an exception.

#### 5.5.2.1 *Direct instrumentation*

In this example, the transformations $old_{down}$ and *sanitize* do not modify the specification, as there are no nested `old` or quantifiers.

In the unoptimized instrumentation, ORTAC collects all terms appearing under `old`, evaluates them in the prestate, and then copies the result. There are four occurrences of the `old` operator, all of which refer to the old version of `uf`. The generated code is showed in listing 5.2.

#### 5.5.2.2 *Optimized Version*

In the optimized version, although the user can still write the specifications in the way that feels the most natural to them, ORTAC preprocesses the terms to propagate the `old` operator upwards, as explained in section 5.3. ORTAC automatically rewrites the terms as if the user had written the following postconditions:

```
ensures num_classes uf ⩽ old (num_classes uf)
ensures old (find uf i <> find uf j)
        -> num_classes uf = old (num_classes uf - 1)
```

```
let union uf i j =
  if not (0 ⩽ i ⩽ size uf) then fail ();
  if not (0 ⩽ j ⩽ size uf) then fail ();
  let old_1, old_2, old_3, old_4 =
    copy (uf, uf, uf, uf)
  in
  (try union uf i j with _ -> fail ());
  if not (num_classes uf ⩽ num_classes old_1)
  then fail ();
  if not (
      not (find old_2 i <> find old_3 j)
      || num_classes uf = num_classes old_4 - 1)
  then fail ();
```

Listing 5.2: Naive instrumentation of `union` ($T_{base}$).

```
let union uf i j =
  if not (0 ⩽ i ⩽ size uf) then fail ();
  if not (0 ⩽ j ⩽ size uf) then fail ();
  let old_1, old_2, old_3 = copy (
    num_classes uf,
    find uf i <> find uf j,
    num_classes uf - 1)
  in
  (try union uf i j with _ -> fail ());
  if not (num_classes uf ⩽ old_1) then fail ();
  if not (not old_2 || num_classes uf = old_3)
  then fail ();
```

Listing 5.3: Instrumentation of `union` with optimized copies ($T_{opt}$).

This rewriting effectively moves to the prestate some computations previously executed in the poststate. Therefore, it only triggers a copy of the result of the computations (two integers and one Boolean in this case) instead of the context necessary for the execution (here, the whole union-find structure). The instrumentation generated by Ortac now has the form showed in listing 5.3.

We do not show the details of the wrapper code at this time, but more details are provided about the generated code in chapter 6, in particular about the `fail` calls, handling of exceptions, etc.

### 5.5.3 *Benchmarks*

We run our maze generator with multiple values of $n$. For each value, we gather the execution time, the number of garbage collections, and the cumulative amount of data copied by `copy`. We ran our benchmarks

on an i7-1165G7 @ 2.80GHz CPU, with 16GB of RAM using the OCaml 4.14.0 compiler. Each data point is an average of 10 runs. We present the results in figure 5.1.

| $n$ | **Instrument.** | **Time (s)** | **GC runs** | **Copies (MB)** |
|---|---|---|---|---|
| 100 | None | 0.0026 | 0 | - |
| | $T_{base}$ | 2.0 | 260 | 190 |
| | $T_{opt}$ | 0.0062 | 0 | 0.038 |
| 200 | None | 0.012 | 0 | - |
| | $T_{base}$ | 30 | 4400 | 3000 |
| | $T_{opt}$ | 0.032 | 2 | 0.15 |
| 400 | None | 0.088 | 1 | - |
| | $T_{base}$ | 680 | 31 000 | 48 000 |
| | $T_{opt}$ | 0.19 | 2 | 0.61 |
| 800 | None | 0.46 | 4 | - |
| | $T_{base}$ | $\infty$ | $\infty$ | $\infty$ |
| | $T_{opt}$ | 0.89 | 4 | 2.4 |
| 1600 | None | 2.2 | 5 | - |
| | $T_{base}$ | $\infty$ | $\infty$ | $\infty$ |
| | $T_{opt}$ | 3.9 | 5 | 9.8 |
| 3200 | None | 11 | 5 | - |
| | $T_{base}$ | $\infty$ | $\infty$ | $\infty$ |
| | $T_{opt}$ | 19 | 6 | 39 |

Figure 5.1: Benchmarks results for the `old` instrumentation

They show that naive instrumentations of the code make it impracticable for large values of $n$, which timed out after one hour of execution. On the other hand, the optimized version significantly reduces the cost of the verifications to a constant factor no larger than 2. The limited amount of data copied and limited use of the GC allow this cost mitigation.

ABOUT COMPLEXITY. Recall that the maze generation calls `union` until there is only one remaining set (*i. e.* exactly $n^2 - 1$ times) so its complexity, when invoked with size $n$, is $O(n^2 \times uf(n))$, where $uf(n)$ is the complexity of `union`. When one properly implements union-find, we get $uf(n) = O(\alpha(n)) \approx O(1)$, so the complexity of the maze generation is $O(n^2)$ in the un-instrumented version.

However, when copying the entire union-find structure (no optimization and shared copies only), the instrumented `union` now needs to copy a structure of size $n^2$. This copy increases $uf(n)$ to $O(n^2)$, which in turn makes the total maze generation complexity $O(n^4)$. Conversely, the `old` propagation optimization does not require copying this much data. Instead, it copies a fixed amount at each call (two integers and one Boolean), and restores the original quadratic complexity of the program.

### RELATED WORK

The efficient evaluation of `old` terms in runtime assertion checking is a well-known and challenging problem for which there is still room for improvement. In the general case, most tools copy the whole memory state before the call to the function [23, 35], while acknowledging the flaws of this approach.

ACSL [6] generalizes the `old` feature by introducing an `\at(t, L)` operator, which lets the user specify arbitrary locations `L` in the code, rather than restricting it to the function prestate. This extension leads to possibly worse performance issues, with even more states needing captures. While initial implementations of E-ACSL [42] used to perform a shallow copy of the variable contents only, which is incorrect in most cases, more recent implementations provide a hybrid method to reduce the copied memory space [40], but this approach has not been detailed yet.

It is also worth mentioning that, as noted in [9], in the presence of preconditions, the evaluation and copy of the `old` terms are meant to be guarded by these preconditions. Accordingly, Ortac only evaluates those once the corresponding preconditions are successfully verified.

Previous work have also explored other optimizations for runtime assertion checkers, such as providing efficient representation of integers [24] or improving the verification of `modifies` clauses [26]. Regarding the former, Ortac benefits from `zarith`, which only switches to arbitrary-precision integers when machine integers are not large enough to store them without overflows. Regarding the latter, Ortac always assumes that the variables provided by the user in the `modifies` clauses are correct and even uses them to optimize the copies. They are in the trusted part of the specification, and verifying these clauses is still future work for Ortac at this point.

### CONCLUSION

In this chapter, we have presented the transformations performed by Ortac to (*a*) be able to execute specifications containing `old` primitives; (*b*) mitigate the cost of copying prestate values during postconditions verifications. We supported this approach with a Coq formalization

and proofs of correctness and showed its efficiency in a practical example.

We presented the transformations in this chapter using MICROSPEL, which specifications are significantly simplified compared to GOSPEL's, to make the presentation and proofs amenable and allow the techniques to generalize in other specification languages. ORTAC, however, goes beyond MICROSPEL and operates on GOSPEL specifications directly. For instance, it moves `old` upwards in predicates, including quantifiers, local variables, and user-defined predicates and functions.

In the next chapter, we discuss other interesting points of translating GOSPEL terms and specifications. In particular, we did not explain here *how* ORTAC implements the `copy` function. We show in the next chapter how it relies on the OCAML type-checker to take into account memory sharing and optimizes for immutable data in order to limit even further the cost of copying.

# 6

# ORTAC: JOINTING OCAML AND GOSPEL

So far, we did not dive into the implementation of Ortac and the design and implementation choices that let us implement specification verifications that respect Gospel's semantics. This chapter shows some challenges when instrumenting the code and how Ortac handles them.

## 6.1 TYPE-GUIDED CODE GENERATION

Ortac leverages the information provided by the typing information—of OCaml in the interface and of Gospel in the specifications—to produce more accurate, or more optimized code.

