JFLA 2021 - 32 èmes Journées Francophones des Langages Applicatifs

Yann Regis-Gianas, Chantal Keller

32ème Journées Francophones des Langages Applicatifs

du 7 au 9 avril 2021 sur l'Internet

LABORATOIRE MÉTHODES FORMELLES

UMR CNRS 9021

nomadic labs

Préface

Les JFLA réunissent chaque année, dans un cadre convivial, concepteurs, développeurs et utilisateurs des langages fonctionnels, des assistants de preuve et des outils de vérification de programmes. Le spectre des travaux présentés aux JFLA est très large : il touche ainsi les aspects les plus théoriques de la conception des langages applicatifs jusqu'à ses applications industrielles. La pandémie a imposé la virtualisation de la 32ème édition des JFLA. Les journées de 2021 se déroulent donc en ce lieu singulier qu'est l'Internet.

Cette année, nous avons sélectionné 9 articles de recherche, 4 articles de recherche courts, et 6 démonstrations. Une fois n'est pas coutume, le programme de ces JFLA aborde de nombreuses thématiques : de la vérification mécanisée de code OCaml et des mathématiques – les nombres premiers, les permutations et les graphes seront à l'honneur – à celles des réseaux de neurones ; de la programmation synchrone certifiée à la sémantique de JavaScript en passant par le modèle mémoire de C++ ; et de la métathéorie des capacités à celle de CiC en passant par celle de Rust. Nous aurons aussi l'occasion de découvrir de nouvelles techniques de programmation fonctionnelle avec les appels terminaux "modulo cons" et l'introspection bien typée. Ces éléments de programmation seront complétés par un peu plus de démonstrations d'outils que d'habitude, déclinés autour de thématiques similaires : un outil pour programmer en OCaml sur FPGA, coqffi pour l'interopérabilité entre Coq et les bibliothèques OCaml impures, MOPSA pour l'interprétation abstraite de programmes, Mlang pour la spécification des impôts, et enfin une évolution de Coq avec des flottants primitifs.

Cette sélection a été faite par les membres du comité de programme, que nous remercions chaleureusement, à partir des 24 soumissions. Nous tenons aussi à exprimer nos vifs remerciements aux rapporteurs externes au comité de programme : Jacques-Henri Jourdan, Pierre Jouvelot, Mário Pereira, Pascal Raymond, et Mattias Roux.

En plus du programme sélectionné, nous aurons le grand plaisir d'écouter un cours de Loïc Correnson présentant la vérification de programmes avec Frama-C ainsi qu'un exposé invité de David Mentré sur les méthodes formelles dans l'industrie.

Nous espérons que cette édition un peu particulière saura malgré tout maintenir l'esprit de convivialité, de curiosité et de bienveillance qui caractérisent habituellement les JFLA.

Programmatiquement vôtre,

Yann Régis-Gianas et Chantal Keller.

Comité de programme

Yann Regis-Gianas	Nomadic Labs
Chantal Keller	Université Paris-Saclay
Adrien Guatto	Université de Paris, CNRS, IRIF
Albin Coquereau	OCamlPro
Assia Mahboubi	INRIA
Claire Dross	AdaCore
Damien Pous	CNRS - ENS Lyon
Evelyne Contejean	Université Paris-Saclay
Marie Kerjean	CNRS, LIPN
Pascale Le Gall	CentraleSupelec
Rodolphe Lepigre	MPI-SWS
Simão Melo de Sousa	Release/LISP – LIACC – Dep. de Informática, Univ. da Beira Interior
Thomas Letan	ANSSI
Timothy Bourke	INRIA

Rapporteurs externes

Pierre Jouvelot
Mário Pereira
Pascal Raymond
Jacques-Henri Jourdan
Mattias Roux

Table des matières

Pratique de la preuve de programme avec Frama-C/WP et Why-3 1
Loïc Correnson

Some views on industrial usages of Formal Methods 2
David Mentré

Strong Automated Testing of OCaml Libraries 3
François Pottier

Cameleer: a Deductive Verification Tool for OCaml 21
Mário Pereira and António Ravara

Démonstration de la plateforme Mopsa d'analyse statique de programmes par interprétation abstraite 45
Matthieu Journault, Antoine Miné, Raphaël Monat et Abdelraouf Ouadjaout

Tail Modulo Cons 48
Frédéric Bour, Basile Clément, and Gabriel Scherer

Type-safe type reflection for OCaml 69
Thierry Martinez

OCaml sur circuit FPGA 72
Jocelyn Sérot and Emmanuel Chailloux

coqffi: Outil pour la génération automatique de FFI Coq/OCaml 75
Thomas Letan and Li-yao Xia

Mécanisation du modèle RC11 et de la propriété DRF-SC 78
Quentin Ladeveze

JSkel: Towards a Formalization of JavaScript's Semantics 95
Adam Khayam, Louis Noizet, and Alan Schmitt

Normalisation vérifiée du langage Lustre 117
Timothy Bourke, Paul Jeanmaire, Basile Pesin, Marc Pouzet

Partitionnement en régions linéaires pour la vérification formelle de réseaux de neurones 134
Julien Girard-Satabin, Aymeric Varasse, Guillaume Charpiat, Zakaria Chihani, and Marc Schoenauer

Lavez vos graphes plus blanc avec HoCL 146
Jocelyn Sérot

Mlang: an Open-Source Toolchain for the Income Tax Computation 155
Denis Merigoux and Raphaël Monat

Cap' ou pas cap' ? Preuve de programmes pour une machine à capacités en présence de code inconnu 157
Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal

Mastering Program Verification using Possession and Prophecies 174
Xavier Denis

Mettons de l'Ordre dans CIC ! 190
Théo Laurent and Kenji Maillard

Théories de permutations avec Why3 202
Alain Giorgetti

Les nombres premiers au crible de la preuve formelle 210
Josué Moreau

Flottants primitifs dans Coq 220
Érik Martin-Dorel, Pierre Roux

Pratique de la preuve de programme avec Frama-C/WP et Why-3

Cours

Loïc Correnson

CEA

Résumé

La vérification d'absence de bug est réputée difficile - à juste titre - mais est particulièrement importante dans le monde de l'informatique embarquée, typiquement dans le contrôle-commande que ce soit pour un avion, un train, une centrale, etc. Dans ce cours, nous étudierons différents aspects d'un code embarqué réalisant un calcul physique extrêmement simple, mais qui illustre parfaitement l'ensemble des problèmes et des défis à relever dans ce domaine. Nous essaierons, avec l'aide et la perspicacité des participants, de relever la plupart de ces défis en combinant les deux plateformes Frama-C/WP et Why3 et en suivant une méthodologie inspirée de travaux effectués notamment à la NASA.

Some views on industrial usages of Formal Methods

Exposé invité

David Mentré

Mitsubishi Electric R&D Centre Europe

Résumé

In this presentation, we will review the motivation to use formal methods in industry nowadays and then the key challenges to address : how to select formal methods, how to use them, especially regarding the integration in an industrial process and last but not least how to convince to use formal methods. Incidentally we will point out in our view the needs on formal methods for a wider deployment in industry. We will take examples from both industry and MERCE experience over the last ten years within Mitsubishi Electric.

Strong Automated Testing of OCaml Libraries

François Pottier

Inria Paris

Abstract

We present Monolith, a programmable tool that helps apply random testing or fuzz testing to an OCaml library. Monolith provides a rich specification language, which allows the user to describe her library's API, and an engine, which generates clients of this API and executes them. This reduces the problem of testing a library to the problem of testing a complete program, one that is effectively addressed by off-the-shelf fuzzers such as AFL.

Prologue: How Ed Got Fired

Ed, an underpaid programmer at a company that sells software components, is assigned the task of implementing a persistent array library in OCaml. An "array", in a broad sense, is a data structure that represents a map of some interval $[0, n)$ to values. A "persistent array" is immutable, or appears to be so. Ed is asked to implement the following signature:

```
type 'a t
val make : int -> 'a -> 'a t
val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> 'a t
```

`make n x` creates an array of size `n` where every cell contains the value `x`, while `get a i` returns the value stored in array `a` at index `i`, and `set a i x` creates a new array that is identical to `a` except that the value stored at index `i` is `x`. Note that `set a i x` does not modify the array `a`.

Because Ed is unhappy with his work, he decides to botch the job. After spending an afternoon at the swimming pool, he returns with the following two-line implementation:

```
include Stdlib.Array
let set a i x = set a i x; a
```

This implementation stores the data in a mutable array: it is incorrect. In the hope of disguising his fraud, Ed compiles it to machine code. Ed knows that the flaw cannot be detected by a "single-threaded" sequence of operations where each operation acts on the most recently-created array. To detect it, one must apply a `get` operation to an array that has already been the subject of a `set` operation. Ed happens to know that the employees of the company's quality assurance department have been laid off a week earlier, and believes his boss incapable of imagining such a tricky scenario, so he thinks he can get away with it.

However, to Ed's surprise, a few minutes after he has handed in his implementation, Bryn, his manager, barges into his office, yelling: "What is this? You're fired!" as he slaps onto the desk a printout that reads as follows:

```
(* ./output/crashes/id:000000,sig:06,src:000000,op:flip16,pos:6 *)
(* @03: Failure in an observation: candidate and reference disagree. *)
(* @01 *) let a0 = make 1 0;;
(* @02 *) let a1 = set a0 0 1;;
(* @03 *) let observed = get a0 0;;
          assert (observed = 0);; (* candidate finds 1 *)
```

Ed is shocked: how could Bryn come up so quickly with this scenario? "How...", Ed begins. "Oh, that was easy," Bryn snarls. "All I had to do was write a reference implementation of persistent arrays, in two lines, like this..."

```
include Stdlib.Array
let set a i x = let a = Array.copy a in set a i x; a
```

"A copy! Why didn't I do a copy," thinks Ed, who realizes that he could have been just as lazy without getting caught so quickly, while Bryn goes on: "... and write down the specifications of the three operations, like this:"

```
open Monolith
module R = Reference (* the above reference implementation *)
module C = Candidate (* Ed's implementation *)
let () =
  (* Specs. *)
  let array   = declare_abstract_type()
  and element = sequential()
  and length  = interval 0 16
  and index a = interval 0 (R.length a) in
  (* Operations. *)
  declare "make" (length ^> element ^> array)                    R.make C.make;
  declare "get"  (array ^>> fun a -> index a ^> element)         R.get  C.get;
  declare "set"  (array ^>> fun a -> index a ^> element ^> array) R.set  C.set;
  (* Run, with 5 units of fuel. *)
  main 5
```

Bryn is gone. While packing, Ed pieces together how Monolith [18] enabled his boss to expose him so easily. In a few lines of code, Bryn declared the existence of an abstract type `array` and of three operations `make`, `get`, `set`. For each operation, he gave one specification and two implementations. For instance, the specification `length ^> element ^> array` indicates that `make` expects an integer length, which must be drawn from the interval $[0, 16)$, an integer element, where elements must be generated in a sequential manner, and returns an array. It is implemented by `C.make` on the candidate side and by `R.make` on the reference side. The specification `array ^>> fun a -> index a ^> element` states that `get` expects an array `a`, a nonnegative integer index that is less than the length of `a`, and returns an integer element. The abstract type `array` is represented by distinct types on the reference side and on the candidate side: indeed, the result type of `R.make` is `int R.t`, whereas the result type of `C.make` is `int C.t`. Bryn did not even need to point this out explicitly; this information was inferred.

Ed imagines that `main` randomly generates scenarios that exploit `make`, `get` and `set`, executes these scenarios using both the reference implementation and the candidate implementation, and compares the results. Either by purely random testing or by giving control of the source of random bits to an efficient gray-box mutation-based fuzzer, such as AFL [24], one can very quickly discover a scenario where the two implementations disagree.

"Unit testing has never been so easy," thinks Ed, sadly, as he exits his former office.

1 Introduction

Fuzz testing, or fuzzing [10, 25], is an extremely effective approach to finding bugs in software. Fuzzing is usually applied to a standalone executable program. This program is expected to read a stream of input bits and to respond by either crashing or not crashing. The fuzzer's job, then, is to craft a sequence of bits that causes a crash.

Fuzz testing is very much akin to random testing. The only difference between the two lies in the fact that the input stream is not random; it is controlled by an adversary. From the point of view of the program under test, this makes no difference. In terms of effectiveness, this can make a big difference: fuzzing can be much more effective at finding bugs than random testing.

Submitting a complete program to random testing or fuzz testing is definitely very interesting and useful. Yet, this begs the question: can these methods be applied to a library, as opposed to a self-contained executable program?

At first sight, this may seem difficult, because a library presents a much more complex interface to the outside world than a standalone program. A library usually publishes many types and operations, and may impose complex constraints on their usage. Manually or automatically constructing a good test suite (that is, a collection of usage scenarios that exercises the library in a wide variety of ways) is difficult [1].

In the setting of OCaml, this paper attempts to answer this question in the affirmative. We present Monolith [18], a programmable tool that helps transform a library into a self-contained program, therefore reducing the problem of testing this library to the problem of testing a complete program. Monolith offers a rich set of combinators that allow a library author to construct a description of her library's API. The author must also provide a reference implementation of her library. Once these two pieces are provided, Monolith takes care of the rest. Its engine generates a valid usage scenario, executes this scenario, and (if an incorrect behavior is detected) prints this scenario and crashes. Thus, the library under test, combined with Monolith's engine, forms a self-contained executable program that can be submitted to random testing or fuzz testing via standard means.

The paper is structured as follows. First (§2), we describe the combinators out of which specifications are built, which form a rich domain-specific language. Then, we briefly describe Monolith's engine (§3), give a few more examples of the use of Monolith (§4), and discuss some of its limitations (§5). Finally, we give a brief review of the related work (§6) and conclude (§7).

2 A Specification Language

To be able to invoke the operations of the library under test, Monolith must have access to their specifications. There are two reasons why this is so: first, because OCaml is a typed language, Monolith must at least know the type of each operation; second, to be able to generate suitable arguments and to determine what to do with the results, Monolith must have access to richer specifications that may carry generators, preconditions, postconditions, and so on.

For this purpose, Monolith gives the user a means of constructing specifications. There is a type `('r, 'c) spec` of specifications, together with a rich collection of combinators that build complex specifications out of simpler ones.

The type `spec` is parameterized with the types `'r` and `'c` of the values that a specification describes. The type `'r` is for the reference side, while `'c` is for the candidate side. Because an abstract type can be implemented in different ways by the reference implementation and by the candidate implementation, an operation may have different types on either side.

To summarize, a specification of type `('r, 'c) spec` is a runtime description of a pair of OCaml values whose respective types are `'r` and `'c`. We refer to such a pair as a *dual value*. The specification tells Monolith in what way the two components of this pair are related: it is in fact a runtime description of a *logical relation* [22].

Within the universe of specifications, we distinguish subsets of *constructible* specifications, which describe values that Monolith is able to construct, and *deconstructible* specifications, which describe values that Monolith can deconstruct or test for equality. We impose a number of rules, such as the following: (1) a function argument must be constructible; (2) a function

```
(* The type of specifications. *)
type ('r, 'c) spec
(* Basic constructible type. *)
val constructible: (unit -> 't) -> ('t, 't) spec
(* Basic deconstructible type. *)
val deconstructible: ('t -> 't -> 'bool) -> ('t, 't) spec
(* Basic abstract type. *)
val declare_abstract_type: unit -> ('r, 'c) spec
(* Combining a constructible spec and a deconstructible spec. *)
val ifpol: ('r, 'c) spec -> ('r, 'c) spec -> ('r, 'c) spec
(* Pair. *)
val ( *** ): ('r1, 'c1) spec -> ('r2, 'c2) spec -> ('r1 * 'r2, 'c1 * 'c2) spec
(* Option. *)
val option : ('r, 'c) spec -> ('r option, 'c option) spec
(* Result. *)
val result : ('r1, 'c1) spec -> ('r2, 'c2) spec -> (('r1, 'r2) result, ('c1, 'c2) result) spec
(* Recursion. *)
val fix: (('r, 'c) spec -> ('r, 'c) spec) -> ('r, 'c) spec
(* Function. *)
val (^>)   : ('r1, 'c1) spec ->         ('r2, 'c2) spec  -> ('r1 -> 'r2, 'c1 -> 'c2) spec
(* Dependent function. *)
val (^>>)  : ('r1, 'c1) spec -> ('r1 -> ('r2, 'c2) spec) -> ('r1 -> 'r2, 'c1 -> 'c2) spec
(* Subset / precondition. *)
val (%): ('r -> bool) -> ('r, 'c) spec -> ('r, 'c) spec
(* Nondeterminism / postcondition. *)
type 'r diagnostic = Valid of 'r | Invalid
val nondet: ('r, 'c) spec -> ('c -> 'r diagnostic, 'c) spec
(* Transformation into or out of a known specification. *)
val map_into:  ('r1 -> 'r2) -> ('c1 -> 'c2) -> ('r2, 'c2) spec -> ('r1, 'c1) spec
val map_outof: ('r1 -> 'r2) -> ('c1 -> 'c2) -> ('r1, 'c1) spec -> ('r2, 'c2) spec
```

Figure 1: The specification language: core combinators

result must be deconstructible; (3) a function is neither constructible nor deconstructible (§5). By lack of space, we omit the details. This discipline is currently enforced at runtime. In the future, we hope to enforce it statically via extra parameters of the type `spec`.

2.1 Core Specification Combinators

A slightly simplified presentation of the core specification combinators appears in Fig. 1. When displaying a scenario, Monolith must be able to print values, operations, and so on. In the figure, for the sake of simplicity, we have removed everything that has to do with printing.

Basic concrete types The function `constructible` (Fig. 1) lets Monolith regard an OCaml type `'t` as constructible: that is, it equips Monolith with a way of constructing values of type `'t`. No runtime description of this type needs be given. A generator, a function of type `unit -> 't`, must be provided. Through an API not shown in this paper, a generator has access to a source of "random" bits, which may be either truly random or controlled by a fuzzer. The values produced by the generator are used both by the reference implementation and by the candidate implementation. For this reason, the return type of `constructible` is `('t, 't) spec`.

The function `deconstructible` lets Monolith regard an OCaml type `'t` as deconstructible:

that is, it equips Monolith with a way of deconstructing values of type `'t`. More accurately, it lets Monolith observe these values, that is, test them for equality. An equality test of type `'t -> 't -> bool` must be provided. It is used by Monolith to compare the values produced by the reference implementation and by the candidate implementation. A discrepancy is regarded as a fatal error, causing a report and a crash.

A constructible specification is used to describe an argument of an operation, whereas a deconstructible specification is used to describe the result of an operation. The combinator `ifpol` allows combining one specification of each kind into a single specification that is both constructible and deconstructible. It can be used to describe base types (§2.2), algebraic data types, first-class function types, and more (§5).

Basic abstract types The function `declare_abstract_type` makes Monolith aware of the existence of an abstract type, represented by the OCaml types `'r` and `'c` on the reference side and on the candidate side, respectively. The two implementations are allowed to use different runtime representations of this abstraction.

The specification returned by `declare_abstract_type` is constructible and deconstructible. Yet, Monolith never generates or observes a value of this type. When it must construct such a value, it *selects* a value of this type from those that have been produced by previous operations. (The source of randomness is used to choose among them.) When it must deconstruct such a value, it simply *records* this value in an environment, for use in a future operation.

An abstract type is by default opaque: when a candidate-side data structure has abstract type, Monolith has no way of checking at runtime that this data structure is well-formed or that it agrees with the data structure maintained by the reference implementation. Yet, it can be desirable to allow Monolith to see under the hood, so to speak, and to perform a well-formedness check. This can be done by supplying a `check` function of type `'r -> 'c -> unit` as an optional argument to `declare_abstract_type`. (This is not shown in Fig. 1.) This function is expected to either silently succeed or fail by raising an exception. It is up to the user to decide how thorough, and how costly, this check should be.

The `check` function is applied by Monolith after *every* operation and to *every* dual value that has been recorded in the environment, even if this value is not ostensibly affected by this operation (that is, even if it is not an argument of this operation). This can be very useful when the candidate implementation involves shared mutable state: if an operation on a candidate data structure c corrupts another candidate data structure c' that happens to share part of its representation with c, this can be detected immediately, whereas in the absence of a `check` function, one or more operations on c' would be required for the problem to become apparent.

Structural types The combinator `***` describes a value of a product type, that is, a pair. The combinators `option` and `result` describe values of two common sum types. These product and sum combinators produce constructible (resp. deconstructible) specifications when they are applied to constructible (resp. deconstructible) specifications.

Monolith is able to construct a value of a product or sum type out of smaller values. When constructing a sum type, this involves a choice: for this purpose, one random bit is read.

Monolith is able to deconstruct a value of a product or sum type into smaller values. A pair is decomposed into its components. When deconstructing a sum type, Monolith checks whether the reference implementation and the candidate implementation agree on the tag. For instance, if one implementation produces `None` while the other implementation produces a value of the form `Some _`, then a discrepancy has been detected, causing a report and a crash. If both implementations produce values of the form `Some _`, then the data constructor `Some` can be stripped off on both sides, and the deconstruction process can continue.

By combining basic concrete types, basic abstract types, products, and sums, one teaches

Monolith how to handle values of composite structural types. For instance, suppose that some operation named `choose` has type `set -> (element * set) option`. Assuming that `element` has been declared as a (constructible and deconstructible) concrete type and assuming that `set` has been declared as an abstract type, the result of this operation can be described by the specification `option (element *** set)`, which is constructible and deconstructible. In particular, to deconstruct the result of `choose`, Monolith checks that either the reference implementation and the candidate implementation both return `None`, or they respectively return `Some (re, rs)` and `Some (ce, cs)`. In the latter case, Monolith further checks that the elements `re` and `ce` are equal and records the existence of a new dual value of abstract type `set`, whose reference-side and candidate-side projections are respectively `rs` and `cs`. This dual value can be passed as an argument to a future operation that requires a set.

Recursive types The combinator `fix` allows the user to construct a recursive specification, that is, a cyclic specification. This feature can be used to describe algebraic data types, such as lists and trees (§5). It can in theory cause Monolith to diverge while attempting to construct a value. This is not a problem in practice, because the probability of divergence is usually zero, and because a limit on execution time is usually imposed. AFL imposes such a limit.

Function types The combinator `^>` constructs a simple (nondependent) function specification. Its two arguments describe the function's domain and codomain. For instance, the operation `choose` that was mentioned above has specification `set ^> option (element *** set)`. Like the function type constructor `->`, the combinator `^>` is right-associative, so curried functions can be easily described: for instance, an operation `add` that inserts an element into a set could have specification `element ^> set ^> set`.

The combinator `^>>` constructs a dependent function specification. In contrast with `^>`, it allows the specification of the codomain to depend on the function's argument. That is, the codomain is not just a specification, of type `('r2, 'c2) spec`, but a function of an argument to a specification, of type `'r1 -> ('r2, 'c2) spec`. This function has access to the reference-side projection of the actual argument generated by Monolith, and can refer to it in the specification of the codomain. This feature is typically exploited in concert with either `constructible` or the subset combinator `%`. Either way, the point is to let the value of an earlier argument influence the choice of a later argument, either by dictating how it must be generated, or, after it has been generated, by rejecting it if it is unsuitable. This is explained next.

Preconditions Broadly speaking, there are two ways of expressing a precondition: either a priori, by influencing the manner in which an argument is generated, or a posteriori, by filtering out unsuitable values for this argument.

Let us illustrate the first approach via an example. Suppose that the library under test involves an abstract type of sequences. Suppose that `get` expects two arguments, namely a sequence `s` and an index `i` into the sequence `s`, and returns an element. The specification of this operation could be `seq ^>> fun s -> interval 0 (R.length s) ^> element`. The dependent function combinator `^>>` is used to bind the variable `s` to the reference-side projection of the first argument. This enables the function call `R.length`, which queries the reference implementation for the length of the sequence `s`. This in turn allows generating an integer index directly in the desired range. The constructible specification `interval i j` describes an integer in the semi-open interval $[i, j)$. It is defined in terms of `constructible`, `deconstructible`, and `ifpol`.

The second approach exploits the subset combinator `%`. This combinator must be applied to a constructible specification and produces a constructible specification. It carries a predicate of type `'r -> bool`, which has access to the reference-side projection of the value that has been constructed and must indicate whether this value is acceptable.

As an example, suppose that one wishes to test a library that involves an abstract type of file descriptors. Suppose that, via `declare_abstract_type`, one has obtained a specification `fd` of type `(R.fd, C.fd) spec`, where `R.fd` and `C.fd` are the types of file descriptors on each side. A file descriptor is either valid or invalid. Suppose that the reference implementation keeps track of this information: the function `R.valid`, of type `R.fd -> bool`, determines whether a reference-side file descriptor is valid. Finally, let us assume that `close` requires a valid file descriptor, and invalidates it. The specification of this operation could be `R.valid % fd ^> unit`. (The combinator `%` binds tighter than `^>`.) The precondition `R.valid` prevents Monolith from applying `close` to an invalid file descriptor, which would not make sense. The fact that `close` invalidates its argument is not visible in the specification; it is visible in the implementation of the functions `R.close` and `R.valid`, which must update and consult the validity information associated with a descriptor. On the candidate side, no function `C.valid` is required: the candidate implementation need not keep track of validity at runtime.

Postconditions Whereas a precondition expresses a property of an argument, which the library under test expects to hold, a postcondition expresses a property of a result, which the library under test must guarantee.

It is often the case that the postcondition of an operation is deterministic, that is, a single value is considered a valid result. As this is the common case, Monolith adopts it as the default. Monolith requires the user to provide a reference implementation of each operation (§3.1) and, by default, expects the reference implementation and the candidate implementation to produce the same result. In that case, no explicit postcondition needs be given.

Sometimes, however, an operation has a nondeterministic specification: that is, several distinct results are permitted. In such a situation, the `nondet` combinator must be used. It produces a deconstructible specification. It is typically placed in the codomain of a function specification, that is, after the rightmost arrow combinator `^>`. Using `nondet` changes the type and the role of the reference implementation. Whereas the reference implementation normally returns a result of type `'r`, it must now return a function of type `'c -> 'r diagnostic` (Fig. 1). This function is a postcondition: it has access to the value of type `'c` produced by the candidate implementation and must return a value of type `'r diagnostic`. Such a value can be viewed as a truth value, enriched with additional information. Indeed, the type `'r diagnostic` is a sum type: the data constructor `Valid` indicates that the candidate result is acceptable, whereas `Invalid` indicates that it is not. In the former case, the diagnostic includes the result of type `'r` that the reference implementation wishes to return. An example use of `nondet` appears in §4.1.

Input and output transformations `map_into` and `map_outof` describe a transformation that must be applied by Monolith to a piece of data. They require the user to supply two transformation functions: a transformation of type `'r1 -> 'r2` for use on the reference side, and a transformation of type `'c1 -> 'c2` for use on the candidate side.

In the case of `map_outof`, the user must provide a specification of type `('r1, 'c1) spec`, that is, a description of the input of the transformation. This means that the transformation can be used to map data out of a type whose structure is known to Monolith. Thus, `map_outof` must be applied to a constructible specification, and returns one. It is typically used to preprocess an argument of an operation.

In the case of `map_into`, the user must provide a specification of type `('r2, 'c2) spec`, that is, a description of the output of the transformation. Thus, the transformation maps data into a type whose structure is known. Therefore, `map_into` must be applied a deconstructible specification, and returns one. It can be used to postprocess the result of an operation. It can also be exploited to wrap an operation in an OCaml context that alters its calling convention: this is illustrated by the manner in which we handle exceptions (§2.2).

```
(* Predefined concrete types. *)
val unit     : (unit, unit) spec
val bool     : (bool, bool) spec
val int      : (int, int) spec
val interval : int -> int -> (int, int) spec
val exn      : (exn, exn) spec
(* Function with an exception effect. *)
val (^!>)  : ('r1, 'c1) spec -> ('r2, 'c2) spec ->
             ('r1 -> 'r2, 'c1 -> 'c2) spec
(* Function with a nondeterminism effect. *)
val (^?>)  : ('r1, 'c1) spec -> ('r2, 'c2) spec ->
             ('r1 -> 'c2 -> 'r2 diagnostic, 'c1 -> 'c2) spec
(* Function with exception and nondeterminism effects. *)
val (^!?>) : ('r1, 'c1) spec -> ('r2, 'c2) spec ->
             ('r1 -> ('c2, exn) result -> ('r2, exn) result diagnostic, 'c1 -> 'c2) spec
(* Moving the second argument to the first place. *)
val rot2   : ('r1 -> 'r2 -> 'r3, 'c1 -> 'c2 -> 'c3) spec ->
             ('r2 -> 'r1 -> 'r3, 'c2 -> 'c1 -> 'c3) spec
```

Figure 2: The specification language: some derived combinators

2.2 Derived Specification Combinators

Predefined concrete types A number of concrete types are predefined using `constructible`, `deconstructible`, and `ifpol`. They include `unit`, `bool`, `int`, `exn` (Fig. 2). All are deconstructible. The types `unit` and `bool` are constructible, whereas `int` and `exn` are not, as it seems desirable to let the user ponder the choice of a suitable generator. `interval i j` is constructible, and causes an integer in the semi-open interval $[i, j)$ to be generated.

Effectful functions By default, Monolith allows neither the reference implementation nor the candidate implementation to raise an exception: both events are considered abnormal. If an operation is expected to raise an exception under normal circumstances, then it must be wrapped in an exception handler. This is done in a couple lines of code, as follows:

```
let handle f x = try Ok (f x) with e -> Error e
let (^!>) a b  = map_into handle handle (a ^> result b exn)
```

The function `handle` turns a function that can raise an exception, whose type is `'a -> 'b`, into one that cannot raise an exception, whose type is `'a -> ('b, exn) result`. The application of `handle` to an operation is hidden underneath a thin layer of sugar: the exceptional function combinator `a ^!> b` is a short-hand for an application of `map_into` that instructs Monolith to use `handle`, both on the reference side and on the candidate side, and to deconstruct the value of type `('b, exn) result` that is thus obtained. When this combinator is used, Monolith checks that either the reference implementation and candidate implementation both return a value, or they both raise an exception. In the latter case, Monolith uses the equality function at type `exn` to check that they raise related exceptions. By default, equality at type `exn` is OCaml's generic equality; however, this can be customized via a mechanism not described in this paper.

To sum up, the combinators `^>` (Fig. 1) and `^!>` (Fig. 2) have the same type. The former forbids raising an exception, while the latter allows the two implementations to raise the same exception (or related exceptions).

It seems convenient to also define `a ^?> b` as a short-hand for `a ^> nondet b`, suggesting that nondeterminism is an effect:

```
let (^?>) a b = a ^> nondet b
```

Finally, it seems useful to define an arrow combinator that combines the exception effect and the nondeterminism effect. The combinator `^!?>` is used to describe an operation that may choose whether it wishes to raise or not raise an exception. It is defined as follows:

```
let (^!?>) a b = map_into (fun x -> x) handle (a ^> nondet (result b exn))
```

The candidate implementation is wrapped with `handle`, so as to catch all exceptions. Thanks to `nondet`, the reference implementation has access to the behavior of the candidate, a value of type `('c2, exn) result`. It must describe its own behavior by returning either `Valid (Ok _)` or `Valid (Error _)`. It is not allowed to raise an exception, so it need not be wrapped with `handle`: it is wrapped with the identity function instead.

Rearranging arguments Sometimes, the arguments of an operation come in an inconvenient order: for instance, the first argument may be subject to a precondition that refers to the second argument. This is the case, for example, of a `remove` operation of type `element -> set -> set`, where the element that is passed as the first argument is required to be a member of the set that is passed as the second argument. When testing this operation, one would like to first pick a set `s` that is at hand, then pick a member `x` of this set. It would not make much sense to first blindly generate `x` and then hope that some set that contains `x` is at hand. Fortunately, it is easy to define a combinator `rot2` that exchanges the order of the two arguments:

```
let rot2 f y x = f x y
let rot2 spec  = map_into rot2 rot2 spec
```

Using `rot2`, it is easy to express the specification of `remove`. It can be written under the form `rot2 (R.nonempty % set ^>> fun s -> choose s ^> set)`, where `choose s` is an abbreviation for `constructible (fun () -> R.choose s)`. It can be read informally as follows: once its arguments are exchanged, `remove` expects a nonempty set `s` and an element that must be chosen in the set `s` and returns a set. We assume that `R.nonempty` tests whether a set is nonempty and that `R.choose`, which has access to the source of randomness, chooses an element in a nonempty set.

3 Monolith's Engine

3.1 Usage

Once the specification language described in the previous section is mastered, using Monolith is very easy. The engine's API consists of just two functions, `declare` and `main` (Fig. 3). `declare` declares the existence of an operation, and must be called once per operation; `main` runs the engine, and must be called once, after all operations have been declared.

```
(* Declaring an operation. *)
val declare : string -> ('r, 'c) spec -> 'r -> 'c -> unit
(* Starting the engine, with some fuel. *)
val main : int -> unit
```

Figure 3: Monolith's engine: main functions

An operation is described by its name, by a specification of type `('r, 'c) spec`, and by its implementations on the reference side and on the candidate side, whose respective types are `'r` and `'c`. OCaml's type discipline guarantees that the specification provided by the user is consistent with the types of the two implementations.

When the engine is started, by invoking `main`, an integer number n, which represents a certain amount of "fuel", must be given. This number bounds the length of the scenarios that Monolith investigates: Monolith generates and executes sequential scenarios of at most n instructions. An instruction is of the form let $p = e$, where e is an OCaml expression, typically an application of an operation to a suitable series of arguments, and p is an OCaml pattern, which deconstructs the result of executing e. Each scenario is executed both under the reference implementation and under the candidate implementation, step by step, synchronously. If a discrepancy is detected, then Monolith prints the entire scenario and crashes. Otherwise, the scenario is abandoned after n instructions have been generated and executed, and Monolith moves on to another scenario.

During this process, every time a choice must be made, the source of randomness is used. This occurs when Monolith chooses an operation or generates an argument for an operation. Thus, viewed from the outside, the complete executable program is a black box that reads a stream of bits and responds (in a deterministic way) by either terminating normally or crashing.

The goal of the testing process, then, is to feed this executable program with random data, or with adversely crafted data, in order to cause a crash.

Random testing can be performed by letting the executable program read data from the pseudo-device `/dev/urandom`. Grey-box mutation-based fuzzing can be performed by requesting the OCaml compiler to instrument the executable program with AFL-specific instructions and by letting AFL drive its execution. A blog post by Rebours [20] explains the workflow. The scripts in `Makefile.monolith`, which is bundled with Monolith, automate most of it.

When the executable program is about to crash, it first prints a scenario on its standard output channel. This is a sequence of OCaml instructions that leads to a discrepancy between the reference implementation and the candidate implementation. The user can copy and paste this scenario directly into the OCaml REPL in order to reproduce the problem.

3.2 Implementation

Monolith's implementation is a few thousand lines of OCaml code. The type `(_, _) spec` is a generalized algebraic data type (GADT). The engine simultaneously generates a well-typed straight-line scenario and executes this scenario both under the reference implementation and under the candidate implementation. Each instruction in this scenario is of the form let $p = e$, where the expression e is built by choosing an operation and constructing a series of arguments for it, while the pattern p is obtained by deconstructing the result of executing this operation. Both of these processes, namely construction of arguments and deconstruction of results, are driven by the structure of the specification provided by the user for this operation.

All of the variables introduced by a pattern p have abstract type. Indeed, a result whose type is concrete need not be bound to a variable; it can be deconstructed, checked for validity, and forgotten. A result whose type is abstract, however, cannot be validated. It must be bound to a variable, so it can later be passed to another operation. Monolith internally maintains a runtime environment where every variable has abstract type and is bound to a dual value.

The fact that the scenario is generated and executed at the same time plays an important role in a few places. When a value of a sum type must be deconstructed, this value is available already, so its tag can be inspected, and a suitable pattern can be constructed. In other words, there is no need for generating case analysis constructs with multiple branches: one branch, the correct branch, is always enough. When a variable of a certain abstract type must be chosen

among those that exist in the environment while respecting a certain precondition, the fact that every variable is already bound to a value is essential: this means that one can effectively test which choices satisfy the precondition and which do not.

4 Examples

We present a few examples of the use of Monolith. In each case, we give a specification and a reference implementation, but do not show the candidate implementation, which is irrelevant.

4.1 Nondeterministic Increasing-Sequence Generators

We wish to specify nondeterministic generators of integer sequences, whose API is as follows:

```
type t
val create : unit -> t
val next : t -> int
```

There is an abstract type `t` of generators. The function `create` returns a fresh generator. The function `next` queries a generator for the next element of the sequence that it represents. The function call `next g` must produce a number that is nonnegative and strictly greater than the number produced by any previous call `next g`.

To test an implementation of this API, one can write the following main program, which declares the type `t` and its operations and runs Monolith:

```
let t = declare_abstract_type() in
declare "create" (unit ^> t) R.create C.create;
declare "next"   (t ^?> int) R.next   C.next;
main 5
```

The specification `t ^?> int` indicates that the operation `next` is nondeterministic. According to it, whereas the candidate function `C.next` has type `C.t -> int`, the reference function `R.next` has type `R.t -> int -> int diagnostic`. A reference implementation can be written as follows:

```
type t = int ref
let create () = ref 0
let next (g : t) (candidate : int) : int diagnostic =
  if !g < candidate then (g := candidate; Valid candidate)
  else Invalid
```

The reference implementation uses mutable state to record the last value produced by the candidate. The function `R.next` reads this state to determine whether `candidate` is an acceptable result. If it is, then it updates this state and returns `Valid _`; otherwise, it returns `Invalid`.

4.2 Semi-Persistent Arrays

Semi-persistent arrays [6] resemble persistent arrays, which we have encountered in the prologue. However, they offer a more restrictive API, which disallows certain access patterns. For this reason, they can be implemented more efficiently. Their API is as follows:

```
type 'a t
val make : int -> 'a -> 'a t
val length : 'a t -> int
val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> 'a t
```

There is an abstract type `'a t` of arrays of elements of type `'a`. The operation `make n x` creates a fresh array of length `n` that holds `n` times the element `x`. The operation `length a` returns the length of the array `a`. The operation `get a i` reads the array `a` at offset `i`, while `set a i x` updates the array `a` at offset `i` with the value `x`, producing a new array, which is considered a *child* of `a`. At any point in time, the arrays that have been created so far, equipped with the child relationship, form a forest. Within each tree, an array is considered *valid* if and only if it is a (reflexive, transitive) parent of the array that was most recently accessed via `get` or `set`. Whereas `length` can be applied to any array, `get` and `set` must be applied to a valid array.

To test an implementation of this API, one can write the following main program:

```
let elt = sequential()
and t = declare_abstract_type() in
declare "make"   (lt 16 ^> elt ^> t)                                    R.make   C.make;
declare "length" (t ^> int)                                             R.length C.length;
declare "get"    (R.valid % t ^>> fun a -> lt (R.length a) ^> elt)      R.get    C.get;
declare "set"    (R.valid % t ^>> fun a -> lt (R.length a) ^> elt ^> t) R.set    C.set;
main 5
```

We use `lt j` as a short-hand for `interval 0 j`. The operation `length`, which can be applied to any array, does not have a precondition, while `get` and `set`, which must be applied to a valid array, have the precondition `R.valid`. This prevents Monolith from generating an access pattern that violates the semi-persistent array API.

There remains to provide a reference implementation of semi-persistent arrays, which must offer not only the four operations `R.make`, `R.length`, `R.get`, and `R.set`, but also the runtime validity test `R.valid` that has been used above as part of the specification. One of the simplest possible implementations, inspired by Conchon and Filliâtre's paper [6], is to associate with each tree a stack of currently valid arrays. This stack represents a path from the most-recently accessed array in this tree up to the root of this tree. To test whether an array is valid, one searches for it in the stack. To invalidate the strict descendants of a valid array, one truncates the stack after this array. Both `get` and `set` perform such an invalidation step.

```
type 'a t = { data: 'a array; stack: 'a t list ref }
(* Stack operations. *)
let peek   stack = match !stack with [] -> assert false | x :: _ -> x
let pop    stack = match !stack with [] -> assert false | _ :: xs -> stack := xs
let push x stack = stack := x :: !stack; x
(* A validity test and an invalidation operation. *)
let valid spa = List.memq spa !(spa.stack)
let invalidate_descendants spa = while peek spa.stack != spa do pop spa.stack done
(* Operations on semi-persistent arrays. *)
let make n x =
  let data = Array.make n x and stack = ref [] in push { data; stack } stack
let length spa =
  Array.length spa.data
let get spa i =
  invalidate_descendants spa; Array.get spa.data i
let set spa i x =
  invalidate_descendants spa;
  let data = Array.copy spa.data and stack = spa.stack in
  Array.set data i x; push { data; stack } stack
```

4.3 Sek

We have used Monolith in conjunction with AFL in order to test Sek [4], a library that offers efficient implementations of ephemeral sequences, persistent sequences, and ephemeral iterators on both kinds of sequences. Sek publishes 4 abstract types and over 150 operations. Its data structures involve complex balancing invariants, shared mutable state, and a subtle ownership policy that determines when an object can be updated in place and when it must be copied. Sek itself represents about 6000 lines of nonblank noncomment lines of code. The reference implementation and the specifications together occupy about 1500 lines of code, that is, about 10 lines per operation on average. As Sek evolved over the course of several months, we found fuzzing extremely effective in uncovering bugs: in our experience, many bugs can be found in minutes, and some more can be found in hours. The fact that Monolith always produces a short scenario is invaluable. It is of course impossible to tell how many bugs in Sek were *not* found. Still, we can confidently say that testing has saved us a lot of trouble and embarrassment.

5 Limitations and Work-Arounds

User-defined algebraic data types Only a fixed collection of product types and sum types are known to Monolith (§2.1). There is no built-in support for tuple types of arity greater than two, for sum types other than `option` and `result`, or for algebraic data types. Such support can however be programmed by the user. Indeed, thanks to the combinators `fix`, `ifpol`, `map_outof`, and `map_into`, the isomorphism between an algebraic data type and a sum of products can be declared and exploited.

The algebraic data type `list`, for instance, could be described in this way. However, such an approach would perhaps be too simple-minded. It would cause Monolith to construct a list by first flipping a coin so as to choose between the data constructors `[]` and `::` and then, if `::` has been chosen, by constructing an element and a sublist, thereby repeating the process. This approach would make it unlikely for long lists to be constructed. In practice, it seems preferable to generate a list by first selecting its length within a certain range, then generating as many elements as necessary. Fortunately, the core combinators that we have presented allow using this strategy on the construction side while sticking with the simple-minded approach on the deconstruction side.

Because lists are so common, Monolith does provide a `list` combinator (not shown in Fig. 1 or Fig. 2). This combinator could in principle be defined by the user outside Monolith.

Functions as arguments Monolith currently does not allow an operation to take a function as an argument: a specification of the form `(a ^> b) ^> c` is rejected. This is a consequence of the rules set forth at the beginning of §2: a function argument must be constructible, and a function is not constructible. It is not clear at this time how one might relax this restriction. Indeed, several of Monolith's design principles seem incompatible with the desire of generating a function that satisfies a certain specification. The fact that Monolith generates code and executes it on the fly, the fact that it generates straight-line code only, and the fact that it generates code without a predetermined goal, all seem incompatible with the idea of generating code for a function with the goal of obeying a predetermined specification.

Fortunately, it is possible to some extent to work around this limitation. Using `constructible` to equip the function type `a -> b` with a custom generator is one possible workaround. Using `map_outof` to transform some other constructible type into the function type `a -> b` is another approach. Finally, one could also declare `a -> b` as an abstract type and equip it with one or more operations that construct values of this type. Although these approaches are limited to generating values in a strict subset of the type `a -> b`, they can be good enough.