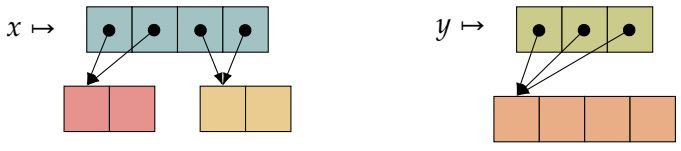
### 6.1.1 *Copies*

In the previous chapter, we explained *what* and *when* Ortac needs to copy in order to execute predicates that contain the `old` primitive. We did not, however, explore *how* this copy is implemented to provide the correct deep-copy semantics at a limited cost.

Let us consider an example where we need to copy an array $x$ of pairs of integers and an array $y$ of arrays of integers.

```
val x : (int * int) array
val y : int array array
```

The array $x$ has size 4 and contains two aliased pairs, respectively, its first two cells and its last two cells. The array $y$ has size 3, and all its cells point to the same array. The memory they use is represented below.




#### 6.1.1.1 *Copying while Preserving Sharing*

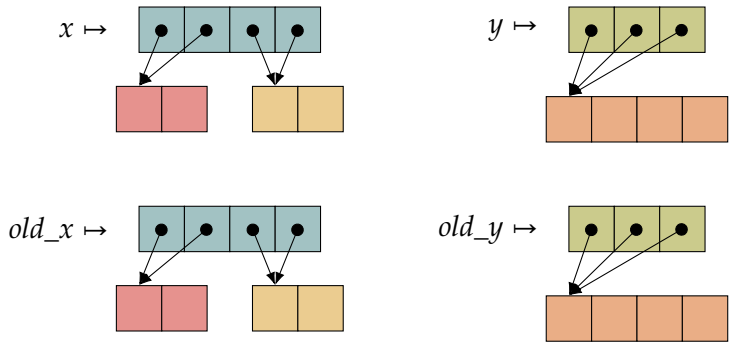
The first feature of `copy` we are interested in is the sharing preservation: shared memory in the original value should be shared in the copy

*Memory sharing is a commonly used feature of OCaml in particular when writing functional data structures.*

for better compacity. A straightforward way of implementing such a copy is to use the `Marshal` module from the OCaml standard library, which implements marshalling. Copying is done by serializing and then immediately deserializing the value:

```
let copy : 'a -> 'a = fun a ->
  Marshal.from_string (Marshal.to_string a [Closures]) 0
```

This `copy` function is polymorphic, preserves sharing (per `Marshal` specification), and works correctly for cyclic values as well —the `Marshal` functions are implemented using `C` code that traverses the memory representation of values.




This approach presents two weaknesses: (*a*) it needs to allocate a string to perform the serialization, which is immediately consumed by the deserialization (although these strings are usually small, they put unnecessary pressure on the GC); (*b*) it copies all memory blocks, regardless of whether they represent mutable or immutable values. The first point is merely an implementation detail of the `Marshal` module. The second point, however, is a legitimate weakness of this method. Indeed, OCaml does not keep the type information at runtime, and therefore exploring the memory does not give any information on the mutability of the data being traversed.

*Rather, it keeps very limited typing information via tags.*

We, however, do have some static information on the mutability of data *via* via the Gospel type-checker and the `modifies` clauses of the specifications. We wish to leverage this information in order to reduce the copied size.

#### 6.1.1.2 *Sharing Immutable Values with the Prestate*

Rather than using `Marshal`, we can use the type of the expression to copy in order to generate a monomorphic copy function by recursively composing primitives. When we encounter an immutable value (or a value that does not appear in the `modifies clause`), `copy` is the identity function.

*Remark* 15. More than inspecting the root type of a value is required to determine immutability. Values of type `t list` are only immutable if `t` is also an immutable type.

For instance, when copying `x : (int * int) array`, we generate:

- a function to copy `(int * int) array` values, given a function to copy `(int * int)` values:

```
let copy_int_pair_array a : int array =
  Array.map copy_int_pair a
```

- a function to copy `(int * int)` values (which are immutables):

```
let copy_int_pair = identity
```

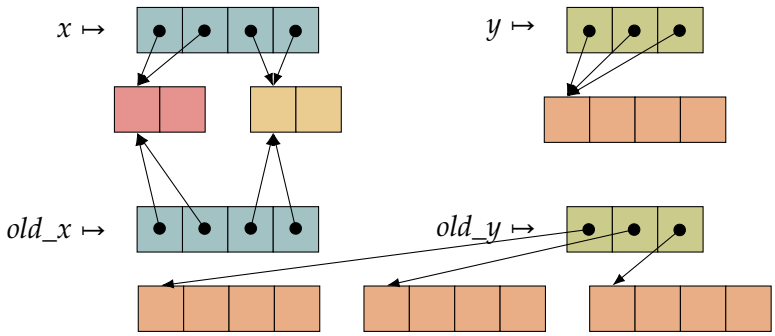
We proceed similarly to generate a function `copy_int_array_array` to copy $y$.

```
let copy_int = identity

let copy_int_array a : int array =
  Array.map copy_int a

(* Final copy function for y *)
let copy_int_array_array a : (int array) array =
  Array.map copy_int_array a
```

After the copy, we obtain the following memory:




Note how *old_x* now has pointers to $x$, rather than duplicating immutable data.

There is, however, a significant drawback: this copying method only preserves sharing between immutable data but not between imperative values. While this is luckily the most common scenario for sharing in OCaml, there are still many examples of sharing between (partially) imperative data, for instance, when one implements polymorphic containers with sharing but instantiates them with mutable types. Therefore, this solution is only partially satisfying.

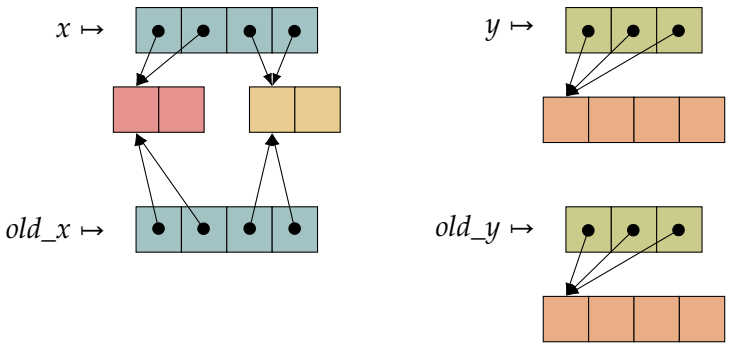
#### 6.1.1.3 *The Best of Both Worlds*

The actual implementation in Ortac attempts to achieve the best of both worlds. It performs similarly to the implementation in `Marshal`: by memoizing the result of the copies using OCaml's physical equality:

```
let copy_int_array =
  let module H = Hashtbl.Make(struct
    type t = int array
    let equal = (==)
    let hash = Hashtbl.hash
  end) in
  let tbl = H.create 0 in
  fun a ->
    try H.find tbl a
    with Not_found ->
      let copy_a = Array.map copy_int a in
      H.add tbl a copy_a;
      copy_a
```

*Remark* 16. The contents of the memoization table must be cleared before each new specification verification so previous copies are discarded.

By implementing this memoization on top of the monomorphic `copy` generation presented previously, we can achieve the most compact deep copy:




It does share a limitation with the previous solutions: this is only possible when the mutable parts of the value to copy are fully known; *e. g.* when it does not involve polymorphic or abstract parts. When it does, generating a `copy` function by following the type definition is no longer possible, and Ortac falls back to the `Marshal` module to copy the whole term, as presented in section 6.1.1.1.

*Remark* 17. One could think of a hybrid solution: use a type-guided approach for as long as the type is known, then fall back to `Marshal` to copy only the abstract or polymorphic parts. In that case, though, the result is neither guaranteed to preserve sharing nor to save copying of immutable data.

### 6.1.2 *Equality Functions*

On the one hand, Gospel provides a polymorphic equality operator `(=)` which semantics corresponds to the logic equality and depends on

the type of the compared values. For instance, two sets provided by the GOSPEL standard library are equal if and only if they contain elements that are equal with respect to their logic equality. We have used this predicate extensively in the examples of chapter 2, and it is a central point of most specifications.