In some cases, testing a higher-order function can be reduced to testing a well-chosen first-order function. For instance, to test a `fold` function that iterates over a collection, one could argue that it suffices to define the function `elements` (which constructs a list of all elements of the collection) in terms of `fold`, and to test `elements`. Indeed, no information other than a sequence of elements can be extracted out of `fold`.[1]

Functions as results One could say that Monolith does not allow an operation to return a function as a result. Indeed, although a specification of the form `a ^> b ^> c` is accepted, it is treated as the specification of a function of arity two. Thus, an operation `op` whose specification is `a ^> (b ^> c)` is always applied to two arguments: Monolith does not perform partial applications. Furthermore, the specification `a ^> (b1 ^> c1) * (b2 ^> c2)` is rejected altogether. This is a consequence of the rules set forth at the beginning of §2: a function result must be deconstructible, and a function is not deconstructible.

Fortunately, these restrictions are superficial and can be worked around. A function type such as `b -> c` can be declared as an abstract type `t` and equipped with an identity operation `id` of type `t ^> b ^> c`. The operation `op` above can then receive the specification `a ^> t`. This yields the desired behavior: Monolith applies `op` to just one argument, binds the resulting function to a variable, and (in subsequent operations) can apply this function several times. For convenience, Monolith offers a derived combinator `declare_semi_abstract_type` (not shown in Fig. 2) that does this on the fly. Thus, the operation `op` can be given the specification `a ^> declare_semi_abstract_type (b ^> c)`.

Sequences OCaml's standard library defines the type `'a Seq.t` to represent on-demand sequences of elements of type `'a`. An element is produced only when the sequence is queried by a consumer. The type of sequences is not an abstract type: instead, it is a function type. It is therefore an example of a procedural abstraction [21].

Sequences can be given Monolith specifications by using the techniques described in the previous two paragraphs. The combinator `ifpol` allows specifying separately how to construct and how to deconstruct sequences. On the construction side, one can generate a sequence essentially in the same way as one generates a list. On the deconstruction side, one can declare a type `seq` of sequences as an abstract type and equip it with a `query` operation whose specification is `seq ^> option (element *** seq)`. This allows Monolith to demand elements one at a time, and also allows it to make multiple queries (possibly interleaved with other instructions) on a single sequence. This is appropriate when dealing with persistent sequences, which can be queried as many times as one wishes.

The treatment of affine sequences, which can be queried at most once, is slightly different. On the construction side, one must construct a sequence that fails at runtime (by raising an exception) if it is queried twice. This allows detecting a situation where an operation queries an affine sequence several times. On the deconstruction side, Monolith must be instructed that an affine sequence must not be queried twice, so as to prevent it from generating illegal usage scenarios. To do so, one must first map a sequence into a custom representation that supports not only querying a sequence, but also testing whether a sequence is valid (that is, whether it has not yet been queried). This custom representation of sequences (which the end user remains unaware of) is then declared as an abstract type `seq`, whose `query` operation is assigned the specification `valid % seq ^> option (element *** seq)`, where the precondition `valid` prevents Monolith from attempting to query a sequence twice.

[1] The reader might remark that this fails to exercise some exotic scenarios, such as a scenario where `fold` is applied to a function that raises an exception, or a scenario where `fold` is applied to a function that invokes `fold` in a reentrant manner. This is true. It is up to the user to think about these scenarios and to decide whether, by not exercising them, she is likely to miss some bugs.

Because sequences are commonly used, Monolith provides two functions `declare_seq` and `declare_affine_seq` (not shown in Fig. 2) that implement these ideas.

Polymorphism Monolith has no explicit support for polymorphic functions or parameterized types. To test a polymorphic function, one must test one or more monomorphic instances of it. Bernardy *et al.* [2] discuss systematic ways of choosing a single monomorphic instance. To work with a parameterized type, such as `'a set`, one can choose to work with a specific monomorphic instance of it, such as `int set`. One can also define a parameterized combinator that represents this parameterized type: as an example, Monolith's `list` combinator has type `('r, 'c) spec -> ('r list, 'c list) spec`.

Shrinking Monolith does not currently perform any shrinking [9, 14]. AFL provides a tool, `afl-tmin`, which performs a certain amount of shrinking, and appears to give relatively good results. However, because this tool is unaware of the structure of the instruction sequences built by Monolith, its effectiveness remains limited. It would be interesting to build a custom shrinking algorithm into Monolith. A related technique that can be used today is iterated narrowing: as soon as a scenario of length n is reported by Monolith, one can reduce the amount of fuel to n so as to narrow the search space and increase the chances that Monolith finds a shorter scenario.

Life without a reference implementation Monolith requires a reference implementation of the library under test. One might think that this is a limitation. Indeed, it can be difficult to write a simple yet reasonably efficient reference implementation. However, it is in fact possible to use Monolith in the absence of a full-fledged reference implementation. To do so, one provides a trivial reference implementation, whose operations do nothing, and one instructs Monolith to use a trivial comparison relation, which always returns `true`, when comparing a candidate result against a reference result.[2] This method allows testing the candidate implementation in isolation; it can detect crashes and violations of runtime assertions.

Another interesting idea, suggested by an anonymous reviewer, is to use version n of a library as a reference implementation while testing version $n+1$ of this library. This technique could conceivably be exploited as part of a continuous integration system.

6 Related Work

Perhaps the best-known and most influential approach to testing in the functional programming community is QuickCheck [9]. QuickCheck offers a library of generators that can construct various kinds of typed data. One popular style of use of QuickCheck is property-based testing, where the user states a universally quantified property, such as `remove x (insert x s) = s`, and the system verifies that a finite number of randomly-selected instances of this property hold. To do so, QuickCheck must generate candidate-side data: if a set `s` is represented in the candidate implementation as a balanced binary search tree, then QuickCheck must be programmed so as to generate such trees. This style is well-suited to a purely functional programming setting, where data structures are immutable and sharing is unobservable, but not to an imperative programming setting, where data structures are mutable and involve sharing. In such a setting, another style of use of QuickCheck, investigated by Claessen and Hughes [5], is model-based testing. This style involves generating sequences of instructions and verifying that the candidate implementation executes them in a manner that is deemed correct with respect to a model. Midtgaard [15] uses this style to test an OCaml implementation of Patricia trees, and finds a

[2] One is likely to need variants of the combinators `^>>` and `%` that give access to the candidate-side projection instead of (or in addition to) the reference-side projection. We plan to provide these combinators in the future.

bug in it. This bug can be found also by Monolith [17], in a few seconds when in random testing mode, and in about one minute when driven by AFL. This requires using an integer generator that has a nonnegligible probability of producing the integer `min_int`.

Gast [13, 12] seems closely related to QuickCheck and supports the same testing methodologies as QuickCheck.

Monolith encourages model-based testing: it automatically generates and executes sequences of instructions, and requires the user to provide a reference implementation. It can generate concrete data, but does not generate inhabitants of abstract types: it obtains them only as the result of executing instructions. If desired, Monolith can also be used to test properties, such as `remove x (insert x s) = s`. To test this property, one embeds it inside a pseudo-operation of type `element ^> set ^> unit`, whose reference implementation does nothing, and whose candidate implementation tests the property and fails if the property is violated.

Dolan introduces support for AFL in the OCaml compiler and is the author of Crowbar [7], a library of data generators that facilitates property-based testing in conjunction with AFL. Crowbar does not offer any support for model-based testing.

Like Monolith, Articheck [3] requires the user to provide a description of each operation, and executes operations in order to construct values of abstract types. However, it differs from Monolith in several key ways. It does not require a reference implementation: it searches for faults in a candidate implementation. It runs for a long time, potentially accumulating many values of each type, whereas Monolith investigates very short scenarios. When it detects a fault, it cannot print a short scenario that exhibits the fault, whereas Monolith can. Finally, it is not designed to be externally controlled by a fuzzer. The authors of Articheck report being able to detect the balancing violation reported by Filliâtre in OCaml's AVL trees [8]. This bug is in fact very easy to find: Monolith, in random testing mode, reliably finds it in a few seconds [16]. This requires writing a `check` function that performs a well-formedness check; the bug cannot otherwise be detected, as it does not endanger functional correctness.

Klein *et al.* [11] perform random testing of Scheme programs that may involve higher-order functions and mutable state. They use higher-order contracts both as generators and as oracles: we use Monolith specifications for the same dual purpose. Whereas we perform code generation and execution at the same time, they separate these tasks. For this reason, code generation has access to the type of every variable, but not to its value, and code generation is performed with the goal of constructing a value of a certain predetermined type. On the positive side, this allows generating an inhabitant of a function type, and allows the generation process to involve backtracking search. On the negative side, this requires exploring a huge search space, and requires generating conditional instructions: this is the case, in particular, when a pre- or postcondition must be met, and when a value of a sum type must be deconstructed.

7 Conclusion

We have presented Monolith, a powerful tool for testing an OCaml library in isolation. Monolith requires a reference implementation and a concise specification of each operation. It supports random testing and fuzzing. Its combinators allow constructing a runtime description of the logical relation that connects the reference and candidate implementations. Monolith's engine is both a program generator and a dual well-typed interpreter: it generates a scenario and runs it under two distinct interpretations of the library's abstract types and operations. Although using GADTs to express unary runtime type descriptions and implement well-typed interpreters is by now common [23, 3, 19], this may be the first example of using a GADT to express a binary logical relation and implement a dual well-typed interpreter.

References

[1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[2] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 125–144. Springer, March 2010.

[3] Thomas Braibant, Jonathan Protzenko, and Gabriel Scherer. Articheck: well-typed generic fuzzing for module interfaces. In *ACM Workshop on ML*, August 2014.

[4] Arthur Charguéraud, François Pottier, and Émilie Guermeur. Sek. `https://gitlab.inria.fr/fpottier/sek/`, 2020.

[5] Koen Claessen and John Hughes. Testing monadic code with QuickCheck. *ACM SIGPLAN Notices*, 37(12):47–59, 2002.

[6] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-persistent data structures. In *European Symposium on Programming (ESOP)*, volume 4960 of *Lecture Notes in Computer Science*, pages 322–336. Springer, April 2008.

[7] Stephen Dolan. Crowbar. `https://github.com/stedolan/crowbar`.

[8] Jean-Christophe Filliâtre. AVL mal equilibrés dans Set. `https://github.com/ocaml/ocaml/issues/8176`, June 2003.

[9] John Hughes. QuickCheck testing for fun and profit. In *Practical Aspects of Declarative Languages (PADL)*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer, January 2007.

[10] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Conference on Computer and Communications Security (CCS)*, pages 2123–2138. ACM, October 2018.

[11] Casey Klein, Matthew Flatt, and Robert Bruce Findler. Random testing for higher-order, stateful programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 555–566, October 2010.

[12] Pieter W. M. Koopman, Peter Achten, and Rinus Plasmeijer. Model based testing with logical properties versus state machines. In *Implementation of Functional Languages (IFL)*, volume 7257 of *Lecture Notes in Computer Science*, pages 116–133. Springer, October 2011.

[13] Pieter W. M. Koopman, Artem Alimarine, Jan Tretmans, and Marinus J. Plasmeijer. Gast: Generic automated software testing. In *Implementation of Functional Languages (IFL)*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer, September 2002.

[14] Fang-Yi Lo, Chao-Hong Chen, and Ying-Ping Chen. Shrinking counterexamples in property-based testing with genetic algorithms. In *IEEE Congress on Evolutionary Computation*, pages 1–8, July 2020.

[15] Jan Midtgaard. QuickChecking Patricia trees. In *Trends in Functional Programming (TFP)*, volume 10788 of *Lecture Notes in Computer Science*, pages 59–78. Springer, 2017.

[16] François Pottier. A demo of Monolith: `faulty/avl`. `https://gitlab.inria.fr/fpottier/monolith/-/tree/master/demos/faulty/avl`, 2020.

[17] François Pottier. A demo of Monolith: `faulty/map`. `https://gitlab.inria.fr/fpottier/monolith/-/tree/master/demos/faulty/map`, 2020.

[18] François Pottier. Monolith. `https://gitlab.inria.fr/fpottier/monolith/`, 2020.

[19] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages*, 2(POPL):16:1–16:34, 2018.

[20] Nathan Rebours. An introduction to fuzzing OCaml with AFL, Crowbar and Bun. `https://`

tarides.com/blog/2019-09-04-an-introduction-to-fuzzing-ocaml-with-afl-crowbar-and-bun, 2019.

[21] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. Technical Report 1278, Carnegie Mellon University, August 1975.

[22] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

[23] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Principles of Programming Languages (POPL)*, pages 224–235, January 2003.

[24] Michal Zalewski. American Fuzzy Lop. https://github.com/google/AFL, 2020.

[25] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book: tools and techniques for generating software tests. https://www.fuzzingbook.org/, 2020.

Cameleer: a Deductive Verification Tool for OCaml*

Mário Pereira and António Ravara

NOVA LINCS & DI, Nova School of Science and Technology, Portugal

Abstract

OCaml is particularly well-fitted for formal verification. On one hand, it is a multi-paradigm language with a well-defined semantics, allowing one to write clean, concise, type-safe, and efficient code. On the other hand, it is a language of choice for the implementation of sensible software, *e.g.*, industrial compilers, proof assistants, and automated solvers. Yet, with the notable exception of some interactive tools, formal verification has been seldom applied to OCaml-written programs. In this paper, we present the ongoing project Cameleer, aiming for the development of a deductive verification tool for OCaml, with a clear focus on proof automation. We leverage on the recently proposed GOSPEL, *G*eneric *O*Caml *SPE*cification *L*anguage, to attach rigorous, yet readable, behavioral specification to OCaml code. The formally-specified program is fed to our toolchain, which translates it into an equivalent program in WhyML, the programming and specification language of the Why3 verification framework. Finally, Why3 is used to compute verification conditions for the generated program, which can be discharged by off-the-shelf SMT solvers. We present successful applications of the Cameleer tool to prove functional correctness of several significant case studies, like FIFO queues (ephemeral and applicative implementations) and leftist heaps, issued from existing OCaml libraries.

1 Introduction

Over the past decades, we have witnessed a tremendous development in the field of deductive software verification [13]. Interactive proof assistants have evolved from obscure and mysterious tools into *de facto* standards for proving industrial-size software projects. Notable examples include the Sel4 verified operating system kernel [25], and the verified compilers CompCert [23] and CakeML [35]. On the other end of the spectrum, the so-called *SMT revolution* and the development of reusable intermediate verification infrastructures contributed decisively to the development of practical automated deductive verifiers. Remarkable applications of automated verification tools include the verified version of the Microsoft's Hypervisor [4] and, more recently, the use of ghost monitors [9] to analyze installation scenarios of the Debian distribution [3].

Despite all the advances in deductive verification and proof automation, little attention has been given to the family of *functional languages* [32]. Taking the example of the OCaml language, if it is a language well-fitted for verification (given its well-defined semantics, clear syntax, and state-of-the-art type system), the community still misses an easy to use framework for the specification and verification of OCaml code.

In this paper, we present Cameleer, a tool for the deductive verification of programs directly written in OCaml, with a clear focus on proof automation. Cameleer uses the recently proposed GOSPEL [6], a specification language for the OCaml language. We believe this is one of the strengths of our approach: firstly, GOSPEL makes a certain number of design choices, that turn it into a clean and digestible specification language; secondly, GOSPEL terms are written in a subset of the OCaml language. In the scope of this work, we have also extended GOSPEL

*This work is partly supported by the HORIZON 2020 Cameleer project (Marie Skłodowska-Curie grant agreement ID:897873) and NOVA LINCS (Ref. UIDB/04516/2020)

to include implementation primitives, such as loop invariants and ghost code, evolving the language from an interface specification language into a more mature proof tool.

Cameleer takes as input an OCaml program annotated with GOSPEL specification and translates it into an equivalent counterpart in WhyML, the programming and specification language of the Why3 framework [18]. Why3 is a toolset for the deductive verification of software, clearly oriented towards automated proof. A distinctive feature of Why3 is that it can interface with several different off-the-shelf theorem provers, namely SMT solvers, which greatly increases proof automation. We believe that proof automation is another strong point of Cameleer, on that can ease its adoption by regular OCaml programmers. In this paper, we present some case studies of automatically verified OCaml modules with our tool.

Contributions. To the best of our knowledge, Cameleer is the first deductive verification tool for annotated OCaml programs. It handles a realistic subset of the language, as demonstrated by a comprehensive set of case studies. Another contribution of this paper is our translation of the OCaml module language into WhyML. While sharing many common syntactic constructions, OCaml and WhyML differ more significantly when it comes to their module systems. This poses interesting challenges to our translation scheme.

Paper structure. This paper is organized as follows. Sec. 2 presents a simple example of a verified OCaml program, intended as a smooth introduction to the Cameleer tool and GOSPEL language. Sec. 3 describes the OCaml to WhyML translation mechanism implemented in the core of Cameleer. Sec. 4 reports on relevant case studies verified with Cameleer, including ephemeral and functorial data structures. Finally, we present some related work in Sec. 5 and conclude with future work in Sec. 6. The source code of Cameleer and verified case studies are publicly available online on the GitHub repository of the project[1].

2 Warmup Example

In this section, we present the verified implementation of the Fibonacci function. We first present a purely applicative version of the mathematical definition, and then an efficient implementation using side-effects. This section is intended as a gentle introduction to the Cameleer framework, the GOSPEL specification language, and certain important verification concepts such as ghost code, loop invariants, and function contracts.

A logical definition. The classical mathematical definition can be readily translated, in OCaml, into the following recursive function:

```
let rec fib n =
  if n <= 1 then n else fib (n-1) + fib (n-2)
```

Though very elegant and concise, the above definition presents some pitfalls and raises a number of questions: is this a *total* function (*i.e.*, it terminates and is defined for every integer argument) and could we avoid repeated computations in recursive calls? We focus now on the former and shall address the latter in a moment.

The Fibonacci function is, traditionally, only defined for a *non-negative* argument and this is exactly the definition we follow here. Such a constraint on a function arguments is a *precondition*, *i.e.*, it limits the range of values the function arguments can take. We shall take this opportunity to show a first piece of OCaml code annotated with GOSPEL specification:

```
let rec fib n =
  if n <= 1 then n else fib (n-1) + fib (n-2)
```

[1] https://github.com/mariojppereira/cameleer

```
(*@ requires n >= 0 *)
```

A GOSPEL specification is attached to the end of a function definition and is given within special comments of the form `(*@ ... *)`. The `requires` clause is used to state the precondition of a function. In order to prove that every call to `fib` halts, we must provide a *variant* that strictly decreases at each recursive call and has a lower-bound. Here, the value of `n` decreases at each recursive call and is bounded from below thanks to the precondition. In GOSPEL, we express the variant of a function as follows:

```
let rec fib n = ...
(*@ requires n >= 0
    variant  n *)
```

The given precondition and the variant form what we call the *function's contract*. Other than serving as rigorous documentation of the function's behavior, the interest of providing a contract is to be able to *formally* prove that the code respects the given specification. This is where the Cameleer toolchain enters the scene.

Assuming function `fib` is contained in the OCaml file `fibonacci.ml`, starting a proof is as easy as typing `cameleer fibonacci.ml` in a terminal. Cameleer translates the input program into an equivalent WhyML counterpart and launches the Why3 graphical integrated development environment. This allows us to visually inspect the verification conditions (VCs) generated by Why3 for the `fib` function: two of them state the variant decreases and is not a negative value, at each recursive call; the other two state the precondition holds for each recursive call. All of these are easily discharged using SMT solvers, *e.g.*, Alt-Ergo [10], CVC4 [2], or Z3 [12].

The provided `fib` function is a naive implementation, as it unnecessarily repeats most intermediate computations. Actually, this is not meant to be used as an *executable* implementation, but rather as a *logical* description of the Fibonacci definition. This means `fib` works more as mathematical function than a programming one. In order to instruct Cameleer to consider `fib` a logical function, we decorate it with the OCaml attribute `[@logic]`, as follows:

```
let [@logic] rec fib n = ...
```

Under this setting, the `fib` function can now be used both inside specification clauses, as well as in regular OCaml code.

A verified efficient implementation. Moving on to an efficient Fibonacci function, a classical approach is to implement it as a loop that goes from 0 to `n-1`, storing in two auxiliary variables `x` and `y` the values for `fib n` to `fib (n + 1)` to compute `fib (n + 2)`. The OCaml code and GOSPEL specification are as follows:

```
let fib_imp n =
  let y = ref 0 in
  let x = ref 1 in
  for i = 0 to n - 1 do
    let aux = !y in
    y := !x; x := !x + aux
  done;
  !y
(*@ r = fib_imp n
      requires n >= 0
      ensures  r = fib n *)
```

As expected, `fib_imp` has the same precondition as `fib`. Here, we name the result of the

`fibonacci` function to mention it in the *postcondition*, introduced via the `ensures` clause. This is exactly where we take advantage of the fact that `fib` can be used as a logical function, since we state the returned value of `fib_imp n` is equal to `fib n`.

As usual in deductive verification, the presence of the `for` loop requires us to supply a loop invariant. Here, it boils down to

```
for i = 0 to n - 1 do
  (*@ invariant !y = fib i && !x = fib (i + 1) *)
```

When fed to Cameleer, four verification conditions are generated for the given `fib_impl` implementation: a loop invariant initialization, which states the invariant holds before the first iteration; a loop invariant preservation, which states the invariant holds after each iteration; two postcondition, one accounting for the case when the loop does not even execute ($\mathtt{n} = 0$), the other when the loop performs at least one iteration. All of them are automatically discharged. The complete OCaml implementation of the Fibonacci implementation can be found online, at the project's GitHub repository[2].

The tale of the ghost code. The `fib_imp` function is a provably correct implementation of the Fibonacci function. After finishing the proof, the specification has no computational interest and should no be part of compiled code. This includes the `fib` function which is only useful as a logical definition. This duality between parts of the code that are compiled and other that have only proof interest is a common trait of deductive verification, commonly known as *ghost code* [16]. Some parts of a program are marked with a special *ghost* status and should be erased from the *regular* code after completing the proof effort. In Cameleer, we use the `[@ghost]` attribute in order to change the status of some functions. For the example of the `fib` definition, this is as simple as `let [@logic] [@ghost] rec fib n`. Building a sound mechanism for ghost code erasure is far from a trivial task, especially in the presence of effectful computations [30]. In Sec. 3.3, we discuss several solutions to deal with ghost code that we could put into practice in the scope of the Cameleer project.

3 Methodology

This section gives an overview on the core of the Cameleer tool, from the OCaml code annotated with GOSPEL specification to the generation of an equivalent WhyML program. In Sec. 3.1, we describe how we use the GOSPEL toolchain to attach specification to certain nodes in the OCaml AST. In Sec. 3.2 we define our OCaml to WhyML translation as a set of inference rules. Finally, Sec. 3.3 explains how we are currently using Why3 as an intermediate verification framework.

3.1 Using GOSPEL toolchain

The Cameleer tool relies on the use of the GOSPEL toolchain[3] to parse and manipulate the OCaml abstract syntax tree. It provides a patched version of the OCaml parser that recognizes GOSPEL special comments and converts them to regular OCaml attributes. For instance, the `fib_imp` specification from Sec. 2 is translated into the following post-item attribute:

```
let fib_imp n = ...
[@@gospel ‘‘r = fib_main n requires n >= 0 ensures r = fib n’’]
```

[2] https://github.com/mariojppereira/cameleer/tree/master/examples/fibonacci.ml
[3] https://github.com/vocal-project/vocal

The payload of a GOSPEL attribute is, hence, a string that contains the user-supplied specification[4]. The GOSPEL attributes are processed by a dedicated parser and type-checker [6], where specifications are attached to nodes of a patched version of the OCaml AST. This custom AST is the entry-point for our OCaml to WhyML translation, which we describe next.

3.2 Translation into WhyML

We present our translation from OCaml to WhyML as set of inference rules. All the auxiliary functions and predicates are total definitions. We focus here on a subset of the OCaml and WhyML languages. The complete definitions are depicted in Fig. 1 and Fig. 2, respectively. On the WhyML side, we omit the definition of t which stands for the logical subset of WhyML. For a comprehensive definition of this part of WhyML, we refer the reader to the Why3 reference manual [36, Chap. 7]. We do not intend to use this section as an heavy formalization of our translation, but rather as a comprehensive presentation of the OCaml subset that Cameleer can handle. Cameleer will report a dedicated error message if a user tries to translate an OCaml program that syntactically falls out of the supported fragment. It is worth noting that our translation is purely syntactic, as we build on the GOSPEL toolchain which is based on a PPX approach. In particular, this means that typing the translated OCaml program is left as a task to Why3. Making our translation type-directed, or at least type-aware, is left as future work.

Expressions. Selected OCaml expressions include variables (x ranges over program variables, while f is used for function names), the conditional `if..then..else`, local bindings of (possibly recursive) expressions, function application, records manipulation (for simplicity, we assume every field to be mutable), treatment of exceptions, loop construction, and finally the `assert false` expression. Values include numerical and Boolean constants, as well as anonymous functions where arguments are annotated with a ghost status. We only consider functions as valid recursive definitions and application is limited to the application of a function name to a list of arguments. The latter is just to ease our presentation; the former is due to recursive definitions in WhyML being limited to functions. Finally, $\mathcal{A}$ notation stands for a (possibly empty) placeholder of OCaml attributes, representing the original place in the expression where GOSPEL elements are introduced. For instance, the first $\mathcal{A}$ in a `let..rec` expression can contain the `[@ghost]` and `[@logic]` attribute, while the second one stands for the function specification. We omit the definition of τ and t, respectively from the OCaml and the WhyML sides. The former stands for the grammar of OCaml types, while the latter is the logical subset of WhyML.

The OCaml and WhyML languages are very similar, hence our translation of expressions is mostly an isomorphism. We give the complete set of rules in Appendix A and explain here only the more subtle aspects of this translation. Let us begin with the translation of an OCaml expression of the form `let..rec..in` into its WhyML counterpart. The corresponding translation rule is the following:

$$(\text{ERec})\ \frac{e_0 \overset{function}{\rightsquigarrow} \overline{\beta x}, e_0' \qquad e_1 \overset{expression}{\rightsquigarrow} e_1' \qquad \neg is\_ghost(\mathcal{A}_0) \qquad kind(\mathcal{A}_0) = \mathcal{K} \qquad \mathcal{A}_1 \overset{function\ spec}{\rightsquigarrow} \mathcal{S}_1}{\texttt{let}\ \mathcal{A}_0\ \texttt{rec}\ f_0 = e_0\ \mathcal{A}_1\ \texttt{in}\ e_1 \overset{expression}{\rightsquigarrow} \texttt{rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S}_1 = e_0'\ \texttt{in}\ e_1'}$$

For the sack of presentation, we use here only a single recursive function and omit any mutually recursive definition. In fact, translating a set of mutual definitions simply amounts to a recursive

[4] In fact, GOSPEL elements can be directly provided using attributes in the source code.

e	$::= x \mid \nu \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{match}\ e\ \texttt{with}\ \overline{[]\ p \Rightarrow e}$ $\mid \texttt{let}\ \mathcal{A}\ x = e\ \mathcal{A}\ \texttt{in}\ e$ $\mid \texttt{let}\ \mathcal{A}\ \texttt{rec}\ f = e\ \mathcal{A}\ \overline{\texttt{and}\ f = e\ \mathcal{A}}\ \texttt{in}\ e \mid f\ \bar{e}$ $\mid \{\overline{f = e}\} \mid e.f \mid e.f \leftarrow e$ $\mid \texttt{raise}\ E\bar{e} \mid \texttt{try}\ e\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e}$ $\mid \texttt{while}\ e\ \texttt{do}\ \mathcal{A}\ e\ \texttt{done} \mid \texttt{assert false}$	Expressions
p	$::= \_ \mid x \mid \overline{p} \mid C(p) \mid p\ []\ p \mid p\ \texttt{as}\ x \mid p : \tau \mid \texttt{exception}\ p$	Patterns
ν	$::= n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fun}\ \mathcal{A}\ (x\beta) \rightarrow e$	Values
β	$::= \texttt{reg} \mid \texttt{ghost}$	Ghost attribute
τ	$::= \alpha \mid \tau \rightarrow \tau \mid \overline{\tau} \mid \overline{\tau}\ C$	Type expression
π	$::= \beta\tau$	Type with ghost status
d	$::= \texttt{exception}\ E : \overline{\pi} \mid \texttt{type}\ td\ \overline{\texttt{and}\ td}$ $\mid \texttt{let}\ \mathcal{A}\ f = e\ \mathcal{A} \mid \texttt{let}\ \mathcal{A}\ \texttt{rec}\ f = e\ \mathcal{A}\ \overline{\texttt{and}\ f = e\ \mathcal{A}}$ $\mid \texttt{module}\ \mathcal{M} = m$	Top-level declarations
td	$::= \overline{\alpha}\ T \mid \overline{\alpha}\ T = \tau \mid \overline{\alpha}\ T = \{\ \overline{f : \pi}\ \}\ \mathcal{A} \mid \overline{\alpha}\ T = \overline{[]\ C\ \texttt{of}\ \overline{\tau}}$	Type definition
m	$::= \texttt{struct}\ \overline{d}\ \texttt{end} \mid \texttt{functor}(\mathcal{X} : mt) \rightarrow m$	Modules
mt	$::= \texttt{sig}\ \overline{s}\ \texttt{end}$	Module types
s	$::= \texttt{val}\ \mathcal{A}\ f : \pi\ \mathcal{A} \mid \texttt{type}\ td\ \overline{\texttt{and}\ td}$	Signatures
p	$::= \overline{d}$	Program

Figure 1: Syntax of core OCaml.

call to our expressions translation procedure, as depicted in Appendix A. Let us consider the following generic expression as a running example to explain the (EREC) rule:

```
let rec foo (x [@ghost]) y = e0
(*@ r = foo x y
      requires ... variant  ... ensures  ... *)
in e1
```

The second premise of the rule translates expression `e1` and it simply amounts to a recursive call to the translation scheme. The first premise is a bit more evolving, hence we explain it in more detail. The definition of `foo` is de-sugared by the OCaml parser into the following Curried expression:

```
let rec foo = fun (x [@ghost]) -> fun y -> e
```

When translating into WhyML, we revert such an operation: we traverse the body of `foo`, building a list of ghost-annotated arguments from the argument of each `fun` construction. The body of the translated function is the body of the last `fun`. This is done in the first premise of the rule, using the $\overset{function}{\rightsquigarrow}$ operation. This conversion into multi-argument functions is justified by the limits of Why3 when it comes to higher-order and anonymous functions. In WhyML, one can only define *pure* anonymous functions, *i.e.*, free of any side effects. Hence,

$$
\begin{array}{lll}
e & ::= x \mid \nu \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{match}\ e\ \texttt{with}\ \overline{[]\ p \Rightarrow e}\ \texttt{end} & \text{Expressions} \\
& \mid \texttt{let}\ \mathcal{K}\ \beta x = e\ \texttt{in}\ e & \\
& \mid \texttt{rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S} = e\ \overline{\texttt{with}\ \mathcal{K}\ f(\overline{\beta x}) = e\ \mathcal{S}}\ \texttt{in}\ e \mid f\ \bar{e} & \\
& \mid \{\overline{f = e}\} \mid e.f \mid e.f \leftarrow e & \\
& \mid \texttt{raise}\ E\bar{e} \mid \texttt{try}\ e\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e}\ \texttt{end} & \\
& \mid \texttt{while}\ e\ \texttt{do}\ \mathcal{I}\ e\ \texttt{done} \mid \texttt{absurd} \mid \texttt{ghost}\ e & \\
p & ::= \_ \mid x \mid \overline{p} \mid Cp \mid p\ []\ p \mid p\ \texttt{as}\ x \mid p : \tau \mid \texttt{exception}\ p & \text{Patterns} \\
\nu & ::= n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fun}\ \mathcal{K}\ (\overline{\beta x})\ \mathcal{S} \rightarrow e & \text{Values} \\
\beta & ::= \texttt{reg} \mid \texttt{ghost} & \text{Ghost status} \\
\tau & ::= \alpha \mid \tau \rightarrow \tau \mid \overline{\tau} \mid C\,\overline{\tau} & \text{Type expression} \\
\pi & ::= \beta\tau & \text{Type with ghost status} \\
\mathcal{K} & ::= \texttt{reg} \mid \texttt{logic} & \text{Function kind} \\
\mathcal{S} & ::= \texttt{requires}\ \bar{t}\ \ \texttt{ensures}\ \bar{t}\ \ \texttt{variant}\ \bar{t} & \text{Function specification} \\
\mathcal{I} & ::= \texttt{invariant}\ \bar{t}\ \ \texttt{variant}\ \bar{t} & \text{Loop specification} \\
d & ::= \texttt{exception}\ E : \overline{\pi} \mid \texttt{type}\ td\ \overline{\texttt{with}\ td} & \text{Top-level declarations} \\
& \mid \texttt{let}\ \mathcal{K}\ f = e & \\
& \mid \texttt{let rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S} = e\ \overline{\texttt{with}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S} = e} & \\
& \mid \texttt{val}\ \mathcal{K}\ \beta f(\overline{x : \pi})\ \mathcal{S} : \pi \mid \texttt{scope}\ \mathcal{M}\ \overline{d}\ \texttt{end} & \\
td & ::= T\overline{\alpha} \mid T\overline{\alpha} = \tau \mid T\overline{\alpha} = \{\ \overline{f : \pi}\ \}\ \texttt{invariant}\ \bar{t} \mid T\overline{\alpha} = \overline{[]\ C\,\overline{\tau}} & \text{Type definition} \\
p & ::= \texttt{module}\ \mathcal{M}\ \overline{d}\ \texttt{end} & \text{Program}
\end{array}
$$

Figure 2: Syntax of core WhyML.

directly translating the Curried definition of `foo` would yield a syntactically correct WhyML expression, however this would be rejected by the language type-and-effect system.

The other three premises deal with specification elements. The first uses the *is_ghost* operation to test whether the `[@ghost]` attribute is provided in $\mathcal{A}_0$. If that is the case (rule (ERecGhost) in Appendix A), the function body is translated into `ghost e`. The kind of a function is either `reg` (only usable as a program function) or `logic` (also usable inside specification). We introduce the $kind(\cdot)$ operation, which also retrieves the function kind from $\mathcal{A}_0$ (in case of a regular function, the attribute can be omitted). The last premise translates the supplied GOSPEL preconditions, variants, and postconditions into the WhyML specification language. We omit the definition of $\xrightarrow{function\ spec}$ since this is a trivial (syntactic) transformation.

We highlight two more interesting cases of expressions translation: the `assert false` construction and local non-recursive bindings. Contrarily to any other `assert` expression, `assert false` is used in OCaml to indicate unreachable points in the code and "is treated in a special way by the OCaml type-checker"[5]. WhyML features the `absurd` construction which has the exact same semantics, which greatly simplifies our translation effort (rule (EAbsurd) in Appendix A). Finally, when translating a `let..in` expression, we need to account for both the

[5]To quote a comment from the OCaml compiler source code itself.

introduction of a local function, as well as the binding of a non-functional value. The translation rule for the latter is as follows:

$$\text{(ELET)}\ \frac{\neg \mathit{is\_ghost}(\mathcal{A}) \qquad \neg \mathit{is\_functional}(e_0) \qquad e_0 \overset{\mathit{expression}}{\rightsquigarrow} e_0' \qquad e_1 \overset{\mathit{expression}}{\rightsquigarrow} e_1'}{\texttt{let}\ \mathcal{A}\ x = e_0\ \mathcal{A}'\ \texttt{in}\ e_1 \overset{\mathit{expression}}{\rightsquigarrow} \texttt{let reg}\ x = e_0'\ \texttt{in}\ e_1'}$$

This rule stands for the sub-case where the bound variable is regular, hence the use of the `reg` kind. Any GOSPEL specification possibly contained in $\mathcal{A}'$ is ignored. Finally, the use of the auxiliary predicate *is_functional*$(\cdot)$ is what allows us to distinguish between locally-bound variables and functions. This predicate decides whether e_0 is a `fun x -> ...` expression, in which case we introduce a WhyML local function (rule (ELETFUN) in Appendix A). This approach does not take partial applications into account, which are directly translated into its WhyML counterpart. If a partial application introduces an effectful computation, this will be rejected by the Why3 type system.

Top-level declarations. Selected top-level declarations include exceptions and type declaration, (mutually-recursive) function definition, and introduction of sub-modules. An exception takes a list of π values, types annotated with a ghost status, to account for the possibility of ghost arguments. In Why3 vocabulary, this is the *mask* of an exception [30, Chap. 3.1]. The complete set of translation rules for top-level declarations and type definitions is given in Appendix B. We highlight here the cases of record type definition and sub-modules.

The attribute $\mathcal{A}$ after a record type definition is used to express in GOSPEL a *type invariant*, *i.e.*, a predicate that every inhabitant of such type must satisfy. Type invariants are readily supported by Why3, as depicted in rule (TDRECORD) in Fig. 5, Appendix B. Each field of the record type is also annotated with a ghost status. This is a common practice in deductive verification: some fields act as *logical models* of the record value; these can be explored within the proof to reason about the represented data structure. It is worth noting that in WhyML, contrarily to OCaml, type arguments are introduced on the right-hand side of the type name.

Finally, translation of a sub-module definition is guided by the following rule:

$$\text{(DMODULE)}\ \frac{m \overset{\mathit{module}}{\rightsquigarrow} \overline{d}}{\texttt{module}\ \mathcal{M} = m \overset{\mathit{declaration}}{\rightsquigarrow} \texttt{scope}\ \mathcal{M}\ \overline{d}\ \texttt{end}}$$

We translate the module expression m into a list of WhyML declarations. WhyML does not feature the notion of sub-module, hence we encapsulate the translated declarations into a *scope*, the WhyML unit for namespaces management. In what follows, we provide a more detailed account of the WhyML module system and its differences with respect to that of OCaml.

Modules. The most interesting cases in our translation is how we deal with the modules language from the OCaml side. A WhyML program is a list of modules, a module is a list of top-level declarations, and declarations can be organized within scopes.

The first module expression we take into account is the `struct..end` construction. This is translated into a WhyML declarations, as depicted in rule (MSTRUCT) (Appendix C). We note this does not change the structure and code organization of the original program, since a `struct..end` expression follows a `module` declaration. Hence, a declaration of the form `module` $\mathcal{M}$ $\overline{d}$ `end` is translated into `scope` $\mathcal{M}$ $\overline{d}$ `end`.

Functors are a central notion when programming in OCaml, so it is out of question to develop a verification tool for OCaml without a (at least minimal) support for functors. WhyML does

not feature a syntactic construction for functors; instead, these are represented as modules containing only abstract symbols [19]. Thus, we propose the following translation rule:

$$(\text{MFunctor})\ \frac{mt \overset{\textit{module type}}{\rightsquigarrow} \overline{d} \qquad m \overset{\textit{module}}{\rightsquigarrow} \overline{d'}}{\texttt{functor}(\mathcal{X} : mt) \to m \overset{\textit{module}}{\rightsquigarrow} \texttt{scope}\ \mathcal{X}\ \overline{d}\ \texttt{end}\ \overline{d'}}$$

Each `functor` expression is translated into a WhyML scope followed by a list of declarations. For instance, the OCaml functor `module Make = functor (X: ...) -> struct ... end`, is translated into the following WhyML excerpt: `scope Make scope X ... end ... end`. The given transformation is the dual of what is actually implemented in the Why3 extraction machinery: every WhyML expression of the form `scope A scope B ...` is translated into `module A (B: ...) ...`, *as long as* scope `B` features only abstract symbols.

Signatures. The argument of a functor is expressed as a *module type*, *i.e.*, a *signature* of the form `sig..end`. This encapsulates a list of declarations belonging to the OCaml signature language, which are translated into a list of WhyML expressions, according to rule (MTSig) (Appendix C). Contrarily to OCaml, WhyML does not impose a separation between signature (interface) and structure (implementation) elements. In particular, the WhyML surface language allows one to include non-defined `val` functions and regular `let` definitions in the same namespace. We give the following translation rule for `val` declarations:

$$(\text{SVal})\ \frac{\neg \textit{is\_ghost}(\mathcal{A}) \qquad \textit{kind}(\mathcal{A}) = \mathcal{K} \qquad \mathcal{A}' \overset{\textit{function spec}}{\rightsquigarrow} \mathcal{S} \qquad \pi, \mathcal{A}' \overset{\textit{function args}}{\rightsquigarrow} \overline{x : \pi'}, \pi_{res}}{\texttt{val}\ \mathcal{A}\ f : \pi\ \mathcal{A}' \overset{\textit{signature}}{\rightsquigarrow} \texttt{val}\ \mathcal{K}\ f(\overline{x : \pi})\ \mathcal{S} : \pi_{res}}$$

The name of the arguments are retrieved from the function specification (Sec. 4.2 features an example of such case). Non-defined functions can also be declared as ghost and/or logical functions. For brevity, the case of ghost `val` is omitted. The complete set of translation rule for signature items can be found in Appendix D.

Programs. An OCaml program is simply a list of top-level declarations. These are translated into a WhyML module, as follows:

$$(\text{Program})\ \frac{\overline{d} \overset{\textit{declaration}}{\rightsquigarrow} \overline{d'}}{\overline{d} \overset{\textit{program}}{\rightsquigarrow} \texttt{module}\ \mathcal{M}\ \overline{d'}\ \texttt{end}}$$

The name $\mathcal{M}$ of the generated module is issued from the OCaml file that contains the original program. If file `foo.ml` contains the program p, it gets translated into `module Foo` p `end`. In summary, we generate a WhyML program containing a single module, which represents the top-level module of an OCaml file. In turn, each sub-module is translated into a WhyML scope, with a special treatment for functorial definitions.

3.3 Interaction with Why3

Why3 front-end. One distinguished feature of the Why3 architecture is that it can be extended to accommodate new front-end languages [36, Chap. 4]. Building on the translation scheme presented in previous section, we use the Why3 API to build an in-memory representation of the WhyML program and to register OCaml as an admissible input format for Why3. We can use any Why3 tool, out of the box, to process a `.ml` file. For instance, one could use directly the command `why3 ide bar.ml` to trigger our OCaml input format and to call the Why3 IDE on the translation of the `bar.ml` file.

Other than the `ide`, we could use the `extract` command to erase any trace of ghost from the original code and print an equivalent OCaml implementation. This would provide us with the necessary guarantees about the semantics and typedness of the extracted program. However, we believe a solution that is directly integrated with the OCaml compilation chain is more organic and natural to the programmer. We currently working on a PPX that generates a new OCaml AST without ghost code, which uses the Why3 extraction as an internal ingredient.

Limitations of using Why3. WhyML and GOSPEL are very similar specification language. Moreover, they share some fundamental principals, namely the arguments of functions are not-aliased by construction and each data structure carries an implicit representation predicate. This makes the translation from GOSPEL to WhyML a very natural process. However, one can use GOSPEL to formally specify some OCaml programs which cannot be translated into WhyML. This is much evident when it comes to recursive ephemeral data structures. Consider, for instance, the `cell` type definition from the `Queue` module of the OCaml standard library[6]:

```
type 'a cell = Nil | Cons of { content: 'a; mutable next: 'a cell }
```

As we attempt to translate such data type into WhyML, we get the following error:

```
This field has non-pure type, it cannot be used in a recursive type definition
```

Recursive mutable data types are beyond the scope of Why3's type-and-effect discipline [15]. The solution would be to resort to an axiomatic memory model of OCaml in Why3 [20], or to employ a richer program logic, *e.g.*, Separation Logic [33] or Implicit Dynamic Frames [34]. We leave such an extension to the Cameleer infrastructure for future work.

4 Case Studies

4.1 FIFO Queue

Our first case study is the implementation of a FIFO queue, implemented as an ephemeral data-structure. The complete OCaml development and GOSPEL specification are presented at Cameleer's GitHub repository [7]. We also publish online a simpler, purely applicative version of this data structure [8], which we picked from the OCamlGraph library[9]. This case study follows the standard approach of using a pair of lists to store the elements of the queue, as follows:

```
type 'a t = { mutable front: 'a list; mutable rear : 'a list;
              mutable view : 'a list [@ghost]; }
(*@ invariant (front = [] -> rear = []) && view = front ++ List.rev rear *)
```

In order to formally express the behavior of the queue data structure, we equip type `t` with a model field and a GOSPEL invariant. The `view` field is used to represent the whole queue as a single list. This is a ghost field since it has no computational interest. Front elements are stored in the correct order and rear elements are stored in *reversed* order. Elements are pushed into the head of the rear list and popped off the head of front. This data structure also maintains the *invariant* that if `front` is empty, then so is `rear`.

In what follows, the specification of operations on a queue is solely given in terms of the `view` field. The `push` operation is implemented and specified as follows:

[6] https://caml.inria.fr/pub/docs/manual-ocaml/libref/Queue.html
[7] https://github.com/mariojppereira/cameleer/blob/master/examples/ephemeral_queue.ml
[8] https://github.com/mariojppereira/cameleer/blob/master/examples/applicative_queue.ml
[9] https://github.com/backtracking/ocamlgraph

```
let push x q =
  if is_empty q then q.front <- [x] else q.rear <- x :: q.rear;
  q.view <- q.view @ [x]
(*@ push x q
      ensures q.view = (old q.view) @ [x] *)
```

The postcondition asserts the updated `view` field of `q` consists of the value of `view` before the call (`old q.view`), extended with the new element `x`. If the queue is empty, then `x` is pushed into the `front` list; otherwise, the new element is added as the new head of `rear`.

Next, we present the implementation of the `pop` operation. This is as follows:

```
let pop q = match q.front with
  | [] -> raise Not_found
  | [x] ->
      q.front <- List.rev q.rear; q.rear <- []; q.view <- tail_list q.view;
      x
  | x :: f ->
      q.front <- f; q.view <- tail_list q.view;
      x
(*@ x = pop q
      raises  Not_found -> is_empty (old q)
      ensures x :: q.view = (old q).view *)
```

If the queue is empty we raise the `Not_found` exception from the OCaml standard library. A `raises` clause is used to introduce what we call an *exceptional postcondition*. We must be careful enough to specify that the `is_empty` property is verified by pre-state of `q`. Otherwise, had we used `is_empty q`, this would propagate into our proof context that the queue is not empty *after* the execution of `pop`. This would prevent proving the safety of the next function.

The most interesting function in our development of ephemeral queues is the concatenation of two such queues. The implementation is as follows:

```
let transfer (q1: 'a t) (q2: 'a t) : unit =
  while not (is_empty q1) do push (pop q1) q2 done
(*@ transfer q1 q2
      raises   Not_found -> false
      ensures  q1.view = [] && q2.view = old q2.view @ old q1.view *)
```

The `transfer` operation takes queues `q1` and `q2` as arguments, and migrates the elements of the former to the end of the latter. Moreover, it clears the contents of its first argument. By design, GOSPEL assumes `q1` and `q2` are two separated queues, *i.e.*, not aliased. The `raises` clause states no `Not_found` exception is raised during execution of `transfer`. In fact, since the `while` loop is guarded by the `not (is_empty q1)` property, we know it is safe to call `pop q1`. Feeding this program to Cameleer generates a total of 8 VCs, from which we are only able to prove the exceptional postcondition and the first clause of the regular postcondition condition. With no surprise, this comes from the fact that we are missing a suitable loop invariant. In what follows, we refine the specification to completely prove the `transfer` function.

In order to prove termination, we use the length of the `view` model from `q1` as a decreasing measure, as follows:

```
(*@ variant List.length q1.view *)
```

In order to prove the remaining VCs, it comes with no surprise that we need a suitable loop

invariant. First, we turn the type invariant property into a loop invariant, as follows:

```
(*@ invariant (q1.front = [] -> q1.rear = []) &&
              (q2.front = [] -> q2.rear = []) *)
(*@ invariant q1.view = q1.front @ List.rev q1.rear &&
              q2.view = q2.front @ List.rev q2.rear *)
```

Although this makes the loop invariant more cumbersome, it is actually almost a mechanical process to include the type invariant in the loop invariant. With such a specification, we are able to discharge every VC of `transfer`, except for the second postcondition.

To complete this proof, we need to be a little more creative. The idea is the following: if we pick an arbitrary loop iteration, at that point of execution we would have already transferred a prefix of the elements from `q1` and would have concatenated those into `q2`. In order to represent such prefix of `q1`, we introduce an auxiliary ghost variable, as follows:

```
let transfer q1 q2 =
  let [@ghost] done_view = ref [] in ...
```

At each iteration, we must update the value of `done_view` as follows:

```
  while not (is_empty q1) do (*@ variant ... invariant ... *)
    done_view := !done_view @ [head_list q1.view];
    push (pop q1) q2
done
```

Once again, the `done_view` is a ghost variable, so any part of the program that manipulates it should be erased from the code. Thus, the above assignment should not incur a penalty on the execution time of `transfer`. Finally, we complete the loop invariant as follows:

```
(*@ invariant old q1.view = !done_view @ q1.view *)
(*@ invariant q2.view = old q2.view @ !done_view *)
```

The first condition maintains that `todo_view` is indeed a prefix of `q1.view`; the second condition states that the current state of `q2` consists of the elements from `todo_view` concatenated to the sequence of original elements of `q2`. With the updated invariant, we are finally able to discharge every generated VC for `transfer`.

4.2 Leftist Heap

Functor definition. The next case study is the implementation of a *heap* data structure. We adopt the *leftist heap* variant [11, 26], which we picked from the OCaml-containers library[10]. This is an applicative implementation, following the approach by C. Okasaki [29, Chap. 3.1].

The fundamental interest of using heaps is to be able to quickly access the minimum element of the collection. Thus, the elements of this data structure must be equipped with a `total` preorder, which is crucial to guarantee the correct behavior of the heap implementation. In OCaml, such kind of restrictions on types are naturally implemented using functors. For leftist heaps, we begin by introducing the following *module type* to represent a total preorder:

```
module type TOTAL_PRE_ORD = sig
  type t
  (*@ function le : t -> t -> bool *)
  (*@ axiom reflexive : forall x. le x x *)
```

[10] https://github.com/c-cube/ocaml-containers

```
  (*@ axiom total     : forall x y. le x y \/ le y x *)
  (*@ axiom transitive: forall x y z. le x y -> le y z -> le x z *)

  val leq : t -> t -> bool
  (*@ b = leq x y
        ensures b <-> le x y *)
end
```

Using GOSPEL, we introduce a purely logical function `le` defined using the classic axioms of reflexivity, totality, and transitivity. Next, the specification of the regular function `leq` should be read as "this function implements the logical function `le`". Using the above `TOTAL_PRE_ORD` type, the implementation of leftist heaps is encapsulated in the following functor `Make`:

```
module Make(E : TOTAL_PRE_ORD) = struct type elt = E.t ... end
```

The type equation `elt = E.t` ensures that elements of type `elt` (used to represent the elements of the heap) are inherit the total preorder relation from module `E`.

Leftist property. We use the following data type definition to represent leftist heaps:

```
type t = E | N of int * elt * t * t
```

A leftist heap is represented as a binary tree where each node is attached a *rank* value. Generally speaking, the rank of a node is defined as the length of the shortest path from that node to an empty node. In GOSPEL, this is as simple as follows:

```
(*@ function rank (h: t) : integer =
      match h with E -> 0 | N _ _ l r -> 1 + min (rank l) (rank r) *)
```

Using the general notion of rank, one can define the notion of *leftist property*: the rank of any left child is always greater or equal to the rank of the right sibling. This is captured by the following GOSPEL definition:

```
(*@ predicate leftist (h: t) = match h with
      | E -> true
      | N n _ l r ->
          n = rank h && leftist l && leftist r && rank l >= rank r *)
```

This property also gives the value of the element storing the rank in the structure. Other than the `leftist` property, leftist heaps should obey the general laws of heaps: the element in each node is less or equal to the elements at its children. The GOSPEL definition is as follows:

```
(*@ predicate is_heap (h: t) = match h with
      | E -> true
      | N _ x l r -> le_root x l && is_heap l && le_root x r && is_heap r *)
```

where `le_root` is a predicate that states a given element is less or equal to the root of a heap:

```
(*@ predicate le_root (e: elt) (h: t) =
      match h with E -> true | N _ x _ _ -> E.le e x *)
```

Finally, we define what is a leftist heap:

```
(*@ predicate leftist_heap (h: t) = is_heap h && leftist h *)
```

Logical definition of minimum element. When describing the logical behavior of a heap data structure, one must pay particular attention to the definition of the minimum element. We need to define what is the *physical* minimum element of the heap, as follows:

```
(*@ function minimum (h: t) : elt *)
(*@ axiom minimum_def: forall l x r n. minimum (N n x l r) = x *)
```

The `minimum` function is only significantly defined for the case of non-empty heaps[11]. We have now the necessary building blocks to formally specify leftist heaps operations.

Heap operations. We present here only two heap operations, `_make_node` and `merge`. The complete OCaml development can be found online[12]. The first function builds a new node, given an element `x` and subtrees `a` and `b`:

```
let _make_node x a b =
  if _rank a >= _rank b then N (_rank b + 1, x, a, b)
  else N (_rank a + 1, x, b, a)
(*@ h = _make_node x a b
    requires leftist_heap a && leftist_heap b && le_root x a && le_root x b
    ensures  leftist_heap h && minimum h = x
    ensures  occ x h = 1 + occ x a + occ x b
    ensures  forall y. x <> y -> occ y h = occ y a + occ y b *)
```

The leftist property is ensured by the `if..then..else` expression, where `_rank` is a function that simply retrieves the rank of a node, *i.e.*, the first argument of the `N` constructor or 0 in case the heap is empty. Both `a` and `b` must be `leftist_heap`s and `x` must no greater than the roots of the two arguments. We give a specification of the heap in terms of the multiset of its elements. The resulting heap `h` is a `leftist_heap`, with minimum element `x`. The number of occurrences of `x` increases by one, whereas the occurrences of any element different from `x` remain the same. Finally, the nuclear operation `merge` is defined as follows:

```
let rec merge t1 t2 = match t1, t2 with
  | t, E | E, t -> t
  | N (_, x, a1, b1), N (_, y, a2, b2) ->
      if E.leq x y then _make_node x a1 (merge b1 t2)
      else _make_node y a2 (merge t1 b2)
(*@ h = merge t1 t2
      requires leftist_heap t1 && leftist_heap t2
      variant  size t1 + size t2
      ensures  leftist_heap t && forall x. occ x h = occ x t1 + occ x t2 *)
```

If the root `x` of heap `t1` is no greater than the root of `t1`, we build a new heap with root `x`; otherwise, we the new root is the root of `t2`. The call to function `_make_node` ensures the `leftist_heap` property for the returned heap. We use the logical function `size` in the variant of `merge`, which is the straightforward definition of the number of nodes in a heap.

Addition and removal of the minimal element are straightforwardly defined. Due to space constraints, we do not present those here and refer the reader to the online repository for the complete OCaml development. This also includes the `filter` and `delete_all` higher-order functions. All the generated VCs for leftist heap operations were automatically discharged.

4.3 Other case studies

Table 1 in Appendix E summarizes the case studies performed with Cameleer. The second column features the number of non-blank lines of OCaml code, while the third one stands for

[11] In GOSPEL, logical functions are total. The value returned by `minimum E` exists, but one can prove no property about it.

[12] https://github.com/mariojppereira/cameleer/blob/master/examples/leftist_heap.ml

the number of non-blank lines of GOSPEL specification. We have put an effort to use Cameleer to verify programs of different natures. These include numerical programs (binary multiplication, different factorial implementations, fast exponentiation, and integer square root), sorting and searching algorithms, data structures implemented as functors, historical algorithms (checking a large routine by Turing, Boyer-Moore's majority algorithm, and binary tree same fringe), and logical algorithms (conversion of a propositional formula into conjunctive normal form).

5 Related Work

Automated deductive verification tools. One can cite Why3, F* [1], Dafny [27], and Viper [28] as well-succeed automated deductive verification tools. Formal proofs are conducted in the proof-aware language of these frameworks, and then executable reliable code can be automatically extracted. In the Cameleer project, we chose to develop a verification tool that accepts as input a program written directly in OCaml, instead of a dedicated proof language. Our specification language, GOSPEL, is very close to the OCaml language itself, hence we believe this does not impose a big burden for the regular OCaml practitioner.

Regarding verification tools that tackle the verification of programs written in mainstream languages, we can cite Frama-C [24] and VeriFast [22]. The former is a framework for the static analysis of C code; the latter can be used to verify functional correctness of C and Java programs. Despite the remarkable case studies verified with these tools, C and Java code can quickly degenerate into a nightmare of pointer manipulation and tricky semantic issues. We argue OCaml is a language better suited for formal verification.

Deductive verification of OCaml programs. To the best of our knowledge, CFML [5] was the only tool available for the deductive verification of code directly written in OCaml. It takes as input an OCaml program and translates it into Coq, together with its *characteristic formulae*. A characteristic formulae is a higher-order statement that captures the semantics of the original program. Proofs are conducted using an embedding of Separation Logic inside Coq. The CFML tool has already been used to verify several interesting OCaml modules, including ephemeral data structures and higher-order functions. Recently, CFML was extended with support for *time credits* and it was successfully applied in the verification of functional correctness and time complexity claims of non-trivial data structures and algorithms [8, 21].

The VOCaL project aims at developing a mechanically verified OCaml library [7]. One of the main novelties of this project is the combined use of three different verification tools: Why3, CFML, and Coq. The GOSPEL specification language was developed in the scope of this project, as a tool-agnostic language that could be manipulated by any of the three mentioned frameworks. It was our participation on the VOCaL project that inspired us, in the first place, to develop Cameleer. We believe Cameleer can be readily included in the VOCaL ecosystem, complementing the toolchains [14] that are currently used to build the verified library.

6 Conclusions and Future Work

In this paper we presented Cameleer, a tool for deductive verification of OCaml-written code. The core of Cameleer is a translation from OCaml annotated code into WhyML, the programming and specification language of the Why3 verification framework. OCaml and WhyML have many common traits (both in their syntax and semantics), which provides us with good guarantees about soundness of Cameleer translation. We have already applied Cameleer to successfully verify functional correctness and safety of 14 realistic OCaml modules. These include implementations issued from existing libraries, and scale up to data structures implemented as functors

and tricky effectful computations. These results encourage us to continue developing Cameleer and apply it to the verification of larger case studies.

What we do not support. Currently, we target a subset of the OCaml language which roughly corresponds to `caml-light`, with basic support for the module language (including functors). Also, WhyML limits effectful computations to the cases where alias is statically known, which limits our support for higher-order functions and mutable recursive data structures. Adding support for the objective layer of the OCaml language would require a major extension to the GOSPEL language and a redesign of our translation into WhyML. Nonetheless, Why3 has been used in the past to verify Java-written programs [17], so in principle an encoding of OCaml objects in WhyML is possible.

GADTs extend usual algebraic data types with a lightweight form of dependent typing in OCaml. The use of GADTs allows one to logically constraint the values that can inhabit a type in a certain point of the program. The WhyML type system does not, currently, include a notion close to GADTs. However, since this a proof-aware language an interesting route for future work would be to explore if it is possible to encode in WhyML the reasoning of GADTs without extending its type system. In particular, if it would be possible to leverage on the notion of type invariant to achieve similar results as those statically provided by GADTs.

Another interesting feature of OCaml we currently do not support are polymorphic variants. Polymorphic variants are more flexible than ordinary variants, as they are not tied to a particular type declaration and can be easily extended according to use scenarios. This flexibility, however, leads to a more complicated typing process for polymorphic variants, when compared to regular ones. Once again, extending Cameleer to deal with polymorphic variants requires extending Why3 itself. This would likely mean a considerable redesign of its type system.

Finally, Why3 constrains the use of higher-order functions to pure computations and this is also the case with Cameleer. A possible solution for such limitation would be to use defunctionalization to convert an higher-order program into an equivalent first-order one. In previous work, we have already explored the use of defunctionalization for verification of stateful higher-order programs in Why3 [31]. However, defunctionalization is a whole-program transformation which severally constrains its applicability. Next, we describe a more robust solution, which amounts at using a richer verification framework to reason about higher-order with effects.

Interface with Viper and CFML. We want to keep improving Cameleer with the ability to verify a growing class of OCaml implementations. This includes pointer-based data structures and effectful higher-order computations. Given the limitations of Why3 to deal with such class of programs, we believe the solution is to extend Cameleer to include translation into different intermediate verification languages. We are considering targeting the Viper infrastructure and the CFML tool. On one hand, Viper is an intermediate verification language based on Separation Logic but oriented towards SMT-based software verification, allowing one to automatically verify heap-dependent programs. On the other hand, the CFML tool already provides a translation from OCaml into Coq, allowing one to verify effectful higher-order functions. Even if it relies on an interactive proof assistant, CFML provides a comprehensive library of tactics that ease the proof effort. Our ultimate goal is to turn Cameleer into a verification tool that can simultaneously benefit from the best features of different verification frameworks. Our motto: we want Cameleer to be able to verify parts of an OCaml module using Why3, others with Viper, and finally some specific functions with CFML.

Acknowledgements. We thank Jean-Christophe Filliâtre and the anonymous reviewers for feedback on previous versions of this paper. We also thank Simon Cruanes for encouraging us to prove the Leftist Heaps implementation from the `ocaml-containers` library.

References

[1] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 515–529. ACM, January 2017.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[3] Benedikt F. H. Becker, Nicolas Jeannerod, Claude Marché, Yann Régis-Gianas, Mihaela Sighireanu, and Ralf Treinen. Analysing installation scenarios of debian packages. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 235–253. Springer, 2020.

[4] Bernhard Beckert and Michał Moskal. Deductive verification of system software in the Verisoft XT project. *KI*, 24(1):57–61, February 2010.

[5] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, pages 418–430, 2011.

[6] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. GOSPEL — Providing OCaml with a Formal Specification Language. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, pages 484–501. Springer, 2019.

[7] Arthur Charguéraud, Jean-Christophe Filliâtre, Mário Pereira, and François Pottier. VOCAL – A Verified OCAml Library, September 2017. ML Family Workshop 2017.

[8] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, 62(3):331–365, 2019.

[9] Martin Clochard, Claude Marché, and Andrei Paskevich. Deductive verification with ghost monitors. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[10] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018.

[11] Clark Allan Crane. *Linear Lists and Priority Queues as Balanced Binary Trees.* PhD thesis, Stanford, CA, USA, 1972. AAI7220697.

[12] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[13] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):397–403, August 2011.

[14] Jean-Christophe Filliâtre, Léon Gondelman, Cláudio Lourenço, Andrei Paskevich, Mário Pereira, Simão Melo De Sousa, and Aymeric Walch. A Toolchain to Produce Verified OCaml Libraries. working paper or preprint, January 2020.

[15] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A pragmatic type system for deductive verification. Research report, Université Paris Sud, 2016. https://hal.archives-ouvertes.fr/hal-01256434v3.

[16] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.

[17] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[18] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[19] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in why3. In Tiziana Margaria and Bernhard Steffen, editors, *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 12476 of *Lecture Notes in Computer Science*, Rhodes, Greece, October 2020. Springer. http://why3.lri.fr/isola-2020/.

[20] Jean-Christophe Filliâtre, Mário Pereira, and Simão Melo de Sousa. Vérification de programmes fortement impératifs avec Why3. In Sylvie Boldo and Nicolas Magaud, editors, *Vingt-neuvièmes Journées Francophones des Langages Applicatifs*, Banyuls-sur-mer, France, January 2018.

[21] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal proof and analysis of an incremental cycle detection algorithm. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[22] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[23] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015.

[24] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects Comput.*, 27(3):573–609, 2015.

[25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[26] Donald E Knuth. *The art of computer programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998.

[27] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[28] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

[29] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[30] Mário José Parreira Pereira. *Tools and Techniques for the Verification of Modular Stateful Code.* Theses, Université Paris Saclay (COmUE), December 2018.

[31] Mário Pereira. Défonctionnaliser pour prouver. In Sylvie Boldo and Julien Signoles, editors, *Vingt-huitièmes Journées Francophones des Langages Applicatifs*, Gourette, France, January 2017.

[32] Yann Régis-Gianas and François Pottier. A hoare logic for call-by-value functional programs. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings*, volume 5133 of *Lecture Notes in Computer Science*, pages 305–335. Springer, 2008.

[33] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, page 55–74, USA, 2002. IEEE Computer Society.

[34] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming*, pages 148–172. Springer, 2009.

[35] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified cakeml compiler backend. *J. Funct. Program.*, 29:e2, 2019.

[36] The Why3 Development Team. *The Why3 platform, version 1.3.3.* LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 1.3.3 edition, September 2020. http://why3.lri.fr/manual.pdf.

A Translation of expressions

$$\text{(EAbsurd)}\ \frac{}{\texttt{assert false} \overset{expression}{\rightsquigarrow} \texttt{absurd}}$$

$$\text{(EFun)}\ \frac{\texttt{ghost}(\mathcal{A}) = \beta \qquad \mathcal{A} \overset{function\ spec}{\rightsquigarrow} \mathcal{S} \qquad e \overset{expression}{\rightsquigarrow} e'}{\texttt{fun}\ \mathcal{A}\ x \to e \overset{expression}{\rightsquigarrow} \texttt{fun}\ (\beta x)\ \mathcal{S} = e'} \qquad \frac{\overline{e} \overset{expression}{\rightsquigarrow} \overline{e'}}{f\,\overline{e} \overset{expression}{\rightsquigarrow} f\,\overline{e'}}\ \text{(EApp)}$$

$$\text{(ERec)}\ \frac{\begin{array}{c}\neg is\_ghost(\mathcal{A}_0) \qquad kind(\mathcal{A}_0) = \mathcal{K} \qquad \mathcal{A}_1 \overset{function\ spec}{\rightsquigarrow} \mathcal{S}_1 \qquad e_0 \overset{function}{\rightsquigarrow} \overline{x\beta}, e'_0 \\ \overline{e_1} \overset{function}{\rightsquigarrow} \overline{\beta y}, \overline{e'_1} \qquad \overline{\mathcal{A}_2} \overset{function\ spec}{\rightsquigarrow} \overline{\mathcal{S}_2} \qquad e_2 \overset{expression}{\rightsquigarrow} e'_2\end{array}}{\texttt{let}\ \mathcal{A}_0\ \texttt{rec}\ f_0 = e_0\ \mathcal{A}_1\ \overline{\texttt{and}\ f_1 = e_1\ \mathcal{A}_2}\ \texttt{in}\ e_2 \overset{expression}{\rightsquigarrow} \texttt{rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S}_1 = e'_0\ \overline{\texttt{with}\ \mathcal{K}\ f_1(\overline{\beta y}) = e'_1\ \mathcal{S}_2}\ \texttt{in}\ e'_2}$$

$$\text{(ERecGhost)}\ \frac{\begin{array}{c}is\_ghost(\mathcal{A}_0) \qquad kind(\mathcal{A}_0) = \mathcal{K} \qquad \mathcal{A}_1 \overset{function\ spec}{\rightsquigarrow} \mathcal{S}_1 \qquad e_0 \overset{function}{\rightsquigarrow} \overline{x\beta}, e'_0 \\ \overline{e_1} \overset{function}{\rightsquigarrow} \overline{\beta y}, \overline{e'_1} \qquad \overline{\mathcal{A}_2} \overset{function\ spec}{\rightsquigarrow} \overline{\mathcal{S}_2} \qquad e_2 \overset{expression}{\rightsquigarrow} e'_2\end{array}}{\texttt{let}\ \mathcal{A}_0\ \texttt{rec}\ f_0 = e_0\ \mathcal{A}_1\ \overline{\texttt{and}\ f_1 = e_1\ \mathcal{A}_2}\ \texttt{in}\ e_2 \overset{expression}{\rightsquigarrow} \texttt{rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S}_1 = \texttt{ghost}\ e'_0\ \overline{\texttt{with}\ \mathcal{K}\ f_1(\overline{\beta y}) = e'_1\ \mathcal{S}_2}\ \texttt{in}\ e'_2}$$

$$\text{(ELet)}\ \frac{\begin{array}{c}\neg is\_ghost(\mathcal{A}) \qquad \texttt{kind}(\mathcal{A}) = \mathcal{K} \qquad \neg is\_functional(e_0) \\ e_0 \overset{expression}{\rightsquigarrow} e'_0 \qquad e_1 \overset{expression}{\rightsquigarrow} e'_1\end{array}}{\texttt{let}\ \mathcal{A}\ x = e_0\ \mathcal{A}'\ \texttt{in}\ e_1 \overset{expression}{\rightsquigarrow} \texttt{let}\ \mathcal{K}\ x = e'_0\ \texttt{in}\ e'_1}$$

$$\text{(ELetGhost)}\ \frac{\begin{array}{c}is\_ghost(\mathcal{A}) \qquad \texttt{kind}(\mathcal{A}) = \mathcal{K} \qquad \neg is\_functional(e_0) \\ e_0 \overset{expression}{\rightsquigarrow} e'_0 \qquad e_1 \overset{expression}{\rightsquigarrow} e'_1\end{array}}{\texttt{let}\ \mathcal{A}\ x = e_0\ \mathcal{A}'\ \texttt{in}\ e_1 \overset{expression}{\rightsquigarrow} \texttt{let}\ \mathcal{K}\ x = \texttt{ghost}\ e'_0\ \texttt{in}\ e'_1}$$

$$\text{(ELetGhostFun)}\ \frac{\begin{array}{c}is\_ghost(\mathcal{A}) \qquad kind(\mathcal{A}) = \mathcal{K} \qquad is\_functional(e_0) \\ \mathcal{A}' \overset{function\ spec}{\rightsquigarrow} \mathcal{S} \qquad e_0 \overset{function}{\rightsquigarrow} \overline{\beta x}, e'_0 \qquad e_1 \overset{expression}{\rightsquigarrow} e'_1\end{array}}{\texttt{let}\ \mathcal{A}\ f = e_0\ \mathcal{A}'\ \texttt{in}\ e_1 \overset{expression}{\rightsquigarrow} \texttt{let}\ \mathcal{K}\ f = \texttt{fun}(\overline{\beta x})\ \mathcal{S} \to \texttt{ghost}\ e'_0\ \texttt{in}\ e'_1}$$

$$\text{(ELetFun)}\ \frac{\begin{array}{c}\neg is\_ghost(\mathcal{A}) \qquad kind(\mathcal{A}) = \mathcal{K} \qquad is\_functional(e_0) \\ \mathcal{A}' \overset{function\ spec}{\rightsquigarrow} \mathcal{S} \qquad e_0 \overset{function}{\rightsquigarrow} \overline{x\beta}, e'_0 \qquad e_1 \overset{expression}{\rightsquigarrow} e'_1\end{array}}{\texttt{let}\ \mathcal{A}\ f = e_0\ \mathcal{A}'\ \texttt{in}\ e_1 \overset{expression}{\rightsquigarrow} \texttt{let}\ \mathcal{K}\ f = \texttt{fun}(\overline{\beta x})\ \mathcal{S} \to e'_0\ \texttt{in}\ e'_1}$$

$$\text{(EWhile)}\ \frac{\mathcal{A} \overset{loop\ annotation}{\rightsquigarrow} \mathcal{I} \qquad e_0 \overset{expression}{\rightsquigarrow} e'_0 \qquad e_1 \overset{expression}{\rightsquigarrow} e'_1}{\texttt{while}\ e_0\ \texttt{do}\ \mathcal{A}\ e_1\ \texttt{done} \overset{expression}{\rightsquigarrow} \texttt{while}\ e'_0\ \texttt{do}\ \mathcal{I}\ e'_1\ \texttt{done}}$$

$$\text{(EIf)}\ \frac{e_0 \overset{expression}{\rightsquigarrow} e'_0 \qquad e_1 \overset{expression}{\rightsquigarrow} e'_1 \qquad e_2 \overset{expression}{\rightsquigarrow} e'_2}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \overset{expression}{\rightsquigarrow} \texttt{if}\ e'_0\ \texttt{then}\ e'_1\ \texttt{else}\ e'_2}$$

$$\text{(EMatch)}\ \frac{e_0 \overset{expression}{\rightsquigarrow} e'_0 \qquad \overline{e_1} \overset{expression}{\rightsquigarrow} \overline{e'_1}}{\texttt{match}\ e_0\ \texttt{with}\ \overline{[]\ p \Rightarrow e_1} \overset{expression}{\rightsquigarrow} \texttt{match}\ e'_0\ \texttt{with}\ \overline{[]\ p \Rightarrow e'_1}\ \texttt{end}}$$

$$\text{(ERaise)}\ \frac{\bar{e} \overset{expression}{\rightsquigarrow} \bar{e}'}{\texttt{raise}\ E\bar{e} \overset{expression}{\rightsquigarrow} \texttt{raise}\ E\bar{e}'} \qquad \frac{e \overset{expression}{\rightsquigarrow} e' \qquad \overline{E\bar{x} \Rightarrow e} \overset{expression}{\rightsquigarrow} \overline{E\bar{x} \Rightarrow e'}}{\texttt{try}\ e\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \overset{expression}{\rightsquigarrow} \texttt{try}\ e'\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e'}\ \texttt{end}}\ \text{(ETry)}$$

Figure 3: Translation of OCaml expressions into WhyML.

B Translation of top-level declarations and type definitions

$$\text{(DMODULE)}\ \frac{m \overset{module}{\rightsquigarrow} \overline{d}}{\texttt{module}\ \mathcal{M} = m \overset{declaration}{\rightsquigarrow} \texttt{scope}\ \mathcal{M}\ \overline{d}\ \texttt{end}}$$

$$\text{(DLETGHOST)}\ \frac{is\_ghost(\mathcal{A}) \qquad kind(\mathcal{A}') = \mathcal{K} \qquad is\_functional(e) \qquad \mathcal{A} \overset{function\ spec}{\rightsquigarrow} \mathcal{S} \qquad e \overset{function}{\rightsquigarrow} \overline{\beta x}, e'}{\texttt{let}\ \mathcal{A}\ f = e\ \mathcal{A}' \overset{declaration}{\rightsquigarrow} \texttt{let}\ f\ \mathcal{K} = \texttt{fun}(\overline{\beta x})\ \mathcal{S} \to \texttt{ghost}\ e'}$$

$$\text{(DLET)}\ \frac{\neg is\_ghost(\mathcal{A}) \qquad kind(\mathcal{A}') = \mathcal{K} \qquad is\_functional(e) \qquad \mathcal{A} \overset{function\ spec}{\rightsquigarrow} \mathcal{S} \qquad e \overset{function}{\rightsquigarrow} \overline{\beta x}, e'}{\texttt{let}\ \mathcal{A}\ f = e\ \mathcal{A}' \overset{declaration}{\rightsquigarrow} \texttt{let}\ f\ \mathcal{K} = \texttt{fun}(\overline{\beta x})\ \mathcal{S} \to\ e'}$$

$$\text{(DREC)}\ \frac{\neg is\_ghost(\mathcal{A}_0) \qquad kind(\mathcal{A}_0) = \mathcal{K} \qquad \mathcal{A}_1 \overset{function\ spec}{\rightsquigarrow} \mathcal{S}_1 \qquad e_0 \overset{function}{\rightsquigarrow} \overline{x\beta}, e_0' \qquad \overline{e_1} \overset{function}{\rightsquigarrow} \overline{y : \pi}, \overline{e_1'} \qquad \overline{\mathcal{A}_2} \overset{function\ spec}{\rightsquigarrow} \overline{\mathcal{S}_2}}{\texttt{let}\ \mathcal{A}_0\ \texttt{rec}\ f_0 = e_0\ \mathcal{A}_1\ \overline{\texttt{and}\ f_1 = e_1\ \mathcal{A}_2} \overset{declaration}{\rightsquigarrow} \texttt{rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S}_1 = e_0'\ \overline{\texttt{with}\ \mathcal{K}\ f_1(\overline{y : \pi}) = e_1'\ \mathcal{S}_2}}$$

$$\text{(DRECGHOST)}\ \frac{is\_ghost(\mathcal{A}_0) \qquad kind(\mathcal{A}_0) = \mathcal{K} \qquad \mathcal{A}_1 \overset{function\ spec}{\rightsquigarrow} \mathcal{S}_1 \qquad e_0 \overset{function}{\rightsquigarrow} \overline{x\beta}, e_0' \qquad \overline{e_1} \overset{function}{\rightsquigarrow} \overline{y : \pi}, \overline{e_1'} \qquad \overline{\mathcal{A}_2} \overset{function\ spec}{\rightsquigarrow} \overline{\mathcal{S}_2}}{\texttt{let}\ \mathcal{A}_0\ \texttt{rec}\ f_0 = e_0\ \mathcal{A}_1\ \overline{\texttt{and}\ f_1 = e_1\ \mathcal{A}_2} \overset{declaration}{\rightsquigarrow} \texttt{rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S}_1 = \texttt{ghost}\ e_0'\ \overline{\texttt{with}\ \mathcal{K}\ f_1(\overline{y : \pi}) = e_1'\ \mathcal{S}_2}}$$

$$\text{(DTYPE)}\ \frac{td_0 \overset{type\ definition}{\rightsquigarrow} td_0' \qquad \overline{td_1} \overset{type\ definition}{\rightsquigarrow} \overline{td_1'}}{\texttt{type}\ td_0\ \overline{\texttt{and}\ td_1} \overset{declaration}{\rightsquigarrow} \texttt{type}\ td_0'\ \overline{\texttt{with}\ td_1'}}$$

$$\text{(DEXN)}\ \frac{}{\texttt{exception}\ E : \overline{\pi} \overset{declaration}{\rightsquigarrow} \texttt{exception}\ E : \overline{\pi}}$$

Figure 4: Translation of OCaml top-level declarations into WhyML.

$$\text{(TDABSTRACT)}\ \frac{}{\overline{\alpha}\ T \overset{type\ definition}{\rightsquigarrow} T\overline{\alpha}} \qquad \frac{}{\overline{\alpha}\ T = \tau \overset{type\ definition}{\rightsquigarrow} T\overline{\alpha} = \tau}\ \text{(TDALIAS)}$$

$$\text{(TDRECORD)}\ \frac{\mathcal{A} \overset{type\ invariant}{\rightsquigarrow} \overline{t}}{\overline{\alpha}\ T = \{\ \overline{f : \pi}\ \}\ \mathcal{A} \overset{declaration}{\rightsquigarrow} T\overline{\alpha} = \{\ \overline{f : \pi}\ \}\ \texttt{invariant}\ \overline{t}}$$

$$\text{(TDVARIANT)}\ \frac{}{\overline{\alpha}\ T = \overline{[]\ C\ \texttt{of}\ \overline{\tau}} \overset{type\ definition}{\rightsquigarrow} T\overline{\alpha} = \overline{[]\ C\ \overline{\tau}}}$$

Figure 5: Translation of OCaml type definitions into WhyML.

C Translation of module expressions and module types

$$\text{(MStruct)}\ \frac{\overline{d} \overset{declaration}{\rightsquigarrow} \overline{d'}}{\texttt{struct}\ \overline{d}\ \texttt{end} \overset{module}{\rightsquigarrow} \overline{d'}}$$

$$\text{(MFunctor)}\ \frac{mt \overset{module\ type}{\rightsquigarrow} \overline{d} \qquad m \overset{module}{\rightsquigarrow} \overline{d'}}{\texttt{functor}(\mathcal{X} : mt) \to m \overset{module}{\rightsquigarrow} \texttt{scope}\ \mathcal{X}\ \overline{d}\ \texttt{end}\ \overline{d'}}$$

$$\text{(MTSig)}\ \frac{\overline{s} \overset{signature}{\rightsquigarrow} \overline{d}}{\texttt{sig}\ s\ \texttt{end} \overset{module\ type}{\rightsquigarrow} \overline{d}}$$

Figure 6: Translation of OCaml module expressions and module types into WhyML.

D Translation of signatures

$$\text{(SVal)}\ \frac{\neg is\_ghost(\mathcal{A}) \qquad kind(\mathcal{A}) = \mathcal{K} \qquad \mathcal{A}' \overset{function\ spec}{\rightsquigarrow} \mathcal{S} \qquad \pi, \mathcal{A}' \overset{function\ args}{\rightsquigarrow} \overline{x : \pi'}, \pi_{res}}{\texttt{val}\ \mathcal{A}\ f : \pi\ \mathcal{A}' \overset{signature}{\rightsquigarrow} \texttt{val}\ \mathcal{K}\ f(\overline{x : \pi})\ \mathcal{S} : \pi_{res}}$$

$$\text{(SValGhost)}\ \frac{\neg is\_ghost(\mathcal{A}) \qquad kind(\mathcal{A}) = \mathcal{K} \qquad \mathcal{A}' \overset{function\ spec}{\rightsquigarrow} \mathcal{S} \qquad \pi, \mathcal{A}' \overset{function\ args}{\rightsquigarrow} \overline{x : \pi'}, \pi_{res}}{\texttt{val}\ \mathcal{A}\ f : \pi\ \mathcal{A}' \overset{signature}{\rightsquigarrow} \texttt{val}\ \mathcal{K}\ \texttt{ghost}\ f(\overline{x : \pi})\ \mathcal{S} : \pi_{res}}$$

$$\text{(SType)}\ \frac{td_0 \overset{type\ definition}{\rightsquigarrow} td_0' \qquad \overline{td_1} \overset{type\ definition}{\rightsquigarrow} \overline{td_1'}}{\texttt{type}\ td_0\ \overline{\texttt{and}\ td_1} \overset{signature}{\rightsquigarrow} \texttt{type}\ td_0'\ \overline{\texttt{with}\ td_1'}}$$

Figure 7: Translation of OCaml signatures into WhyML.

E Summary of Cameleer case studies

Case Study	Lines of Code	Lines of Specification
Applicative Queue	25	21
Binary Multiplication	10	6
Binary Search	15	10
Checking a Large Routine	25	15
CNF Convertion	113	47
Ephemeral Queue	39	32
Factorial	10	13
Fast Exponentiation	19	47
Fibonacci	46	58
Insertion Sort	13	80
Integer Square Root	8	14
Leftist Heap	68	108
Mjrty	45	32
Same Fringe	23	16
total	459	499

Table 1: Case studies verified with the Cameleer tool.

Démonstration de la plateforme MOPSA d'analyse statique de programmes par interprétation abstraite *

Matthieu Journault[1], Antoine Miné[1,2],
Raphaël Monat[1] et Abdelraouf Ouadjaout[1]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France `firstname.lastname@lip6.fr`
[2] Institut Universitaire de France, F-75005, Paris, France

L'analyseur statique MOPSA

MOPSA [10, 7] est un logiciel d'analyse statique pour la vérification de programmes informatiques basé sur la théorie de l'interprétation abstraite [3]. Il infère automatiquement des invariants et signale, dès la compilation, les erreurs possibles d'exécution. Pour cela, MOPSA effectue une interprétation par induction sur la syntaxe qui collecte les exécutions de programmes sur tous les chemins possibles, en raisonnant à un niveau abstrait qui oublie certains détails sémantiques et permet un calcul efficace — par exemple, en ne retenant que les bornes des variables. L'analyse est sûre (toutes les propriétés prouvées par l'analyse sont vraies) mais incomplète : elle peut signaler des fausses alarmes, ce qui peut être corrigé en adaptant l'abstraction utilisée.

Les outils de vérification basés sur l'interprétation abstraite connaissent une popularité grandissante, notamment à travers la commercialisation d'outils utilisés dans l'industrie, comme Polyspace Verifier [16], Astrée [9], Sparrow [13], Julia [15] et le développement de plate-formes libres comme Frama-C [5] ou Infer [2]. Ces outils sont limités à l'analyse de langages statiques, comme le C. MOPSA se distingue en ciblant également les langages dynamiques, comme Python. Une autre particularité de MOPSA est sa conception modulaire et extensible. Il comporte une plate-forme indépendante des choix de langage et d'abstraction, dans laquelle peuvent être ajoutées et combinées des abstractions arbitraires (numériques, de pointeurs, de mémoire, etc.) et des itérateurs de syntaxe pour de nouveaux langages. MOPSA encourage le développement d'abstractions indépendantes et leur intégration dans une analyse grâce à l'utilisation de signatures uniformes de domaines et de mécanismes de communication génériques. Mopsa est un logiciel libre [1] écrit en majorité en OCaml (62 Klignes).

MOPSA supporte actuellement des sous-ensembles significatifs des langages C99 et Python 3. [2] En C, MOPSA détecte les comportements indéfinis ainsi que les utilisations invalides de la bibliothèque standard C (spécifiée dans un langage de modélisation dédié). Nous avons appliqué [14] MOPSA à l'analyse d'une partie des benchmarks C de Juliet [1] et des programmes de la suite Coreutils [6]. En Python, MOPSA effectue une analyse de types (capable d'exploiter les annotations de type de la bibliothèque standard) ainsi que des analyses de valeurs et d'exceptions [11, 12]. L'analyse de Python est extrêmement difficile à cause de la sémantique complexe et très dynamique du langage ; néanmoins, MOPSA est déjà capable d'analyser de petits programmes Python réels, comme PathPicker (2,5 Klignes). MOPSA est également un outil académique ayant pour but d'encourager l'enseignement et la recherche en analyse statique par interprétation abstraite.

*Ce travail a été en partie financé par le Conseil Européen de la Recherche dans le cadre de la bourse "Consolidator Grant" 681393 - MOPSA.

1. Sources disponibles sur : `https://gitlab.com/mopsa/mopsa-analyzer`
2. À l'exception, en C, des fonctions récursives, de la concurrence, des longjumps, des bitfields, des tableaux multi-dimensionnels à longueur variable et de l'assembleur et, en Python, des fonctions récursives, d'`eval`, de `super`, de l'asynchrone et des métaclasses.

```
1  #include <string.h>
2  #include <stdlib.h>
3  char *rand_string(int l) {
4    char *p = malloc(l + 1);
5    if (!p) exit(1);
6    for(int i=0; i<l; i++)
7      p[i] = 32 + rand() % 95;
8    p[l] = '\0';
9    return p;
10 }
11 void main() {
12   int n = rand() % 100;
13   char *s = rand_string(n);
14   int len = strlen(s);
15 }
```

```
mopsa >>> break example.c:14
mopsa >>> continue
  example.c:14.2-23: int len = strlen(src);
mopsa >>> next
  example.c:15.2-10: return 0;
mopsa >>> print n,len,s
  s -> {@Memory:1538e028d}
  offset(s) -> [0,0]
  n -> [0,99]
  n = len
  len = string-length(@Memory:1538e028d)
  bytes(@Memory:1538e028d) = n + 1
mopsa >>> info alarms
  No alarm
```

Figure 1 – Exemple de programme C (à gauche) et session interactive Mopsa (à droite).

Démonstration

Nous souhaitons démontrer, à travers l'analyse de courts programmes C et d'un exemple en Python, les capacités d'analyse de Mopsa et les avantages de sa conception modulaire.