On the other hand, OCAML also provides a polymorphic equality operator (`=`). Its semantics, however, is defined in different terms. It is a structural equality, meaning that it recursively compares the contents of the values, including when they are mutable. This definition needs to be revised to fully understand the function's behaviour: it is worth detailing *what* structure is compared, *i.e.* the structure of the memory representation of the values. For instance, two sets from the OCAML standard library are equal if and only if they are represented by equal—structurally, not physically—equal binary tree internally.

Although their semantics are defined differently, they coincide for many basic types and type compositions, *e.g.* machine integers, lists, arrays, records, and variants. Therefore, it is tempting to implement GOSPEL's polymorphic equality using OCAML's polymorphic equality, which would also have the advantage of being predictable for OCAML developers. This is not, however, what ORTAC does.

WHY AVOID OCAML'S POLYMORPHIC EQUALITY? There are several reasons why ORTAC avoids implementing GOSPEL's logic equality with OCAML's polymorphic equality provided in the standard library.

- When applied to functional values, it fails at runtime with an `Invalid_argument` exception. While the generated code would still be valid (ORTAC reports that it cannot translate the property), this case can be detected and reported at the instrumentation time rather than on each run.

- When applied to cyclic values, it may not terminate (mainly if the values *are* equal) or raise an exception (*e.g.* if it overflows the stack), depending on the structure of the arguments. When it does not terminate, the process also cannot be stopped with a `SIGTERM` signal because the program running is C code from the OCAML runtime.

- On a conceptual level, the polymorphic equality breaks the module abstraction barrier (and even the OCAML memory representation abstraction!). GOSPEL and ORTAC are higher-level languages that tend to avoid this.

- Finally, as we already mentioned, OCAML's equality implements a different relation than GOSPEL's logic equality. It is agnostic of the type of the compared values and compares their memory representations. It is specific to the implementation users choose for their types and can even be affected by compiler optimizations!

*This is a frequent source of bugs in OCAML and using the polymorphic equality generally be used with caution.*

Therefore, when translating GOSPEL's polymorphic equality, ORTAC implements ad-hoc polymorphism, generating the corresponding monomorphic functions. It substitutes them in the instrumentation, similar to the one described for copies (see section 6.1.1). Primitive types are known to ORTAC, along with their equality predicates. It composes the equality functions isomorphically to the structure of the type, just like `copy`. When a type is unknown or does not have an attached equality predicate (*e.g.* type variables in polymorphic types, abstract types, functional types), the equality predicate is deemed non-executable and is rejected.

Note that this solution does not prevent equality tests of cyclical values from not terminating. However, it does mitigate the issue in two ways: (*a*) before comparing values structurally with their types, it compares them physically, so cyclic values that are physically equal (or have physically equal parts) are properly compared; (*b*) since the generated implementation is pure OCAML code and does not contain C bindings, the program still properly responds to `SIGTERM` signals by exiting.

*The `compare` function from the standard library also implements this trick based on physical comparison.*

### 6.1.3 *Printing, Hashing, Comparing*

The functions `copy` and `equal` are necessary to implement the specifications that involve GOSPEL's `(=)` and `old`. In some cases, it is also helpful to generate monomorphic `print`, `hash` or `compare` functions. Unlike copies or equality, these functions are optional to the instrumentation, but they are helpful to generate more performant code or provide more useful feedback to users. The generation process is the same: it is guided by the types and may fail statically (*e.g.* at the generation time) and issue a warning if it reaches polymorphic or abstract types.

COMPARING AND HASHING. Comparing and hashing are performance helpers that ORTAC uses when it generates verifications. For instance, ORTAC needs them to implement efficient sets when translating GOSPEL's standard library's sets or to generate memoized recursive functions as we showed in chapter 3. The polymorphic `compare` and `hash` functions provided by the standard library are sometimes enough for these use cases as they are less restrictive. For instance, the `hash` function is guaranteed to terminate. However, even when we apply them to types that make their execution safe, they might not be compatible with the equality functions for these types. For instance, `compare nan nan` evaluates to `0` and `Hashtbl.hash nan = Hashtbl.hash nan` evaluates to `true`, but `nan = nan` evaluates to `false`, in accordance with the IEEE 754 standard for floating point arithmetics [22]. Therefore, using them in the generated code can introduce inconsistencies with respect to the equality predicate and erroneous verifications.

PRINTERS. Finally, ORTAC also generates printers for the values involved in the specifications so that the error messages presented in the error messages (see chapter 3) contain the actual arguments.

```
$ ./fib 4242
File "fibonacci.mli", lines 8-13, characters 0-30:
 Runtime error when executing fib 4242:
 - the precondition
     fibonacci 4242 ⩽ Int.max_int
   did not hold.
```

This makes it easier for users to understand them and to replay failed test cases. When printers cannot be inferred, the error messages only show the variable names given in the specifications:

```
$ ./fib 4242
File "fibonacci.mli", lines 8-13, characters 0-30:
 Runtime error when executing fib n:
 - the precondition
     fibonacci n ⩽ Int.max_int
   did not hold.
```

### 6.1.4 *Limitations and Fallbacks*

This approach of type-based generation applies well to fully known types. However, it is not applicable in all cases, and three *scenarii* prevent it from succeeding: abstract types, polymorphic types, and functional types. If the type of a term is abstract, polymorphic, or functional (or contains parts that are), it is no longer possible to generate monomorphic functions based on the type definition. ORTAC provides two ways to overcome this and execute the specifications.

USER-PROVIDED FUNCTIONS. In many OCAML interfaces exposing a type, developers also expose an `equal` function that implements meaningful equality (as opposed to the structural equality of memory representations). In those cases, users may say so using the `equality` clause in the function specification. GOSPEL checks that it indeed has the type of an equality function (possibly parametrized by higher order equality functions when the type is polymorphic), and ORTAC uses it as the equality function for this type in its monomorphic function generation.

While exposing an `equal` in an interface is idiomatic, exposing a `hash` function is pretty rare. When optional functions (*e. g.* `print`, `compare`, or `hash`) are missing, ORTAC can also give up on the corresponding optimization and fallback to a less efficient (or comfortable) implementation. For instance, when `hash` is missing but `compare` is present, the memoization is implemented with a reference pointing to a finite map from OCAML's standard library; when only `equal` is available, GOSPEL's sets are implemented with lists. When ORTAC falls back to

inefficient implementations, it reports it to the user as a warning during the instrumentation phase.

POLYMORPHIC FUNCTIONS. When the user does not provide those functions, the only fallback is to use the standard library polymorphic functions. ORTAC does not do it by default for all the reasons discussed earlier, but it provides an option to allow this behaviour.

MIXING APPROACHES. All these options are mutually exclusive: ORTAC ensures that all the functions used in its generated code come from the same source (type, user, or standard library) to reduce the risk of incompatible functions. Note that ORTAC does not check that all these functions are compatible: when the user provides functions, it is their responsibility to ensure this property.

*Verifying at runtime that these functions are compatible is possible but was not implemented.*

**Example 24** (User-provided functions and generated functions may be incompatible)**.** For instance, consider the following example of a module implementing rational numbers using a non-reduced couple of integers and exposing an `equal` function that judges whether two rational numbers are equal:

```
type rat = int * int

val equal: rat -> rat -> bool
(*@ equality *)
```

In this module, the same number (in the sense of `equal`) can be represented with multiple different OCAML values. For instance, `(1, 2)` and `(3, 6)` both represent $\frac{1}{2}$. However, the type-generated (or the polymorphic) `compare` will judge whether these values differ.
If one now wishes to implement sets of rational numbers using OCAML's sets built with such a `compare` function, they cannot retrieve values as expected. The following code fails the assertion check:

```
module Set = Set.Make (struct
  type t = rat
  let compare = Stdlib.compare
end)

let () =
  let s = Set.singleton (1, 2) in
  assert (Set.mem (3, 6) s)
```

## 6.2 TYPE INVARIANTS

In GOSPEL, the specifier can attach *invariants* to types. They describe properties that hold (*a*) for all values of this type; (*b*) at all times outside of the abstraction barrier. In particular, values stored inside containers (*e. g.* lists, arrays, or records) must also obey their invariants. However, it is possible for internal functions to temporarily break these invariants, so long as they are restored before the function returns. This applies to function arguments but also global values and internal states.