À titre d'illustration, la figure 1 donne un exemple de programme C construisant une chaîne de caractères aléatoire. Mopsa propose des abstractions offrant différents compromis entre efficacité et précision, qui sont combinées par l'utilisateur pour créer des analyses. Une première analyse utilise un modèle mémoire simple qui décompose les variables C en séquences de cellules numériques ou pointeurs dont l'abstraction est déléguée à des domaines non-relationnels (e.g., les intervalles). Sur les 62 vérifications nécessaires à prouver l'absence d'erreur arithmétique et de pointeur dans le programme, Mopsa signale 6 (fausses) alarmes. Une deuxième analyse utilise le même modèle mémoire mais ajoute un domaine numérique relationnel, les polyèdres [4], qui infère les relations entre les variables `l`, `n`, `i` et la variable représentant la taille du bloc alloué par `malloc` à la ligne 4. Ceci permet d'éliminer les fausses alarmes dans les déréférencements `p[i]` et `p[l]`. L'analyse signale une seule (fausse) alarme, lors de l'appel à `strlen(s)` à la ligne 14 : le modèle de `strlen` contient une précondition, $\exists$`i` $\in [0, $`bytes(s)` $-$ `offset(s)`$[:$ `s[i] == '\0'`,[3] qui ne peut pas être prouvée par le modèle mémoire simple car la position du `'\0'` n'est pas constante. Une troisième analyse enrichit l'abstraction de mémoire avec un domaine symbolique [8] qui maintient, dans une variable auxiliaire `string-length(.)`, la position du premier caractère `'\0'` dans un bloc, ce qui suffit à éliminer l'alarme et prouver l'absence d'erreur sur le programme.

Nous démontrerons le mode interactif de Mopsa. Ce mode, similaire à un débogueur, permet d'exécuter pas à pas l'interprétation abstraite, d'observer les valeurs abstraites inférées et de mieux comprendre l'analyse. La figure 1 donne, à droite, une session pour la dernière analyse du programme de gauche. Elle montre l'invariant inféré ligne 15 : `n = len = string-length(@Memory:1538e028d) = bytes(@Memory:1538e028d) - 1`, où `@Memory:1538e028d` est un nom choisi par l'abstraction de l'allocation dynamique pour représenter le bloc alloué par `malloc` ligne 4.

Nous montrerons comment un utilisateur peut spécifier les abstractions à inclure via un fichier de configuration JSON. Un exemple d'analyse de Python et l'étude de sa configuration mettront en évidence le grand nombre d'abstractions et d'itérateurs partagés avec l'analyse du

3. `bytes(s)` est la taille en octets du bloc dans lequel `s` pointe, et `offset(s)` est l'offset de `s` par rapport au début du bloc.

C. Cette réutilisation d'abstractions pour l'analyse de langages aussi différents que C et Python démontre la flexibilité de Mopsa et sa capacité d'extension vers d'autres langages.

Références

[1] P. E. Black. Juliet 1.3 test suite : Changes from 1.2. Technical Report NIST TN – 1995, NIST, Jun. 2018.

[2] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NFM*, pages 3–11. Springer, 2015.

[3] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM, Jan. 1977.

[4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the 5th Annual ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

[5] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : A software analysis perspective. *Formal Aspects of Computing*, 27 :573–609, 2012.

[6] GNU. Coreutils : GNU core utilities.

[7] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Proc. of the 11th Working Conference on Verified Software : Theories, Tools, and Experiments (VSTTE19)*, volume 12031 of *Lecture Notes in Computer Science (LNCS)*, pages 1–18. Springer, Jul. 2019.

[8] M. Journault, A. Miné, and A. Ouadjaout. Modular static analysis of string manipulations in C programs. In *Proc. of the 25th International Static Analysis Symposium (SAS'18)*, volume 11002 of *Lecture Notes in Computer Science (LNCS)*, pages 243–262. Springer, Sep. 2018.

[9] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée : Proving the absence of runtime errors. In *Proc. of ERTS2 2010*, May 2010.

[10] A. Miné, A. Ouadjaout, and M. Journault. Design of a modular platform for static analysis. In *Proc. of 9h Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, Lecture Notes in Computer Science (LNCS), page 4, 28 Aug. 2018.

[11] R. Monat, A. Ouadjaout, and A. Miné. Static type analysis by abstract interpretation of Python programs. In *Proc. of the 34th European Conference on Object-Oriented Programming (ECOOP'20)*, Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl Publishing, Jul. 2020.

[12] R. Monat, A. Ouadjaout, and A. Miné. Value and allocation sensitivity in static Python analyses. In *Proc. of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP'20)*, pages 8–13. ACM, Jun. 2020.

[13] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. *SIGPLAN Not.*, 47(6) :229–238, June 2012.

[14] A. Ouadjaout and A. Miné. A library modeling language for the static analysis of C programs. In *Proc. of the 27th International Static Analysis Symposium (SAS'20)*, Lecture Notes in Computer Science (LNCS), page 25. Springer, Nov. 2020.

[15] F. Spoto. Julia : A generic static analyser for the Java bytecode. In *Proc. of FTfJP'2005*, page 17, July 2005.

[16] The Mathworks. Polyspace static analyzer. http://www.mathworks.fr/products/polyspace/.

Tail Modulo Cons

Frédéric Bour[1,2], Basile Clément[1], and Gabriel Scherer[1]

[1] INRIA
[2] Tarides

Abstract

OCaml function calls consume space on the system stack. Operating systems set default limits on the stack space which are much lower than the available memory. If a program runs out of stack space, they get the dreaded "Stack Overflow" exception – they crash. As a result, OCaml programmers have to be careful, when they write recursive functions, to remain in the so-called *tail-recursive* fragment, using *tail* calls that do not consume stack space.

This discipline is a source of difficulties for both beginners and experts. Beginners have to be taught recursion, and then tail-recursion. Experts disagree on the "right" way to write `List.map`. The direct version is beautiful but not tail-recursive, so it crashes on larger inputs. The naive tail-recursive transformation is (slightly) slower than the direct version, and experts may want to avoid that cost. Some libraries propose horrible implementations, unrolling code by hand, to compensate for this performance loss. In general, tail-recursion requires the programmer to manually perform sophisticated program transformations.

In this work we propose an implementation of "Tail Modulo Cons" (TMC) for OCaml. TMC is a program transformation for a fragment of non-tail-recursive functions, that rewrites them in *destination-passing style*. The supported fragment is smaller than other approaches such as continuation-passing-style, but the performance of the transformed code is on par with the direct, non-tail-recursive version. Many useful functions that traverse a recursive datastructure and rebuild another recursive structure are in the TMC fragment, in particular `List.map` (and `List.{filter,append}`, etc.). Finally those functions can be written in a way that is beautiful, correct on all inputs, and efficient.

In this work we give a novel modular, compositional definition of the TMC transformation. We discuss the design space of user-interface choices: what degree of control for the user, when to warn or fail when the transformation may lead unexpected results. We mention a few remaining design difficulties, and present (in appendices) a performance evaluation of the transformed code.

1 Introduction

1.1 Prologue

"OCaml", we teach our students, "is a functional programming language. We can write the beautiful function `List.map` as follows:"

```
let rec map f = function
| [] -> []
| x :: xs -> f x :: map f xs
```

"Well, actually", we continue, "OCaml is an effectful language, so we need to be careful about the evaluation order. We want `map` to process elements from the beginning to the end of the input list, and the evaluation order of `f x :: map f xs` is unspecified. So we write:

```
let rec map f = function
| [] -> []
| x :: xs ->
  let y = f x in
  y :: map f xs
```

"Well, actually, this version fails with a `Stack_overflow` exception on large input lists. If you want your `map` to behave correctly on all inputs, you should write a *tail-recursive* version. For this you can use the accumulator-passing style:"

```
let map f li =
  let rec map_ acc = function
  | [] -> List.rev acc
  | x :: xs -> map_ (f x :: acc) xs
  in map_ [] f li
```

"Well, actually, this version works fine on large lists, but it is less efficient than the original version. It is noticeably slower on small lists, which are the most common inputs for most programs. We measured it 35% slower on lists of size 10. If you want to write a robust function for a standard library, you may want to support both use-cases as well as possible. One approach is to start with a non-tail-recursive version, and switch to a tail-recursive version for large inputs; even there you can use some manual optimizations to reduce the overhead of the accumulator. For example, the nice Containers library does it as follows:".

```
let tail_map f l =
  (* Unwind the list of tuples, reconstructing the full list front-to-back.
     @param tail_acc a suffix of the final list; we append tuples' content
     at the front of it *)
  let rec rebuild tail_acc = function
    | [] -> tail_acc
    | (y0, y1, y2, y3, y4, y5, y6, y7, y8) :: bs ->
      rebuild (y0 :: y1 :: y2 :: y3 :: y4 :: y5 :: y6 :: y7 :: y8 :: tail_acc) bs
  in
  (* Create a compressed reverse-list representation using tuples
     @param tuple_acc a reverse list of chunks mapped with [f] *)
  let rec dive tuple_acc = function
    | x0 :: x1 :: x2 :: x3 :: x4 :: x5 :: x6 :: x7 :: x8 :: xs ->
      let y0 = f x0 in let y1 = f x1 in let y2 = f x2 in
      let y3 = f x3 in let y4 = f x4 in let y5 = f x5 in
      let y6 = f x6 in let y7 = f x7 in let y8 = f x8 in
      dive ((y0, y1, y2, y3, y4, y5, y6, y7, y8) :: tuple_acc) xs
    | xs ->
      (* Reverse direction, finishing off with a direct map *)
      let tail = List.map f xs in
      rebuild tail tuple_acc
  in
  dive [] l
```

```
let direct_depth_default_ = 1000

let map f l =
  let rec direct f i l = match l with
    | [] -> []
    | [x] -> [f x]
    | [x1;x2] -> let y1 = f x1 in [y1; f x2]
    | [x1;x2;x3] ->
      let y1 = f x1 in let y2 = f x2 in [y1; y2; f x3]
    | _ when i=0 -> tail_map f l
    | x1::x2::x3::x4::l' ->
      let y1 = f x1 in
      let y2 = f x2 in
      let y3 = f x3 in
      let y4 = f x4 in
      y1 :: y2 :: y3 :: y4 :: direct f (i-1) l'
  in
  direct f direct_depth_default_ l
```

At this point, unfortunately, some students leave the class and never come back. (These days they just have to disconnect from the remote-teaching server.)

We propose a new feature for the OCaml compiler, an explicit, opt-in "Tail Modulo Cons" transformation, to retain our students. After the first version (or maybe, if we are teaching an advanced class, after the second version), we could show them the following version:

```
let[@tail_mod_cons] rec map f = function
| [] -> []
| x :: xs -> f x :: map f xs
```

This version would be as fast as the simple implementation, tail-recursive, and easy to write.

The catch, of course, is to teach when this `[@tail_mod_cons]` annotation can be used. Maybe we would not show it at all, and pretend that the direct `map` version with `let y` is fine. This would be a much smaller lie than it currently is, a `[@tail_mod_cons]`-sized lie.

Finally, experts would be very happy. They know about all these versions, but they would not have to write them by hand anymore. Have a program perform (some of) the program transformations that they are currently doing manually.

1.2 TMC transformation example

A function call is in *tail position* within a function definition if the definition has "nothing to do" after evaluating the function call – the result of the call is the result of the whole function at this point of the program. (A precise definition will be given in Section 2.) A function is *tail recursive* if all its recursive calls are tail calls.

In the definition of `map`, the recursive call is not in tail position: after computing the result of `map f xs` we still have to compute the final list cell, `y :: □`. We say that a call is *tail modulo cons* when the work remaining is formed of data *constructors* only, such as `(::)` here.

```
let[@tail_mod_cons] rec map f = function
| [] -> []
| x :: xs ->
  let y = f x in
  y :: map f xs
```

Other datatype constructors may also be used; the following example is also tail-recursive *modulo cons*:

```
let[@tail_mod_cons] rec tree_of_list = function
| [] -> Empty
| x :: xs -> Node(Empty, x, tree_of_list xs)
```

The TMC transformation returns an equivalent function in *destination-passing* style where the calls in *tail modulo cons* position have been turned into *tail* calls. In particular, for `map` it gives a tail-recursive function, which runs in constant stack space; many other list functions also become tail-recursive. The transformed code of `map` can be described as follows:

```
let rec map f = function
| [] -> []
| x::xs ->
  let y = f x in
  let dst = y :: Hole in
  map_dps dst 1 f xs;
  dst
```

```
and map_dps dst i f = function
| [] ->
  dst.i <- []
| x::xs ->
  let y = f x in
  let dst' = y :: Hole in
  dst.i <- dst';
  map_dps dst' 1 f xs
```

The transformed code has two variants of the `map` function. The `map_dps` variant is in *destination-passing style*, it expects additional parameters that specify a memory location, a *destination*, and will write its result to this *destination* instead of returning it. It is tail-recursive. The `map` variant provides the same interface as the non-transformed function, and internally calls `map_dps` on non-empty lists. It is not tail-recursive, but it does not call itself recursively, it jumps to the tail-recursive `map_dps` after one call.

The key idea of the transformation is that the expression `y :: map f xs`, which contained a non-tail-recursive call, is transformed into first the computation of a *partial* list cell, written `y :: Hole`, followed by a call to `map_dps` that is asked to write its result in the position of the `Hole`. The recursive call thus happens after the cell creation (instead of before), in tail-recursive position in the `map_dps` variant. In the direct variant, the value of the destination `dst` has to be returned after the call.

The transformed code is in a pseudo-OCaml, it is not a valid OCaml program: we use a magical `Hole` constant, and our notation `dst.i <- ...` to update constructor parameters in-place is also invalid in source programs. The transformation is implemented on a lower-level, untyped intermediate representation of the OCaml compiler (Lambda), where those operations do exist. The OCaml type system is not expressive enough to type-check the transformed program: the list cell is only partially-initialized at first, each partial cell is mutated exactly

once, and in the end the whole result is returned as an *immutable* list. Some type system are expressive enough to represent this transformed code, notably Mezzo (Pottier and Protzenko, 2013).

1.3 Other approaches

1.3.1 More general transformations

Instead of a program transformation in *destination-passing* style, we could perform a more general program transformation that can make more functions tail-recursive, for example a generic *continuation-passing* style (CPS) transformation. We have three arguments for implementing the TMC transformation:

- The TMC transformation generates more efficient code, using mutation instead of function calls. On the OCaml runtime, the difference is a large constant factor.[1]
- The CPS transformation can be expressed at the source level, and can be made reasonably nice-looking using some monadic-binding syntactic sugar. TMC can only be done by the compiler, or using safety-breaking features.
- TMC is provided as an opt-in, on-demand optimization. We can add more such optimizations, they are not competing with each other, especially if they are to be rather used by expert programmers. Someone should try presenting CPS as an annotation-driven transformation, but we wanted to look at TMC first.

1.3.2 Different runtimes

Using the native system stack is a choice of the OCaml implementation. Some other implementations of functional languages, such as SML/NJ, use a different stack (the OCaml bytecode interpreter also does this), or directly allocate stack frames on their GC-managed heap. This approach makes "stack overflow" go away completely, and it also makes it very simple to implement stack-capture control operators, such as continuations, or other stack operations such as continuation marks.

On the other hand, using the native stack brings compatibility benefits (coherent stack traces for mixed OCaml+C programs), and seems to noticeably improve the performance of function calls (on benchmarks that are only testing function calls and return, such as Ackermann or the naive Fibonacci, OCaml can be 4x, 5x faster than SML/NJ.)

Lazy (call-by-need) languages will also often avoid running into stack overflows: as soon as a lazy datastructure is returned, which is the default, functions such as `map` will return immediately, with recursive calls frozen in a lazy thunk, waiting to be evaluated on-demand as the user traverses the result structure. User still need to worry about tail-recursivity for their strict functions (if the implementation uses the system stack); strict functions are often preferred when writing performant code.

1.3.3 Unlimiting the stack

Some operating systems can provide an unlimited system stack; such as `ulimit -s unlimited` on Linux systems – the system stack is then resized on-demand. Then it is possible to run non-tail-recursive functions without fear of overflows. Frustratingly, unlimited stacks are not available on all systems, and not the default on any system in wide use. Convincing all users to

[1] On a toy benchmark with large-sized lists, the CPS version is 100% slower and has 130% more allocations than the non-tail-recursive version.

setup their system in a non-standard way would be *much* harder than performing a program transformation or accepting the CPS overhead for some programs.

1.4 Related Work

Tail-recursion modulo cons was well-known in the Lisp community as early as the 1970s. For example the REMREC system (Risch, 1973) would automatically transform recursive functions into loops, and supports modulo-cons tail recursion. It also supports tail-recursion modulo associative arithmetic operators, which is outside the scope of our work, but supported by the GCC compiler for example. The TMC fragment is precisely described (in prose) in Friedman and Wise (1975).

In the Prolog community it is a common pattern to implement destination-passing style through unification variables; in particular "difference lists" are a common representation of lists with a final hole. Unification variables are first-class values, in particular they can be passed as function arguments. This makes it easy to write the destination-passing-style equivalent of a context of the form `List.append li` $\Box$, as the difference list `(List.append li X, X)`. In constrast, we only support direct constructor applications. However, this expressivity comes at a performance cost, and there is no static checking that the data is fully initialized at the end of computation.

In general, if we think of non-tail recursive functions as having an "evaluation context" left for after the recursive call, then the techniques to turn classes of calls into tail-calls correspond to different reified representations of non-tail contexts, as long as they support efficient composition and hole-plugging. TMC comes from representing data-construction contexts as the partial data itself, with hole-plugging by mutation. Associative-operator transformations represent the context $1 + (4 + \Box)$ as the number 5 directly. (Sometimes it suffices to keep around an abstraction of the context; this is a key idea in John Clements' work on stack-based security in presence of tail calls.)

Minamide (1998) gives a "functional" interface to destination-passing-style program, by presenting a partial data-constructor composition `Foo(x,Bar(□))` as a use-once, linear-typed function `linfun h -> Foo(x,Bar(h))`. Those special linear functions remain implemented as partial data, but they expose a referentially-transparent interface to the programmer, restricted by a linear type discline. This is a beautiful way to represent destination-passing style, orthogonal to our work: users of Minamide's system would still have to write the transformed version by hand, and we could implement a transformation into destination-passing style expressed in his system. Pottier and Protzenko (2013) supports a more general-purpose type system based on separation logic, which can directly express uniquely-owned partially-initialized data, and its implicit transformation into immutable, duplicable results. (See the List module of the Mezzo standard library, and in particular `cell`, `freeze` and `append` in destination-passing-style).

1.5 Contributions

This work is in progress. We claim the following contributions:

- A formal grammar of which programs expressions are in the "Tail Modulo Cons" fragment.
- A novel, modular definition of the transformation into destination-passing-style.
- Discussion of the user-interface issues related to transformation control.
- A performance evaluation of the transformation for `List.map`, in the specific context of the OCaml runtime.

A notable non-contribution is a correctness proof for the transformation. We would like to work on a correctness proof soon; the correctness argument requires reasoning on mutability

and ownership of partial values, an excellent use-case for separation logic.

2 Tail Calls Modulo Constructors

$$
\begin{aligned}
\mathsf{Exprs} \ni e, d ::=\ & x, y \\
\mid\ & n \in \mathbb{N} \\
\mid\ & f\ e \\
\mid\ & \texttt{let}\ x = e\ \texttt{in}\ e' \\
\mid\ & K\,(e_i)^i \\
\mid\ & \texttt{match}\ e\ \texttt{with}\ (p_i \to e_i')^i \\
\mid\ & d.e \leftarrow e'
\end{aligned}
\qquad
\begin{aligned}
\mathsf{FunctionNames} \ni f \\
\mathsf{Patterns} \ni p ::=\ & x \mid K\,(p_i)^i
\end{aligned}
$$

$$
\mathsf{Stmt} \ni s ::= \ \texttt{let rec}\ (f_i\ x = e_i)^i
$$

Figure 1: A first-order programming language

In order to simplify the presentation of the transformation, we consider a simple untyped language, described in Figure 1. This language, which we present with a syntax similar to OCaml, embeds function application and sequencing let-binding, as well as constructor application and pattern-matching. In addition, to implement the imperative DPS transformation, we include a special operator: $d.e \leftarrow e'$ is an imperative construct which updates d's e-th argument in-place. All those constructs (and more) are present in the untyped intermediate language used in the OCaml compiler where we implemented the transformation. One notable missing construct is function abstraction. In fact, this model requires that all functions be called by name, and functions can only be defined through a toplevel `let rec` statement. Our implementation supports the full OCaml language, but it cannot specialize higher-order function arguments for TMC.

In the following, we will use syntaxic sugar for some usual constructs; namely, we can desugar $e1; e2$ into the let-binding $\texttt{let}\ \_ = e1\ \texttt{in}\ e2$ (where _ is a fresh variable name) and $(e_1, \ldots, e_n)$ into the constructor application $\texttt{Tuple}\,(e_1, \ldots, e_n)$.

The multi-hole grammar of tail contexts, where each hole indicates a tail position, for this simple language is depicted in Figure 2. To interpret a multi-hole context T with n holes, we denote by $T\,[e_1, \ldots, e_n]$ the term obtained by replacing each of the holes in T from left to right with the expressions $e_1, \ldots, e_n$. In a decomposition $e = T\,[e_1, \ldots, e_n]$, each of the e_i is in tail position; in particular, a call $f\ e$ is in tail position (i.e. it is a tail call) in expression e' if there is a decomposition of e' as $T\,[e_1, \ldots, e_j, f\ e, e_{j+1}, \ldots, e_n]$.

One can remark that, for a language construct, the holes in the tail context are precisely the complement of holes in the evaluation context. For instance, the construct $\texttt{let}\ x = e\ \texttt{in}\ e'$ has evaluation contexts of the form $\texttt{let}\ x = E\ \texttt{in}\ e'$ and tail contexts $\texttt{let}\ x = e\ \texttt{in}\ T$. In some sense, the tail contexts are "guarded" by the evaluation context: a reduction can only occur in a tail position after the construct has been reduced away, and the subterm in tail position is now at toplevel. This guarantees that when we start reducing inside the tail context, there is no remaining computation to be performed in the surrounding context. The "depth" of the surrounding context is a source-level notion that is directly related to call-stack size: tail calls do not require reserving frames on the call stack.

Our first goal is to figure out what the proper grammar is for properly defining tail calls modulo cons. The lazy way would be to allow, in tail position, a single constructor application

$$
\begin{aligned}
\mathsf{TailCtx} \ni T ::= & \\
& \mid \square \\
& \mid e; T \\
& \mid \texttt{let}\ x = e\ \texttt{in}\ T \\
& \mid \texttt{match}\ e\ \texttt{with}\ (p_j \to T_j)^j
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{ConstrCtx} \ni C\left[\square\right] ::= & \ \square \\
& \mid K\left((e_i)^i, C\left[\square\right], (e_j)^j\right)
\end{aligned}
$$

Figure 2: Tail multicontexts, constructor contexts

itself containing a tail position, that is, add a case $K\left((e_i)^i, \square, (e_j)^j\right)$ to the definition of T. This would capture most of the cases presented above. However, such a lazy implementation would be brittle: for instance, a partially unrolled version of `map` below would not benefit from the optimization.

```
let rec umap f xs =
  match xs with
  | [] -> []
  | [x] -> [f x]
  | x1 :: x2 :: xs ->
      f x1 :: f x2 :: umap f xs
```

This would make the performance-seeking developer unhappy, as they would have to choose between our optimization or the performance benefits of unrolling. To make them happy again, we at least need to consider **repeated** constructor applications. Using the grammar C from Figure 2, we could consider all the decompositions $T\left[C\right]$, and extract the inner hole from the nested C context.

However, this approach would still be somewhat fragile, and some very reasonable program transformations on the nested TMC call would break it. For instance, it is not possible to locally let-bind parts of the function application, or to perform a match ultimately leading to a TMC call inside a constructor application. In our new grammar U, instead of adding a case $K\left((e_i)^i, \square, (e_j)^j\right)$ that forces constructors to occur only at the "leaves" of a context, we add a case $K\left((e_i)^i, U, (e_j)^j\right)$, allowing arbitrary interleavings of constructors and tail-preserving contexts. This gives a more natural and less surprising definition of the tail positions modulo cons. This grammar is depicted in Figure 3.

$$
\begin{aligned}
\mathsf{TailModConsCtx} \ni U ::= & \\
& \mid \square \\
& \mid e; U \\
& \mid \texttt{let}\ x = e\ \texttt{in}\ U \\
& \mid \texttt{match}\ e\ \texttt{with}\ (p_j \to U)^j \\
& \mid K\left((e_i)^i, U, (e_j)^j\right)
\end{aligned}
$$

Figure 3: Tail modulo cons contexts

If an expression e_0 is of the form $U\left[(e_i)^i, f\ e, (e_j)^j\right]$, we say that the plugged subterms are in tail position *modulo constructor* in e_0, and in particular $f\ e$ is a tail call *modulo constructor*. We also define tail positions *strictly* modulo cons as the tail positions modulo cons which are not regular tail positions.

Notice that there is a subtlety here: the term $K\,(f_1\ e_1, f_2\ e_2)$ admits two *distinct* context decompositions, one with $U_1 := K\,(\Box, f_2\ e_2)$ where $f_1\ e_1$ is a tail call modulo cons, the other with $U_2 := K\,(f_1\ e_1, \Box)$ where $f_2\ e_2$ is a tail call modulo cons. (This is intentional, obtained by allowing a single sub-context in the constructor rule of U.) We can transform this term such that either one of the calls become tail-calls, but not both. In other words, the notion of being "in tail position modulo cons" depends on the decomposition context U.

Our implementation has to decide which context decomposition to perform. It does not make choices on the user's behalf: in such ambiguous situations, it will ask the user to disambiguate by adding a `[@tailcall]` attribute to the one call that should be made a tail-call.

Remark: our grammar for U is *maximal* in the sense that in each possible context decomposition of a term, all tail positions modulo cons are inside a hole of the U context. It would be possible to abandon maximality by allowing arbitrary terms e (containing no hole) as a context. We avoided doing this, as it would introduce ambiguities in the grammar of context (the program $(a; b)$ can be parsed using this e case directly, or using the $e; U$ rule first), so that operations defined on contexts would depend on the parse tree of the context in the grammar.

3 TRMC functions of interest

Many functions that consume and produce lists are tail-recursive-modulo-cons, in the sense that all they have a TMC decomposition where all recursive calls are in TMC position. Notable functions include `map`, as already discussed, but also for example:

```
let[@tail_mod_cons] rec filter p = function
| [] -> []
| x :: xs -> if p x then x :: filter p xs else filter p xs

let[@tail_mod_cons] rec merge cmp l1 l2 =
  match l1, l2 with
  | [], l | l, [] -> l
  | h1 :: t1, h2 :: t2 ->
      if cmp h1 h2 <= 0
      then h1 :: merge cmp t1 l2
      else h2 :: merge cmp l1 t2
```

TMC is not useful only for lists or other "linear" data types, with at most one recursive occurrence of the datatype in each constructor.

A non-example Consider a `map` function on binary trees:

```
let[@tail_mod_cons] rec map f = function
| Leaf v -> Leaf (f v)
| Node(t1, t2) -> Node(map f t1, (map[@tailcall]) f t2)
```

In this function, there are two recursive calls, but only one of them can be optimized; we used the `[@tailcall]` attribute to direct our implementation to optimize the call to the right child. This is actually a *bad* example of TMC usage in most cases, given that

- If the tree is arbitrary, there is no reason that it would be right-leaning rather than left-leaning. Making only the right-child calls tail-calls does not protect us from stack overflows.
- If the tree is known to be balanced, then in practice the depth is probably very small in both directions, so the TMC transformation is not necessary to have a well-behaved function.

Yes-examples from our real world There *are* interesting examples of TMC-transformation on functions operating on tree-like data structures, when there are natural assumptions about which child is likely to contain a deep subtree. The OCaml compiler itself contains a number of them; consider for example the following function from the `Cmm` module, one of its lower-level program representations:

```
let[@tail_mod_cons] rec map_tail f = function
  | Clet(id, exp, body) ->
      Clet(id, exp, map_tail f body)
  | Cifthenelse(cond, ifso, ifnot) ->
      Cifthenelse(cond, map_tail f ifso, (map_tail[@tailcall]) f ifnot)
  | Csequence(e1, e2) ->
      Csequence(e1, map_tail f e2)
  | Cswitch(e, tbl, el) ->
      Cswitch(e, tbl, Array.map (map_tail f) el)
  [...]
  | Cexit _ | Cop (Craise _, _, _) as cmm ->
      cmm
  | Cconst_int _ | Cvar _ | Ctuple _ | Cop _ as c ->
      f c
```

This function is traversing the "tail" context of an arbitrary program term – a meta-example! The `Cifthenelse` node acts as our binary-node constructor, we do not know which side is likely to be larger, so TMC is not so interesting. The recursive calls for `Cswitch` are not in TMC position. But on the other hand the `Clet`, `Csequence` cases are very beneficial to have in TMC: while they have several recursive subtrees, they are in practice only deeply nested in the direction that is turned into a tailcall by the transformation. The OCaml compiler does sometimes encounter machine-generated programs with a unusually long sequence of either constructions, and the TMC transformation may very well avoid a stack overflow in this case.

Another example would be #9636, a patch to the OCaml compiler proposed by Mark Shinwell, to get a partially-tail-recursive implementation of the "Common Subexpression Elimination" (CSE) pass. Xavier Leroy remarked that the existing implementation in fact fits the TMC fragment. Not all recursive calls become tail-calls (this would require a more powerful transformation or a longer, less readable patch), but the behavior of TMC on the unchanged code matches the tail-call-ness proposed in the human-written patch.

4 Modular TMC transformation

A good way of thinking about our TMC transformation is as follows. We want to transform a tail context modulo cons U into a regular tail context T, where tail calls modulo cons have been replaced by regular tail calls, but to the DPS version of the callee. More precisely, given a term e, we will build its DPS version as follows:

- First, we find a decomposition of e as $U\,[e_1, \ldots, e_n]$ which identifies the tail positions modulo cons. We want this decomposition to capture as much of the TMC calls to DPS-enabled functions (functions marked with the `[@tail_mod_cons]` attribute) as possible.
- Once the decomposition is selected, we transform the context $x.y \leftarrow U$ (where x and y are fresh variables not appearing in U) into a context $T\,[x_1.y_1 \leftarrow \Box, \ldots, x_n.y_n \leftarrow \Box]$. This transformation is effectively moving the assignment from the root to the leaves of the context U.
- Finally, we replace the assignments $x.y \leftarrow f\ e$ by calls $f^{\mathsf{dps}}\ ((x, y), e)$ when the callee f has a DPS version f^{dps}, introducing calls in tail position.

However, this transformation is not enough: we also need to transform the code of the original function to call f^{dps} in the recursive case. There is a naive way to do it, which is suboptimal, as we explain, before showing how we do it, in Section 4.2 (The "direct" transformation).

Finally, Section 4.3 (Compression of nested constructors) highlights an optimization made by our implementation which generates cleaner code for TMC calls inside nested constructor applications.

4.1 The DPS transformation

We now present in detail the transformation used to build the body of the transformed f^{dps} function from a function definition `let rec` $f\ x = e$. As a reminder, the semantics of $f^{\mathsf{dps}}\ ((d, i), e')$ should be the same as $d.i \leftarrow f\ e'$, and the body of f^{dps} should replace tail calls modulo cons in e with regular tail calls to the DPS versions of the callee. We only present the transformation for unary functions: the general case follows by using tuple for n-ary functions. Our implementation handles fully applied functions of arbitrary arity, treating them similarly as the equivalent unary function with a tuple argument.

We first find a decomposition of e as $U\,[e_1, \ldots, e_n]$. Recall that the TMC transformation depends on this decomposition, as we have multiple choices for the decomposition of a constructor $K\,(e_1, \ldots, e_n)$. We use the following logic:

- If none of the e_i contains calls in TMC position to a function which has a DPS version (a *TMC candidate*), use the decomposition $e = \Box\,[e]$.
- If exactly one of the e_i contains such a TMC candidate, or if exactly one of the e_i contains a TMC candidate marked with the `[@tailcall]` annotation, name it e_j and use the decomposition $e = (K\,(e_1, \ldots, e_{j-1}, \Box, e_{j+1}, \ldots, e_n)\,[e_j]$.
- Otherwise, report an error to the user, indicating the ambiguity, and requesting that one (or more) `[@tailcall]` annotations be added.

For other constructs, we only decompose them if at least one of their component has a TMC candidate; for instance, `let` $x = e$ `in` e' gets decomposed into $\Box\,[$`let` $x = e$ `in` $e']$ unless e' contains TMC candidates. This avoids needlessly duplicating assignments.

Once we obtained the decomposition as a tail context modulo constructors, we transform said context into a tail context where each expression in tail position is an assignment. We write $d.n \leftarrow U \rightsquigarrow_{dps} T[d_i.n_i \leftarrow \Box]^i$ to signify that the context $T[d_i.n_i \leftarrow \Box]^i$ is obtained by performing the DPS transformation on the TMC context U with destination $d.n$. The rules describing this transformation are shown below.

$$\frac{}{d.n \leftarrow \Box \rightsquigarrow_{dps} \Box\,[d.n \leftarrow \Box]} \qquad \frac{d.n \leftarrow U \rightsquigarrow_{dps} T[d_i.n_i \leftarrow \Box]^i}{d.n \leftarrow \texttt{let}\ x = e\ \texttt{in}\ U \rightsquigarrow_{dps} \texttt{let}\ x = e\ \texttt{in}\ T[d_i.n_i \leftarrow \Box]^i}$$

$$\frac{\forall j, \quad d.n \leftarrow U_j \rightsquigarrow_{dps} T_j\left[d_{i_j}.n_{i_j} \leftarrow \Box\right]^{i_j}}{d.n \leftarrow \texttt{match}\ e\ \texttt{with}\ (p_j \to U_j)^j \rightsquigarrow_{dps} \texttt{match}\ e\ \texttt{with}\ \left(p_j \to T_j\left[d_{i_j}.n_{i_j} \leftarrow \Box\right]^{i_j}\right)^j}$$

$$\frac{n' = |I| + 1 \qquad d'.n' \leftarrow U \rightsquigarrow_{dps} T[d'_l.n'_l \leftarrow \Box]^l}{d.n \leftarrow K\left((e_i)^{i \in I}, U, (e_j)^j\right) \rightsquigarrow_{dps} \begin{array}{l}\texttt{let}\ d' = K\left((e_i)^{i \in I}, \texttt{Hole}, (e_j)^j\right)\ \texttt{in} \\ \quad d.n \leftarrow d'; \\ \quad T[d'_l.n'_l \leftarrow \Box]^l\end{array}}$$

Most cases are straightforward: for constructs whose tail context modulo cons is also a regular tail context, we simply apply the transformation into said tail context. The two important cases are the one of a hole, where we introduce an assignment to the result of evaluating the hole, and the case of a constructor, where we "reify" the hole by using a placeholder value, fill the current destination, and recursively transform the TMC context with the newly created hole as a new destination $d'.n'$.[2]

After this transformation, we can now build the term $T[d_i.n_i \leftarrow e_i]^i$ where the e_i are the subterms in the initial decomposition $U[e_i]^i$ of the body of our recursive function. Finally, for each e_i with shape $f_i\ e_i'$ where f_i has a DPS version f_i^{dps}, we can replace $d_i.n_i \leftarrow e_i$ with ${f_i}^{dps}\ ((d_i, n_i), e_i)$, yielding the final result of the DPS transformation.

We remark again that the *only* calls in tail position in the transformed term are the assignments we have just transformed into calls to a DPS version of the original callee. Indeed, any other tail call in the initial decomposition has been replaced by an assignment. We "lose" a tail call when we go from the "transformed" world back to the "direct-style" world – a CPS transformation would work similarly, transforming $f\ e$ into $k\ (f\ e)$ if f has no CPS version.

4.2 The "direct" transformation

As we just noted, tail calls in the original function are tail calls in the DPS version only if their callee also have a DPS version (e.g. in the common case of a recursive function). Tail calls where the callee didn't have a DPS version are no longer in tail position. As such, the simple and lazy way to call into f^{dps} in the body of f, namely, introducing a destination and calling f^{dps}, is suboptimal, as it could make previously-tail recursive paths in f no longer tail recursive, and the programmer may rely on those being tail recursive. Instead, we will ensure that calls to f^{dps} only happen inside a constructor application: this way, all the pre-existing tail calls will be left untouched.

The transformation is very similar as the DPS transformation (in fact, all of the "boring" cases are identical), and we will reuse the same context decomposition of e as $U\,[e_1, \ldots, e_n]$. We again perform a context rewriting on U, but now the output is an arbitrary context E.

$$\frac{}{\Box \rightsquigarrow_{direct} \Box} \qquad \frac{U \rightsquigarrow_{direct} E}{\texttt{let}\ x = e\ \texttt{in}\ U \rightsquigarrow_{direct} (\texttt{let}\ x = e\ \texttt{in}\ E)}$$

$$\frac{\forall j,\ U_j \rightsquigarrow_{direct} E_j}{\texttt{match}\ e\ \texttt{with}\ (p_j \to U_j)^j \rightsquigarrow_{direct} (\texttt{match}\ e\ \texttt{with}\ (p_j \to E_j)^j)}$$

$$\frac{n = |I| + 1 \qquad d.n \leftarrow U \rightsquigarrow_{dps} T[d_l.n_l \leftarrow \Box]^l}{K\,((e_i)^{i \in I}, U, (e_j)^j) \rightsquigarrow_{direct} \begin{array}{l}\texttt{let}\ d = K\,((e_i)^{i \in I}, \texttt{Hole}, (e_j)^j)\ \texttt{in} \\ \quad T[d_l.n_l \leftarrow \Box]^l; \\ \quad d\end{array}}$$

This transformation leaves the regular tail positions unchanged, but switches to the DPS version for tail positions strictly modulo cons. We then replace again tail assignments of a call to a tail call to the DPS version of the callee. Note that this time, calls to the DPS version

[2] It may look like d' is only used once in the right-hand-side of the conclusion of this rule, so its binding could be inlined, but this d' is used in the last premise and will occur in U.

are not in tail position (we need to return the computed value): we simply introduce a fresh destination so that we can call into the DPS version.

The presentation by Minamide (1998) suggests a slightly different encoding, where we would pass a third extra argument to the DPS version: the location of the final value to be returned. This would allow tail calls into the DPS version, at the cost of an extra argument. This may look compelling, but in practice not so much, because calls from the DPS version back into the "direct" world will never be in tail position, and we only end up paying a constant factor more frames.

4.3 Compression of nested constructors

Consider a function such as the partially unrolled `map` shown above. It has two nested constructor applications, and the DPS transformation as described above will generate the code on the left below for the `umap_dps` version. This is unsatisfactory, as it introduces needless writes that the OCaml compiler does not eliminate. Instead, we would want to generate the nicer code on the right.

```
let rec umap_dps dst i f = function
| [] ->
       dst.i <- []
| [x] ->
       dst.i <- [f x]
| x1 :: x2 :: xs ->
       let dst1 = f x1 :: Hole in
       dst.i <- dst1;
       let dst2 = f x2 :: Hole in
       dst1.1 <- dst2;
       umap_dps dst2 1 f xs
```

```
let rec umap_dps dst i f = function
| [] ->
       dst.i <- []
| [x] ->
       dst.i <- [f x]
| x1 :: x2 :: xs ->
       let y1 = f x1 in
       let y2 = f x2 in
       let dst' = y2 :: Hole in
       dst.i <- y1 :: dst';
       umap_dps dst' 1 f xs
```

Notice that in the nicer code, we need to let-bind constructor arguments to preserve execution order. We implement this optimization by keeping track, in the rules for the *dps* transformation, of an additional "constructor context". Before, we were conceptually preserving the semantics of $d.n \leftarrow U$; now, we will be preserving the semantics of $d.n \leftarrow C\,[U]$ for some constructor context C. C represents delayed constructor applications, which we will perform later — typically, immediately before calling a DPS-transformed function.

The new rules, written $d.n \leftarrow C\,[U] \rightsquigarrow_{dps} T[d_i.n_i \leftarrow C_i]^i$, are shown in Figure 4. The rules for `let` and `match` are unchanged, except that we pass the unchanged constructor context C recursively.

The constructor rule is now split in two parts: the DPS-Constr-Opt rule adds the new constructor to the delayed constructor context, and the DPS-Reify rule generates an assignment for the constructor context. Note that the rules for $\rightsquigarrow_{dps}$ are no longer deterministic: the DPS-Reify can apply whenever the delayed constructor stack is nonempty. We perform the reification in two cases. The first one is before generating a call to a DPS-transformed version, because we need a concrete destination for that. The second one is that we keep track of whether a subterm would duplicate the delayed context (e.g. due to a `match`) and immediately apply the reification after a constructor in that case.

Notice that a similar optimization could be made in the *direct* transformation (in fact, our implementation does just that): the goal of switching to the *dps* mode in a constructor context is simply to provide a destination to the inner TMC calls. We can, without loss of generality, only switch to *dps* when there is a TMC call in tail position in the recursive argument (i.e. there will be no opportunities to introduce a destination in a subterm).

$$\text{DPS-Hole-Opt}\quad \frac{}{d.n \leftarrow C\,[\Box] \rightsquigarrow_{dps} \Box\,[d.n \leftarrow C]}$$

$$\text{DPS-Reify}\quad \frac{n' = |I| + 1 \qquad d'.n' \leftarrow \Box\,[U] \rightsquigarrow_{dps} T[d_l.n_l \leftarrow C_l]^l}{d.n \leftarrow C\left[K\left((e_i)^i, \Box, (e_j)^j\right)\right][U] \rightsquigarrow_{dps} \begin{array}{l}\texttt{let } d' = K\left((e_i)^{i\in I}, \texttt{Hole}, (e_j)^j\right) \texttt{ in}\\ \quad d.n \leftarrow C\,[d'];\\ \quad T[d_l.n_l \leftarrow C_l]^l\end{array}}$$

$$\text{DPS-Constr-Opt}\quad \frac{n' = |I| + 1 \qquad d'.n' \leftarrow C\left[K\left((v)_i^{i\in I}, \Box, (e_j)^j\right)\right][U] \rightsquigarrow_{dps} T\left[(d_l.n_l \leftarrow C_l)^l\right]}{d.n \leftarrow C\left[K\left((e_i)^{i\in I}, U, (e_j)^j\right)\right] \rightsquigarrow_{dps} \begin{array}{l}\texttt{let } (v_i = e_i)^{i\in I} \texttt{ in}\\ \quad \texttt{let } (v_j = e_j)^j \texttt{ in}\\ \quad\quad T[d_l.n_l \leftarrow C_l]^l\end{array}}$$

$$\frac{d.n \leftarrow \Box\,[U] \rightsquigarrow_{dps} \Box\,[d.i \leftarrow C_i]}{d.n \leftarrow U \rightsquigarrow_{dps} \Box\,[d.i \leftarrow C_i]}$$

Figure 4: DPS transformation, with constructor compression

Finally, we note that in our implementation we perform all of the transformations (dps, direct, as well as the computation of the auxiliary information such as whether there are TMC opportunities in the term and whether we need to provide a destination to benefit from a switch to the DPS mode) in a single pass over the terms.

5 Design issues

5.1 Non-issue: Flambda and Multicore

Some readers may wonder whether introducing mutation to build immutable data structures could be an issue with other subsystems of the OCaml implementation that perform fine-grained mutability reasoning, notably the Flambda optimizer and the Multicore runtime.

The answer is that there is no issue, move along! The OCaml value model (even under Multicore) already contains the notions that (immutable) values may start "uninitialized" and eventually be filled by "initializing" writes – this is how immutable values are constructed from the C FFI, for example. In 2016, in preparation for our TMC work, Frédéric Bour extended the Lambda intermediate representation with an explicit notion of "initializing" write (#673), so that the information is explicit in the generated code, and thus perfectly understood by Flambda.

5.2 Non-linear control operators

The TMC transformation makes the assumption that partially-initialized values (with a `Hole` placeholder) have a unique owner, and will be initialized into complete values by a single write. This assumption can be violated by the addition of control operators to the language, such as

`call/cc` or `delim/cc`. The problem comes from the *non-linear* usage of continuations, where the same continuation is invoked twice.

In practice this means that we cannot combine the external (but magical) delim/cc library with our TMC transformation. Currently the only solution is to disable the TMC transformation for delimited-continuation users (at the risk of stack overflows), but in the future we could perform a more general stack-avoiding transformation, such as a continuation-passing-style transformation, for `delim/cc` users.

(We considered intermediate approaches, based on detecting multi-writes to a partially-initialized value, and copying the partial value on the fly. This works for lists, where the position of the hole is, but we did not manage to define this approach in the general case of arbitrary constructors.)

5.3 Evaluation order

If a call inside a constructor is in TMC position, our transformation ensures that it is evaluated "last". For example, in the program $K\,(e_1, f\ e_2, e_3)$, if f is the TMC call, we know that e_1 and e_3 will be evaluated (in some order) before $f\ e_2$ in the transformed program.

In OCaml, the order of evaluation order of constructor arguments is implementation-defined; the evaluation-order resulting from the TMC transform is perfectly valid for the source program. However, in this case it is also *different* from the one you would typically observe on the unmodified program – most implementations use either left-to-right or right-to-left.

We consider that it is reasonable that an *explicit* transformation would change the evaluation order – especially as it remains a valid order for the source program. Reviewers have found this to be an issue, and suggested instead to forbid having potentially-side-effectful arguments in TMC constructor applications: in this example, if we restrict e_1 and e_3 to be values (or variables), we cannot observe a difference anymore.

In several cases this forces the user to `let`-bind the arguments beforehand, explicitly expressing an evaluation order. This is a sensible design, but in our experience many functions that would benefit from the TMC transformation are not written in this style, and converting them to be in this style would be a bothersome and invasive change, raising the barrier to entry of the `[@tail_mod_cons]` annotation – without much benefits in terms of evaluation-order, as functions that need to enforce a specific evaluation order should use explicit `let`-bindings anyway. This is for example the case of the most interesting example of Section 3 (TRMC functions of interest), the `tail_map` function on a compiler intermediate representation.

5.4 Stack usage guarantees

This document presents a precisely-defined subset of functions that can be TMC-transformed (they must be decomposable through a TMC context U, with a TMC-specialized function call in tail-position or, preferably, strictly in tail-position modulo cons).

For many recursive functions in this subset, the TMC-transformation returns a "tail-recursive function", using a constant amount of stack space. However, many other functions are not "tail-recursive modulo cons" (TRMC) in this sense. Initially we wanted to restrict our transformation to reject non-TRMC functions: the success of the transformation would guarantee that the resulting function never stack-overflows.

However, we realized that many functions we want to TMC-transform are not in the TRMC fragment. The interesting `tail_map` function from Section 3 (TRMC functions of interest) does not provide this guarantee – for instance, its stack usage grows linearly with the nesting of `Cifthenelse` constructors in the `then` direction.

5.5 Transformation scope

If a given function f is marked for TMC transformation, what is the scope of the code in which TMC calls to f should be transformed? If one tried to give a maximal scope, an answer could be: all calls to f in the program. This requires including information on which functions have a DPS version in module boundaries, and thus in module types (to enable separate compilation). We tried to find a smaller scope that is easy to define, and would not require cross-module information.

The second idea was the following "minimal" scope: when we have a group of mutually-recursive functions marked for TMC, we should only rewrite calls to those functions in the recursive bodies themselves, not in the rest of the program. In `let rec` $(f_i\ x = e_i)^i$ `in` e', the calls to the f_i in some e_j would get rewritten, but not in e'. This restriction makes sense, given that generally the stack-consuming calls are done within the recursive bodies, with only a constant number of calls in e' that do not contribute to stack exhaustion.

However, consider the `List.flatten` function, which flattens lists of lists into simple lists.

```
let rec flatten = function
| [] -> []
| xs :: xss -> xs @ flatten xs
```

This function is not in the TRMC fragment. However, it can be rewritten to be TMC-transformable by performing a local inlining of the `@` operator to flatten two lists:

```
let[@tail_mod_cons] rec flatten = function
| [] -> []
| xs :: xss ->
  let[@tail_mod_cons] rec append_flatten xs xss =
    match xs with
    | [] -> flatten xss
    | xs :: xss -> xs :: append_flatten xs xss
  in append_flatten xs xss
```

This definition contains a toplevel recursive function and a local recursive function, and both are in the TMC fragment. However, to get constant-stack usage it is essential that the call to `append_flatten` that is *outside* its recursive definition be TMC-specialized. Otherwise it is not a tail-call anymore in the transformed program.

For now we have decided to extend the "minimal" scope as follows: for a recursive definition `let rec` $(f_i\ x = e_i)^i$ `in` e', TMC calls to the f_i in the body e' are not rewritten *unless* the whole term is itself in the body of a function marked for TMC. In other words, the scope is "minimal" at the toplevel, but "maximal" within TMC-marked functions.

Another alternative is to stick to the minimal scope, and warn on the `flatten` implementation above. It is still possible to write a TMC `flatten` in the minimal scope, by extruding the local definition into a mutual recursion:

```
let[@tail_mod_cons] rec flatten = function
| [] -> []
| xs :: xss -> append_flatten xs xss

and[@tail_mod_cons] append_flatten xs xss =
  match xs with
  | [] -> flatten xss
  | xs :: xss -> xs :: append_flatten xs xss
```

Indeed, for local definitions it is always possible to rewrite the term `let rec` $(f_i\ x = e_i)^i$ `in` e' by moving e' inside the mutually-recursive part: `let rec` $((f_i\ x = e_i)^i, g\ \_ = e')$ `in` $g\ ()$. The

question would be whether we want to force users to perform this transformation manually, and how to tell them that we expect it.

5.6 Future work: Higher-order transformation

We formalized the TMC-transformation on a first-order language, and our implementation silently ignores the higher-order features of OCaml. What would it mean to DPS-transform a higher-order function such as (`let` app $f\ x = f\ x$)?

Our answer would be that the higher-order DPS-transformation takes a pair of function f and returns a pair $(f^{\mathsf{direct}}, f^{\mathsf{dps}})$, allowing to define

$$\texttt{let app}^{\mathsf{direct}} \qquad (f^{\mathsf{direct}}, f^{\mathsf{dps}})\ x = f^{\mathsf{direct}}\ x$$
$$\texttt{let app}^{\mathsf{dps}} \quad (d, i)\ (f^{\mathsf{direct}}, f^{\mathsf{dps}})\ x = f^{\mathsf{dps}}\ (d, i)\ x$$

5.7 Future work: Multi-destination TMC?

The program below on the left, frustratingly, cannot be TMC-transformed with our current definition or implementation. One can manually express a *multi-destination* DPS version, on the right, but it is still unclear how to specify the fragment of input programs that would be transformed in this way. (We discussed this with Pierre Chambart.)

```
let rec partition p = function
| [] -> ([], [])
| x::xs ->
  let (yes, no) = partition p xs in
  if p x
  then (x :: yes, no)
  else (yes, x :: no)
```

```
let rec partition_dps
  dst_yes i_yes dst_no i_no p xs
 = match xs with
| [] -> dst_yes.i_yes <- []; dst_no.i_no <- []
| x :: xs ->
  let dst_yes', i_yes', dst_no', i_no' =
    if p x then
      let dst' = x :: Hole in
      dst_yes.i_yes <- dst';
      dst', 1, dst_no, y_no
    else
      let dst' = x :: Hole in
      dst_no.i_no <- dst';
      dst_yes, i_yes, dst', 1
  in
  partition_dps
    dst_yes' i_yes' dst_right' i_right' p xs
```

References

Daniel P. Friedman and David S. Wise. Unwinding stylized recursions into iterations. Technical report, Indiana University, 1975.

Yasuhiko Minamide. A functional representation of data structures with a hole. In *POPL*, 1998.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *ICFP*, September 2013.

Tore Risch. REMREC - a program for automatic recursion removal in LISP. Technical report, Uppsala Universitet, 1973.

A Performance evaluation

In this section we present preliminary performance numbers for the TMC-transformed version of `List.map`, which appear to confirm the claim that this version is "almost as fast as the naive, non-tail-recursive version" – and supports lists of all length.

The performance results were produced by Anton Bachin's benchmarking script faster-map, which internally uses the core-bench library that is careful to reduce measurement errors on short-running functions, and also measures memory allocation. They are "preliminary" in that they were run on a personal laptop (running an Intel Core i5-4500U – Haswell), we have not reproduced results in a controlled environment or on other architectures. We ran the benchmarks several times, with variations, and the qualitative results were very stable.

A.1 The big picture

We graph the performance ratio of various `List.map` implementation, relative to the "naive tail-recursive version", that accumulates the results in an auxiliary list that is reversed at the end – the first tail-recursive version of our Prologue. We pointed out that this implementation is "slow" compared to the non-tail-recursive version: for most input lengths it is the slowest version, with other versions taking around 60-80% of its runtime (lower is better). One can also see that it is not *that* slow, it is at most twice as slow.

The other `List.map` versions in the graph are the following:

base The implementation of Jane Street's Base library (version 0.14.0). It is hand-optimized to compensate for the costs of being tail-recursive.

batteries The implementation of the community-maintained Batteries library. It is actually written in destination-passing-style, using an unsafe encoding with `Obj.magic` to unsafely cast a mutable record into a list cell. (The trick comes from the older Extlib library, and its implementation has a comment crediting Jacques Garrigue for the particular encoding used.)

containers Is another standard-library extension by Simon Cruanes; it is the hand-optimized tail-recursive implementation we included in the Prologue.

trmc is "our" version, the last version of the Prologue: the result of applying our implementation of the TMC transformation to the simple, non-tail-recursive version.

stdlib is the non-tail-recursive version that is in the OCaml standard library. (All measurements used OCaml 4.10)

stdlib unrolled 5x, trmc unrolled 4x are the result of manually unrolling the simple implementation (to go in the direction of the Base and Containers implementation); in the **trmc** case, the result is then TRMC-transformed.

Our expectation, before running measurements, are that **trmc** should be about as fast as **stdlib**, both slower than manually-optimized implementations (they were tuned to compete with the **stdlib** version). We hoped that **trmc unrolled 4x** would be competitive with the manually-optimized implementations. Finally, **batteries** should be about on-par with **trmc**, as it is using the same implementation technique (as a manual transformation rather than a compiler-supported transformation).

Actual results In the first half of the graph, up to lists of size 1000, the results are as we expected. There are three performance groups. The slow tail-recursive baseline alone. Then **batteries**, **stdlib** and **trmc** close to each other (**batteries** does better than the two other, which is surprising, possibly a code-size effect). Then the versions using manual optimizations: **base**, **containers**, and our **unrolled** versions.

At the far end of the graph, with lists of size higher than 10^5, the results are interesting, and very positive: **trmc** and **batteries** are the fastest versions, **containers** is slower. **base** falls back to the slow tail-recursive version after a certain input length, so its graph progressively joins the baseline on larger lists. (Note: later versions of Base switched to use an implementation closer to **containers** in this regime.)

We also got performance results on **stdlib** and **stdlib unrolled** on large lists, by configuring the system to remove the call-stack limit to avoid the stack overflows; their performance profile is interesting and non-obvious, we discuss it in Section A.3 (ulimit).

In the third quarter of the graph, for list sizes in the region $[5.10^3; 10^5]$, we observe surprising results where the destination-passing-style version (**trmc** and **batteries**) become, momentarily, sensibly slower than the non-tail-recursive **stdlib** version. We discuss this strange effect in details in Section A.2 (Promotion effects), but the summary is that it is mostly a GC effect due to the micro-benchmark setting (this strange region disappears with a slightly different measurement), and would not show up in typical programs.

A.2 Promotion effects

What happens in the $[5.10^3; 10^5]$ region, where destination-passing-style implementations seem to suddenly lose ground against **stdlib**? Promotion effects in the OCaml garbage collector.

Consider a call to `List.map` on an input of length N, which is in the process of building the result list. With the standard `List.map`, the result is built "from the end": first the list cell for the very last element of the list is allocated, then for the one-before-last element, etc., until the whole list is created. With the TMC-transformed `List.map`, the result list is built "from the beginning": a list cell is first allocated for the first element of the list (with a "hole" in tail positive) and written in the destination, then a list cell for the second element is written in the first cell's hole, etc., until the whole list is created.

The OCaml garbage collector is generational, with a small minor heap, collected frequently, and a large major heap, collected infrequently. When the minor heap becomes full, it is collected, and all its objects that are still alive are promoted to the major heap.

What if the minor heap becomes full in the middle of the allocation of the result list? Let's consider the average scenario where promotion happens at cell $N/2$. In both cases (**stdlib** or **trmc**), one half of the list cells are already allocated (the second or the first half, respectively), and they get promoted to the major heap. What differs is what happens *next*. With the non-tail-recursive implementation, the next list cell (in position $N/2 - 1$) is allocated, pointing to the $(N/2)$-th cell, and the process continues unchanged. In the destination-passing case, the next list cell $(N/2 + 1)$ is allocated, and it is written in the hole of the $(N/2)$-th cell.

At each minor collection, the garbage collector (GC) needs to know which objects are live in the minor heap. An object is live if it is a root (a global value in the program, a value on the call stack, etc.), if it is pointed to by a live object in the minor heap, or a live object on the *major* heap. The GC cannot afford to traverse the large major heap to determine the latter category, so it keeps track of all the pointers from the major to the minor heap; this is called "write barrier". At this point in our new list's life, writing the $(N/2+1)$-th cell to the $(N/2)$-th cell hits this write barrier, and the $(N/2)$-th cell is added to the "remembered set", of minor objects that are live due to the major heap. Trouble!

Hitting the write barrier has a small overhead, but this only happens once in the middle of constructing a very large list. When the next cell, the $(N/2 + 2)$-th list cell, gets written in the hole of the $(N/2 + 1)$-th cell, both are in the minor heap, so the write barrier does not come into play. Nothing happens until the next minor collection.

For the sake of simplicity, let's consider that the next minor collection only happens after the whole result list has been created. At this point, in the destination-passing-style version, the $(N/2 + 1)$-th cell is in the remembered set, so it will be promoted by the garbage collector to the major heap, along with all the objects that it itself references; those are (at least) the list cells from the middle to the end of the result list – in total $N/2 - 1$ list cells get promoted. In the **stdlib** version, they corresponds to the $N/2 - 1$ list cells at the beginning of the result list, allocated after the previous minor collection. They will *also* get promoted... if the result list is still alive at this point!

To recapituate, the "remainder" of the result list is promoted in all cases in the destination-passing version; it is only promoted in the non-tail-recursive version if the result list is still alive.

"But this is silly", you say, "who would call `List.map` on a very large list and drop the result immediately after?" Well, this code:

```
let make_map_test name map_function argument : Core_bench.Bench.Test.t =
  Core_bench.Bench.Test.create ~name
    (fun () ->
```

```
map_function ((+) 1) argument
|> ignore)
```

Remark 1. *There* are *real-world example where large results are short-lived. For example, consider a processing pipeline that calls* `map`*, and then* `filter` *on the result, etc.: the result of* `map` *may become dead quickly, if the lists are not too large. It is less likely to hit this promotion effect with medium-sized lists, but if this is all your application code is doing you will still see the effect once every L/M calls, where L is the length of your lists and M the size of the minor heap, adding to a small but observable promotionoverhead.*

If we change the code to a version that guarantees that the result is kept until at the next minor promotion, then there should be no difference in promotion behavior. We did exactly this, and the new graph looks like this:

This is the version that we consider representative of most applciations, where data is long-lived enough that the subtle promotion effects do not get into play. Notice in particular that, on this version, **trmc unrolled** is robustly better than all other implementations. (**stdlib unrolled** is still somewhat faster in the previous "awkward region", and we are not sure why, nor do we care very much.)

A.3 ulimit

Using `ulimit -s unlimited`, we removed the call-stack limit on our test machine, to test the speed of the non-tail-recursive `List.map` on large lists. These behaviors tend to be unobserved by OCaml users, which just get a program crash with a `Stack_overflow`.

It is interesting that the **stdlib** version gets progressively slower on large inputs, until it matches the performance of the tail-recursive baseline. Notice that the **stdlib unrolled** variant is also getting slower, although it starts with a good safety margin.

Pierre Chambart suggested that this slowdown may result from stack scanning: when the GC performs a minor collection, it scans the call stack to find root pointers to minor objects. When the result list gets much larger than the minor-heap size, many minor collections will occur during the `List.map` call, with progressively larger call stacks. The total overhead of the call-stack scanning is thus quadratic; scanning the stack is *fast*, but it eventually slows down those implementations noticeably. (In theory it would only get slower and slower as we increased the input size.)

Some implementations use the heap rather than a call stack; either the tail-recursive `List.map` in OCaml, or non-tail-recursive versions in a language that allocates call frames on the heap. Notice that those implementations do not suffer from such a quadratic slowdown, thanks to the generational GC: when the corresponding "heap frames" are scanned, they get moved to the major heap, and they will not get scanned again by minor collections, avoiding the quadratic behavior at this level – eventually enough memory is consumed that the major heap will need to be traversed regularly. It would be possible to change the OCaml runtime to use a similar approach for the system call stack: the runtime could keep track of which portions of the call stack have been traversed already, to not scan them again on the next minor collection. In fact, it seems that the OCaml runtime implements such an optimization on Power architectures only (?!). In our case it is of little pratical interest, however, speeding up scenarios that we never observe due to the system stack limit.

Type-safe type reflection for OCaml

Thierry Martinez

Inria, France

Abstract

This is a demonstration proposal for the `refl` package, which provides type-safe type reflection for OCaml. Generalized algebraic datatypes (GADTs) and extensible variant types are used to encode types and type declarations into values, covering most of the OCaml type system, including GADTs. The encoding of the type declarations into type witnesses allows printers, serializers, visitors, to be defined as regular functions parameterized by the type witnesses. The encoding is uniform so as to ensure separate compilation: type declarations and definitions of functions that operate on corresponding type witnesses can span distinct modules and be compiled separately. Moreover, the encoding offers strong type safety guarantees: the signature of functions can require that the type witnesses satisfy some properties regarding parameter variance, type system features or attributes. On the other hand, the correctness and the completeness of these functions is proved by type checking: values of the corresponding type are constructed and destructed without any run-time check.

https://github.com/thierry-martinez/refl

1 Introduction

Type-driven code generation is a technique of choice for writing code more declaratively: for a given data-structure, lots of companion functions, such as printing, serializing, visiting, etc. can be automatically deduced from the shape of the type declaration, reducing repetitive and error-prone manual coding. Type-driven code generation is baked into the Haskell specification since the `Haskell` 1.0 report [6], and tools such as `DrIFT` [14] or `Data.Derive` [11] have then added the ability to extend the mechanism with user-defined schemes for new derived functions. These tools have been ported to other programming languages, such as `ppx_deriving` for `OCaml` [13].

Thanks to these tools, the type declarations and the schemes of derived functions are kept separated. However, the programmer has to anticipate at the type declaration site which are the derivers to apply to the types being declared. This can be particularly problematic in the case of a data structure declared in a library, where the choice of the derivers applied to the data structure is fixed by the library itself. It would have been more convenient that the definition of the data structure and the definition of companion functions be orthogonal, so that the user can choose independently which data structure to use and which companion functions to apply.

The purpose of the `refl` package is to provide a generic deriver, which associates to each type a value that represents the type at run-time, so that companion functions can be defined as usual functions that takes the type witness as an additional parameter. The library is able to express all the standard derivers of `ppx_deriving` as companion functions: this covers `show` and `iter`, but also `compare`, `map`, etc.

The library is more general than `ppx_deriving` in several dimensions: companion functions such as `iter` and `map` are n-ary, records with polymorphic fields are supported, and GADTs are supported as well. The type information carried by the witness is precise enough for the companion functions to be defined without runtime check: that means that the OCaml compiler proves that (assuming termination) these functions are total over the values that their type accept as arguments.

Related work

The "Scrap Your Boilerplate" Haskell infrastructure proposed the *spiny* encoding [5] to fold and unfold types into generic views, relying on GADTs and higher-kinded polymorphic types. While OCaml type system does not support higher-kinded polymorphic types natively, a restricted form of them can be encoded thanks to extensible variant types [16], and the SYB framework has been ported to OCaml in some extents in several implementations: for instance, `ocaml-syb` [15], `generic` [1], `tpf` [2], and `lrt` [8]. None of these approaches cover GADTs nor records with polymorphic fields: the spiny encoding is name-less in the sense that type variables are encoded as type variables in the meta-representation, which compromises the ability of meta-reasoning on the variables, which is essential to handle GADTs or polymorphic fields, or even to implement a generic `map` function. Moreover, the spiny encoding does not reflect the type structure itself, which forces to make run-time checks for reasoning on multiple values.

2 Encoding OCaml type structures in GADTs

Given a type `t`, `refl` provides a value `[%refl: t]`, the *type witness* that represents the type `t`. The type of the type witness `[%refl: t]` is a GADT, the parameters of which reflect the structure of `t`. The first parameter is `t` itself: generic functions can perform *ad-hoc polymorphism* by destructing the type witness, each constructor refining the type of `t`. A type witness with a single parameter for `t` would be enough to express unary generic functions such as iterators or printers that accept to visit any types. The other parameters of the type witness enable the generic functions to express constraints on the accepted types: *e.g.*, the generic function `map` can express that its input type `'a t` and its output type `'b t` are two instances of the same type constructor; the generic function `compare` can restrict its input to only accept types without non-opaque closures; etc. Expressing such constraints involve some type-level data structures such as lists of type parameters and set of occurrences for type variables.

3 Applications, limits of the approach and future work

`refl` is used by the library `clangml` [3] which provides bindings for the `Clang` [10] front-end, and in particular an AST for the C [9] and C++ [7] languages, with more than 400 types and 3,000 constructors. The size of the data types involved in `clangml` has guided some choices in the type witness representation of `refl`: *e.g.*, using flat sequences and Peano natural numbers for representing and indexing data constructors makes the type-checking time more than quadratic in the number of constructors, which has been solved by using type-level binary trees. Moreover, while type reflection reveals the type structure, `clangml` requires `refl` to support some forms of abstraction: data structures have many abstract types and the whole AST is defined by a functor, the parameter of which specifies whether every node in the AST is strict or `lazy`.

`refl` supports all the builtin types, abstract types, array types, arrow types with and without labels, tuples, records with and without polymorphic fields, variants and polymorphic variants and GADTs and objects. First-class modules and extensible variants are not supported yet. No proper benchmarking has been done but switching from `ppx_deriving` to `clangml` in `MemCAD` made no observable run-time performance penalty, even on large programs. Interestingly, compile time, for the type-checking in particular, has increased, but efforts have been made to keep it down. Future works include the support of first-class modules and the ability to write generic functions on functor parameters: in particular, a generic `map` should be able to convert between `F(X).t` and `F(Y).t`, providing functions between the types of `X` and `Y`.

References

[1] Florent Balestrieri and Michel Mauny. Generic Programming in OCAML. In *The OCaml Users and Developers Workshop*, sep 2016.

[2] dkm pqwy. tpf, 2019–2020. https://github.com/pqwy/tpf/.

[3] Thierry Martinez et al. Clangml, 2018–2020. https://gitlab.inria.fr/memcad/clangml.

[4] Thierry Martinez et al. Clangml transforms, 2018–2020. https://gitlab.inria.fr/memcad/clangml-transforms.

[5] Ralf Hinze, Andres Löh, and Bruno C.d.S. Oliveira. "Scrap Your Boilerplate" reloaded. In Masami Hagiya and Philip Wadler, editors, *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer Berlin / Heidelberg, apr 2006.

[6] Paul Hudak, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, Kevin Hammond, John Hugues, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, and Jonathan Young. Haskell, a non-strict, purely functional language, version 1.0. *Available online https://wiki.haskell.org/Language_and_library_specification*, 1990.

[7] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, fifth edition, December 2017.

[8] Patrick Keller. LexiFi runtime types, 2020. https://github.com/lexifi/lrt.

[9] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[10] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[11] Neil Mitchell et al. Data.Derive, 2006–2017. http://hackage.haskell.org/package/derive.

[12] Xavier Rival. Abstract domains for the analysis of programs manipulating complex data-structures, 2011. Habilitation Thesis, https://www.di.ens.fr/~rival/papers/hdr.pdf.

[13] whitequark. ppx_deriving, 2014–2020. https://github.com/ocaml-ppx/ppx_deriving.

[14] Noel Winstanley and John Meacham. DrIFT, 2004–2006. https://hackage.haskell.org/package/DrIFT.

[15] Jeremy Yallop. ocaml-syb, 2015–2019. https://github.com/yallop/ocaml-syb.

[16] Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 119–135, Cham, 2014. Springer International Publishing.

OCaml sur circuit FPGA

Type d'article : démonstration d'outil

Jocelyn Sérot[1] and Emmanuel Chailloux[2]

[1] *Institut Pascal, UMR 6602 UCA/CNRS/SIGMA*
jocelyn.serot@uca.fr
[2] *Sorbonne Université, CNRS, LIP6, F-75005 Paris, France*
emmanuel.chailloux@lip6.fr

Résumé

On propose une démonstration d'O2B (OCaml On Board), un prototype d'infrastructure logicielle permettant l'exécution de programmes OCaml sur des circuits reprogrammables de type FPGA de la famille Intel. La démonstration s'effectuera avec la chaîne de développement Quartus Lite disponible gratuitement sur le site d'Intel [1]et une carte DE10-Lite (embarquant un FPGA Intel Max 10) de Terasic. L'ensemble des outils utilisés et les exemples utilisés sont téléchargeables à l'adresse https://github.com/jserot/O2B.

1 Introduction

Largement utilisés depuis un vingtaine d'années par la communauté des concepteurs de circuits numériques, les architectures de type FPGA (Field Programmable Gate Array) font désormais l'objet d'un intérêt croissant de la part des programmeurs au sens large. En offrant la possibilité de « tailler sur mesure » le matériel aux besoins du logiciel, ce type d'architecture offre en effet un moyen inédit de contourner les limitations des architectures de processeurs classiques. La programmation de ce type de circuit, dans l'état de l'art, passe toutefois par l'usage de langages dédiés à la description de matériel comme VHDL ou Verilog, à faible niveau d'abstraction et difficiles à utiliser par des programmeurs non familiarisés avec les architectures matérielles [4]. Un certain nombre de travaux ont cherché à utiliser des langages fonctionnels pour pallier cette difficulté. Deux voies ont été explorées. La première consiste à transformer directement le programme en une représentation au niveau *transfert de registres* (RTL), description intermédiaire sous la forme de portes logiques et de registres de mémorisation, classiquement utilisée en amont des outils de synthèse physique sur les circuits. C'est l'approche suivie dans les projets Lava [3], Cλash [2] (pour le langage Haskell) et HardCaml [2] (pour le langage OCaml). La seconde consiste à exécuter une machine virtuelle, interprétant le *bytecode* résultant de la compilation du programme, sur un ou plusieurs processeurs *softcore* implémenté sur le FPGA. Cette machine virtuelle peut alors être décrite classiquement en C. C'est l'approche suivie par le projet Hume [1] et l'outil dont on fera la démonstration.

2 Description

L'infrastructure logicielle mise en place est décrite sur la figure 1. Les programmes OCaml sont d'abord compilés en *bytecode* par le compilateur `ocamlc` puis le *bytecode* produit est transformé en un tableau C par l'outil `bc2c` de l'environnement OMicroB [5, 6] qui définit en C une machine virtuelle OCaml et son *runtime*. L'exécutable généré embarque alors la machine virtuelle OCaml et le *bytecode* du programme. Il est associé aux fonctions du *Board Support*

1. https://fpgasoftware.intel.com
2. https://github.com/janestreet/hardcaml

Package donnant accès aux ressources matérielles de la carte cible et compilé en un code binaire pour le processeur *softcore* instancié sur le FPGA. L'architecture de ce processeur, ainsi que le nombre et la nature des périphériques associés, est définie séparément. Dans notre cas, on utilise l'outil QSys de la chaîne Quartus. Cette étape génère un ensemble de fichiers VHDL qui constituent la description de la plateforme matérielle. C'est cette description, une fois synthétisée, toujours avec les outils de la chaîne Quartus, qui est utilisée pour configurer le FPGA (et en particulier instancier le processeur *softcore*, un Nios2 dans notre cas).

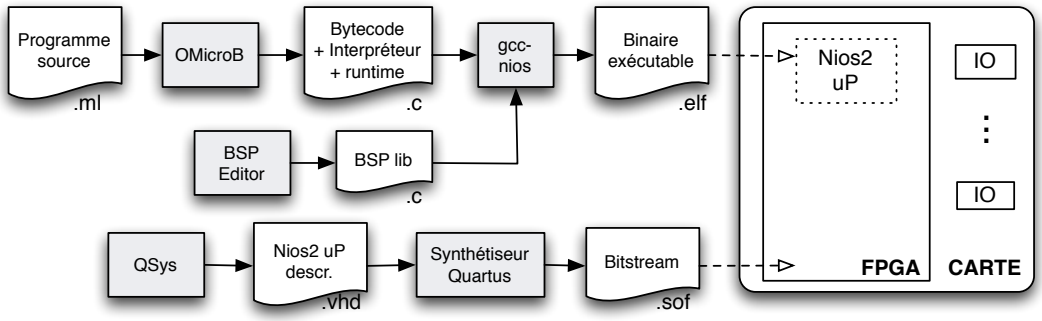


Figure 1 – Infrastructure logicielle de l'outil

Utilisée telle quelle, l'infrastructure décrite ci-dessus permet déjà d'augmenter sensiblement le niveau d'abstraction des programmes. Les opérations accédant aux dispositifs d'entrées-sorties peuvent en particulier être encapsulées au sein de modules pour une plateforme donnée. Plusieurs programmes mettant ces possibilités en évidence et s'exécutant sur une plateforme matérielle constituée d'une carte DE10-Lite – une cible d'entrée de gamme embarquant un FPGA de type Max10 (50000 blocs logiques, 2 Mo RAM interne, 128 Mo SDRAM externe) – sont disponibles sur le site d'O2B et seront présentés lors de la démo.

Il est aussi possible de définir des plateformes matérielles dédiées au sein desquelles certaines opérations du *bytecode* sont implémentées soit directement en C (via la mécanisme d'appel de fonctions externes d'OCaml) soit sous la forme de *custom instructions* du processeur Nios2, décrites au niveau RTL (en VHDL par exemple) et implémentées en matériel au niveau porte sur le FPGA. On montrera comment cette seconde approche, sur un simple programme de calcul du PGCD permet d'accélérer de manière considérable son exécution, en ramenant le nombre de cycles d'horloges requis par itération de l'algorithme de 4300 à 1.

3 Perspectives

Même s'il ne s'agit à ce stade que d'une simple démonstration de faisabilité, le travail illustré ici ouvre de nombreuses perspectives. On évoquera ainsi deux extensions possibles et riches d'applications. La première consiste à dériver automatiquement le code VHDL RTL des fonctions à accélérer à partir du programme OCaml (ce code est actuellement écrit à la main et intégré à la plateforme via l'outil QSys). La seconde consiste à instancier sur le FPGA non pas un seul processeurs *soft core* mais plusieurs, afin d'exécuter les programmes de manière parallèle.

Références

[1] Wim Vanderbauwhede Abdallah Al Zain et greg Michaelson. Hume to fpga. In P. Kitsos, editeur, *Proceedings of 10th International Symposium on Trends in Functional Programming*, University of Oklaoma, May 2010.

[2] Marco Egbertus Theodorus Gerards, C.P.R. Baaij, Jan Kuper, et Matthijs Kooijman. Higher-order abstraction in hardware descriptions with clash. In P. Kitsos, editeur, *Proceedings of the 14th EUROMICRO Conference on Digital System Design, DSD 2011*, pp. 495–502, United States, August 2011. IEEE Computer Society.

[3] Mary Sheeran Per Bjesse, Koen Claessen et Satnam Singh. Lava : Hardware Design in Haskell. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, ICFP'98, pp. 174–184. ACM, 1998.

[4] Jocelyn Sérot. *La programmation des circuits FPGA et le langage VHDL. Une introduction pour les programmeurs et par l'exemple.* Ellipse, 2019.

[5] Benoit Vaugon Steven Varoumas, Basile Pesin et Emmanuel Chailloux. Programming microcontrollers through high-level abstractions. In *Proceedings of the 12th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL@SPLASH 2020*, 2020.

[6] Steven Varoumas. *Modèles de programmation de haut niveau pour microcontrôleurs à faibles ressources. (High-level programming models for microcontrollers with scarce resources).* PhD thesis, Sorbonne University, Paris, France, 2019.

`coqffi`: Outil pour la génération automatique de FFI Coq/OCaml

Thomas Letan[1] and Li-yao Xia[2]

[1] Agence nationale de la sécurité des systèmes d'information (ANSSI)
Paris, France
thomas.letan@ssi.gouv.fr
[2] University of Pennsylvania, Philadelphia, Pennsylvania, U.S.A.
xialiyao@seas.upenn.edu

Résumé

Dans cet article, nous présentons `coqffi`, un outil pour générer automatiquement des FFI Coq à partir d'interfaces OCaml. Plus précisément, `coqffi` génère le code verbeux permettant la liaison à ces interfaces OCaml via le mécanisme d'extraction de Coq. Utilisé conjointement avec le système de compilation `dune`, dont les versions récentes permettent de compiler et d'extraire des théories Coq, `coqffi` permet de mélanger de manière transparente des modules OCaml et Coq au sein d'une même base de code.

1 Introduction

Le mécanisme d'extraction de Coq [5] permet de *transpiler* du Gallina en OCaml, afin de tirer bénéfice du compilateur optimisant OCaml pour exécuter efficacement des programmes certifiés en Coq. Gallina étant un langage de programmation fonctionnel *pur*, c'est-à-dire sans effets de bord, l'utilisation de Coq a tendance à rester confinée à des composants très spécifiques — mais aussi critiques — de logiciels. CompCert [2], un compilateur pour le langage C, est sans doute l'exemple le plus emblématique de cet usage du mécanisme d'extraction. Ses passes de compilation certifiées sont des fonctions pures écrites en Gallina.

Depuis plusieurs années, de nombreux travaux de recheche se sont penchés sur l'écriture et la vérification de code impur en Gallina. C'est le cas de `Coq.io` [1], FreeSpec [3] et Interaction Tree [6] par exemple. Ces trois cadriciels sont compatibles avec le mécanisme d'extraction de Coq et permettent d'envisager un développement logiciel où Gallina est le langage « principal » et où OCaml est cantonné à l'implémentation de bibliothèques logicielles « bas-niveau ».

Dans les deux cas, le mécanisme d'extraction est malheureusement lourd à utiliser, faisant appel à des commandes vernaculaires rébarbatives et sujettes à erreurs. Mal utilisé, il peut ainsi être la source de bogues et vulnérabilités dans le programme OCaml généré. Dans cet article, nous présentons `coqffi`, un outil permettant de générer automatiquement des FFI Coq-OCaml. L'objectif de cet outil est de réduire drastiquement la difficulté d'utilisation du mécanisme d'extraction de Coq. `coqffi` a été publié au sein de l'organisation `coq-community` sur GitHub afin de favoriser son adoption par la communauté Coq [4] [1].

2 Présentation des fonctionnalités

`coqffi` prend en entrée une interface de module OCaml compilée (`.cmi`) et génère en sortie un module Coq (`.v`). Pour chaque déclaration présente dans l'interface de l'entrée, le module généré en sortie contient deux commandes vernaculaires Coq : une définition Coq possédant

1. https://github.com/coq-community/coqffi

```
val send_msg :
    string -> unit [@@ impure]
```

(**1**) Interface OCaml

```
Axiom ml_send_msg : string -> IO unit.
Extract Constant ml_send_msg =>
    "(fun x k -> k (MyLib.send_msg x))".
```

(**2**) Interface Coq générée

un type « compatible » (du point de vue du mécanisme d'extraction) et la configuration du mécanisme d'extraction. `coqffi` traduit récursivement les types OCaml en types Coq. Un type polymorphique OCaml est ainsi introduit avec l'opérateur `forall` Coq. Les fonctions OCaml sont traduites par des fonctions en Gallina. `coqffi` traduit un ensemble de « types primitifs » OCaml bien identifiés vers des types « compatibles » Coq. Enfin, les types OCaml définis par l'interface sont laissés inchangés.

Par ailleurs, `coqffi` distingue deux types de valeurs OCaml, selon si elles sont déclarées pures (par défaut) ou impures (par le biais d'une annotation `[@@impure]`). En ce qui concerne les valeurs impures, `coqffi` axiomatise une monade `IO` et des directives d'extraction en conséquence. Afin de demeurer générique et de laisser la porte ouverte à l'utilisation d'autres monades, `coqffi` génère par ailleurs une classe de types qui généralise l'interface à d'autres monades, et une instance de cette classe pour le type `IO`.

Par défaut, les valeurs et types OCaml sont exposés en Coq sous la forme d'axiomes. Les utilisateurs de `coqffi` peuvent par la suite introduire des hypothèses sur les comportements de ces définitions, si leur but est de conduire un travail de vérification formelle sur le logiciel qu'ils sont en train d'implémenter. Il est aussi possible d'assigner à une valeur pure OCaml un modèle implémenté en Coq (par le biais de l'annotation `[@@coq_model <model>]`). Dans ce cas là, le terme Coq défini pour la valeur concernée sera définie comme un alias vers son modèle, plutôt que comme un axiome. `coqffi` est aussi capable de générer des types inductifs pour un sous-ensemble de types OCaml[2]. Dans ce cas, le mécanisme d'extraction est configuré afin de projeter les types inductifs ainsi définis vers les types OCaml.

Enfin, `coqffi` s'utilise facilement avec `dune`, ce qui permet de mélanger au sein du même dossier du code Coq et OCaml. Nous souhaitons nous rapprocher des développeurs de `dune` afin de déterminer s'il est envisageable de proposer une *stanza* dédiée, similaire à `coq.theory` et `coq.extraction`.

3 Conclusion

Le principal objectif de `coqffi` est de faciliter l'utilisation du mécanisme d'extraction de Coq, dont la configuration est rapidement lourde et sujette à erreurs. Couplé avec `dune`, il permet de mélanger facilement, au sein de la même base de code, des modules Coq et OCaml. Nous espérons que `coqffi` saura trouver son public au sein de la communauté et favorisera l'utilisation de Coq dans des développement logiciels souhaitant tirer parti de ses fonctionnalités d'assistant de preuves.

Références

[1] Guillaume Claret and Yann Régis-Gianas. Mechanical Verification of Interactive Programs Specified by Use Cases. In *Proceedings of the Third FME Workshop on Formal Methods in Software Engineering*, pages 61–67. IEEE Press, 2015.

2. Il s'agit d'une fonctionnalité expérimentale, désactivée par défaut.

[2] Xavier Leroy et al. The CompCert Verified Compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[3] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular Verification of Programs with Effects and Effects Handlers in Coq. In *22st International Symposium on Formal Methods (FM 2018)*, volume 10951. Springer, 2018.

[4] Thomas Letan, Yann Régis-Gianas, Li-yao Xia, and Yannick Zakowski. coqffi : Coq to ocaml ffi made easy. https ://coq.inria.fr/.

[5] Pierre Letouzey. A new extraction for coq. In *International Workshop on Types for Proofs and Programs*, pages 200–219. Springer, 2002.

[6] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction Trees : Representing Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL) :1–32, 2019.

Mécanisation du modèle RC11 et de la propriété DRF-SC

Quentin Ladeveze

INRIA, France
quentin.ladeveze@inria.fr

Résumé

Un modèle mémoire décrit les comportements possibles d'un programme concurrent réalisant des accès à la mémoire partagée. Dans le modèle mémoire SC, ces comportements sont ceux que l'on pourrait observer si l'on exécutait séquentiellement un entrelacement des instructions des différents *threads* du programme. Ce modèle simple est généralement celui qui est attendu par le programmeur. Cependant, il exclut des comportements, dits relâchés, induits par les processeurs et les compilateurs lorsque ces derniers réalisent des optimisations. Pour que les utilisateurs puissent bénéficier de la simplicité de SC, tout en permettant de bonnes performances, on souhaite que les modèles mémoire de ces processeurs et de ces langages de programmation vérifient la propriété DRF-SC. Cette propriété garantit que si les exécutions conformes à SC d'un programme ne contiennent aucune *race*, le programme ne présentera aucune exécution non conforme à SC. Autrement dit, en respectant une condition simple, l'utilisateur peut ignorer le modèle mémoire de son langage et de son architecture. Le modèle RC11 est un modèle mémoire formel des programmes C/C++ respectant le standard C11. Les auteurs de ce modèle accompagnent ce dernier d'une preuve qu'il respecte la propriété DRF-SC. Nous présentons la mécanisation dans l'assistant de preuves Coq de ce modèle mémoire et de cette preuve. De plus, nous explorons les limites d'application de cette propriété dans le cadre des programmes C11. Enfin, nous discutons des façons dont cette preuve pourrait être adaptée à d'autres modèles et modifiée pour s'appliquer à une classe moins restreinte de programmes.

1 Introduction

Un modèle mémoire décrit le comportement de programmes réalisant des accès concurrents à une mémoire partagée. Dans le modèle *séquentiellement consistant* (SC) [8], le résultat de l'exécution d'un programme est tel qu'il serait si l'on exécutait ses instructions séquentiellement, en respectant l'ordre dans lequel elles apparaissent dans le programme. C'est généralement ce qui est attendu et souhaité par le programmeur.

Cependant, les programmes peuvent présenter des comportements non conformes au modèle SC. En effet, le compilateur traduisant ces programmes en instructions assembleur et le processeur exécutant ces instructions peuvent tous deux réaliser des optimisations.

```
x = 1;     || y = 1;
print(y);  || print(x);
```

(a) Possibilité d'optimisation matérielle

```
x = 1;  || print(x);
x = 2;  ||
```

(b) Possibilité d'optimisation logicielle

FIGURE 1 – Programmes pouvant présenter des comportement inattendus à cause d'optimisations

Dans le programme 1a par exemple, les écritures peuvent être d'abord réalisées dans un cache (un *store buffer*) des processeurs sur lesquels elles sont exécutées. Les lectures qui les suivent peuvent ensuite être effectuées avant que ces écritures aient atteint la mémoire centrale. Ce programme pourrait avoir pour résultat d'afficher deux fois 0 (la valeur initial de `x` et `y`).

Dans le programme 1b, l'écriture de la valeur 1 à l'emplacement `x` peut être supprimée par le compilateur, car elle précède directement l'écriture d'une autre valeur au même emplacement.

Ce programme ne pourrait donc jamais avoir pour résultat d'afficher la valeur 1, alors que c'est un des comportement que l'on peut attendre.

Ces optimisations n'affectent pas le résultat des programmes séquentiels, mais peuvent donner des comportements différents aux programmes concurrents par rapport à ceux attendus dans le modèle SC.

Un comportement non conforme à SC correspond le plus souvent à ce qu'un des *threads* du programme perçoive les effets des accès mémoire d'un autre *thread* dans un autre ordre que celui du programme. On dit que des instructions du programme peuvent être *réordonnées*. Lorsque de tels comportements sont possibles, on se trouve alors dans un modèle mémoire dit *relâché*.

Pour l'écriture et l'utilisation d'un compilateur d'un langage de haut niveau, le modèle mémoire doit être pris en compte à deux niveaux. Premièrement, il est souhaitable pour ce compilateur d'offrir un modèle mémoire uniforme, quelle que soit l'architecture cible. Le compilateur doit donc prendre en compte le modèle mémoire de chacune des architectures qu'il supporte, et générer un code assembleur en conséquence. Deuxièmement, le programmeur doit prendre en compte le modèle mémoire fourni par le compilateur pour anticiper les comportements de ses programmes.

Afin d'éviter les ambiguïtés, on peut exprimer les comportements possibles dans un modèle mémoire formalisé mathématiquement, qui définit précisément l'ensemble des résultats attendus lorsque l'on exécute un programme donné. Ce modèle mémoire formel définit notamment l'ensemble des instructions qui peuvent être réordonnées.