For instance, the specification of the type of union-find presented in section 2.3 could mention that there are always fewer classes in the structure than its full size, and these are both positive integers:

```
type t
(*@ ephemeral
    with self
      invariant 0 <= num_classes self <= size self *)
```

The first line, `with self`, introduces the name of the value of the type `t`, so we can use it in the invariant itself in the second line.

*Remark* 18. At first sight, this definition looks suspiciously cyclic: the type `t` is defined in terms of `num_classes` and `size`, which are defined over the type `t`. However, one can always rewrite it using a GOSPEL axiom instead of an invariant, which would detach the property from the type definition and break the cyclicity:

```
type t
(*@ ephemeral *)

(*@ axiom t_inv : forall (self: t).
      0 <= num_classes self <= size self *)
```

### 6.2.1 *How Invariants are Checked*

The first, easy step to check these invariants is to translate them into OCAML expressions and generate a checking function for them, which we may call whenever we want them to hold. Again, this step may statically fail if the invariant is not deemed executable, in which case the invariant is ignored, and a warning is issued by ORTAC.

However, being able to verify the invariants on *one* value of the type is not sufficient. Indeed, the values are not always immediately accessible for verification, for instance, if they are stored inside containers or are part of a larger value. For instance, one may write a function returning a list of values of type `t`:

```
val f : unit -> t list
```

In that case, all the values in the returned list must satisfy the invariant. In general, values of type `t` can be nested into arbitrary types, which

can be part of the GOSPEL standard library (if the function is a logic function), the OCAML standard library, or written by the user.

In order to perform the correct verifications, we, therefore, have to (*a*) determine for each input and output whether they carry values of types that have invariants attached; (*b*) in that case, generate iterators to access these values. The iterator generation is performed the same way as `copy` (see section 6.1.1): it is guided by the type definition and composes the `iter` functions available for each type. For instance, for the function `f` shown before, ORTAC generates the following iterator:

```
let iter_t_list (f: t -> unit) (x: t list) =
  List.iter f x
```

The generator works similarly as long as the container type is well-known. Here is another example of values of type `((t * 'a) list) Set.t` appear in the module. Note that the type need not be fully known (this one is polymorphic); only the container (and its associated `iter` function) is required.

```
let iter_t_a_pair (f: t -> unit) (x: t * 'a) =
  f (fst x)

let iter_t_a_pair_list (f: t -> unit) (x: (t * 'a) list) =
  Set.iter (iter_t_a_pair f) x

let iter_t_a_pair_list_set (f: t -> unit)
                           (x: ((t * 'a) list) Set.t) =
  Set.iter (iter_t_a_pair_list f) x
```

In both examples, the containers are either part of the language (pairs), defined in the OCAML standard library (lists) or in the GOSPEL standard library (sets). All of those are known to ORTAC; therefore, the iterators generation is not a problem. Sometimes, however, the container is part of the user codebase and is not exposed:

```
type 'a container
val g : unit -> t container
```

*Recall that runtime values do not hold type information.*

In that case, there is no fallback polymorphic solution like for `copy` or `equal`. The user may expose an `iter` function, annotated with an `iterator` specification (this is only used by ORTAC and is not part of the GOSPEL language):

```
val iter : ('a -> unit) -> 'a container -> unit
(*@ iterator *)
```

If the user does not provide such a function, the invariant cannot be checked on the return values of `g` (it may still be checkable in other functions), and a warning is issued.

Now that we have a function generated by ORTAC that checks the invariants of a value, we have to decide *when* to call the checking function to enforce the invariants.

### 6.2.2 *When Invariants are Checked*

The most straightforward solution seems to follow the definition: we can check the invariants on all accessible values of that type at every function's entrance and exit (normal and exceptional). We can achieve this by maintaining a list of weak pointers (*i. e.* pointers that the GC does not follow and may collect as soon as the pointee is collected) of previously seen values at the interface of the functions in the module. However, this would be very costly and is often unnecessary.

*Remark* 19. When a function does not have a contract attached, it still must maintain all the type invariants. Therefore, ORTAC produces wrappers even for functions that are not specified.

ABSTRACT TYPES. Abstract types can only be modified by the module's functions where they are defined. Therefore, checking the invariants at the function *exits* only is sufficient to ensure they hold everywhere.

IMMUTABLE VALUES. In the case of immutable values, checking the invariant once is enough. More generally, values that are 'locally immutable', *i. e.* values that do not appear in the `modifies` clause also do not need to be checked since these clauses are in the trust base of the specification (see chapter 3).

*One could also maintain a set of already checked values to ensure that the invariants are checked exactly* once.

THE CASE OF PUBLIC TYPES. In the case of public types (or types that contain publicly modifiable parts), checking the invariants at every function entry is also necessary since they might have been modified from outside the module.

CONSUMED VALUES. GOSPEL features a `consumes` clause in function contracts that lets users specify that a value must not be used anymore after the call. For instance, when specifying an interface for file descriptors (or OCAML channels) manipulation, the `close` operation *consumes* its argument, which is now invalid for further use.

```
val close_out : out_channel -> unit
(*@ close_out ch
    consumes ch *)
```

ORTAC instruments these clauses by keeping track of all consumed values with a set of weak pointers to these values. This set typically remains small since pointers to consumed values are usually not kept alive for long (when the code is correct); the impact on performance should be minimal. Before each function call, it checks if the arguments were previously consumed and reports it to the user. When it comes to type invariants, they are not checked on consumed values.

## 6.3 EXCEPTIONS

This section shows how ORTAC instruments the code to consider exceptions, whether raised by the instrumented implementation or by the generated code itself.

### 6.3.1 *Exceptions Raised when Executing Specifications*

In chapter 3, we showed that ORTAC sometimes statically fails when it determines that it cannot translate a term into an executable OCaml expression, *e.g.* when it faces a logic function with no definition. In that case, it shows a warning to the user and does not translate that part of the specification. However, there are situations where a term *seems* executable, but its execution raises an exception instead.

#### 6.3.1.1 *How Executing Pure Specifications Can Raise Exceptions*

In GOSPEL, the logic domain is pure. Therefore it may seem unexpected that executing a specification can raise an exception at all. However, the OCAML instrumentation by ORTAC creates potential sources of exceptional behaviour.

EXCEPTIONS LINKED TO THE ENVIRONMENT. First, no matter how pure the generated OCAML expression is, it is always possible that its OCaml execution raises an exception due to the execution environment. Indeed, memory allocations may result in `Out_of_memory`, recursive functions calls can raise `Stack_overflow`, or a user interrupt *via* `CTRL-c` will raise `Sys.Break`. When these asynchronous exceptions are raised, we can deduce nothing from the execution of the specification, and the only reasonable behaviour is to report it back to the user.

GOSPEL'S TOTAL LOGIC AND GOSPEL STANDARD LIBRARY. Second, some functions from the GOSPEL standard library are pure because GOSPEL's logic is total, but their actual implementation in ORTAC is not. Let us consider the function `Sequence.get` from the standard library, which returns the *i*th element of a sequence.

```
(*@ val get : 'a t -> int -> 'a *)
```

If *i* is outside the bounds of the sequence, the function cannot raise an exception since it is a logic function. Instead, it always returns an arbitrary value of the correct type with only one piece of information available: it is equal to itself. Moreover, since logic functions provide referential transparency, it is equal to all other calls to `Sequence.get` with the same arguments. Implementing such a mechanism for total functions is generally not possible for multiple reasons:

- If the return type is polymorphic, then the type-checker does not allow crafting an arbitrary value of that type.

*It is possible by tricking the type-checker with* `Obj.magic` *or the like, but we do not want to go down that path.*

- When crafting a value is possible (*e. g.* because the type is known), picking a default one is still unsound. Indeed, a single default value does not allow us to enforce that it equal to itself and other same calls, yet different from default values created by different calls. For instance, according to GOSPEL's semantics, we must ensure that `1/0 = 1/0` holds but `1/0 = 2/0` does not.
- Some types have invariants attached to them, making crafting values even more complex, if not impossible.
- Even if crafting a default value is not necessary, the memoization mechanism required to ensure referential transparency is only possible if we can generate at least an `equal` function for all the argument types, which is not always possible. In the case of `Sequence.get`, that would mean using OCAML's polymorphic equality. We already discussed the issues in that case in section 6.1.2.