Un des formalismes possibles pour décrire un modèle mémoire est le formalisme axiomatique [1,2]. Dans ce cadre, une exécution est représentée par un ensemble de relations entre des accès à la mémoire partagée. On définit ensuite la *conformité* d'une exécution au modèle mémoire par un ensemble de contraintes, généralement d'acyclicité, sur ces relations.

Même lorsqu'ils sont formalisés précisément, les modèles mémoires des architectures et des langages les plus répandus restent souvent complexes à utiliser pour le programmeur. Pour cette raison, ces modèles sont généralement conçus pour vérifier la propriété DRF-SC : si aucune des exécutions conformes au modèle SC d'un programme ne contient de *races*, alors toutes les exécutions de ce programme conformes à un modèle qui respecte la propriété DRF-SC sont conformes au modèle SC. Une *race* est un accès simultané par deux *threads*, dont au moins une écriture, à un même emplacement mémoire (pour une définition formelle de cette notion, voir la section 3.2). Cette propriété permet au programmeur d'ignorer complètement le modèle mémoire relâché et de considérer le modèle SC à la place, à condition de respecter certaines règles.

Pour faciliter la vie du programmeur, certains langages proposent plusieurs types d'accès à la mémoire partagée. On peut les classer en deux grandes catégories : non-atomiques et atomiques.

Les premiers sont prévus pour accéder aux données locales à un *thread*, et une *race* entre deux accès de ce type est considérée comme une erreur de programmation.

Les seconds sont spécifiquement conçus pour la communication entre *threads*. Le compilateur fournit donc des garanties sur l'ordre dans lequel ils seront exécutés. Pour ce faire, le compilateur utilise des instructions assembleur supplémentaires destinées à synchroniser ces accès, comme par exemple des barrières. Ces accès à la mémoire sont donc moins efficaces que des accès mémoire non-atomiques.

RC11 [7] est un modèle mémoire axiomatique pour les programmes C/C++ respectant le standard C11. Ce modèle mémoire est accompagnée d'une preuve stipulant qu'il vérifie bien la condition DRF-SC. Nous présentons la formalisation dans l'assistant de preuves Coq de ce modèle et de cette preuve.

2 Le modèle RC11

Nous commencerons par présenter les détails du modèle RC11. Nous décrirons dans un premier temps les fonctionnalités spécifiques du standard C11 qui aident l'utilisateur à contrôler les comportements relâchés de son modèle mémoire. Nous poursuivrons par la formulation axiomatique du modèle qui se découpe en deux étapes. On génère d'abord l'ensemble des *exécutions* (représentées par des graphes) possibles d'un programme. On filtre ensuite ces exécutions pour ne garder que celles qui sont conformes au modèle en leur appliquant des prédicats de conformité.

2.1 Contrôle de la conformité

Le standard C11 contient plusieurs outils destinés à aider les programmeurs à écrire des programmes concurrents correctement synchronisés.

Premièrement, il définit plusieurs types d'accès atomiques. Chacun d'entre eux fournit des garanties plus ou moins fortes sur l'ordre d'exécution des accès, avec une pénalité de performance en conséquence :

1. Les atomiques SC sont les plus forts et les plus coûteux à implémenter. La sémantique souhaitée est qu'un programme qui ne comprend des *races* que sur des accès atomiques SC ait un comportement conforme à SC.
2. Les atomiques *release-acquire* (RA) ont pour objectif de permettre de passer des messages entre plusieurs threads, mais sans le coût d'accès SC. Les écritures en mémoire peuvent être des opérations *release* et les lectures peuvent être des opérations *acquire*. La sémantique souhaitée pour ces accès est la suivante : lorsqu'une lecture *acquire* lit une valeur écrite par une écriture *release* ou une écriture atomique qui suit une écriture *release* dans le programme, les effets de tous les accès à la mémoire précédant l'écriture *release* dans l'ordre du programme sont visibles pour tous les accès mémoire qui suivent la lecture *acquire* dans l'ordre du programme.
3. Les atomiques relâchés sont traduits par le compilateur en un accès unique à la mémoire, sans l'ajout d'aucune primitive de synchronisation dans le code assembleur. L'atomicité de ces accès est garanti, ce qui peut être important lorsque l'on accède une valeur de 64 bits sur une machine 32bits par exemple.

Deuxièmement, le standard fournit des *barrières* utilisables dans le langage de haut niveau, qui permettent d'indiquer au compilateur qu'il doit insérer des barrières au niveau assembleur à un emplacement spécifique. Comme pour les accès atomiques, il existe plusieurs types de barrières :

1. Les barrières peuvent être *release*, *acquire* ou les deux. Ces barrières peuvent se combiner de trois façons.
   - lorsqu'une barrière *release* précède une écriture atomique dont la valeur est lue par une lecture atomique suivie d'une barrière *acquire*, les effets des accès précédant la barrière *release* sont visibles pour les accès qui suivent la barrière *acquire*.
   - lorsqu'une barrière *release* précède une écriture atomique donc la valeur est lue par une lecture *acquire*, les effets des accès précédant la barrière *release* sont visibles pour les accès qui suivent la lecture *acquire*.
   - lorsque la valeur d'une écriture *release* est lue par une lecture atomique suivie d'une barrière *acquire*, les effets des accès précédant la lecture *release* sont visibles pour les accès qui suivent la barrière *acquire*.
2. Une barrière SC interdit tout réordonnancement entre des accès effectués par les instructions qui la précèdent et celles qui la suivent.

Enfin, le standard définit une opération `atomic_exchange` qui permet de lire la valeur stockée à un emplacement, puis d'y écrire une nouvelle valeur. On nomme ces opérations des accès *read-modify-write*. Le standard garantit que la valeur inscrite à cet emplacement mémoire ne sera pas modifiée entre la lecture et l'écriture qui correspondent à cette opération.

2.2 Notations

Avant de décrire l'ensemble des exécutions associées à un programme, nous introduisons quelques notations. On note R^+, $R^?$ et R^* respectivement la clôture transitive, réflexive et réflexive-transitive d'une relation binaire R. On note R^{-1} l'inverse de la relation R. On note $R_1; R_2$ la composition de deux relations R_1 et R_2. L'opérateur de composition est prioritaire par rapport à l'opérateur d'union. Une relation R est acyclique si R^+ est irréflexive.

On note $[A]$ la relation identité restreinte à l'ensemble A. On note $\mathsf{dom}(R)$ et $\mathsf{codom}(R)$ respectivement le domaine et le codomaine de la relation R.

Pour une fonction f, on note $=_f$ l'ensemble des paires d'éléments dont l'image par f est identique et $\neq_f$ l'ensemble des paires d'éléments dont l'image par f est différente. On note $R|_f$, la relation $(R \cap =_f)$ et $R|_{\neq f}$ la relation $(R \cap \neq_f)$.

Si R est un ordre partiel strict, $R|_{imm}$ désigne les *liens immédiats* dans R, c'est à dire les paires $\langle a, b\rangle \in R$ telles que pour tout c, $\langle a, c\rangle \in R$ implique $\langle b, c\rangle \in R^?$ et $\langle c, b\rangle \in R$ implique $\langle c, a\rangle \in R^?$. L'intuition derrière cette notion de lien immédiat est de dire qu'il n'y pas d'éléments "*entre*" a et b dans R quand $\langle a, b\rangle \in R|_{imm}$.

On suppose également un ensemble Loc d'emplacements mémoire et un ensemble Val de valeurs. On utilise comme méta-variables x, y, z pour des emplacements mémoires et v pour des valeurs.

On note `na`, `rlx`, `acq`, `rel`, `acqrel` et `sc` les différents modes d'accès à la mémoire, respectivement les accès non-atomiques, relâchés, *acquire*, *release*, *release-acquire* et SC. La figure 2 montre la façon dont les accès à la mémoire sont ordonnés par $\sqsubset$.

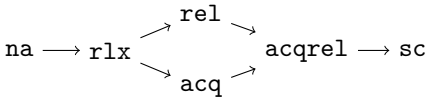


FIGURE 2 – Ordonnancement des modes d'accès par $\sqsubset$

2.3 L'ensemble des exécutions

2.3.1 Évènements

La première composante d'une exécution est un ensemble fini `E` d'évènements. La figure 3 présente la définition en COQ d'un évènement. On peut voir que le type `Event` peut représenter une lecture, une écriture ou une barrière.

```
Inductive Event : Type :=
| Read (eid: nat) (m: Mode) (l: Loc) (v: Val)
| Write (eid: nat) (m: Mode) (l: Loc) (v: Val)
| Fence (eid: nat) (m: Mode).
```

FIGURE 3 – La définition en COQ d'un évènement

Chaque évènement a un identifiant, représenté par un entier naturel et un mode d'accès (non-atomique ou atomique d'un certain type). Les lectures et les écritures sont également

constituées d'un emplacement mémoire et d'une valeur écrite à ou lue de cet emplacement. Dans cette formalisation, les emplacements mémoire et les valeurs sont représentés par des entiers naturels. L'ensemble $\mathtt{E}$ des évènements est bien formé si tous ses éléments ont un identifiant différent, et si les modes d'accès sont cohérents (une écriture ne peut pas avoir pour mode `rel` par exemple). Ces deux conditions sur l'ensemble des évènements sont rassemblées dans la définition `valid_evts`.

On peut noter que cette formalisation est presque identique à la définition d'origine dans le modèle RC11 [7, section 3.1]. La seule différence réside dans l'encodage des identifiants. La définition d'origine représente les évènements d'une exécution par une fonction qui associe à chaque identifiant un label, composé d'un mode d'accès, d'une valeur et d'un emplacement. Cette définition est équivalente à la définition en Coq que l'on a présentée, associée à la condition que chaque évènement a un identifiant différent.

Six fonctions $\mathtt{typ}$, $\mathtt{mod}$, $\mathtt{id}$, $\mathtt{loc}$, $\mathtt{val_r}$ et $\mathtt{val_w}$ permettent d'accéder respectivement au type ($\mathtt{R}$, $\mathtt{W}$ ou $\mathtt{F}$), au mode d'accès, à l'identifiant, à l'emplacement affecté et à la valeur lue ou écrite d'un évènement. Les fonctions $\mathtt{val_r}$, $\mathtt{val_w}$ et $\mathtt{loc}$ sont partielles, car elles ne s'appliquent pas aux barrières. De même, $\mathtt{val_r}$ ne s'applique qu'aux lectures et $\mathtt{val_w}$ qu'aux écritures.

On utilise a, b, c comme méta-variables pour des évènements. Pour chaque type d'évènement $\mathtt{T} \in \{\mathtt{R}, \mathtt{W}, \mathtt{F}\}$, $\mathtt{T}$ désigne également l'ensemble $\{a \in \mathtt{E} \mid \mathtt{typ}(e) = \mathtt{T}\}$. Les ensembles d'évènements peuvent également être annotés par un emplacement en indice et par un exposant pour le mode. Par exemple, $\mathtt{W}_x^{\sqsupseteq \mathtt{rlx}}$ désigne l'ensemble des écritures à l'emplacement x dont le mode est au moins relâché.

L'ensemble $\mathtt{E}$ contient un ensemble d'évènements d'initialisation $\mathtt{E}_0 = \{\mathtt{W}^{\mathtt{na}}(i, x, 0) \mid x \in \mathsf{Loc}\}$ où les identifiants i sont choisis tels que $\mathtt{E}$ est bien formé.

2.3.2 Relations

Une exécution est également constituée de quatre relations entre évènements :

1. $\mathtt{sb}$ (*sequenced-before*) est une relation d'ordre strict (parfois aussi notée $\mathtt{po}$ pour *program order*), telle que $\mathtt{sb} \subseteq \mathtt{E} \times \mathtt{E}$. Cette relation ordonne les évènements d'un même *thread* dans l'ordre dans lequel ils apparaissent dans le programme. Elle doit également ordonner les évènements d'initialisation avant tous les autres évènements (on a $\mathtt{E}_0 \times (\mathtt{E} \setminus \mathtt{E}_0) \subseteq \mathtt{sb}$). Ces conditions de validité sont rassemblées dans la définition `valid_sb`.
2. $\mathtt{rf}$ (*reads-from*) est une relation binaire qui associe à une écriture a, les lectures qui lisent de la mémoire la valeur écrite par a. Cette relation doit satisfaire quatre conditions :
   (a) $\mathtt{rf} \subseteq [\mathtt{W}]; =_{\mathtt{loc}}; [\mathtt{R}]$.
   (b) Pour tout $\langle a, b\rangle \in \mathtt{rf}$, on a $\mathtt{val_w}(a) = \mathtt{val_r}(b)$.
   (c) $\langle a_1, b\rangle, \langle a_2, b\rangle \in \mathtt{rf}$ implique $a_1 = a_2$.
   (d) $\mathtt{R} \subseteq \mathsf{codom}(\mathtt{rf})$.

   Ces conditions imposent que la relation lie une écriture à une lecture, que ces deux évènements affectent un même emplacement et qu'elles lisent ou écrivent la même valeur. De plus, $\mathtt{rf}$ doit associer une unique écriture à chaque lecture, signifiant qu'une valeur lue en mémoire doit nécessairement y avoir été écrite. Ces conditions de validité sont rassemblées dans la définition `valid_rf`.
3. $\mathtt{mo}$ (*modification order*) est une relation d'ordre strict sur $\mathtt{W}$ (parfois aussi notée $\mathtt{co}$ pour *coherence order*). Elle relie des évènements affectant le même emplacement mémoire. Pour tout $x \in \mathit{Loc}$, la restriction de $\mathtt{mo}$ aux évènements lisant de ou écrivant sur x (notée $\mathtt{mo}_x$) est une relation d'ordre total sur $\mathtt{W}_x$. Cette relation décrit l'ordre dans lequel les effets des écritures en mémoire deviennent visibles à tous les *threads* du programme. Ces conditions de validité sont rassemblées dans la définition `valid_mo`.

4. rmw (*read-modify-write*) est une relation binaire qui permet de traduire les accès *read-modify-write*. Notre formalisation traduit ces opérations en une lecture suivie immédiatement par une écriture dans sb. On a $\texttt{rmw} \subseteq [\texttt{R}]; (\texttt{sb}|_{imm} \cap =_{loc}); [\texttt{W}]$. Pour toute paire d'évènements $\langle a, b\rangle \in \texttt{rmw}$, on a :

$$\langle \texttt{mod}(a), \texttt{mod}(b)\rangle \in (\langle \texttt{rlx}, \texttt{rlx}\rangle, \langle \texttt{rel}, \texttt{rlx}\rangle, \langle \texttt{rlx}, \texttt{acq}\rangle, \langle \texttt{rel}, \texttt{acq}\rangle, \langle \texttt{sc}, \texttt{sc}\rangle)$$

On note At le sous-ensemble des évènements de E reliés par rmw. Ces conditions de validité sont rassemblées dans la définition valid_rmw.

On notera respectivement $G.\texttt{sb}$, $G.\texttt{rf}$, $G.\texttt{mo}$ et $G.\texttt{rmw}$ les relations *sequenced-before*, *reads-from*, *modification order* et *read-modify-write* d'une exécution G. De plus, on note $G.\texttt{rf}|_{\texttt{sc}}$ la relation $([G.\texttt{E}^{\texttt{SC}}]; G.\texttt{rf}; [G.\texttt{E}^{\texttt{SC}}])$. La notation équivalente existe pour sb, mo et rmw.

Dans cet article, nous considérons uniquement les programmes concurrents qui sont un ensemble fini de programmes séquentiels, exécutés en parallèle chacun sur un *thread*. La construction d'une exécution d'un programme est alors définie inductivement sur la structure des programmes séquentiels qui le composent.

Bien sûr, cette définition dépend du langage source, mais elle suit toujours le même schéma. À l'ensemble des accès mémoire réalisés lors d'une exécution d'un programme, on ajoute une relation sb qui encode l'ordre des instructions ayant généré ces accès dans le programme. Puis on ajoute des relations rf, mo et rmw arbitraires, mais respectant les conditions ci-dessus.

On peut retrouver les définitions des évènements et des relations d'une exécution dans le fichier exec.v du développement Coq.

2.4 La conformité d'une exécution

La validité d'une exécution se décompose en quatre conditions à vérifier. Pour les exprimer de façon compréhensible, nous allons d'abord introduire d'autres relations, dérivées de celles qui composent une exécution.

2.4.1 Les relations rb et eco

$$\texttt{rb} \triangleq \texttt{rf}^{-1}; \texttt{mo} \qquad (\textit{reads-before})$$

La relation rb (*reads-before*, parfois notée fr pour *from-read*) relie les lectures d'une valeur en mémoire aux écritures qui sont effectuées au même emplacement après cette lecture.

$$\texttt{eco} \triangleq (\texttt{rf} \cup \texttt{mo} \cup \texttt{rb})^{+} \qquad (\textit{extended coherence order})$$

La relation eco contient l'ensemble des paires d'évènements telles que l'effet du premier évènement est visible pour le second évènement. En effet, l'effet d'une écriture est visible pour les lectures qui lisent cette valeur, ce qui est exprimé par rf. Par définition, mo décrit l'ordre dans lequel les écritures sont visibles les unes pour les autres. Et enfin, rb (*reads-before*) assure que la lecture d'une valeur en mémoire soit effectuée avant que cette valeur ne soit changée par une autre écriture.

2.4.2 La relation hb

La relation hb (*happens-before*) est utilisée pour exprimer la sémantique des atomiques *release-acquire*. Une paire $\langle x, y\rangle \in \texttt{hb}$ signifie que l'effet de l'accès mémoire x doit être visible de l'évènement y, et ce à cause d'une paire d'évènements *release-acquire*. La relation hb est définie à partir de deux relations plus basiques :

$$\texttt{rs} \triangleq [\texttt{W}]; \texttt{sb}|^{?}_{\texttt{loc}}; [\texttt{W}^{\sqsupseteq\texttt{rlx}}]; (\texttt{rf}; \texttt{rmw})^{*} \qquad (\textit{release-sequence})$$

$$\texttt{sw} \triangleq [\texttt{E}^{\sqsupseteq\texttt{rel}}]; ([\texttt{F}]; \texttt{sb})^{?}; \texttt{rs}; \texttt{rf}; [\texttt{R}^{\sqsupseteq\texttt{rlx}}]; (\texttt{sb}; [\texttt{F}])^{?}; [\texttt{E}^{\sqsupseteq\texttt{acq}}] \qquad (\textit{synchronizes-with})$$

$$\texttt{hb} \triangleq (\texttt{sb} \cup \texttt{sw})^{+} \qquad (\textit{happens-before})$$

On peut voir que hb ordonne les paires d'éléments appartenant à sw, c'est à dire les paires d'éléments *release-acquire* qui se *synchronisent* l'un avec l'autre. Il ordonne également tous les évènements qui précèdent le premier élément d'une paire dans sw avec ceux qui suivent le second élément de cette même paire.

$$\underbrace{[\texttt{E}^{\sqsupseteq\texttt{rel}}]}_{①}; \underbrace{([\texttt{F}]; \texttt{sb})^{?}}_{②}; \underbrace{[\texttt{W}]; \texttt{sb}|^{?}_{\texttt{loc}}}_{③}; \underbrace{[\texttt{W}^{\sqsupseteq\texttt{rlx}}]}_{④}; \underbrace{(\texttt{rf}; \texttt{rmw})^{*}}_{⑤}; \underbrace{\texttt{rf}}_{⑥}; \underbrace{[\texttt{R}^{\sqsupseteq\texttt{rlx}}]}_{④}; \underbrace{(\texttt{sb}; [\texttt{F}])^{?}}_{②}; \underbrace{[\texttt{E}^{\sqsupseteq\texttt{acq}}]}_{①}$$

La figure ci-dessus correspond à la définition de sw dans laquelle on a déplié la définition de rs et numéroté les parties importantes de la définition. Dans sa forme la plus simple, la relation sw relie deux évènements *release-acquire* (①) par une relation rf (⑥). Mais il y a des complications : le premier évènement de la paire peut être une écriture *release* (④), une écriture *release* qui précède une écriture atomique au même emplacement (③), ou une barrière *release* qui précède une écriture atomique (②). Le second évènement de la paire peut être une lecture *acquire* (④) ou une barrière *acquire* (②).

Quelle que soit la situation, il existe une paire composée d'une écriture et d'une lecture atomique qui entrent en jeu dans sw (④). La lecture atomique peut lire la valeur écrite par l'écriture atomique (⑥). Elle peut également lire la valeur écrite par le dernier évènement d'une chaîne de *read-modify-write* si le premier évènement de cette chaîne a lu la valeur écrite par l'écriture atomique (⑤).

2.4.3 La relation psc

La relation psc est utilisée pour exprimer la condition des atomiques SC. Intuitivement, on souhaite qu'il y ait un ordre total entre tous les évènements sc, qui correspond à l'ordre dans lequel ils sont exécutés et donc perçus par les autres *threads*.

Cet ordre total ne doit pas contredire ni l'ordre du programme (sb), ni l'ordre dans lequel des variables doivent être visibles les unes par rapport aux autres (eco). L'ordre imposé par hb doit également être pris en compte. Enfin, la sémantique des barrières sc doit être exprimée.

Pour qu'un ordre total qui respecte ces conditions existe, il est suffisant qu'un ordre partiel qui respecte ces conditions et qui soit acyclique existe [3]. Cela est dû au fait qu'un ordre partiel acyclique sur un ensemble peut toujours être étendu en un ordre total sur ce même ensemble.

La relation psc décrit cet ordre partiel sur les évènements sc.

$$\texttt{scb} \triangleq \texttt{sb} \cup (\texttt{sb}|_{\neq\texttt{loc}}; \texttt{hb}; \texttt{sb}|_{\neq\texttt{loc}}) \cup \texttt{hb}|_{\texttt{loc}} \cup \texttt{mo} \cup \texttt{rb}$$

$$\texttt{psc}_{base} \triangleq ([\texttt{E}^{\texttt{sc}}] \cup ([\texttt{F}^{\texttt{sc}}]; \texttt{hb}^{?})); \texttt{scb}; ([\texttt{E}^{\texttt{sc}}] \cup (\texttt{hb}^{?}; [\texttt{F}^{\texttt{sc}}]))$$

La relation $\texttt{psc}_{base}$ est l'équivalent d'un ordre partiel sur les évènements sc qui est cohérente avec (sb ∪ eco) et hb. rf a été exclue de eco, car lorsque deux éléments sc sont reliés par rf, ils le sont également par hb.

La raison pour laquelle on remplace hb par $(\texttt{sb} \cup \texttt{sb}|_{\neq\texttt{loc}}; \texttt{hb}; \texttt{sb}|_{\neq\texttt{loc}} \cup \texttt{hb}|_{\texttt{loc}})$, est qu'imposer que cet ordre partiel respecte strictement l'ordre imposé par hb rend la compilation correcte vers l'architecture Power impossible [7, section 2.1].

Cette relation $\mathtt{psc}_{base}$ impose également que les effets d'un évènement qui doit s'exécuter après une barrière `sc` ne doivent pas pouvoir être observés par les évènements dont les effets doivent s'exécuter avant la barrière.

$$\mathtt{psc}_{\mathtt{F}} \triangleq [\mathtt{F}^{\mathtt{sc}}]; (\mathtt{hb} \cup \mathtt{hb}; \mathtt{eco}; \mathtt{hb}); [\mathtt{F}^{\mathtt{sc}}]$$

Cependant, la relation $\mathtt{psc}_{base}$ n'exprime pas l'intégralité de la sémantique des barrières `sc`. On souhaiterait qu'un programme dans lequel tous les évènements sont séparés par des barrières `sc` soit conforme à SC. Or, l'acyclicité de $\mathtt{psc}_{base}$ ne suffit pas à garantir que cette propriété est respectée [7, section 2.2]. La relation $\mathtt{psc}_{\mathtt{F}}$ représente les contraintes imposées par ces barrières.

$$\mathtt{psc} \triangleq \mathtt{psc}_{base} \cup \mathtt{psc}_{\mathtt{F}}$$

La relation `psc` est simplement l'union de ces deux relations et représente l'ensemble des contraintes qu'un ordre total sur les évènements `sc` doit respecter.

2.4.4 Les quatre conditions de validité

Une exécution est conforme au modèle RC11 à quatre conditions :

1. $\mathtt{hb}; \mathtt{eco}^?$ est irréflexif
2. $\mathtt{rmw} \cap (\mathtt{rb}; \mathtt{mo}) = \varnothing$
3. `psc` est acyclique
4. $\mathtt{sb} \cup \mathtt{rf}$ est acyclique

La condition 1 garantit que plusieurs comportements seront évités. Tout d'abord, elle garantit que la relation $(\mathtt{hb}; \mathtt{eco})$ est irréflexive. Cette condition exprime la sémantique des accès *release-acquire*, en excluant les exécutions où les effets de certains évènements sont observés (exprimé par `eco`) dans un sens interdit par une paire d'atomiques *release-acquire* (exprimé par `hb`). Cependant, `hb` n'exprime pas uniquement l'ordre imposé par ces accès atomiques, puisqu'elle contient également `sb`. Le fait que la relation $(\mathtt{sb}; \mathtt{eco})$ soit irréflexive signifie qu'un évènement ne peut pas observer les effets des évènements de son propre *thread* dans un ordre différent de celui du programme. Enfin, la condition garantit que la relation `hb` est irréflexive.

La condition 2 exprime la sémantique des paires *read-modify-write*. Elle exclut les exécutions dans lesquelles la valeur contenue dans un emplacement mémoire est modifiée entre la lecture d'une paire *read-modify-write* à cet emplacement et son écriture.

La condition 3 impose que l'ordre partiel `psc` sur les évènements `sc` est acyclique, et donc qu'il peut être étendu en un ordre total.

La condition 4 résout radicalement un problème majeur du modèle mémoire du standard C11, à savoir les comportements *out-of-thin-air*.

$$\begin{array}{l|l} a :=_{\mathtt{rlx}} x & b :=_{\mathtt{rlx}} y \\ y :=_{\mathtt{rlx}} a & x :=_{\mathtt{rlx}} b \end{array}$$

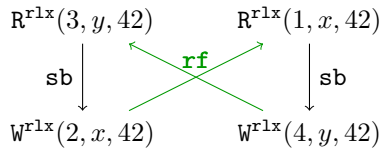


FIGURE 4 – Exécution (à droite) *out-of-thin-air* d'un programme (à gauche)

La figure 4 présente un exemple de programme qui peut donner lieu à une exécution contenant un cycle dans $(\mathtt{sb} \cup \mathtt{rf})$. Dans ce programme, les valeurs écrites aux emplacements x et y dépendent de valeurs lues des mêmes emplacements. Dans l'exécution associée, ces lectures peuvent lire des valeurs qui viennent "de nulle part", puisqu'il y a une dépendance cyclique entre les 4 accès mémoire de l'exécution. Bien sûr, aucune implémentation du standard C11

ne présente ce genre de comportement. Cependant, formaliser précisément ce qui constitue un comportement *out-of-thin-air* sans interdire des optimisations connues des concepteurs de processeurs et de compilateurs est notoirement difficile [5,6]. Des solutions existent, mais elles sont complexes [9,10]. Imposer l'acyclicité de $(\texttt{sb} \cup \texttt{rf})$ est une solution simple à ce problème, mais qui implique de moins bonnes performances lorsque l'on compile vers certaines architectures (par exemple Power [7, section 5]). On peut retrouver la définition du modèle RC11 dans le fichier `rc11.v` du développement Coq.

3 Vérification du théorème DRF-SC

Maintenant que notre modèle mémoire est rigoureusement défini, nous présentons la preuve qu'il respecte la propriété DRF-SC. Nous souhaitons donc montrer que si toutes les exécutions conformes au modèle SC ne contiennent pas de *races*, alors toutes les exécutions de ce programme qui sont conformes au modèle RC11 sont également conformes au modèle SC.

Trois définitions nous manquent pour que cet énoncé soit précis. D'abord une définition axiomatique du modèle mémoire SC, puis une définition précise de ce qu'est une *race* et enfin une définition de l'ensemble des exécutions associées à un programme. Nous présenterons donc ces trois définitions, suivies de la preuve en elle-même.

3.1 Le modèle SC

Dans le modèle SC, on souhaite que tous les accès à la mémoire, qu'ils soient atomiques ou non, soient observés dans l'ordre dans lequel ils apparaissent dans le programme. On souhaite donc que la relation eco ne soit jamais en contradiction avec sb. On souhaite également que l'atomicité des opérations *read-modify-write* soit préservée. On caractérise donc le modèle SC par ces deux conditions :

1. $\texttt{rmw} \cap (\texttt{rb}; \texttt{mo}) = \varnothing$
2. $(\texttt{sb} \cup \texttt{eco})$ est acyclique.

On peut retrouver la définition de la conformité à SC dans le développement Coq. On peut également noter deux propriétés importantes relatives aux modèles SC et RC11 :

Lemme 1 (). *Toutes les exécutions conformes au modèle SC sont également conformes à RC11*

Cette propriété nous assure que RC11 n'exclut aucun comportement SC, et ne fait qu'en admettre des nouveaux. Cela garantit que le modèle est "réaliste", et qu'il n'exclut pas de comportements auxquels un programmeur pourrait légitimement s'attendre.

Lemme 2 (). *Si tous les accès d'un programme sont des atomiques de type* sc*, toutes ses exécutions conformes à RC11 sont également conformes à SC*

3.2 Les *races*

```
Definition race (ex: Execution) : rlt Event :=
  fun x => fun y =>
    (is_write x \/ is_write y) /\
    x <> y /\ get_loc x = get_loc y /\
    ~((hb ex) x y) /\ ~((hb ex) y x).
```

FIGURE 5 – La définition en Coq d'une *race*

Jusqu'à présent, nous avons décrit deux évènements formant une race comme deux accès au même emplacement mémoire, pouvant être exécutés simultanément, et dont au moins un est une écriture.

Nous appelons *évènements en conflit* les paires distinctes d'accès à un emplacement mémoire dont au moins un est une écriture. Comment peut-on exprimer que deux de ces éléments peuvent s'exécuter simultanément ?

Une autre façon de dire que deux instructions peuvent s'exécuter simultanément est de dire que le modèle mémoire n'impose pas que les effets d'une des instructions doivent être visibles pour l'autre accès sur la mémoire, et donc n'impose pas que leurs effets affectent la mémoire dans un ordre précis. Dans le cas de RC11, cet ordre peut être imposé grâce aux atomiques, dans deux cas :

1. lorsque une première instruction est reliée à une seconde par `hb`. Les effets de la première instruction sur la mémoire doivent alors être visibles pour la seconde.
2. lorsque deux évènements sont `sc`, l'acyclicité de `psc` impose que leurs effets soient vus dans le même ordre par tous les *threads* du programme.

Les évènements x et y forment donc une race, ce qu'on note $\langle x, y\rangle \in$ `race`, lorsque x et y sont en conflit, qu'ils ne sont pas reliés par `hb` (ni dans un sens, ni dans l'autre), et que l'un d'entre eux n'est pas un atomique `sc`.

On peut remarquer dans la figure 5 que la définition de *race* de notre formalisation Coq n'est pas exactement la même, puisqu'elle n'exclut pas les paires d'évènements `sc`. Dans l'article original, les auteurs du modèle RC11 définissent une *race* de cette façon, et formulent la garantie DRF-SC de façon légèrement différente de ce que nous avons présenté jusqu'à présent. Le théorème tels qu'ils le formulent [7, théorème 4, section 8], stipule que si dans toutes les exécutions conformes à SC, les évènements qui forment une *race* sont tous les deux `sc`, alors toutes les exécutions du programme conformes à RC11 sont conformes à SC.

Pour que la formalisation en Coq de la preuve de la garantie DRF-SC soit la plus directe possible, nous avons conservé la formulation initiale dans la formalisation Coq, comme on peut le voir dans les hypothèses dans le lemme final `drf_sc_final`.

Ajouter le fait que deux évènements `sc` ne peuvent pas former une *race* directement dans la définition d'une *race* ou le préciser dans les hypothèses de notre théorème final sont deux façons équivalentes de prouver la même chose. Nous pensons que la première façon expose plus clairement la signification de ce qu'est une *race*, et c'est pourquoi nous avons introduit la notion de cette façon dans cet article.

3.3 L'ensemble des exécutions

Comme nous l'avons mentionné dans la partie 2.3.2, on ne dispose pas d'une façon unique de transformer un programme en un ensemble d'exécutions, puisque cette transformation dépend du langage source que l'on considère. Cela pose un problème lorsque l'on veut formuler notre théorème DRF-SC, puisqu'il nous faut considérer l'ensemble des exécutions conformes à SC et RC11.

Pour contourner cette difficulté, nous définissons de façon axiomatique ce que signifie pour deux exécutions le fait d'être issues d'un même programme. Nous expliquerons pourquoi ces axiomes sont raisonnables et correspondent à une idée intuitive de ce que sont deux exécutions issues d'un même programme.

3.3.1 Préfixes d'exécution

Une exécution G' est un préfixe d'une autre exécution G lorsque l'ensemble des évènements de G' est un sous-ensemble $\mathtt{E}$ des évènements de G tel que :

— Les évènements de G' contiennent l'ensemble $\mathtt{E}_0$ des évènements d'initialisation de G.

— Pour tout évènement $b \in G'.\mathtt{E}$, si un évènement a est tel que $\langle a, b\rangle \in G.(\mathtt{sb} \cup \mathtt{rf})$, alors $a \in G'.\mathtt{E}$. $G'.\mathtt{E}$ est donc clos par rapport à $G'.(\mathtt{sb} \cup \mathtt{rf})^{-1}$.

Intuitivement, le préfixe d'une exécution correspond à une exécution incomplète. Pour chaque *thread*, on choisit un des évènements qu'il contient et on ne garde que les évènements qui le précèdent dans le programme. Le sous-ensemble choisi n'est pas clos seulement par rapport à $\mathtt{sb}^{-1}$, mais aussi par rapport à $\mathtt{rf}^{-1}$ pour éviter que les choix de découpage pour chaque *thread* soient incohérents entre eux, et que certaines lectures lisent en mémoire des valeurs qui n'y ont jamais été écrites.

Dans le cadre de la preuve de la propriété DRF-SC, nous considérons une formulation différente mais équivalente de la notion de préfixe [1]. On considère qu'une exécution est *cohérente* si les identifiants uniques associés à chacun des évènements de l'exécution sont tels que pour toute paire d'évènements $\langle x, y\rangle \in (\mathtt{sb} \cup \mathtt{rf})$, on a $\mathtt{id}(x) \leq \mathtt{id}(y)$.

Lorsque G est une exécution cohérente, G' est une *délimitation* de G par un entier naturel k si l'ensemble des évènements de G' est un sous-ensemble $\mathtt{E}$ des évènements de G tel que :

— Les évènements de G' contiennent l'ensemble $\mathtt{E}_0$ des évènements d'initialisation de G.

— Pour tout évènement $x \in G'.\mathtt{E}$, $\mathtt{id}(x) \leq k$.

Pour toute exécution cohérente et quel que soit l'entier k, une délimitation de l'exécution G par k, notée G_k est un préfixe de l'exécution. Il est à noter que pour toute exécution conforme à RC11 (ou SC par extension), il existe des identifiants qui font de l'exécution une exécution cohérente, et ce grâce à la condition qui interdit les comportements *out-of-thin-air* dans RC11. L'intérêt de cette définition alternative est qu'elle permet de comparer les préfixes d'exécutions. Pour j, k tels que $j \leq k$, G_j est un plus petit préfixe de G que G_k.

Pour comprendre pourquoi cet axiome est raisonnable, il est utile d'anticiper un tout petit peu sur la preuve en elle-même. Notre raisonnement consistera à montrer la contraposée de notre énoncé de DRF-SC, c'est à dire que si un énoncé est conforme à RC11 mais pas à SC, il existe une autre exécution du même programme, conforme à SC et qui contient une *race*. Dans ce cadre, il n'est pas nécessaire de prouver qu'il existe une autre exécution "complète" du programme. Puisque l'on cherche à montrer que cette autre exécution est conforme à SC et contient une *race*, il suffit de montrer qu'un préfixe d'exécution respectant ces conditions existe. En effet, si une exécution incomplète d'un programme conforme à SC contient une *race*, on peut imaginer qu'il suffit de continuer à exécuter les instructions du programme séquentiellement, pour obtenir une autre exécution complète, qui sera toujours conforme à SC, et dans laquelle la *race* sera toujours présente.

3.3.2 Changement d'une lecture sb-finale

Soit G une exécution qui contient une lecture $a \notin \mathsf{dom}(G.\mathtt{sb})$ (a est le dernier évènement de son propre *thread*). L'évènement a lit une valeur v de l'emplacement mémoire x. Cette valeur a été écrite à cet emplacement par une écriture b. Supposons maintenant qu'il existe dans G une écriture c qui écrit la valeur v' à l'emplacement x. Nous pouvons construire l'exécution G' dans laquelle la valeur lue par a est changée pour v', et où la paire $\langle b, a\rangle$ de $G.\mathtt{rf}$ est remplacée par une paire $\langle c, a\rangle$.

Nous considérons alors que G et G' sont deux exécutions d'un même programme. Considérer que ce changement résulte en une autre exécution d'un même programme est raisonnable, car la valeur lue par une lecture ne peut pas avoir une influence sur ce qui s'est passé avant la lecture dans le *thread*. Il est possible de modéliser précisément les évènements suivant la lecture qui peuvent être modifiés, grâce une relation de *dépendance*. Mais dans notre cas ce n'est pas nécessaire, car aucun évènement ne suit la lecture.

1. Pour la preuve de cette équivalence, voir le lemme Coq `bounded_exec_is_prefix`.

3.3.3 Égalité modulo mo

Si deux exécutions sont identiques à l'exception de leur relation mo, on considère que ce sont deux exécutions d'un même programme.

Cet axiome est également conforme à l'intuition que l'on peut avoir de deux exécutions d'un même programme. En effet, lorsqu'on ne change ni les instructions d'un programme, ni les valeurs qui sont lues ou écrites par ces instructions, le comportement du programme reste le même. Lorsque l'on change la relation mo, et donc l'ordre dans lequel sont perçues les écritures, seule la conformité de l'exécution avec un modèle mémoire, et donc la possibilité de l'existence d'une telle exécution, peut changer.

3.3.4 Formalisation en Coq

```
Inductive sameP (res ex: Execution) : Prop :=
  | sameP_pre : prefix res ex -> sameP res ex
  | sameP_res_chval : forall j k bound c v l,
      minimal_conflicting_pair ex bound j k ->
      numbering k > numbering j ->
      res_chval_k ex res bound j k c l v -> sameP res ex
  | sameP_mo : eq_mod_mo res ex -> sameP res ex
  | sameP_trans : forall c, sameP res c -> sameP c ex -> sameP res ex.
```

FIGURE 6 – La définition en Coq de deux exécutions d'un même programme

Les conditions dans lesquelles deux exécutions sont issues d'un même programme sont rassemblées dans un prédicat inductif `sameP` présenté dans la figure 6. Les conditions nécessaires pour que le changement d'une lecture `sb`-finale soit possible sont plus restreintes que ce que nous avons présenté dans la section 3.3.2.

Ces conditions ne prennent en compte que le changement de lectures `sb`-finales dans un cas particulier, qui est le cas précis qui se présentera dans la preuve. L'important est de noter que ce cas particulier implique que la lecture dont la valeur lue est modifiée est bien `sb`-finale dans l'exécution.

3.4 La preuve

```
Lemma drf_sc (e: Execution):
  complete_exec e /\ numbering_coherent e /\ numbering_injective e ->
  (forall e', sameP e' e /\ sc_consistent e' ->
              (forall a b, (race e') a b ->
                           (get_mode a = Sc /\ get_mode b = Sc))) ->
  (forall e', sameP e' e /\ rc11_consistent e' ->
              sc_consistent e').
```

FIGURE 7 – Énoncé final du théorème DRF-SC

Le principe général de la preuve est le suivant : nous allons prouver la contraposée de notre énoncé initial, tel qu'il est présenté dans la figure 7. Nous supposons une exécution G conforme à RC11, mais pas à SC, et nous cherchons à prouver qu'il existe une autre exécution du même programme qui est elle conforme à SC, mais qui contient une race (ou dont les éléments formant une *race* sont tous les deux `sc`. Voir 3.2).

Pour présenter cette preuve de façon claire et permettre au lecteur d'accéder aux lemmes importants du développement Coq, la figure 8 présente le raisonnement sous forme d'arbre.

Les nœuds de cet arbre représentent les différentes étapes de la preuve, et seront annotés par des liens vers les lemmes importants concernant l'exécution que l'on considère à cette étape de la preuve. Les flèches pleines représente un passage d'une exécution à une autre en restant dans les exécutions d'un même programme. Les flèches en pointillés représentent des embranchements dans le raisonnement lorsque l'on considère différents cas.

Aux feuilles de cet arbre, on trouvera les lemmes finaux, c'est à dire ceux qui prouvent que les exécutions que nous avons obtenues dans chaque cas sont conformes à SC et contiennent une *race*.

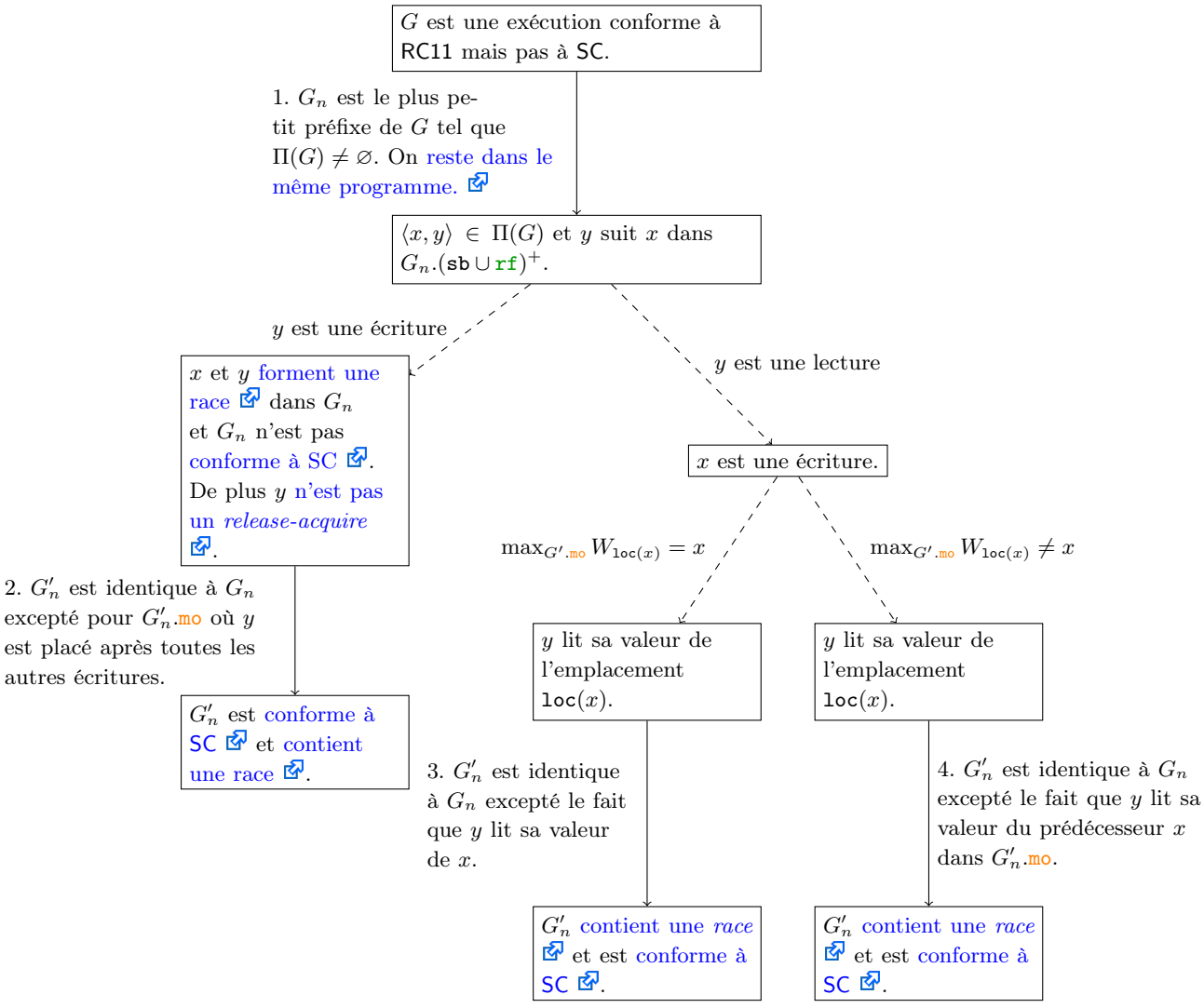


FIGURE 8 – Schéma résumant les différentes étapes du raisonnement de la preuve DRF-SC pour RC11

Avant d'exposer le raisonnement de la preuve un peu plus en détails, nous introduisons $\Pi(G)$ l'ensemble des paires d'évènements $\langle a, b\rangle$ telles que :

— a et b sont deux évènements de l'exécution ($a \in G.\mathtt{E}$ et $b \in G.\mathtt{E}$),

— a et b sont en conflit,

— a et b ne sont pas tous les deux des atomiques `sc`, et

— ni $\langle a, b\rangle$, ni $\langle b, a\rangle$ n'appartiennent à $(G.\texttt{sb} \cup G.\texttt{rf}|_{\texttt{sc}})^+$.

Pour tout $i \in \mathbb{N}$ tel que G_i est une délimitation de G, si $\Pi(G_i) = \varnothing$, alors G_i est conforme à SC.

On suppose donc une exécution G conforme à RC11 mais pas à SC. Puisque G n'est pas conforme à SC, on a $\Pi(G) = \varnothing$. Il existe alors G_n, la plus petite délimitation de G (pour tout $i < n$, on a $\Pi(G_i) = \varnothing$), dont les évènements incluent une paire d'évènements $\langle x, y\rangle \in \Pi(G_n)$ telle que $\texttt{id}(y) = n$. Cela correspond à l'étape 1 présentée dans la figure 8.

Puisque tous les préfixes de G_n sont conformes à SC, et que G_n respecte la condition d'atomicité des *read-modify-write*[2], G_n n'est pas conforme à SC à cause d'un cycle dans $G_n.(\texttt{sb} \cup \texttt{rf} \cup \texttt{mo} \cup \texttt{rb})$ dont y fait partie. On considère alors deux cas :

1. Si y est une écriture, on a x et y qui forment une *race* dans G_n. De plus, l'écriture y n'étant suivie par aucun évènement dans $G_n.\texttt{sb}$, l'étape qui la suit dans le cycle ne peut faire partie que de la relation $G_n.\texttt{mo}$. On modifie donc la relation $G_n.\texttt{mo}$ pour obtenir l'exécution G'_n dans laquelle y n'est suivie par aucune écriture dans $G'_n.\texttt{mo}$. On obtient alors une exécution conforme à SC et contenant une *race*. Cela correspond à l'étape 2 présentée dans la figure 8. Il est également important de noter que y ne pouvait pas être une écriture atomique, et que donc l'atomicité des *read-modify-write* n'est pas affectée par notre modification de la relation mo.
2. Si y est une lecture, on a forcément x qui est une écriture. Puisque y est une lecture qui n'est suivie par aucun évènement dans $(G_n.\texttt{sb} \cup G_n.\texttt{rf})$, l'étape qui la suit dans le cycle ne peut faire partie que de la relation $G_n.\texttt{rb}$. Cela signifie que y lit sa valeur depuis une lecture qui est suivie par un autre évènement dans $G_n.\texttt{mo}$.
   On considère alors deux cas :
   (a) $\max_{G'.\texttt{mo}} W_{\texttt{loc}(x)} = x$. Dans ce cas, on prend le prédécesseur immédiat d de x dans $G_n.\texttt{mo}$ est on remplacer la valeur lue par y par celle de d pour obtenir l'exécution G'_n. Cela correspond à l'étape 3 présentée dans la figure 8. Or, par définition, x n'est suivi par aucun évènement dans $G'_n.\texttt{mo}$ et de plus il n'est suivi par aucun autre évènement dans $(G'_n.\texttt{sb} \cup G'_n.\texttt{rf})$. Le cycle est donc brisé est on obtient une exécution conforme à SC et contenant une *race*. Nous avons pris le prédécesseur de x car créer un lien dans $G'_n.\texttt{rf}$ entre x et y, aurait brisé la *race* entre ces deux évènements.
   (b) $\max_{G'.\texttt{mo}} W_{\texttt{loc}(x)} \neq x$. Dans ce cas, on remplace la valeur lue par y par celle de $\max_{G'.\texttt{mo}} W_{\texttt{loc}(x)}$. Cela correspond à l'étape 4 présentée dans la figure 8. Puisque cette écriture n'est suivie par aucun évènement dans $G'_n.\texttt{mo}$, le cycle est brisé et on obtient une exécution conforme à SC et contenant une *race*.

4 Limitation et généralisation

En dehors d'apporter un degré de confiance plus élevé, la mécanisation de cette preuve peut avoir selon nous deux intérêts majeurs.

Premièrement, nous pensons que c'est un bon point de départ pour repousser une limitation du théorème DRF-SC. Nous présentons cette limite, qui concerne un idiome de programmation courant, et nous posons quelques pistes pour étendre l'application de cette propriété à cet idiome.

Deuxièmement, nous voudrions abstraire cette preuve pour qu'elle ne dépende plus de détails du modèle RC11, mais seulement de propriétés, qui pourraient éventuellement être partagées par d'autres modèles mémoire.

2. Le préfixe conserve la propriété d'atomicité des exécutions et G respecte l'atomicité car conforme à RC11.

4.1 DRF-SC et le *message-passing*

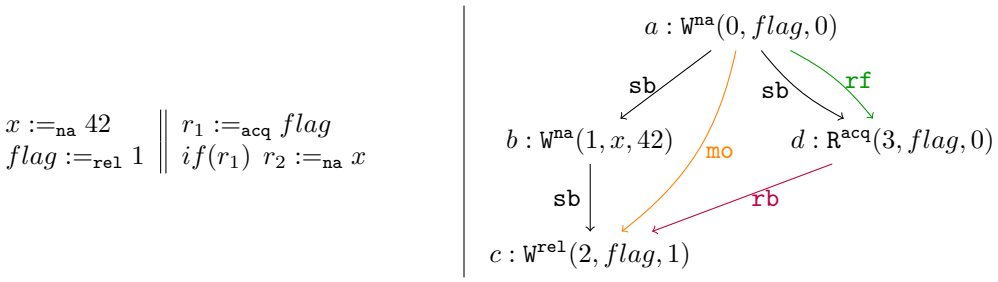


FIGURE 9 – Un programme de *message-passing* et son exécution contenant une *race*

Nous entendons par limitation l'existence de programmes qui n'ont que des comportements SC, mais sur lesquels la propriété DRF-SC ne s'applique pas. La figure 9 est un exemple d'un tel programme. Dans ce programme, on utilise un flag auquel on accède avec des lectures et écritures *release-acquire*, pour protéger l'accès à des données (ici le contenu de x). Toutes les exécutions de ce programme qui sont conformes à RC11 le sont également avec SC.

Mais l'exécution présentée, qui est conforme à SC, contient une *race* entre l'écriture du flag (c) et sa lecture dans le second *thread* (d). Cette *race* ne peut pas donner lieu à un comportement non conforme à SC car le *modification-order* mo est imposé dans cette exécution. La *race* pourrait donner lieu à un comportement non SC si l'écriture (c) pouvait être perçue comme affectant la mémoire avant l'écriture initiale (a). Or cela est impossible, car cela créerait un cycle dans ($\mathtt{hb}; \mathtt{eco}^?$), et donc l'exécution ne serait pas conforme à RC11. Il existe donc un ordre de perception des écritures par les autres *threads* imposé par RC11 et qui n'est pas pris en compte dans la notion de *race*.

Peut-on restreindre la définition de *race* pour que les situations de ce type ne soient pas considérées comme une *race* tout en conservant la propriété DRF-SC ? Nous pensons que la formalisation en Coq que nous venons de présenter sera un atout pour répondre à cette question, car elle permettra de faire évoluer la notion de *race* et la preuve de DRF-SC associée au modèle en parallèle, en étant sûr que cette preuve reste valide.

4.2 Généralisation

Nous pensons que cette preuve mécanisée pourrait être adaptée à d'autres modèles. Pour cela, nous pensons qu'il serait efficace d'encapsuler chaque modèle mémoire axiomatique dans une *typeclass*. Ce procédé permettrait de s'abstraire des différents types d'évènements que le modèle mémoire doit prendre en compte, ainsi que de l'implémentation de relations dérivées des relations de base sb, rf, mo et rmw.

En étudiant la preuve de la propriété DRF-SC pour le modèle RC11, nous pouvons constater plusieurs difficultés :

— la preuve dépend beaucoup de la présence d'atomiques sc dans le modèle RC11. L'ensemble $\Pi(G)$ notamment, est construit à partir de la relation $\mathtt{rf}|_{\mathtt{sc}}$. Pour pouvoir s'en abstraire, nous devons définir l'ensemble des évènements sc, comme un ensemble d'évènements partageant une certaine propriété.

— la preuve dépend également beaucoup des quatre conditions de conformité du modèle RC11. On ne peut donc utiliser la preuve que pour des modèles strictement plus forts que RC11, dont les conditions de conformité impliquent celles de RC11. Or, il existe peut-être des modèles plus relâchés, qui satisfont également la propriété DRF-SC. L'objectif est donc de trouver le modèle le plus relâché possible qui respecte la propriété DRF-SC.

Nous pensons que la première étape de cet effort d'abstraction, avant d'essayer d'adapter la preuve à un autre modèle, est de reformuler le modèle RC11, pour que la preuve se repose moins sur ces détails. Nous pensons que l'assistant de preuves sera un avantage pour garantir que cette reformulation du modèle est équivalente à celle que nous avons présentée.

5 Conclusion

Nous avons présenté dans cet article la formalisation d'un modèle, défini dans [7], qui décrit le comportement des programmes C/C++11 concurrents. Nous avons ensuite présenté la formalisation dans l'assistant de preuve Coq d'une preuve existante et issue du même article, stipulant que ce modèle respecte la propriété DRF-SC. Cette propriété permet aux programmeurs d'ignorer complètement les détails du modèle mémoire, qui peuvent être complexes et contre-intuitifs. Nous avons formalisé cette preuve dans l'assistant de preuves Coq.

La formalisation dans des assistants de preuves de modèles mémoire est un domaine actif, et des modèles mémoires proches de celui de C11 ont récemment été formalisés en Coq [4, 12]. Cependant, ces deux modèles vérifient une variante plus «faible» de la propriété DRF-SC. Cette variante stipule que si toutes les exécutions d'un programme conformes à un modèle qui respecte DRF-SC "faible" ne contiennent pas de *races*, alors toutes les exécutions de ce programme sont conformes à SC. Cette propriété est moins intéressante, car elle demande au programmeur de raisonner sur toutes exécutions conformes au modèle mémoire, et pas seulement sur celles conformes à SC pour y détecter les éventuelles *races*.

Le développement Coq final comptabilise au final environ 8000 lignes de code. Nous signalons l'utilisation intensive de la librairie `relation-algebra` [11] au sein de ce développement. Cette librairie fournit une tactique `kat`, qui est une procédure de décision sur les énoncés d'inclusion (et d'équivalence par extension) entre relations (sous certaines conditions). Cette librairie s'est avérée être un outil particulièrement précieux pour travailler sur les modèles mémoires axiomatiques.

Si la preuve du fait que RC11 respecte cette propriété DRF-SC apparaissait déjà dans l'article présentant ce modèle [7], sa mécanisation n'a pas été un simple travail de traduction. Certaines parties non triviales du raisonnement n'étaient pas explicitées dans cette preuve. La preuve du lemme `change_val_eco_ac2` par exemple, nécessite de prendre en compte de très nombreux cas, mais n'est pas donnée dans la preuve non-mécanisée. Pour ces raisonnements non détaillés, la sûreté apportée par un assistant de preuves est d'autant plus importante.

En outre, nous pensons que cette mécanisation de la preuve sera un outil précieux dans la réalisation de deux objectifs : étendre cette propriété pour qu'elle s'applique à plus de programmes et abstraire cette preuve de RC11 pour qu'elle s'applique à des modèles différents. Cette preuve mécanisée pourrait également être utilisée dans le cadre d'une autre preuve formelle. Par exemple, il serait possible de vérifier la spécification d'un programme C11 en se basant sur une sémantique SC si on prouve que ses exécutions ne contiennent aucune *races* et d'obtenir par extension une preuve de cette spécification dans le modèle mémoire RC11.

Références

[1] Jade Alglave. A shared memory poetics. *These de doctorat, L'université Paris Denis Diderot*, 2010.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats : Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2) :1–74, 2014.

[3] Mark Batty, Alastair F Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 634–648, 2016.

[4] John Bender and Jens Palsberg. A formalization of Java's concurrent access modes. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA) :1–28, 2019.

[5] Hans-J. Boehm. P1217r0 : Out-of-thin-air, revisited, again, 2018.

[6] Hans-J Boehm and Brian Demsky. Outlawing ghosts : Avoiding out-of-thin-air results. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, pages 1–6, 2014.

[7] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++ 11. *ACM SIGPLAN Notices*, 52(6) :618–632, 2017.

[8] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9) :690–691, 1979.

[9] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0 : global optimizations in relaxed memory concurrency. In *PLDI*, pages 362–376, 2020.

[10] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. *ACM SIGPLAN Notices*, 51(1) :622–633, 2016.

[11] Damien Pous. Kleene algebra with tests and Coq tools for while programs. In *International Conference on Interactive Theorem Proving*, pages 180–196. Springer, 2013.

[12] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. Repairing and mechanising the JavaScript relaxed memory model. *arXiv preprint arXiv :2005.10554*, 2020.

JSkel: Towards a Formalization of JavaScript's Semantics

Adam Khayam, Louis Noizet, and Alan Schmitt

Inria

Abstract

We present JSkel, a formalization of the semantics of JavaScript in Skel, the concrete language used to write skeletal semantics. We describe the improvements to Skel we designed and implemented to significantly simplify the formalization. We show the formalization is both close to the specification and executable.

1 Introduction

Mechanizing the semantics of complex languages is useful, not only to make it precise, but also to obtain a framework in which one may prove properties of programs or of the language itself. The success of such a mechanization may be evaluated in two ways: is the actual semantics of the language really captured, and can the formalization be used in practice?

There are two main ways to make sure the semantics is captured: first, the mechanization can be textually close to the specification of the language, if it exists. Second, if the mechanization is executable, it should successfully run tests provided by the language. Thus, a suitable tool to mechanize semantics should make it easy to stay close to language definitions, be they algorithmic (e.g., for JavaScript) or based on inference rules. Furthermore, the tool should also provide an easy way to derive an executable version of the semantics to run code, including tests.

The goal of this paper is to show that the Skeletal Semantics framework [2] is suitable to mechanize semantics. To this end, we describe the ongoing formalization of JavaScript as a skeletal semantics. In this paper, we only address the question of capturing the semantics. We leave the evaluation of the usability of the formalization for future work.

JSkel is the first formalization of a real-world programming language in Skel and it has been a core support in the evolution of Skel, from the introduction of the notion of polymorphism to the addition of monads and first-class functions. The evolution of Skel is motivated by the need of having a formal semantics and a language expressive enough to be able to capture the behavior of JavaScript, ensuring a visual and a behavioral match between the formalization and the specification.

We present existing approaches to mechanize languages in Section 2. We then introduce Skeletal Semantics as well as our concrete syntax in Section 3. Our main contribution is the description of the mechanization of JavaScript in Section 4, where we also present some crucial features of skeletal semantics that significantly improve the readability of the semantics. We conclude and explore future work in Section 5.

2 Context

We first motivate why we chose JavaScript to evaluate the mechanization of a language using skeletal semantics. The three defining features of JavaScript in this regard are the following. First, it is complex, hence a good candidate to see if our solution scales up. Second, it has a precise specification, called ECMAScript (ES), and a large suite of test cases, hence we do not

have to guess what its semantics is. And third, it has been mechanized in several frameworks, which facilitates the comparison to other approaches.

In addition, we have a previous experience in mechanizing JavaScript in Coq, called JSCert [1]. Defining a semantics in a proof assistant is the most direct way of mechanizing it. It has a major drawback, however. If the design choices are not correct, one cannot manipulate the mechanization to modify it, and one must instead redo it using different choices. JSCert is a pretty-big-step [4] Coq definition of ECMAScript 5.1. It consists of an inductive definition, a recursive definition, and a correctness proof showing they match. The goal of the inductive definition is to prove properties of the language and of JavaScript programs, whereas the recursive definition can provide an OCaml interpreter using Coq's extraction mechanisms. The mechanization is fairly close to the specification for people who can read Coq code, and the test suite can be run using the extracted interpreter. JSCert has two flaws, unfortunately. It is difficult to maintain, as one has to update two formalizations and a proof. In addition, and most importantly, JSCert uses Coq's induction to represent the recursive evaluation of the language. This shallow embedding results in a definition of a semantics as an inductive of about 1000 rules, which is too large to prove properties of the language. The exploration of ways to address these issues was the motivation to create skeletal semantics.

To simplify the mechanization, [6] introduces a core language λ_{JS} that embodies JavaScript essentials, and a desugaring process that transforms a JS program to a λ_{JS} one. This work is based on ECMA 3 and 5. The authors empirically prove that the desugared version of JS is compliant with the implementations of JS itself using JavaScript's test suites. In spite of its efficiency and correctness, λ_{JS} is a feature-based redefinition of ECMA. This formalization does not follow the specification, making it hard to gather evidence it actually captures the language beyond running the tests. We attempted to formalize λ_{JS} in Coq[1], uncovering several issues with the desugaring process that were not witnessed by testing.

An alternative approach to the formalization of JavaScript is the use of an existing framework. $\mathbb{K}$JS [11] is a complete mechanization of ECMAScript 5.1 in the $\mathbb{K}$ framework. It provides an executable interpreter of JS directly from the semantics with no additional effort. This framework is not suitable to analyse the language itself as it only provides tools to reason about the execution of programs. In addition, there is no evidence that the mechanization can be easily maintained: although JavaScript has significantly evolved since ES 5.1 (the specification has more than doubled in size), the mechanization has not been updated. Our experience in updating JSExplain [3], a JavaScript interpreter written in OCaml, from ES5.1 to ES6 has shown us it is far from a trivial issue. The power of the $\mathbb{K}$ framework comes at the cost of additional complexity in the description of languages in it, which hinders maintainability and closeness to the specification.

3 Skeletal Semantics

Skeletal semantics is a *syntax* to define the semantics of programming languages in a concise yet powerful way, with a light formalism. This provides a way to easily manipulate the semantics, for instance to convert it into a Coq formalization or an OCaml interpreter. One of the strengths of skeletal semantics is the possibility to leave some constructions undefined, to let them be implementation dependent or for gradual specification.

The theoretical concept behind skeletal semantics was presented in [2]. The version we show here has been significantly improved, but it is still work in progress, in particular regarding some

[1] https://github.com/tilk/LambdaCert

$$\texttt{eval\_stmt}\ (\sigma, \texttt{While}\ (e,t)) :=$$
$$\quad \text{let}\, b = \texttt{eval\_expr}\ (\sigma, e)\, \text{in}$$
$$\quad \begin{pmatrix} \text{let}\, \texttt{True} = b\, \text{in let}\, \sigma' = \texttt{eval\_stmt}\ (\sigma, t)\, \text{in}\, \texttt{eval\_stmt}\ (\sigma', \texttt{While}\ (e,t)) \\ \text{let}\, \texttt{False} = b\, \text{in}\, \sigma \end{pmatrix}$$

Figure 1: skeleton for the `While` constructor

theoretical aspects such as the definition of abstract interpretations.

The "Skel" language, which has been defined to describe skeletal semantics, serves as support for the necro ecosystem [9], which provides a generator of OCaml interpreters [7], a generator of gallina formalization [8], and a (work in progress) generator of TEX inference rules [10].

The Skel formalization of JavaScript, which constitutes the main topic of this article, has had a lot of impact on the language itself, as it prompted us to make several improvements to Skel, such as adding polymorphism and first-class functions.

Figure 1 shows an example of a skeleton for the evaluation of a *while* block. We see all the main elements of skeletal semantics and we can observe that they are actually the main elements of any semantics.

- Recursion (`eval_stmt` and `eval_expr`) lets us define the semantics depending on the semantics of other terms (frequently subterms, but not always, as we can see in this example with the call $\texttt{eval\_stmt}\ (\sigma', \texttt{While}\ (e,t))$).
- There are auxiliary functions and auxiliary types such as booleans, and possibility to match a boolean variable against a given constructors (e.g `True`).
- The `let...in` construct is used to perform a sequence of operations.
- The branching (represented as a parenthesized system in Figure 1) is a choice between two possible rules. Often, the branches are mutually exclusive and a pattern matching at the start of the branch determines which branch is taken. Non-mutually exclusive branches lets us also represent non-deterministic semantics (such as λ-calculus with no evaluation strategy).

3.1 Formalism

We ignore polymorphism in the formal definition below to keep it readable. The use of polymorphism is pretty intuitive though, and its semantics is the usual one. We first give the syntax of skeletal semantics.

$$\begin{array}{rrcl} \textsc{Term} & t & ::= & x_i \mid C\ t \mid (t, \ldots, t) \mid \lambda x : \tau \cdot S \\ \textsc{Skeleton} & S & ::= & x_i\ t_1 \ldots t_n \mid \text{let}\, p = S\, \text{in}\, S \mid (S..S) \mid t \\ \textsc{Pattern} & p & ::= & x_i \mid \_ \mid C\ p \mid (p, \ldots, p) \\ \textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau, \ldots, \tau) \end{array}$$

A term is either a variable, a constructor applied to a term, a tuple of terms, or an abstraction whose body is a skeleton (where the type of the abstracted variable is explicitly given). Terms can be viewed as expressions in their normal form (that is with no reduction possible). A skeleton is either the application of a variable to several terms, a let binding, where the bound skeleton is matched against a pattern, a branching of several skeletons, or simply a term (sometimes, we write $\mathrm{ret}\, t$ to say explicitly that we consider the skeleton and not the term). A pattern is either a variable name, a wildcard, a constructor applied to a pattern, or a tuple of patterns. Finally, a type is either a base type, that is a type declared by the user (either specified or unspecified), an arrow type, or a tuple of types.

This syntax is very close to Abstract Normal Forms [5]. The main difference is that we add branching as a non-deterministic choice. We also provide constructors and tuple that are not in the basic ANF, and we allow let bindings in let bindings. Of course, the let bindings can always be extracted by using a fresh variable name, which the function `necrotrans extractletin` [9] actually does.

A skeletal semantics is a list of type *declarations*, either *unspecified* or *specified* (by giving its constructors), and a list of term declarations, also either unspecified and containing only a name and type, or specified with an additional term.

We give the following four examples, written in Skel :

```
type int

type nat = | Zero | Succ nat
```

```
term add: int → int → int

term two:nat = Succ (Succ Zero)
```

The possibility to declare non-specified types and terms is a really powerful tool. When defining a semantics, we sometimes do not want to go into details on how every type and every function works. Or sometimes, we want it to remain unspecified. The partial specifications allows for that.

As shown above, the Skel language is really light (close to λ-calculus), yet very powerful. A good proof of this power is the semantics of JS given in Section 4.

To furthermore improve readability, the Skel language also provide notation for binding operators. That is, assuming a term declaration for `bind`, one can write `let%bind p = x in v` for `bind x (λ v -> let p = v in s)`. It is particularly useful when writing a semantics that heavily uses monadic constructions, such as the one for JavaScript.

In itself, the language is only a syntactic construct, but there are multiple ways to derive meaning out of a skeletal semantics. This enables the writing of a single skeletal semantics to obtain multiple semantics. The usual way to interpret a skeletal semantics is the concrete semantics, which corresponds to a natural (big step) semantics. We describe it in Section 3.3, but first we focus on typing.

3.2 Typing

Skel is strongly typed. As mentioned above, types are threefold: user-defined types (specified or unspecified), product types for tuples, and arrow types for functions.

We give the typing rules for terms and skeletons in Figure 2. They are respectively of the form $\Gamma \vdash_t t : \tau$ and $\Gamma \vdash_S S : \tau$, where Γ is a typing environment (a partial functions that binds variable names to types). The initial typing environment Γ_0 is the function that binds every term name (both specified and non-specified) to the associated type given in the term declaration list.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_t x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash_t t : \tau \qquad \mathit{ctype}(\mathrm{C}) = (\tau, \tau')}{\Gamma \vdash_t \mathrm{C}\, t : \tau'}\ \textsc{Const}$$

$$\frac{\Gamma \vdash_t t_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash_t t_n : \tau_n}{\Gamma \vdash_t (t_1, \ldots, t_n) : (\tau_1, \ldots, \tau_n)}\ \textsc{Tuple} \qquad \frac{\Gamma + x \leftarrow \tau \vdash_S S : \tau'}{\Gamma \vdash_t (\lambda x : \tau \cdot S) : \tau \to \tau'}\ \textsc{Clos}$$

$$\frac{\Gamma \vdash_t t : \tau}{\Gamma \vdash_S \mathrm{ret}\, t : \tau}\ \textsc{Ret} \qquad \frac{\Gamma \vdash_S S_1 : \tau \qquad \ldots \qquad \Gamma \vdash_S S_n : \tau}{\Gamma \vdash_S (S_1 \ldots S_n) : \tau}\ \textsc{Branch}$$

$$\frac{\Gamma \vdash_S S : \tau \qquad \Gamma + p \leftarrow \tau \vdash_S S' : \tau'}{\Gamma \vdash_S \mathrm{let}\, p = S \,\mathrm{in}\, S' : \tau'}\ \textsc{LetIn}$$

$$\frac{\forall i\ \Gamma \vdash_t t_i : \tau_i \qquad \Gamma(x) = \tau_1 \to \cdots \to \tau_n \to \tau}{\Gamma \vdash_S (x\ t_1 \ldots t_n) : \tau}\ \textsc{App}$$

Figure 2: Typing rules of Skeletal Semantics

To type a variable, we look for its type in Γ. To type a constructor applied to a term, we type the given term, check that it matches the requirement for the constructor, and return the output type of the constructor. Both are given using $\mathit{ctype}(C)$ which returns the pair of the declared input type and output type for the constructor C as found in the type declaration list. To type a tuple, we type each component and return the tuple of the types. Finally, to type an abstraction, we type the skeleton, and return the arrow from the type of the argument to the type of the skeleton.

To type a skeleton that is a term, we simply type the term. To type a branching, we require that every branch has the same type τ, and we return τ (an empty branch therefore can be typed to any given type). To type a let binding, we first type the bound skeleton, then we type the continuation in the extended environment. To this end, we introduce the extensions of Γ with a matched type.

- $\Gamma + x \leftarrow \tau$ is the partial function Γ' s.t. $\Gamma'(x) = \tau$ and for all $y \neq x$, $\Gamma'(y) = \Gamma(y)$.
- $\Gamma + \_ \leftarrow \tau = \Gamma$.
- $\Gamma + C\ p \leftarrow \tau' = \Gamma + p \leftarrow \tau$ if $\mathit{ctype}(C) = (\tau, \tau')$.
- $\Gamma + (p_1, \ldots, p_n) \leftarrow (\tau_1, \ldots, \tau_n) = \Gamma + p_1 \leftarrow \tau_1 + \cdots + p_n \leftarrow \tau_n$.

Finally, to type n-ary function application, we require the variable in function position to be an arrow type and the types of the applied terms to match the input types of the function.

We now briefly describe how we provide polymorphic types in practice, since they are heavily used in the JSkel formalization below. Polymorphism is always explicitly declared. Thus, when declaring a polymorphic type or term, one has to give its type arguments. For example, here are the declaration of the `list` type and a `map` term.

```
type list<a> =
| Nil
| Cons (a, list<a>)
```

```
term map<a,b>: (f: (a → b)) → (l: list<a>) → list<b> =
        branch
                let Nil = l in Nil<b>
        or
                let Cons (v, q) = l in
                let w = f v in
                Cons<b> (w, q)
        end
```

As we can see, the type arguments must be explicitly given when applying a function or a constructor. On the other hand, they are not given in a pattern, because they can always be safely inferred.

3.3 Concrete Interpretation

The concrete interpretation gives a natural, or big-step, semantics to a skeletal semantics. To define the concrete interpretation, we first explain how types are translated into sets of concrete values. Then we define the evaluation relation, which relates every term or skeleton to concrete values of their type.

Formally, for each type τ we define the set of concrete values V_τ in the following way.

- If τ is a non-specified type, V_τ must be given.
- If τ is a specified type, V_τ is the set freely generated by the constructors of type τ, where the argument of each constructor is recursively generated.
- $V_{(\tau_1,\ldots,\tau_n)} = V_{\tau_1} \times \cdots \times V_{\tau_n}$.
- $V_{\tau_1 \to \tau_2} = \mathcal{R}(V_{\tau_1}, V_{\tau_2}) = \mathcal{P}(V_{\tau_1} \times V_{\tau_2})$.

Skel being a strongly typed language, every term and every skeleton has a unique type, with the exception of empty branchings that can have any type. To recover type uniqueness, we require that empty branchings be explicitly annotated with their type. Every term of type τ can be evaluated to zero, one, or several values of V_τ. In a sense, evaluation of a term is non-deterministic. Formally, the evaluation is a relation between terms/skeletons and values.

We define in Figure 3 the evaluation of a term and of a skeleton in a given environment E, which is a partial function that binds variables to values. The initial environment E_0 is an environment that binds every unspecified term to a value. We denote this evaluation as $\Downarrow_t$ for terms and $\Downarrow_S$ for skeletons.

To evaluate a variable, we look for its value in E. To evaluate a constructor applied to a term, we evaluate the term to a value and return the constructor applied to this value. To evaluate a tuple, we evaluate each component, and return the tuple of the values. Finally, the evaluation of an abstraction is a subset of the relation R in which v relates to w if and only if the skeleton can be evaluated to w, given that the abstracted variable is bound to v in the environment. It is convenient to only return a subset of the relation as we can then only consider the arguments to which the function is actually applied.

To evaluate a skeleton which is simply a term, we use the evaluation rules for terms. To evaluate a branching, we return the value v of a branch that successfully evaluates to v. To evaluate the application of a variable to terms, we look up the variable in the environment and get a relation. We then evaluate the first term and check that the returned value is in the relation, which provides a result value. If the evaluation is partial and additional terms are present, this value is in turn a relation and we continue. We thus write $R^*(v_1, \ldots, v_n)$ for

$$\frac{E(x) = v}{E, x \Downarrow_t v}\ \textsc{Var} \qquad \frac{\texttt{term x = t} \qquad E_0, t \Downarrow_t v}{E, x \Downarrow_t v}\ \textsc{Term} \qquad \frac{E, t \Downarrow_t v}{E, (\mathrm{C}\, t) \Downarrow_t \mathrm{C}\, v}\ \textsc{Const}$$

$$\frac{E, t_1 \Downarrow_t v_1 \qquad \ldots \qquad E, t_n \Downarrow_t v_n}{E, (t_1, \ldots, t_n) \Downarrow_t (v_1, \ldots, v_n)}\ \textsc{Tuple}$$

$$\frac{R \subseteq \{(v, w) \mid v \in V_\tau \wedge (E + x \leftarrow v), S \Downarrow_S w\}}{E, (\lambda x : \tau \cdot S) \Downarrow_t R}\ \textsc{Clos} \qquad \frac{E, t \Downarrow_t v}{E, \mathrm{ret}\, t \Downarrow_S v}\ \textsc{Ret}$$

$$\frac{E, S_i \Downarrow_S v}{E, (S_1 \ldots S_n) \Downarrow_S v}\ \textsc{Branch} \qquad \frac{E, S \Downarrow_S v \qquad (E + p \leftarrow v), S' \Downarrow_S w}{E, \mathrm{let}\, p = S \,\mathrm{in}\, S' \Downarrow_S w}\ \textsc{LetIn}$$

$$\frac{\forall i\ \ E, t_i \Downarrow_t v_i \qquad E(x) = R \qquad R^*(v_1, \ldots, v_n, w)}{E, (x_i\ t_1 \ldots t_n) \Downarrow_S w}\ \textsc{App}$$

Figure 3: Concrete Interpretation of Skeletal Semantics

$\exists R_1 \ldots \exists R_n . R(v_1, R_1) \wedge R_1(v_2, R_2) \wedge \cdots \wedge R_n(v_n, w)$ corresponding to the curried application. To evaluate a let binding, we evaluate the first skeleton as a value, extend the environment doing pattern matching, and evaluate the second skeleton in this extended environment. Pattern matching is recursively defined as follows.

- $E + x \leftarrow v$ is the partial function E' s.t $E'(x) = v$ and for all $y \neq x$, $E'(y) = E(y)$.
- $E + \_ \leftarrow v = E$.
- $E + (C\ p) \leftarrow C\ v = E + p \leftarrow v$.
- $E + (p_1, \ldots, p_n) \leftarrow (v_1, \ldots, v_n) = E + p_1 \leftarrow v_1 + \cdots + p_n \leftarrow v_n$.

To evaluate any term or skeleton, we assume given an initial environment E_0 that maps every non-specified term to a concrete value of the appropriate type.

It is easy to check that if $\Gamma \vdash_t t : \tau$ and $E, t \Downarrow_t v \in V_\tau$, then it entails that $v \in V_\tau$, given that Γ matches E, that is $\forall x$, $\Gamma(x) = \tau \rightarrow E(x) \in V_\tau$.

4 JSkel

In this section, we present the Skel formalization of the `GetValue`[2] method written in Skel. As a first step, we present the JSkel types and helper functions designed for a visually faithful formalization. We use a monadic approach to propagate information implicitly. We claim the result is a simple yet powerful tool.

JSkel is still work in progress. Nevertheless, we are already able to instantiate our interpreter to run basic code, namely a subset of the Expression[3] grammar production, statements such as

[2] https://tc39.es/ecma262/#sec-getvalue
[3] https://tc39.es/ecma262/#prod-Expression

the Expression Statement[4], the Variable Statement[5], and the Empty Statement[6], and Lexical Declarations[7]. We define a specification entry-point that first creates the ES initial environment, then parses and evaluates the script.

4.1 Towards a formalization

4.1.1 ECMAScript

ECMAScript is a large vernacular specification written in an imperative style. It is divided into 28 chapters and 6 appendices. Despite its complexity and verbosity, it provides a complete specification of the behavior of JavaScript. Some choices are left to the implementation, however, so a formalization has to take into account design choices that cannot be considered standard. We explain below how we deal with them.

After an introductive part in chapters 1 to 5, where the notational and algorithmic conventions are defined, the document can be divided into three main functional groups.

Chapter 6 to 9 give a taxonomy of the ES' *Data Types and Values*, providing each taxon with a definition of its operations and related invariant, followed by the definition of *abstract operations* (type conversion, comparison, object and iterator operations), the *runtime environment*, and detailed classification of the type *Object* and its internal methods. This block of chapters gives a complete overview of the execution environment in which an ES program should be executed.

The central part of the specification, chapter 10 to chapter 16, describes the actual ES programming language. Chapters 10 and 11 focus on lexical units. Language constructs are given in chapters 12 to 15, where each construct is given with its syntactic specification and its evaluation. These describe expressions, statements, functions, and scripts. Modules are defined in chapter 16.

Chapters 17 to 27 introduce the default components of the *Global Object*, which can be considered as the standard library of JavaScript. In addition, a memory model is given in Chapter 28.

4.1.2 Challenges of the Formalization

The first step of the mechanization in our purely functional Skel language is to deal with the imperative nature of the specification. This raises two issues.

First, there is a notion of implementation-dependent state that can be mutated. More precisely, we define by *state* the aggregation of all the imperative data manipulated by the specification. We design it as a record that includes the Execution Context stack, a strictness boolean flag, and a pool of Maps holding *Execution Contexts*, *Environment Records*, *Realms*, *Script Records*, and *Objects*. This record is left unspecified, as well as the functions to access and set it. This is representative of our general approach to have all the "implementation dependent" parts of the ES specification kept unspecified. To remain close to the imperative specification, we design a Skel state monad in Figure 4. This monad lets us implicitly pass the state around.

Second, the specification often breaks the usual control flow by having `return` in the middle of algorithms. We can capture such control flows by using nested branches (see below), but this

[4] https://tc39.es/ecma262/#prod-ExpressionStatement
[5] https://tc39.es/ecma262/#prod-VariableStatement
[6] https://tc39.es/ecma262/#prod-EmptyStatement
[7] https://tc39.es/ecma262/#prod-LexicalDeclaration

```
type state (*implementation dependent*)
type st<α> := state → (α, state)

term st_bind<α,β>: (v: st<α>) → (f: α → st<β>) → st<β> =
     λ s: state → let (v', s') = v s in f v' s'

term st_ret<α>: (v: α) → st<α> = λ s :state → (v, s)
```

Figure 4: State Monad in Skel

Field Name	Value
[[Type]]	One of **normal**, **break**, **continue**, **return**, or **throw**
[[Value]]	Any **ECMAScript language value** or **empty**
[[Target]]	Any **ECMAScript string** or **empty**

```
type completionType =
| Normal
| Break
| Continue
| Return
| Throw
```

```
type completionValue<α> =
| Ok α
| Abruption maybeEmpty<value>

type completionTarget :=
     maybeEmpty<string>
```

Figure 5: ES Completion Record and Skel Formalization

significantly reduces the legibility of the mechanization. We thus define a *control flow* monad to simplify the mechanization.

In addition, the specification itself introduces operators that behave much like an exception monad, to deal with `break`, function returns, or exceptions.

We thus propose in Section 4.1.3 an exception monad that captures the behavior of the ES's monadic shorthands `?` and `!`, and in Section 4.1.4 its extension to handle control-flow features.

We claim the combination of the state monad with the control-flow and the exception ones greatly simplifies our code, making JSkel easy to write, maintain and to visually compare to ES. This is shown is Section 4.2.

4.1.3 Completion Record and the ECMAScript Error Handling (?!) monad

Most ES operations do not directly return values, they instead return *completion records*. A Completion Record describes the runtime propagation of values and control flow. This record is composed of three fields, as depicted in Figure 5. In a nutshell, a completion record indicates what to do (this is a result, a break out of a loop, a return of a function, an exception being thrown), it contains an optional value, and in the case of a break or continue, it contains a target.

In theory, a completion record should only hold ES language values (Null, Undefined, Boolean, Number, BigInt, String, Symbol, and Object) or be empty. In practice, it is used to return many other constructions. If we consider the $getValue(V)$ abstract operation, the completion record given as input can hold either a Value or a Reference in its `[[Value]]` field. Many such examples litter the spec, hence we consider completion records to be polymorphic in what their "value" field holds. We declare such completion records as `type completionRecord<a>`.

```
type out<α> =
| Success completionRecord<α>
| Anomaly anomaly
```

```
type anomaly =
| AbruptAnomaly completionRecord<()>
| StringAnomaly string
| NotImplemented
```

Figure 6: Out and Anomaly Declarations

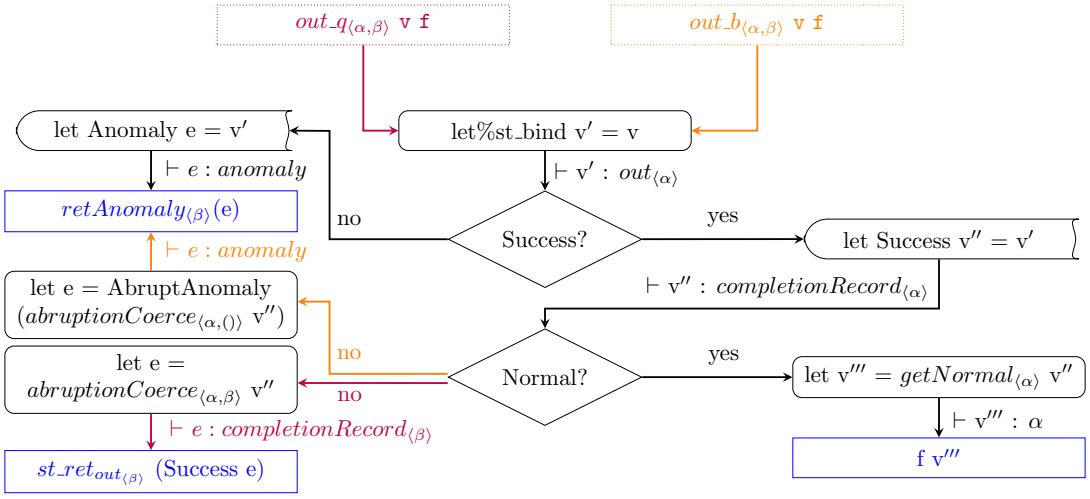


Figure 7: JSkel Model of `?` and `!`. Functions are presented in Figure B1

Figure 5 defines the types corresponding to the contents of this record: `completionType` holds control flow information, `completionValue` is either an `Ok` polymorphic constructor that contains the value of a non-abrupted computation, or an abruption. An ES abrupted Completion Record is one whose type is not `Normal`. For instance, a `Throw` record has an exception object as completion value, a `Return` record has an optional value, and a `Break` record has `empty`. Due to the aforementioned specification issues where non-values may be returned, we store the optional value in a separate constructor to be able to return it independently of the completion type. Finally, the `completionTarget` holds the optional string representing the target. An ES completion record for values thus has type `completionRecord<maybeEmpty<value>>`.

We next define a type `out` composed of two constructors, `Success` for *successful computations*, and `Anomaly` for anomalies, which intuitively corresponds to a failure of the specification. A successful computation is one that returns an ES completion record, either `Normal` or `Abrupt`. Incorrect computations are captured by the `Anomaly` constructor. In the specification, anomalies can be raised by assertion failures, or when it is explicitly written that an abstract operation call must not return an `Abrupt`. The specification authors informally guarantee that these failures never occur. We make them explicit so that we can formally express their absence and thus pave the way for a formal certification of this property.

The `Anomaly` constructor is of type `anomaly`. We define the latter with three type constructors: `AbruptAnomaly` that holds a completion record in case an evaluation returns an unexpected abruption, `StringAnomaly` that contains a textual information about an anomaly—when an assertion of the specification is broken or when an implementation-dependent operation fails, and `NotImplemented` that signals that we have not yet implemented some feature.

ES abruptions are propagated through an abstract method called `ReturnIfAbrupt`. Basi-

```
type controlFlow<α,β> =
| ContinueControl α
| ReturnControl β

term cf_ret<β> : β → controlFlow<(),β>
term cf_cont<β> : () -> controlFlow<(),β>
term cf_assign<α,β> : α → controlFlow<α,β>

term cf_bind<α,β,γ> : controlFlow<α,β> → (α → controlFlow<γ,β>) → controlFlow<γ,β>
term cf_res<α,β,γ> : controlFlow<α,β> → (α → β) → β
```

Figure 8: The controlFlow type with binders and setters

cally, this method gets a completion record, and either returns the value of the `[[Value]]` field in case of a normal completion or propagates the abruption.