*Even though this is rarer, there is no guarantee that the type is inhabited at all!*

Therefore, ORTAC does not try to mimic GOSPEL's totality when implementing the standard library. Instead, it uses exceptions like they are used in the OCAML standard library: for instance, executing `Sequence.get s (-1)` raises an exception.

OCAML PURE FUNCTIONS. Third, some functions from the OCAML domain are marked as `pure` (and therefore are allowed in specifications). However, their implementation can be faulty or contract preconditions can be violated. Let us explore this possibility through the following specification for the `Array.get` function:

```
type 'a array
(*@ model { m : 'a Sequence.t } *)

val get : 'a array -> int -> 'a
(*@ v = get a i
    requires 0 <= i < Sequence.length a.m
    pure
    ensures v = a.m[i] *)
```

In this example, the function `get` is pure *on the condition that its precondition is satisfied*. Now consider the following specification for `copy`.

*Recall that we do not statically know if a function's precondition is satisfied.*

```
val copy : 'a array -> 'a array
(*@ y = copy x
    ensures forall i,
              0 <= i < length x ->
              get x i = get y i *)
```

If the implementation of `copy` is erroneous, then `y` can be a different, smaller size from `x`. In that case, the call `get y i` violates the precondition for `get`, and the function's behaviour is undefined. Under GOSPEL's totality logic, it should return an arbitrary value of type `'a`, but

this is not what `get` does in the standard library. The `get` implementation is such that executing that call will raise an `Invalid_argument` exception instead.

Whether the responsibility lies on the caller, the callee, or the environment, ORTAC cannot statically determine that an exceptional behaviour will occur. Therefore, we must account for that when generating the wrapper and deal with the potential exceptions when executing specifications appropriately.

#### 6.3.1.2 *The Instrumentation of Terms*

There are multiple ways of dealing with exceptions raised by the execution of specification terms. We already determined in the last section that replacing the faulty term result with a 'default value of the correct type' is not technically practicable. Another solution is to silence the exception and falsify the predicate that uses the failing term. For instance, `v = a.m[i]` would evaluate to `false` if the evaluation of `a.m[i]` raises an exception. This approach is arguably a risky choice: the predicate `not (v <> a.m[i])` would evaluate to... `true`! ORTAC does not implement any of these methods: silencing exceptions introduces *implicit* computation rules that users are unfamiliar with, providing unpredictable and misleading results.

The approach taken by ORTAC is to consider the whole *root* clause to have failed if one of the sub-terms raised an exception. Because the nature of the failure differs from a contract violation (a failure to check for a violation), we report this using a specific exception. In the generated code, ORTAC wraps all clauses evaluation in a `try with` block to catch and report exceptions. Let us take back the function `Queue.unsafe_pop` presented in listing 2.4 and illustrate this on the verification of its precondition.

```
 val unsafe_pop: 'a t -> 'a
(*@ v = unsafe_pop q
    requires elements q <> []
    modifies q
    ensures old elements q = (elements q) @ [v] *)
```

The instrumentation of that function is a wrapper of the following form that catches and reports execution failures in the specification.

*Like before, the details of the failure depend on the testing frontend (see chapter 3).*

```
let unsafe_pop q =
  if not
    try (elements q <> [])
    with e -> undefineness_failure e
  then correctness_failure ();
  ... (* Call the function and check the rest of
         the contract *)
```

One may also propose a hybrid reporting about such failures: (*a*) on the one hand, exceptions raised by the execution of terms *always* report a specific trigger to the user to warn that the specification could not be fully checked; (*b*) on the other hand, when the instrumenter can determine that the clause is violated *regardless of the result of that execution*, a regular specification violation is *also* reported. This approach would require a finer analysis to determine the position (negative or positive) in the root predicate. It has not been implemented in ORTAC.

THE SPECIAL CASE OF OLD. In this context, we must take special care of terms under the `old` operator. In chapter 5, we showed how ORTAC evaluates these terms: it evaluates it as much as possible in the pre-state, then copies the result of the computation and uses it in the post-state verification. The instrumentation would therefore be of the following form:

```
let unsafe_pop q =
  ... (* Check the precondition *)
  let old_elements_q = copy (elements q) in
  let v = unsafe_pop q in
  if not
    try old_elements_q = List.append (elements q) [v]
    with e -> undefineness_failure e
  then correctness_failure ()
```

Consider the case where the `elements q` under `old` raises an exception. With this instrumentation, the exception is not caught by the instrumentation code and is leaked directly to the user before the function is executed. Wrapping the `copy` inside a `try with` reporting block would also be incorrect: it would prevent the function from being executed and the other postconditions from being verified. Moreover, the `old` term may not even be needed in the execution at all because of some lazy operations (*e. g.* implying conditionals), in which case the exceptions should not be reported at all. Instead, we must ensure that the execution carries on, and the failure must only be reported in the post-state, where it is used. Therefore, when evaluating a term `old` *t*, we try to evaluate *t* and store the result (whether it is a value or an exception) in an OCAML `result` instead, then we copy it. Errors are never reported at this point. Then, when it appears in the predicate, rather than using `old_elements_q` directly, we have to unwrap it and re-raise the exception, which will, in turn, be caught by the `try with` block surrounding the postcondition check. This is where the reporting happens.

```
let unwrap = function
| Ok v -> v
| Error e -> raise e
```

*Note that we also deep-copy the exception* `e`*, as it could also carry mutable values.*

In the end, we obtain the following instrumentation.

```
let unsafe_pop q =
  ... (* Check the precondition *)
  let old_elements_q = copy (
    try Ok (elements q) with e -> Error e
  ) in
  let v = unsafe_pop q in
  if not
    try
    unwrap old_elements_q  = List.append (elements q) [v]
    with e -> undefineness_failure e
  then correctness_failure ()
```

### 6.3.2 *Exceptions Raised by Functions*

Apart from exceptions raised by the generated code while verifying the specifications, ORTAC also has to deal with the exceptions the instrumented functions may raise. Functions with a specification attached should not raise exceptions unless specified otherwise. Therefore, the basic instrumentation around functions wraps calls in `try with` blocks that catch exceptions and report them to the user as forbidden exceptions.

By default, ORTAC makes a particular case for the three exceptions mentioned in section 6.3.1.1 and lets them through as they might not be related to the program logic at all, but this is easily customizable by the user.

```
let unsafe_pop q =
  ... (* Check the precondition and compute old *)
  let v =
    try Queue.unsafe_pop q
    with
    | (Sys.Break | Stack_overflow | Out_of_memory) as e ->
      raise e
    | e -> unexpected_exception_failure e
  in
  ... (* Check the postcondition *)
  v
```

Some clauses in the function specification can allow it to raise exceptions: *via* exceptional postconditions (`raises`) or defensive preconditions (`checks`).

#### 6.3.2.1 *Exceptional Postconditions*

Consider now the variant `pop_exn`, which raises `Empty` if the input queue is empty and, in that case, does not modify it.

```
val pop_exn: 'a t -> 'a
(*@ v = pop_exn q
    modifies q
    ensures old elements q = (elements q) @ [v]
    raises Empty -> elements q = old (elements q) = [] *)
```

The translation of the exceptional postcondition is rather straightforward. When an exception appears in a `raise` clause with an associated predicate, we generate a case at the top of the exception matching that checks the associated predicate and re-raises the exception if it holds.

*Note that the precautions described in the previous section are also applied when exceptional postconditions are checked.*

```
let pop_exn q =
  let elements_q = copy (
    try Ok (elements q) with e -> Error e
  ) in
  let v =
    try Queue.pop_exn q
    with
    | Empty as e ->
      if not
        try (elements q = unwrap elements_q
             && unwrap elements_q = [])
        with e -> undefineness_failure e
      then correctness_failure ();
      raise e
    | (Sys.Break | Stack_overflow | Out_of_memory) as e ->
      raise e
    | e -> unexpected_exception_failure e
  in
  ... (* Check the normal postcondition *)
  v
```

*Remark* 20. When the function also involves types with invariants attached, those must be checked in all the exceptional branches we just generated.

#### 6.3.2.2 *Defensive Functions and Invalid Arguments*

In function specifications, the user may also write defensive preconditions in `checks` clauses. As explained in chapter 2, when a function has `checks` $P$ in its contract, we have to check that the function raises `Invalid_argument` if and only if the predicate $P$ does not hold *in the prestate*. Let us switch to the `pop` variant, which defensively checks that the input queue is not empty.

*If multiple* `checks` *clauses are present, their conjunction must hold.*

```
val pop: 'a t -> 'a
(*@ v = pop q
    checks elements q <> []
```

```
    modifies q
    ensures old elements q = (elements q) @ [v] *)
```

*The evaluation of the predicate leads to a value of type* `bool`*: it does not need to be copied.*

The `checks` clause is a precondition, so we *evaluate* the predicate in the prestate. However, we do not *check* it until the poststate: we are interested in the function reaction to the input rather than the condition on the input itself. After the function execution, we add a case for `Invalid_argument` in the error matching and check that (*a*) if the function raised `Invalid_argument`, then the precondition was not verified (otherwise, it is a false negative); (*b*) in all other cases (normal exit and other exceptional exits), the precondition was verified (otherwise, it should have raised `Invalid_argument`, and it is a false positive). For the same reasons we discussed in the case of `old`, the result (or the exception) is wrapped into an OCaml `result` so that the code reports the undefineness issues at the time of the actual checks. The generated code for `pop` is similar to the one presented in listing 6.1.

*Remark* 21. Note that a `checks` clause is incompatible with a `raises Invalid_argument` clause.

```
val f : int -> int
(*@ y = f x
    checks P
    raises Invalid_argument _ -> Q *)
```

In that case, if `f` correctly raises `Invalid_argument`, (*a*) it can be *because* `P` holds; (*b*) it can *imply* that `Q` holds; (*c*) or both at the same time. Since there is no way of deciding one case from the other, instrumenting this function would not bring guarantees regarding these clauses. For this reason, specifications that combine these clauses are rejected by Ortac.

```
let pop q =
  let check =
    try Ok (elements q <> []) with e -> Error e
  in
  let v =
    try Queue.pop_exn q
    with
    | Invalid_argument _ as e ->
      if (* Fail if the condition *was* verified. *)
        try unwrap check
        with e -> undefineness_failure e
      then correctness_failure ();
      (* Otherwise, it is correct: re-raise it. *)
      raise e
    | (Sys.Break | Stack_overflow | Out_of_memory) as e ->
      if not  (* Fail if it *was not* verified. *)
        try (unwrap check)
        with e -> undefineness_failure e
      then correctness_failure ();
      raise e
    | e ->
      if not  (* Same here *)
        try (unwrap check)
        with e -> undefineness_failure e
      then correctness_failure ();
      unexpected_exception_failure e
  in
  if not   (* Same here *)
    try (unwrap check)
    with e -> undefineness_failure e
  then correctness_failure ();
    ... (* Check the normal postcondition *)
  v
```

Listing 6.1: The handling of OCaml exceptions in the instrumented code produced for `Queue.pop`.

# 7

# PERSPECTIVES

Ortac is still a few steps away from a full-fledged tool that can be deployed in a high-scale setting. To conclude this thesis, we show two independent directions that can guide the development of the tool in the future. One is an open scientific question: how can we execute and verify logic models at runtime? In essence, these models do not exist in the implementation and therefore are never created or updated by the programs. The second is more of an engineering challenge: how can we integrate Ortac's workflow into the build system and package manager of OCaml so that instrumentations compose correctly with system dependencies, transitive dependencies, and the packaging of libraries?

## 7.1 HANDLING GOSPEL MODELS

With the mutable queues in chapter 3, we showed an example of how the specification style can impact its executability. Logic models are a central feature of Gospel; writing logic views of abstract types and using them in function specifications is a natural way of writing specifications. They are also well supported by other tools that build upon Gospel (*e.g.* the Why3 plugin and Cameleer).

They are, however, not easy to check at runtime since they do not exist in the program space. For this reason, they are currently not supported by Ortac. The main challenge is that Ortac does not have any information on *how* the models are updated after each function call and cannot retrieve the model from the value it is attached to.

Therefore, developers must write specifications differently, for instance, using pure functions, if they wish to use Ortac. Sadly, this is not always a natural way of writing specifications. It may even force developers to expose functions they otherwise would not have exposed, which hurts our initial goal of little intrusion. It is also not always possible, and models are generally more expressive: they may carry information that is *impossible* to retrieve from the implementation. Therefore, handling models is essential to supporting as many Gospel specifications as possible.

At this point, we do not have a plan for supporting *all* models in specifications. However, when writing specifications with models, a

specific shape of function postconditions naturally arises: postconditions that express the new value of the model directly, often as a function of its old version: `v.model = <term>`. For instance, in the mutable queues example, we naturally wrote the specifications of `create` and `push` that way:

```
type 'a t
(*@ mutable model elements: 'a sequence *)

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

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

When such postconditions are present, and the term is executable by Ortac, we could use it to compute the new value of the model after a call to the function. Ortac would then maintain the models associated with each value and could verify the clauses that use them.

ALL OR NOTHING. It is important to note that instrumenting a model using this technique is only possible if *all* the functions in the module have a compatible postcondition. Indeed, the model Ortac generates must be in sync with the actual OCaml value at all times to be meaningful. If one (or more) function in the interface does not have a postcondition for this model, or if the postcondition is not of a compatible form, then updating the model is not possible. In that case, the model must be deemed not executable and ignored for *all* functions and invariants in the interface. For a model to be instrumentable this way, all the functions in the interface must be specified with a compatible postcondition.

A SHIFT IN THE TRUST BASE. Using relational postconditions to generate and update the models corresponding to OCaml values has important implications regarding the verifications and guarantees Ortac provides. In this context, such postconditions would *not* be verified by the instrumented code: they would *trusted*. The generated models can still be used to check other preconditions and type invariants that involve them. However, these checks are only meaningful if the postconditions are correct, which Ortac cannot check.

THE BEST-CASE SCENARIO FOR HYBRID VERIFICATION. Let us zoom out and consider the big picture of Gospel capabilities. In particular, Gospel interfaces well with Why3 *via* the interface refinment plugin—for WhyML implementations—or Cameleer—for OCaml

implementations. Both allow models to exist in the implementation files, and some *ghost* code in the function's body must update them to let WHY3 prove the corresponding postconditions. In the case of our relational postconditions, the code that updates the model and the relational postcondition are identical, which makes the proof trivial for automatic provers. Therefore, this scenario is close to a best-case scenario to verify the parts that ORTAC assumes using WHY3 and obtain correct instrumentation.

*Ghost code only exists for specifications and cannot otherwise affect the program computations.*

A BETTER SYNTAX FOR RELATIONAL POSTCONDITIONS. Suppose ORTAC consider these specific postconditions differently for RAC. In that case, it also makes sense to differentiate them at the GOSPEL language level. It would immediately make relational postconditions visible to both tools and users. One possible syntax option is to extend the `modifies` clause as follows:

```
val push: 'a -> 'a t -> unit
(*@ push v q
    modifies q.elements <- cons v (old q.elements) *)
```

It regroups two pieces of information conceptually significantly linked, yet currently separated: *which* models are modified and *how* they are modified. Note that this may not compose very well with exceptional behaviours: `modifies` clauses—unlike `ensures` ones—apply to normal *and* exceptional behaviours, but model mutations are often different.

## 7.2 INTEGRATING ORTAC TO THE OCAML PLATFORM

In chapter 3, we showed that ORTAC provides a few options to control its output depending on the context it is used in. However, we only showed how to apply it to very basic scenarios with single-file libraries. This structure is, of course, not realistic. Real-world programs combine multiple libraries—some are local to the project, some are installed on the system—into a final client code that may be a library or an executable (or worse, a dynamic plugin). When these dependencies have GOSPEL specifications, one must decide which ones to check, instrument them with ORTAC, compile them, and link the final code with the correct implementation (instrumented or not). Developers should also decide whether they package their libraries with the runtime verifications enabled or not. Manually performing these tasks is daunting and should ultimately be dealt with by a build system and a package manager. In this section, we highlight some challenges ahead—and propose some design options—to better integrate ORTAC to some of the most essential tools in the OCAML community: its build system DUNE and its package manager OPAM.

FIRST THINGS FIRST: BETTER TOOLING FOR THE GOSPEL LANGUAGE. A new language—even if 'only' a specification language—means re-

newed needs for developer tools. At this stage, we proposed several proofs of concept of possible integrations into some existing tools.

For instance, GOSPEL can pre-process specifications to be added to the documentation comments, so users may see them in the HTML documentation generated by ODOC. However, ODOC does not know about GOSPEL specifically, so the current workaround is to insert them as verbatim code in the documentation string. We can also manually write custom rules to let DUNE handle GOSPEL specifications type-checking as part of the compilation chain. For instance, the following rule automatically type-checks the specifications during the package tests:

```
(rule
 (alias runtest)
 (action
  (run gospel check %{dep:fibonacci.mli})))
```

Ultimately, GOSPEL could integrate seamlessly with these tools—and others—to provide better user experience: editor integration *à la* MERLIN, specification formatting with OCAMLFORMAT, automatic integration with DUNE, understanding and printing clauses with cross-reference links in generated documentations.

A challenging long-term plan is to replace part of the GOSPEL type-checker to use the type information produced by the OCAML compiler during the compilation instead. There is, however, no frontend for the OCAML type-checker, so that would mean processing the compilation artifacts (*e. g.* the `*.cmti` files) to gather the information. The plan forward is not obvious at this stage, as this process would integrate poorly with the current compilation chain of OCAML programs, especially when involving ORTAC instrumentation as well. In any case, it is not expressible with custom DUNE rules.

INSTRUMENTING AND PACKAGING LIBRARIES. As for ORTAC integration, the main challenge also lies in the integration to the build system and package manager. Ideally, it is the build system's responsibility to understand the structure of the project to instrument and, therefore, to invoke ORTAC appropriately—*e. g.* on all interface files, and only when interface files change. It should also integrate the generated files into the compilation chain to produce the instrumented version of the libraries.

There are multiple paths ahead to producing instrumented libraries. One possible way is a DUNE profile similar to `dev` or `release` that produces an instrumented version. Another interesting possibility is for DUNE to generate a library variant—a library with a different implementation but the same interface. That would allow library consumers to switch between an runtime checked libary (*e. g.* for development stages) and a 'vanilla' one (for production settings) by simply changing

the dependency in the `dune` file. No modification would be necessary in the OCaml client code at all.

```
; Depend on the vanilla fibonacci library
; This is the current available workflow without Ortac
(libraries fibonacci)

; OR
; Depend on the instrumented fibonacci library
; This one is generated by Ortac + Dune
(libraries fibonacci.rac)
```

As for the packaging of these libraries, the integration with Opam is also not trivial: one should decide which version of the libraries to package and install. Ideally, when instrumenting a given 'project', $(a)$ the libraries that are local to the project should be fully instrumented; $(b)$ the direct dependencies should be instrumented with the preconditions mode only; $(c)$ the transitive dependencies should not be instrumented. Note that some direct dependencies might also be transitive dependencies. In that case, they should probably be instrumented for local calls, but uninstrumented for external calls.

In the way Opam currently works, the behaviour we just described is not achievable: it pulls dependencies source code, then compiles them once and keeps the installed version along with the compiler in a *switch*. However, in our description, we need libraries to potentially be compiled multiple times, with different instrumentation levels, depending on their dependency status in the project. It is possible that `opam-monorepo` can bring part of a solution to this issue, since it operates differently: it pulls all the sources locally and vendors them in the project, then compiles them at the same time as the project itself using Dune. Regardless, the correct design to achieve still requires more experimentations and expertise to mature.

# EPILOGUE

My work as a PhD candidate is now over, but the Gospel and Ortac projects are not.

In the past years, we have built a small yet potent team of Gospel developers who will carry on working on this verification ecosystem. It involves researchers and engineers from the Formal Methods Laboratory (LMF), Inria Paris, Tarides, and Nomadic Labs.

In December 2022, we were awarded an ANR grant to (*a*) further develop the Gospel language to increase its expressiveness; (*b*) develop a complete verification and test ecosystem around it, and improve its integration to developer tools (*e. g.* IDE, build system, documentation generators); (*c*) apply these tools to practical use cases and demonstrate their ability to scale up.

As for the Ortac project I started, it is now being actively developed by Nicolas Osborne and Samuel Hym, who keep improving the support for more Gospel specifications and more OCaml programs, and experiment with complex and strangely exciting use cases.

# BIBLIOGRAPHY


[1] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. *Data Structures and Algorithms.* Boston, MA, USA: John Wiley & Sons, Ltd, 1983 (cit. on p. 87).

[2] Mike Barnett. 'Code Contracts for .NET: Runtime Verification and So Much More'. In: *Runtime Verification*. Ed. by Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky and Nikolai Tillmann. Vol. 6418. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 16–17. DOI: 10.1007/978-3-642-16612-9_2 (cit. on p. 25).

[3] Mike Barnett, K. Rustan M. Leino and Wolfram Schulte. 'The Spec# Programming System: An Overview'. In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Ed. by Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet and Traian Muntean. Vol. 3362. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2005, pp. 49–69. DOI: 10.1007/978-3-540-30569-9_3 (cit. on p. 25).

[4] Mike Barnett and Wolfram Schulte. *Contracts, Components, and their Runtime Verification on the .NET Platform*. Tech. rep. MSR-TR-2002-38. Apr. 2002, p. 33 (cit. on p. 25).

[5] Mike Barnett and Wolfram Schulte. 'Runtime verification of .NET contracts'. In: vol. 65. 3. Elsevier BV, Mar. 2003, pp. 199–208. DOI: 10.1016/s0164-1212(02)00041-9 (cit. on p. 25).

[6] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. 2008. URL: https://frama-c.cea.fr/acsl.html (cit. on pp. 25, 91).

[7] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino and Erik Poll. 'An overview of JML tools and applications'. In: *International Journal on Software Tools for Technology Transfer* 7.3 (June 2005), pp. 212–232. DOI: 10.1007/s10009-004-0167-4 (cit. on p. 25).

[8] 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. Baltimore, Maryland, United States, 1990, pp. 392–402. DOI: http://doi.acm.org/10.1145/255471.255563. URL: http://doi.acm.org/10.1145/255471.255563 (cit. on p. 25).

[9] Patrice Chalin and Frédéric Rioux. 'JML Runtime Assertion Checking: Improved Error Reporting and Efficiency Using Strong Validity'. In: *Lecture Notes in Computer Science*. Ed. by Jorge Cuéllar, T. S. E. Maibaum and Kaisa Sere. Vol. 5014. Berlin, Heidelberg: Springer Berlin Heidelberg, June 2008, pp. 246–261. DOI: 10.1007/978-3-540-68237-0_18 (cit. on p. 91).

[10] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço and Mário Pereira. 'GOSPEL—Providing OCaml with a Formal Specification Language'. In: *Lecture Notes in Computer Science*. Ed. by Maurice H. ter Beek, Annabelle McIver and José N. Oliveira. Vol. 11800. Porto, Portugal: Springer International Publishing, Oct. 2019, pp. 484–501. DOI: 10.1007/978-3-030-30942-8_29. URL: https://hal.inria.fr/hal-02157484v2/file/final.pdf (cit. on p. 7).

[11] Yoonsik Cheon. 'A runtime assertion checker for the Java Modeling Language'. In: (Aug. 2018). DOI: 10.31274/rtd-180813-9872 (cit. on p. 25).

[12] Alonzo Church. 'An Unsolvable Problem of Elementary Number Theory'. In: *Journal of Symbolic Logic* 1.2 (1936), pp. 73–74. DOI: 10.2307/2268571 (cit. on p. 83).

[13] Koen Claessen and John Hughes. 'QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs'. In: *SIGPLAN Not.* 35.9 (Sept. 2000), pp. 268–279. DOI: 10.1145/357766.351266. URL: https://doi.org/10.1145/357766.351266 (cit. on p. 45).

[14] David R. Cok. *Does your software do what it should? User guide to specification and verification with the Java Modeling Language and OpenJML*. Version draft as of July 1, 2022. 2022. URL: https://www.openjml.org/documentation/OpenJMLUserGuide.pdf (visited on 01/07/2023) (cit. on p. 46).

[15] David R. Cok. 'OpenJML: JML for Java 7 by Extending OpenJDK'. In: *Lecture Notes in Computer Science*. Ed. by Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann and Rajeev Joshi. Vol. 6617. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 472–479. DOI: 10.1007/978-3-642-20398-5_35 (cit. on p. 25).

[16] Simon Cruanes, Grinberg Rudi, Jacques-Pascal Deplaix, Jan Midtgaard and Valentin Chaboche. *QCheck*. Github. 2013. URL: https://github.com/c-cube/qcheck/ (visited on 01/07/2023) (cit. on p. 45).

[17] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles and Boris Yakobowski. 'Frama-C'. In: *Software Engineering and Formal Methods*. Ed. by George Eleftherakis, Mike Hinchey and Mike Holcombe. Vol. 7504. Springer Berlin

Heidelberg, May 2012, pp. 233–247. DOI: 10.1007/978-3-642-33826-7_16 (cit. on p. 25).

[18] Mickaël Delahaye, Nikolai Kosmatov and Julien Signoles. 'Common specification language for static and dynamic analysis of C programs'. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13*. Ed. by Sung Y. Shin and José Carlos Maldonado. ACM Press, Mar. 2013, pp. 1230–1235. DOI: 10.1145/2480362.2480593 (cit. on p. 25).

[19] Stephen Dolan. *Crowbar*. Github. 2017. URL: https://github.com/stedolan/crowbar (visited on 01/07/2023) (cit. on p. 43).

[20] Jean-Christophe Filliâtre and Andrei Paskevich. 'Why3 — Where Programs Meet Provers'. In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Mar. 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-37036-6_8.pdf (cit. on p. 25).

[21] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt and Marc Heuse. 'AFL++ : Combining Incremental Steps of Fuzzing Research.' In: *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. Ed. by Yuval Yarom and Sarah Zennou. WOOT'20. USA: USENIX Association, 2020. DOI: 10.5555/3488877.3488887. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi (cit. on p. 43).

[22] 'IEEE Standard for Floating-Point Arithmetic'. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229 (cit. on p. 100).

[23] Piotr Kosiuczenko. 'An Abstract Machine for the Old Value Retrieval'. In: *Lecture Notes in Computer Science*. Ed. by Claude Bolduc, Jules Desharnais and Béchir Ktari. Vol. 6120. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 229–247. DOI: 10.1007/978-3-642-13321-3_14 (cit. on p. 91).

[24] Nikolai Kosmatov, Fonenantsoa Maurica and Julien Signoles. 'Efficient Runtime Assertion Checking for Properties over Mathematical Numbers'. In: *Runtime Verification*. Ed. by Jyotirmoy Deshmukh and Dejan Nickovic. Vol. 12399. Cham: Springer International Publishing, 2020, pp. 310–322. DOI: 10.1007/978-3-030-60508-7_17 (cit. on p. 91).

[25] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby and Bart Jacobs. 'JML: notations and tools supporting detailed design in Java'. In: *OOPSLA 2000 Companion, Minneapolis, Minnesota*. 2000, pp. 105–106 (cit. on p. 25).

[26] Hermann Lehner and Peter Müller. 'Efficient Runtime Assertion Checking of Assignable Clauses with Datagroups'. In: *Fundamental Approaches to Software Engineering*. Ed. by David S. Rosenblum and Gabriele Taentzer. Vol. 6013. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 338–352. DOI: 10.1007/978-3-642-12029-9_24. URL: https://link.springer.com/content/pdf/10.1007%2F978-3-642-12029-9_24.pdf (cit. on p. 91).

[27] K. Rustan M. Leino. 'Dafny: An Automatic Program Verifier for Functional Correctness'. In: *LPAR-16*. Vol. 6355. 2010, pp. 348–370 (cit. on p. 25).

[28] Xavier Leroy and François Pessaux. 'Type-based analysis of uncaught exceptions'. In: *ACM Transactions on Programming Languages and Systems* (*TOPLAS*) 22.2 (2000), pp. 340–377. DOI: 10.1145/349214.349230. URL: https://inria.hal.science/hal-01499948 (cit. on p. 41).

[29] Jun Li, Bodong Zhao and Chao Zhang. 'Fuzzing: a survey'. In: *Cybersecurity* 1 (Dec. 2018). DOI: 10.1186/s42400-018-0002-y (cit. on p. 45).

[30] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, Aug. 2015. DOI: 10.1017/cbo9781139629294 (cit. on pp. 25, 46).

[31] B. Meyer. 'Applying 'design by contract''. In: *Computer* 25.10 (Oct. 1992), pp. 40–51. DOI: 10.1109/2.161279. URL: http://www.inf.ethz.ch/~meyer/publications/computer/contract.pdf (cit. on pp. 25, 46).

[32] P. Müller, M. Schwerhoff and A. J. Summers. 'Viper: A Verification Infrastructure for Permission-Based Reasoning'. In: *Verification, Model Checking, and Abstract Interpretation* (*VMCAI*). Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62 (cit. on p. 25).

[33] Clément Pascutto. *Ortac*. Github. 2020. URL: https://github.com/ocaml-gospel/ortac (visited on 01/07/2023) (cit. on p. 8).

[34] United States Patent and Trademark Office. 'DESIGN BY CONTRACT'. 2911197. Inc. Interactive Software Engineering. 2004 (cit. on p. 5).

[35] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov and Julien Signoles. 'Instrumentation of Annotated C Programs for Test Generation'. In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. Victoria, Canada: IEEE, Sept. 2014, pp. 105–114. DOI: 10.1109/scam.2014.19 (cit. on p. 91).

[36] François Pottier. *Monolith*. Gitlab. 2020. URL: https://gitlab.inria.fr/fpottier/monolith/ (cit. on p. 43).

[37] François Pottier. 'Strong Automated Testing of OCaml Libraries'. In: *JFLA 2021 - 32es Journées Francophones des Langages Applicatifs*. Saint Médard d'Excideuil, France, Feb. 2021. URL: https://inria.hal.science/hal-03049511 (cit. on p. 43).

[38] The Gospel Project. *The Gospel Documentation*. 2023. URL: http://ocaml-gospel.github.io/gospel (visited on 01/07/2023) (cit. on p. 22).

[39] The Vocal Project. *Vocal*. Github. 2018. URL: https://github.com/ocaml-gospel/vocal (visited on 01/07/2023) (cit. on p. 7).

[40] Julien Signoles. 'The e-ACSL perspective on runtime assertion checking'. In: *Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution*. Ed. by Wolfgang Ahrendt, Davide Ancona and Adrian Francalanza. VORTEX 2021: Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution. (online), Denmark: ACM, July 2021, pp. 8–12. DOI: 10.1145/3464974.3468451 (cit. on p. 91).

[41] Julien Signoles, Basile Desloges and Kostyantyn Vorobyov. *The E-ACSL User Manual*. Version 27.0. 2023. URL: https://frama-c.com/download/e-acsl/e-acsl-manual.pdf (visited on 01/07/2023) (cit. on p. 46).

[42] Julien Signoles, Nikolai Kosmatov and Kostyantyn Vorobyov. 'E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper)'. In: *Kalpa Publications in Computing*. Ed. by Giles Reger and Klaus Havelund. Vol. 3. EasyChair, Jan. 2018, pp. 164–173. DOI: 10.29007/fpdh. URL: https://easychair.org/publications/open/t6tV (cit. on pp. 25, 91).

[43] Alan Mathison Turing. 'On Computable Numbers, with an Application to the Entscheidungsproblem'. In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: https://doi.org/10.1112/plms/s2-42.1.230 (cit. on p. 83).

[44] Michał Zalewski. *Technical "whitepaper" for afl-fuzz*. 2013. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt (visited on 01/07/2023) (cit. on p. 43).

This thesis, including its LaTeX source code, figures, and Coq proofs is available online at https://github.com/pascutto/phd-thesis.