In cases of an abstract operation or a recursive evaluation, the specification uses the prefix `?` to indicate that `ReturnIfAbrupt` has to be applied to the resulting completion. This operator is basically a monadic bind for an exception monad. The other operator used in the specification is `!`: it behaves like `?` on a normal result, but it asserts an abruption cannot occur. We thus model it as transforming an abruption into an anomaly. These behaviors are reflected in the flow-chart presented in Figure 7, presenting respectively $\texttt{out\_q}_{\langle\alpha,\beta\rangle}$(red arrows), and $\texttt{out\_b}_{\langle\alpha,\beta\rangle}$(orange arrows). We define them as the monadic binders of the combination of `st` and `out`.

```
term (out_q|out_b)<α,β> : st<out<α>> → (α → st<out<β>>) → st<out<β>>
term out_ReturnIfAbrupt<α,β> : out<α> → (α → st<out<β>>) → st<out<β>>
```

The state monad is required for getting the result in case it is stored in the data structures in it, such as when manipulating references to values in the heap. We define, in Figure B1, getters (`getNormal, getAbrupt, ...`) for each `completionRecord`'s type, and returns (`retAnomaly, retAbrupt, retNormal, ...`) for successful and anomaly computations. Note the use of the `abruptionCoerce` operation that is only defined for completion records that are abruptions. It is then the identity, but it changes the type parameter of the completion record.

Given two algorithmic steps,

1. let bar be ? AbstractOperation(foo)
2. Return bar

we represent them as

```
1 let%out_q bar = abstractOperation(foo) in
2 retNormal<τ_bar> bar
```

where τ_{bar} is the type of `bar`.

4.1.4 A Control-Flow monad

As said earlier, the ES specification uses imperative control flow, such as returning in the middle of an algorithm. We introduce, in Figure 8, the type `controlFlow<α,β>` for computations that either continue with an argument of type α or that terminate with a result of type β. It is composed of two constructors: `ReturnControl`, to return a result of type β, and `ContinueControl` to continue the execution with a value of type α to be given to the continuation.

We define a monadic binder `cf_bind`, a function `cf_res` for extracting the value from a `controlFlow<α,β>` term, and three return functions: `cf_ret` for `ReturnControl`, `cf_cont` for signaling a unit `ContinueControl`, and `cf_assign` for an α `ContinueControl`. We could

1. If foo is `true` Return `1`
2. If bar is `true` Return `2`
3. If baz is `true` Return `3`
4. Return `4`

```
(*no control-flow monad*)
branch let True = foo in 1
or     let False = foo in
       branch let True = bar in 2
       or     let False = bar in
              branch let True = baz in 3
              or     let False = baz in 4
              end
       end
end
```

```
(*control-flow monad*)
let%cf_res result =
  branch let True = foo in cf_ret<int> 1
  or     let False = foo in cf_cont<int> ()
  end;%cf_bind
  branch let True = bar in cf_ret<int> 2
  or     let False = bar in cf_cont<int> ()
  end;%cf_bind
  branch let True = bar in cf_ret<int> 3
  or     let False = bar in cf_cont<int> ()
  end;%cf_bind
  cf_ret<int> 4
in result
```

Figure 9: Code with and without Control Flow Monad. See Figure B2 for Functions

implement `cf_cont` using `cf_assign`, but we find that making the distinction explicit clarifies the code.

We illustrate in Figure 9 the translation of algorithmic steps in Skel, with and without the control-flow monad. In the examples, we use the form `let (True|False) = ... in ...` to test whether a value is `true` or `false`. Despite the slight overhead in notation, the control-flow approach is closer to the algorithmic steps and can be consistently applied to deal with the common case of a conditional that returns without an else branch. One can achieve a similar behavior without the control flow monad, at the cost of nested branching (highlighed on the left-hand side of Figure 9). It is possible to avoid the nesting of branching by hoisting all the branches at top-level. This results in the following code.

```
branch let True = foo in 1 (*1*)
or     let False = foo in let True = bar in 2
or     let False = foo in let False = bar in let True = baz in 3
or     let False = foo in let False = bar in let False = baz in 4
end
```

The reason previous conditions are repeated is because there is no guarantee the evaluation of a branching will consider branches in declaration order. In addition, using a collecting semantics (where all branches are considered) would give the wrong result if we did not restate all conditions.

We define `st<cf<`α`,out<`β`>>>` type as the combination of the three monadic types, and accordingly, the definitions of the binders and return functions as `bind`, `cf_out`, `cont`, and `ret`. Our general approach is to encapsulate all the algorithmic steps in this type, returning an `st<out<`β`>>` at the end of every function that may raise an exception. To this end, we start each algorithm with `let%cf_out result =` to enter the full monad with control, and we exit the control monad at the end of the algorithm with `in result`.

Figure 10 gives a variant of Figure 9 with exceptions. Notice that the only thing that changes is the first step and the use of the appropriate monadic binders for type `st<cf<`α`,out<`β`>>>`.

Despite the closeness to the algorithmic steps, the latter figure, in lines 4, 6 and 8, shows redundancy of code in the `or` branches. Indeed, each time the condition is not satisfied, we have to explicitly state that the evaluation continues. To avoid having to do so, we define two

1. If foo is `true` throw `FooError`
2. If bar is `true` Return 2
3. If baz is `true` Return 3
4. Return 4

```
let%cf_out result =
    branch let True = foo in
           let%throw fe=fooError<int> in fe
    or     let False = foo in cont<int> () end;%bind
    branch let True = bar in ret<int> 2
    or     let False = bar in cont<int> () end;%bind
    branch let True = bar in ret<int> 3
    or     let False = bar in cont<int> () end;%bind
    ret<int> 4
in result
```

Figure 10: Variant of the Figure 9 with Exceptions. See Figure B3 for Functions.

boolean binder terms, `ifpTrue` and `ifpFalse`, for introducing partiality in branchings. The `ifpTrue` binder, in the case v is `True`, applies the bind function `f` to a unit, otherwise, in the case v is `False`, the branch returns a continuation of type α. The `ifpTrue` term definition, and its application to the Figure 10 example, results in the following code.

```
term ifpTrue<α>:
 (v: boolean) →
 (f: () → st<controlFlow<(),out<α>>>
 ) →
 st<controlFlow<(),out<α>>> =

   branch let True = v in f ()
   or     let False = v in cont<α>()
   end
```

```
let%cf_out result =

   branch foo;%ifpTrue
          let%throw fe=fooError<int> in fe end;%bind
   branch bar;%ifpTrue ret<int> 2 end;%bind
   branch bar;%ifpTrue ret<int> 3 end;%bind
   ret<int> 4

in result
```

4.2 The `GetValue(V)` example

In this section, we illustrate our design choices through the mechanization in Skel of the `GetValue` abstract method. This method, presented in Figure 11, is one of the most referenced methods in the specification, as it is often called after the evaluation of syntactic constructors of the language. It takes V as input, a completion record containing either a value or a reference, and it returns a value as output. In a nutshell, if V is a value, `GetValue` returns it, and if it is a reference, `GetValue` acts like a binding resolver to obtain a primitive value, Object, or Environment Record. The reference type is shown in Figure 12. It is a record with a field `[[BaseValue]]` of type `environmentRecord` or `value`, and additional fields not relevant here. Our mechanization in Skel leaves this type unspecified. We describe the types and terms used in our formalization of `getValue` in Figure B4.

We now describe step by step how we formalize this method (the code is collected as a single function in Figure B5). The type mixing values and references is the specified type `valref` defined as `Value value | Reference reference`. The argument of `GetValue` thus has type `out<valref>`. The first step of the abstract method applies `ReturnIfAbrupt` on V. In case it is an abruption, the method propagates the completion record to the caller, otherwise it extracts the `completionValue`. To model this, we use the `%returnIfAbrupt` monadic binder.

```
let%returnIfAbrupt v = v
```

If V is a normal completion, then the newly bound V will have type `valref`. We could use a different type for `returnIfAbrupt` to be able to write `let%bind v = returnIfAbrupt v`, but we would then need to specify the polymorphic type components of `returnIfAbrupt`. We are working on type inference to enable such a change in the future.

1. ReturnIfAbrupt(V).
2. If Type(V) is not Reference, return V.
3. Let base be GetBase(V).
4. If IsUnresolvableReference(V) is **true**, throw a **ReferenceError** exception.
5. If IsPropertyReference(V) is **true**, then
   a. If HasPrimitiveBase(V) is **true**, then
      i. Assert: In this case, base will never be **undefined** or **null**.
      ii. Set base to ! ToObject(base).
   b. Return ? base.[[Get]](GetReferencedName(V), GetThisValue(V)).
6. Else,
   a. Assert: base is an Environment Record.
   b. Return ? base.GetBindingValue(GetReferencedName(V), IsStrictReference(V)).

Figure 11: The ECMAScript's $GetValue(V)$

Field Name	Value
[[BaseValue]]	**Value** or **Environment Record**
[[Strict]]	Boolean Flag
[[ThisValue]]	**Value** or **Empty**
[[Name]]	String Value

Figure 12: The Reference type

The next step inspects the type of V. If it is not a reference, hence it is a value, we return it. Otherwise, the `ifpFalse` implicitly propagate a continuation.

```
2 branch valref_Type(v, T_Ref);%ifpFalse let Value v = v in ret<value> v end;%bind
```

The `GetBase` method in the third instruction takes the `[[BaseValue]]` from the reference V, and assigns it to base. The previous step guarantees that V has type `reference`, so we procede:

```
3 let Reference v = v in let base = getBase(v) in
```

Now base has type `ref_bv`. The step 4 follows the same pattern as step 2:

```
4 branch isUnresolvableReference v;%ifpTrue
5        let%throw refErr = referenceError<value> in refErr end;%bind
```

An *unresolvable* reference is the one that has `Undefined` as `[[BaseValue]]`. In this case, a `ReferenceError` is thrown. Otherwise, step 5 inspects whether V is a *property reference*. Being a property reference means that a reference have a non-null, defined base value of type `value`.

```
6 branch let True = isPropertyReference v in
```

Step 5.a. checks whether the reference V has a *primitive* `[[BaseValue]]`. Having a primitive base means that the reference points to a primitive value, i.e., a value of type `Boolean`, `String`, `Symbol`, `BigInt`, or `Number`. After some assertions, this algorithmic step sets base as `ToObject`

applied to base. Note the use of the %b monadic operator corresponding to the ! in the call: ToObject must return a normal result, otherwise an anomaly is raised. There is no else branch, but since base is not primitive in that case, it has to be an Object. We extract it using pattern matching. In both case, the resulting loc_Object is returned to the binder let%bind base using assign.

```
7     let%bind base =
8         branch let True = hasPrimitiveBase v in let R_Value base = base in
9                val_Type(base,T_Null);%assF val_Type(base,T_Undefined);%assF
10               let%b base = toObject(base) in assign<loc_Object,value>(base)
11        or     let False = hasPrimitiveBase v in
12               let R_Value (Obj base) = base in assign<loc_Object,value>(base)
13        end in
```

In line 9, we use the monadic binder assF⟨*value*⟩ for "assert false". If the result of the val_type application is **true**, an anomaly is raised.

Once base is set, the algorithmic step 5.b calls the *object internal method* [[Get]]. The JSkel formalization is straightforward with it, using base as the first argument to o_Get.

```
14    let name = getReferencedName(v) in let%q thisVal = getThisValue(v) in
15    let%q v' = o_Get(base, name, thisVal) in ret<value> v'
```

Note that we make explicit the order of the calls to getReferencedName and getThisValue, as Skel requires function application to be to fully computed terms. We could faithfully model different execution orders using branching, but this would make the mechanization confusing. The call to getReferencedName is pure, although the specification contains an assertion[8]. Since we know that the assertion is satisfied by typing, we do not include it. The call to getThisValue, however, is not pure as the assertion there[9] is not captured by typing (although we have checked that the reference is indeed a property reference at step 5). We thus use the %q binder in that case to extract the pure value or propagate the anomaly, if any.

If V is not a property reference, then base must be an Environment Record. In this last step, we check that this is the case using an assertion, then we apply the getBindingValue operation.

```
16or      let False = isPropertyReference v in ref_Type(base, T_R_EnvRec);%assT
17        let R_EnvironmentRecord base = base in
18        let name = getReferencedName(v) in let strict = isStrictReference(v) in
19        let%q v' = er_GetBindingValue(base, name, strict) in ret<value> v'
20end
```

As with most of the algorithms in our mechanization, we need to return a result that is out of the control monad. The whole algorithm is thus surrounded by the following piece of code.

```
let%cf_out result = (*GetValue's algorithmic steps*) in result
```

5 Conclusion and Future Work

We have described how the Skel language can be used to mechanize complex semantics. In particular, we have shown how carefully chosen monads can help in keeping the specification and

[8] https://tc39.es/ecma262/#sec-getreferencedname
[9] https://tc39.es/ecma262/#sec-getthisvalue

its mechanization very similar. The current state of our formalization of JavaScript can be found online, at `https://gitlab.inria.fr/skeletons/jskel`. It includes the Skel formalization of the language, boilerplate code to integrate with existing JavaScript parsers (see Appendix A), and an implementation of all the unspecified terms. Using `necro-ml` [7], we have all that is needed to generate an executable semantics.

To this point, our main effort has been in mechanizing the foundations of the language, of which `GetValue` is a good example. We are now ready to formalize more constructs than simple expressions and statements.

We also have been able to start the evaluation of maintainability of the approach. This project began with the formalization of ECMAScript 2020, but after few months, we switched to the newer ECMAScript 2021. This process took only a day of work. This may change once we have a more complete formalization of the specification, but as we can easily produce a list of differences between specifications, the amount of work is proportional to the size of changes and not to the size of the specification. In particular, we only have one mechanization to change, instead of two formal developments and a correctness proof as is the case for JSCert.

A short term goal is to formalize enough of JavaScript to have all the features needed to run the ECMA-262 test suite. A longer term goal is to validate that we can use our mechanized semantics to prove properties of the language. A first property of interest is the guarantee that assertions in the specification are actually satisfied. To achieve this goal, we will use the `necro-coq` tool [8]. As the formalization of JavaScript helped us refine the Skel language, we believe the generation of a Coq semantics will provide many opportunities to improve `necro-coq`.

References

[1] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javascript specification. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 49:87–100, 01 2014.

[2] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal Semantics and their Interpretations. *Proceedings of the ACM on Programming Languages*, 44:1–31, 2019.

[3] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. JSExplain: A Double Debugger for JavaScript. In *The Web Conference 2018*, pages 1–9, Lyon, France, Apr 2018.

[4] Arthur Charguéraud. Pretty-big-step semantics. In *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*, pages 41–60. Springer, 2013.

[5] Olivier Danvy. Three steps for the cps transformation. Technical Report CIS-92-2, Kansas State University, 1991.

[6] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. *ECOOP, Lecture Notes in Computer Science*, pages 126–150, 06 2010.

[7] Enzo Crance Martin Bodin, Nathanael Courant and Louis Noizet. Necro Ocaml Generator, https://gitlab.inria.fr/skeletons/necro-ml.

[8] Louis Noizet. Necro Gallina Generator, https://gitlab.inria.fr/skeletons/necro-coq.

[9] Louis Noizet. Necro Library, https://gitlab.inria.fr/skeletons/necro.

[10] Louis Noizet. Necro Library, https://gitlab.inria.fr/skeletons/necro-tex.

[11] Daejun Park, Andrei Stefănescu, and Grigore Roşu. Kjs: A complete formal semantics of javascript. *ACM SIGPLAN Notices*, 50:346–356, 06 2015.

```
type expression =
| Expr_AssExpr assignmentExpression
| Expr_Comma (expression, assignmentExpression)
```

Figure A1: Type definition of expressions in JSkel

A The parsing process

In JSkel, we formalize the JavaScript syntax as it is defined in ECMAScript. For instance, we give in Figure A1 the description of the Expression type[10], which is the same as its ES counterpart.

Unfortunately, no existing parser of JavaScript provides such a faithful representation. We thus had to choose between implementing our own parser or translating the AST provided by on-the-shelf parsers. We chose the latter.

More precisely, we use a parser that conforms to the *SpiderMonkey*'s Parser API[11], which is followed by all parsers we have found. We chose the Flow Parser[12] library because it is written in OCaml, which is the language we can instantiate our interpreter into, and because it is used to manipulate JavaScript in industrial setting.

When instantiating the interpreter, we define a transformation that, given a Flow AST, produces a well-typed ES AST. In the long term, we would like to write a parser faithful to the specification, but the complexity of ES syntax makes it challenging.

In Figure A2, we show the resulting AST of a program produced by the Flow Parser. Its transformation is presented in Figure A3.

An execution of our interpreter on a JS program first performs `InitializeHostDefinedRealm`[13], then parses the source code, and finally calls `ScriptEvaluation`[14].

[10] https://tc39.es/ecma262/#prod-Expression
[11] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API
[12] https://github.com/facebook/flow
[13] https://tc39.es/ecma262/#sec-initializehostdefinedrealm
[14] https://tc39.es/ecma262/#sec-runtime-semantics-scriptevaluation

```
(({Loc.source = None; start = {Loc.line = 1; column = 0};
   _end = {Loc.line = 1; column = 16}},
  {Flow_ast.Program.statements =
    [({Loc.source = None; start = {Loc.line = 1; column = 0};
       _end = {Loc.line = 1; column = 10}},
      Flow_ast.Statement.VariableDeclaration
       {Flow_ast.Statement.VariableDeclaration.declarations =
         [({Loc.source = None; start = {Loc.line = 1; column = 4};
            _end = {Loc.line = 1; column = 9}},
           {Flow_ast.Statement.VariableDeclaration.Declarator.id =
             ({Loc.source = None; start = {Loc.line = 1; column = 4};
               _end = {Loc.line = 1; column = 5}},
              Flow_ast.Pattern.Identifier
               {Flow_ast.Pattern.Identifier.name =
                 ({Loc.source = None; start = {Loc.line = 1; column = 4};
                   _end = {Loc.line = 1; column = 5}},
                  {Flow_ast.Identifier.name = "x"; comments = None});
                annot =
                 Flow_ast.Type.Missing
                  {Loc.source = None; start = {Loc.line = 1; column = 5};
                   _end = {Loc.line = 1; column = 5}};
                optional = false});
            init =
             Some
              ({Loc.source = None; start = {Loc.line = 1; column = 8};
                _end = {Loc.line = 1; column = 9}},
               Flow_ast.Expression.Literal
                {Flow_ast.Literal.value = Flow_ast.Literal.Number 1.;
                 raw = "1"; comments = None})})];
        kind = Flow_ast.Statement.VariableDeclaration.Let; comments = None});
     ({Loc.source = None; start = {Loc.line = 1; column = 11};
       _end = {Loc.line = 1; column = 16}},
      Flow_ast.Statement.Expression
       {Flow_ast.Statement.Expression.expression =
         ({Loc.source = None; start = {Loc.line = 1; column = 11};
           _end = {Loc.line = 1; column = 16}},
          Flow_ast.Expression.Binary
           {Flow_ast.Expression.Binary.operator =
             Flow_ast.Expression.Binary.Plus;
            left =
             ({Loc.source = None; start = {Loc.line = 1; column = 11};
               _end = {Loc.line = 1; column = 12}},
              Flow_ast.Expression.Identifier
               ({Loc.source = None; start = {Loc.line = 1; column = 11};
                 _end = {Loc.line = 1; column = 12}},
                {Flow_ast.Identifier.name = "x"; comments = None}));
            right =
             ({Loc.source = None; start = {Loc.line = 1; column = 15};
               _end = {Loc.line = 1; column = 16}},
              Flow_ast.Expression.Literal
               {Flow_ast.Literal.value = Flow_ast.Literal.Number 1.;
                raw = "1"; comments = None});
            comments = None});
        directive = None; comments = None})];
   comments = None; all_comments = []}),
 [])
```

Figure A2: Parsing of "`let x = 1; x + 1`"

```
Script
 (VSome (ScriptBody (STMTList
     (STMTList_Item
         (STMTListIt_Decl
           (LexicalDeclaration
(*LexicalDeclaration :
    LetOrConst BindingList
*)
             (LexDecl
               (Let,
                LexBind (LexBind_ID (BI_identifier (IdentifierName "x"),
                VSome (Init (AssE_CondE (CondE_ShortCircuit (SCE_LOE
                  (LOE_LAE (LAE_BOR (BOE_BXE (BXE_BAE (BAE_Equality
                     (EqExp_Relational (RelExp_Shift (ShExp_Additive
                       (AddE_Multiplicative (MulE_Exponentiation
                         (EE_UnaryExpression (UnaExp (UpdExp
                           (LHSE_NewExpression (NE_MemberExpression
                             (ME_PrimaryExpression (PE_Literal
                               (Lit_NumericLiteral (DecimalLiteral 1.))
                      ))))))))))))))))))))
          )))))
         ),
       STMTListIt_STMT (ExpressionStatement (ExprSTMT (Expr_AssExpr
         (AssE_CondE (CondE_ShortCircuit (SCE_LOE (LOE_LAE (LAE_BOR (BOE_BXE
           (BXE_BAE (BAE_Equality (EqExp_Relational (RelExp_Shift
             (ShExp_Additive
(*AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression
*)
              (AddE_Sum
                (AddE_Multiplicative (MulE_Exponentiation
                  (EE_UnaryExpression (UnaExp
                    (UpdExp (LHSE_NewExpression (NE_MemberExpression
                      (ME_PrimaryExpression (PE_IdentifierReference
                        (IR_identifier (IdentifierName "x"))))))))),
                 MulE_Exponentiation
                   (EE_UnaryExpression (UnaExp (UpdExp (LHSE_NewExpression
                     (NE_MemberExpression (ME_PrimaryExpression (PE_Literal
                       (Lit_NumericLiteral (DecimalLiteral 1.))
                   )))))))
              ))
            ))))))))))))))))))
```

Figure A3: The transformation of Figure A2

B Unspecified Terms

For introductive purposes to JSkel, we have left these terms unspecified, presenting only their definition. All the terms presented in this section are fully implemented in the `skel` language.

```
(*Operations of the completionRecord type*)
term getNormal<α> : completionRecord<α> → α
term getAbrupt<α> : completionRecord<α> → (maybeEmpty<value>, completionType)
term abruptionCoerce<α,β> : completionRecord<α> → completionRecord<β>

(*out type getters and setters*)
term getAnomaly<α> : out<α> → anomaly
term retAnomaly<α> : anomaly -> st<out<α>>
term retAbrupt<α> : (maybeEmpty<value>, completionType) -> st<out<α>>
term retNormal<α> : α -> st<out<α>>
```

Figure B1

```
(*Types*)

type boolean =
| False
| True

(*Binders*)

term cf_bind<α,β,γ> :
     controlFlow<α,β> →
     (α → controlFlow<γ,β>) →
     controlFlow<γ,β>

term cf_res<α,β> :
     controlFlow<α,β> →
     (α → β) →
     β

(*Returns*)

term cf_ret<α> : α → controlFlow<st<out<α>>>
term cf_cont<α> : () → controlFlow<st<out<α>>>
```

Figure B2

```
(*Binders*)

term cf_out<β,γ> :
     st<controlFlow<(),β>> →
     (β → γ) →
     st<γ>

term throw<β> :
     (() → st<out<β>>) →
     (st<controlFlow<(),out<β>>> → st<controlFlow<(),out<β>>>) →
     st<controlFlow<(),out<β>>>

term bind<α,β,γ> :
     st<controlFlow<α,out<β>>> →
     (α → st<controlFlow<γ, out<β>>>) →
     st<controlFlow<γ, out<β>>>

(*Returns*)

term cont<β> : () → st<controlFlow<(), out<β>>>
term ret<β> : β → st<controlFlow<(), out<β>>>

(*Functions*)

term fooError<α> : () → st<out<α>>
```

Figure B3

```
(*Types*)

type valref = | Reference reference | Value value
type type_valref = | T_Ref | T_Val
type ref_bv = | R_Value value | R_EnvironmentRecord loc_EnvironmentRecord
type type_ref_bv = | T_R_EnvRec | T_R_Value
type value = | Null | Undefined | String string | Number number
             | BigInt bigInt | Object loc_Object | Symbol value
             | Boolean boolean
type type_value = | T_Null | T_Undefined | T_String | T_Number
                  | T_BigInt | T_Object | T_Symbol | T_Boolean

(*Binders*)

term returnIfAbrupt<α,β,γ> :
     out<α> → (α →
     st<controlFlow<γ, out<β>>>) →
     st<controlFlow<γ, out<β>>>

term assF<α,β> :
     boolean →
     (boolean → st<controlFlow<α,out<β>>>) →
     st<controlFlow<α,out<β>>>

term assT<α,β> :
     boolean →
     (boolean → st<controlFlow<α,out<β>>>) →
     st<controlFlow<α,out<β>>>

(*Returns*)

term assign<α,β> : α → st<controlFlow<α,out<β>>>

(*Boolean type checkers*)

term valref_Type : (valref, type_valref) → boolean
term val_Type : (value, type_value) → boolean
term ref_bv_Type : (ref_bv, type_ref_bv) → boolean

(*Reference type abstract methods*)

term hasPrimitiveBase : reference → boolean
term getBase : reference → ref_bv
term isUnresolvableReference : reference → boolean
term isPropertyReference : reference → boolean
term getReferencedName : reference → string
term isStrictReference : reference → boolean

(*Type conversion operations*)

term toObject : value -> loc_Object
(*Object internal method*)
term o_Get : (loc_Object, string, value) → st<out<value>>

(*Environment Record's abstract method*)

term er_GetBindingObject : (loc_EnvironmentRecord, string, boolean) → st<out<value>>

(*Errors*)

term referenceError<α> : () → st<out<α>>
```

Figure B4

```
term getValue : (v : out<valref>) -> st<out<value>> =
   let%cf_out result =
     let%returnIfAbrupt v = v in
     branch valref_Type(v, T_Ref);%ifpFalse let Value v = v in ret<value> v end;%bind
     let Reference v = v in let base = getBase(v) in
     branch isUnresolvableReference v;%ifpTrue
            let%throw re = referenceError<value> in re end;%bind
     branch let True = isPropertyReference v in
            let%bind base =
               branch let True = hasPrimitiveBase v in let R_Value base = base in
                      val_Type(base,T_Null);%assF val_Type(base,T_Undefined);%assF
                      let%b base = toObject(base) in assign<loc_Object,value>(base)
               or     let False = hasPrimitiveBase v in
                      let R_Value (Obj base) = base in assign<loc_Object,value>(base)
               end in
            let name = getReferencedName(v) in let%q thisVal = getThisValue(v) in
            let%q v' = o_Get(base,name,thisVal) in ret<value> v'
     or     let False = isPropertyReference v in ref_bv_Type(base,T_R_EnvRec);%assT
            let R_EnvironmentRecord base = base in
            let name = getReferencedName(v) in let strict = isStrictReference(v) in
            let%q v' = er_GetBindingValue(base,name,strict) in ret<value> v'
     end
   in
   result
```

Figure B5

Normalisation vérifiée du langage Lustre

Timothy Bourke[1,2], Paul Jeanmaire[1,2], Basile Pesin[1,2], Marc Pouzet[2,1]

[1] Inria Paris, France
[2] Département d'informatique de l'École normale supérieure, CNRS, PSL University, Paris, France

Résumé

Lustre est un langage synchrone à flots de données conçu pour programmer des systèmes embarqués. Dans le cadre du projet Vélus, nous avons développé et formalisé dans Coq un compilateur qui accepte une forme normalisée du langage et la compile vers du code impératif. Si cette forme réduite prend en charge un code généré depuis une interface utilisateur basée sur les schémas-blocs, nous voulons offrir au programmeur la possibilité de manipuler le langage complet.

Dans cet article nous présentons l'étape de normalisation, qui transforme le langage de programmation en langage normalisé. Cette transformation est décomposée en trois étapes afin de simplifier les preuves de correction. Pour établir la préservation de la sémantique, il est nécessaire de démontrer que les trois passes préservent certaines propriétés statiques et dynamiques du langage. En particulier, il faut prouver le lien entre le typage des horloges et la sémantique dynamique pour pouvoir raisonner sur la suite de la compilation.

1 Introduction

La conception de systèmes embarqués critiques est souvent basée sur des modèles de type schémas-blocs qui permettent la définition et l'interconnexion de comportements temporels. Un tel modèle peut être compris comme une composition récursive de séquences infinies, les flots de valeurs à calculer pas à pas, et compilé dans un code impératif pour une exécution cyclique. C'est l'idée principale derrière le langage académique Lustre [13] et son descendant industriel Scade 6 [10]. Le projet Vélus [1] [5, 6] vise à formaliser les particularités de ces langages, leurs systèmes de types [11] et leurs schémas de compilation [3] dans l'assistant de preuve Coq [12].

Les précédents travaux sur Vélus ont traité une forme restreinte du langage. Cette forme suffit pour traduire les programmes représentés graphiquement et est une étape préalable à la génération de code impératif. Mais elle ne représente qu'un sous-ensemble du vrai langage source et elle est moins commode pour les programmes écrits ou lus dans un éditeur. En outre, le traitement de la forme plus générale dans un assistant de preuve soulève d'intéressantes questions techniques. On s'intéresse donc ici au passage de la forme générale du langage à la forme normalisée. La transformation en elle-même est presque banale ; le défi est de vérifier sa correction dans un assistant de preuve. Notre contribution est de présenter une implémentation qui facilite cette tâche et les invariants principaux qui sous-tendent notre preuve formelle.

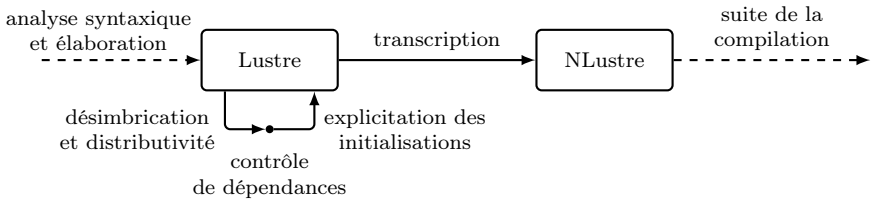


FIGURE 1 – Passes de normalisation

1. https://velus.inria.fr

```
1  node count_down(res:bool; n:int)
2  returns (cpt:int)
3  let
4    cpt = if res then n else (n fby (cpt - 1));
5  tel
6
7  node rising_edge_retrigger(i:bool; n:int)
8  returns (o : bool)
9  var edge, ck:bool; v:int;
10 let
11   edge = i and (false fby (not i));
12   ck = edge or (false fby o);
13   v = merge ck
14         (count_down((edge, n) when ck))
15         (0 when not ck);
16   o = v > 0;
17 tel
```

FIGURE 2 – Exemple : source Lustre

```
1  node count_down(res:bool, n:int32)
2  returns (cpt:int32)
3  var $114:int32, $215:int32, $214:bool;
4  let
5    $214 = true fby false;
6    $215 = 0 fby (cpt - 1);
7    $114 = if $214 then n else $215;
8    cpt = if res then n else $114;
9  tel
10
11 node rising_edge_retrigger(i:bool, n:int32)
12 returns (o:bool)
13 var edge:bool, ck:bool, v:int32,
14     $122:bool, $121:bool,
15     $124:int32 when ck;
16 let
17   $122 = false fby (not i);
18   edge = i and $122;
19   $121 = false fby o;
20   ck = edge or $121;
21   $124 = count_down(edge when ck, n when ck);
22   v = merge ck $124 (0 when not ck);
23   o = v > 0;
24 tel
```

FIGURE 3 – Exemple : NLustre généré

i	F	**T**	F	F	F	F	F	T	F	T	F	F	F	F	···
n	3	**3**	3	3	3	3	3	3	3	3	3	3	3	3	···
edge	F	**T**	F	F	F	F	F	T	F	T	F	F	F	F	···
ck	F	**T**	**T**	**T**	**T**	F	F	T	T	T	T	T	T	F	···
edge when ck		**T**	F	F	F			T	F	T	F	F	F		···
count_down(...)		**3**	**2**	**1**	**0**			3	2	3	2	1	0		···
v	0	**3**	**2**	**1**	**0**	0	0	3	2	3	2	1	0	0	···
o	F	**T**	**T**	**T**	**F**	F	F	T	T	T	T	T	F	F	···

FIGURE 4 – Exemple d'une trace du nœud `rising_edge_retrigger`

Les parties du compilateur Vélus concernées par cette transformation sont schématisées dans la figure 1. L'analyse syntaxique suivie d'une passe d'élaboration ajoutant les annotations de typage produit un programme dans la syntaxe abstraite *Lustre*. La passe de normalisation est chargée de mettre le programme sous forme normalisée. On la sépare en deux transformations source à source. La première désimbrique les éléments qui ne doivent se trouver qu'à la racine d'une expression et distribue les opérateurs sur les listes d'arguments. La deuxième explicite des initialisations pour simplifier la forme des opérateurs séquentiels. Après la désimbrication, on vérifie que le programme ne contient pas de dépendance circulaire. Ensuite, la passe de transcription transforme le programme dans la syntaxe abstraite *NLustre* qui encode les exigences de la forme normalisée et sert de point d'entrée pour la suite de la compilation.

Exemple. Le programme en figure 2 définit un opérateur typique de l'automatique [14, §1.2.2] qui est fourni par les bibliothèques de Scade et de Simulink. Il est composé de deux fonctions de

flots, également appelées *nœuds*. La normalisation le transforme dans le programme en figure 3.

Le premier nœud s'appelle `count_down`, prend en entrée deux flots, `res` et `n`, et rend en sortie un flot `cpt`. Dans le corps du nœud, une équation définit le flot associé à `cpt` avec une expression conditionnelle qui prend la valeur de l'entrée `n` lorsque `res` est vrai et sinon la valeur définie par la sous-expression `n fby (cpt - 1)`. L'opérateur `fby`, « *followed by* », produit un flot en décalant d'un instant le flot à droite et en remplissant le vide initial ainsi créé avec la première valeur du flot à gauche. Ici, le résultat est un flot `cpt` qui compte à rebours de la valeur courant de `n` en recommençant chaque fois que `res` est vrai.

La normalisation de `count_down` donne le nœud du même nom dans la figure 3. Dans l'équation de `cpt`, le `fby` a été désimbriqué et remplacé par la nouvelle variable locale `$114`. Ensuite, une variable d'initialisation `$214` a été ajoutée et utilisée dans la définition de `$114` pour choisir entre la valeur initiale venant de `n` et toutes les autres valeurs venant de `$215`. Par conséquent, tous les `fby`s sont initialisés par une constante dans l'expression à gauche. Pour ce nœud, il n'était pas nécessaire de distribuer un opérateur sur une liste d'arguments.

Le deuxième nœud, `rising_edge_retrigger`, attend un front montant, c'est-à-dire, un F suivi directement d'un T, sur son entrée `i` et émet ensuite la valeur T n fois sur son unique sortie. Une trace est donnée dans la figure 4. On y voit en caractères gras le premier front montant sur l'entrée `i`, la valeur de `n` à cet instant et, sur la dernière ligne, la réponse du nœud sur `o`. Le corps du nœud est composé de quatre équations, dont trois pour les variables locales du nœud. La première équation détecte les fronts montants en considérant chaque paire de valeurs successives d'`i`. L'équation de `ck` détermine si un compte à rebours est actif. C'est le cas si un front montant vient d'être détecté ou si la sortie était vraie dans l'instant antérieur.

L'équation de `v` utilise les deux opérateurs d'échantillonnage `when` et `merge`. Un `when` filtre un premier flot en fonction de la valeur d'un deuxième. Par exemple, l'expression `edge when ck` prend la valeur de celle de `edge` seulement lorsque `ck` est vrai, comme on peut voir dans la trace. On dit alors que le flot est « présent ». Aux autres instants, laissés vides dans la figure, on dit que le flot est « absent ». Dans l'expression `count_down((edge, n) when ck)`, le premier nœud est appliqué à deux flots qui sont tous les deux filtrés par `ck`. Le cadencement de cette instance est donc plus lent que celui de son contexte. Un `merge` fusionne deux flots en fonction de la valeur de son premier argument. Lorsque le premier flot est vrai, une valeur est prise du deuxième flot qui doit être présent et le troisième flot doit être absent, et inversement. Donc, la valeur de `v` vient de l'instance de `count_down` quand `ck` est vrai et d'un flot de zéros sinon.

Enfin, la dernière équation assure que la sortie n'est vraie que lorsque le compte à rebours a une valeur positive.

Le résultat de la normalisation de `rising_edge_retrigger` est présenté dans la figure 3. Dans l'équation de `v`, l'instance de nœud a été desimbriquée et remplacée par la variable `$124`. Dans l'expression définissant celle-ci se trouve la seule application de la distributivité dans cet exemple : l'opérateur `when` a été distribué sur la liste d'arguments `(edge, n)`. En général, la distributivité s'applique aussi aux arguments de `merge`, `if-then-else` et `fby`.

Les `fby`s dans les définitions d'`edge` et de `ck` ont été désimbriqués mais, ayant déjà des constantes à gauche, ils n'ont pas été autrement modifiés.

Sommaire. La syntaxe et la sémantique de Lustre et de NLustre formalisé dans Vélus sont présentées dans la section 2. La présentation des passes indiquées en figure 1 commence dans la section 3 avec la dernière, la transcription, avant de revenir dans les sections 4 et 5 sur la désimbrication/distributivité et explicitation des initialisations. Cette organisation, qui ne reflète pas l'ordre de traitement d'un programme, facilite la description d'une propriété sémantique qui est nécessaire et pour le passage entre Lustre et NLustre et pour l'explicitation des initialisations.

2 Lustre dans Vélus

Dans cette partie, nous présentons la formalisation de Lustre dans Vélus. Nous introduisons d'abord les objets syntaxiques qui seront manipulés par l'algorithme de normalisation, puis le modèle sémantique du langage.

2.1 Syntaxes

La syntaxe des expressions et équations du langage Lustre est donnée en figure 5. Une équation associe une liste d'identifiants à une liste d'expressions. Les `fby`s et les instanciations de nœuds peuvent apparaître sans restriction. Par ailleurs, les opérateurs `when`, `merge`, `if-then-else` et `fby` peuvent être appliqués à des listes de sous-expressions. Une expression peut donc produire plusieurs flots. Cette fonctionnalité permet d'imbriquer librement les instances de nœud, qui peuvent rendre plusieurs flots, dans d'autres expressions.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \diamond\ e \\
& | & e \oplus e \\
& | & e^+ \texttt{ fby } e^+ \\
& | & e^+ \texttt{ when } x \\
& | & \texttt{merge } x\ e^+\ e^+ \\
& | & \texttt{if } e \texttt{ then } e^+ \texttt{ else } e^+ \\
& | & f\ \texttt{(}\ e^+\ \texttt{)} \\
\\
eq & ::= & x^+ \texttt{ = } e^+ \texttt{ ;}
\end{array}
$$

FIGURE 5 – Equations de Lustre

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \diamond\ e \\
& | & e \oplus e \\
& | & e \texttt{ when } x \\
\\
ce & ::= & e \\
& | & \texttt{merge } x\ ce\ ce \\
& | & \texttt{if } e \texttt{ then } ce \texttt{ else } ce \\
\\
eq & ::= & x \texttt{ = } ce \\
& | & x \texttt{ = } c \texttt{ fby } e \\
& | & x^+ \texttt{ = } f\ \texttt{(}\ e^+\ \texttt{)}
\end{array}
$$

FIGURE 6 – Equations de NLustre

$$
\begin{array}{rcl}
n & ::= & \texttt{node } f\ \texttt{(}\ d^+\ \texttt{) returns (}\ d^+\ \texttt{)} \\
& & \texttt{var } d^*\ \texttt{;} \\
& & \texttt{let } eq^*\ \texttt{tel}
\end{array}
\qquad
\begin{array}{rcl}
d & ::= & x^{ck}_{ty} \\
\\
G & ::= & n^+
\end{array}
$$

FIGURE 7 – Syntaxe des nœuds

La syntaxe des nœuds est décrite dans la figure 7. Un nœud peut recevoir plusieurs entrées (au moins une) et produire plusieurs sorties (au moins une). De plus, il peut déclarer des variables locales. Le nœud est ensuite constitué d'une liste d'équations.

Dans la syntaxe abstraite du langage, les expressions sont annotées par leur type et leur horloge statique. On notera plus loin dans cet article e^{ck} pour l'expression e annotée par l'horloge ck. Dans Lustre le système d'horloges est, à l'instar d'un système de types [11], un moyen de vérifier statiquement qu'un programme est synchrone et donc qu'il s'exécute dans un mémoire de taille bornée. Ainsi, les opérandes d'une opération arithmétique binaire doivent avoir la même horloge tandis que les horloges des membres d'un `merge` doivent être complémentaires, comme c'est le cas dans l'exemple de la figure 2, ligne 13. Une expression comme `x + (x when c)` n'est pas synchrone car la taille du tampon nécessaire au calcul dépend directement des valeurs prises par `c`, qui ne sont en général pas déterminées à la compilation.

Une horloge est définie par $ck ::= \bullet \mid ck\ \texttt{on}\ x$. Elle peut être l'horloge de base, s'activant à chaque instant, ou bien représenter un échantillonnage sur les valeurs d'une variable booléenne x.

Le système d'horloges contraint ces annotations. Par exemple, si l'expression `e` a pour horloge $\bullet$, alors l'expression `e when x` a pour horloge $\bullet$ `on x`.

Le langage normalisé présenté dans la figure 6 est plus contraint que le langage Lustre. Les expressions y sont hiérarchisées en expressions simples et expressions de contrôle. On y catégorise trois types d'équations, contenant soit une expression de contrôle, soit un `fby`, soit une instanciation. Par ailleurs, on n'y trouve plus de listes de sous-expressions. Cela signifie que dans le langage normalisé, toutes les expression sauf les instanciations représentent exactement un flot. Les nœuds normalisés sont, comme ceux de Lustre, composés d'une liste d'équations. Cette forme normalisée est moins commode pour un programmeur, mais constitue une étape essentielle de la compilation puisqu'elle permet d'isoler les `fby`s et instanciations de nœuds qui nécessitent un traitement particulier lors des transformations ultérieures vers du code impératif.

L'objectif de l'algorithme de normalisation présenté plus loin sera donc de transformer un programme de la première syntaxe vers la seconde, tout en préservant la sémantique que nous détaillons maintenant.

2.2 Modèle sémantique de Lustre

La sémantique de Lustre associe à chaque expression une liste de flots de valeurs. La formalisation présentée ici impose des relations entre flots infinis de valeurs représentés par le type co-inductif `stream value`, où `value` représente des valeurs présentes ou absentes. On notera les valeurs présentes $\langle v \rangle$ et les absences $\langle\rangle$. On note le jugement $G, H, bs \vdash e \Downarrow \boldsymbol{vs}$ pour « dans le programme G, sous l'historique H et sur le rythme bs, l'expression e produit les flots $\boldsymbol{vs}$ ». Dans ce jugement, le paramètre global G contient les nœuds du programme. L'historique H associe des flots aux noms apparaissant dans le nœud. L'historique correspond en fait aux chronogrammes donnés plus hauts. Enfin, le flot de base bs donne le rythme d'exécution du nœud. C'est un flot de booléens (`stream bool`), vrai si l'une au moins des entrées du nœud est présente. Il est défini par la fonction `base-of : list (stream value)` $\rightarrow$ `stream bool` qui calcule l'union des présences à chaque instant.

De manière générale, la typographie en **gras** dénote une liste. On note ainsi, par simplicité, $G, H, bs \vdash \boldsymbol{es} \Downarrow \boldsymbol{vs}$ pour le cas où $\boldsymbol{es}$ est une liste d'expressions. Dans ce cas, $\boldsymbol{vs}$ correspond à la concaténation des listes de flots issues de chaque expression. Dans les jugements ci-dessous $\boldsymbol{vs}$ est bien une liste de flots, qu'on parle d'une seule expression e ou de plusieurs expressions $\boldsymbol{es}$ car, comme on l'a vu plus haut, l'une des subtilités de notre mécanisation est qu'une expression peut correspondre à plusieurs flots.

La sémantique des expressions est donnée par des fonctions ou relations co-inductives, qui apparaissent ensuite dans des règles sémantiques définies par induction sur la syntaxe des expressions. Ces opérateurs sont appliqués point-à-point sur des listes de flots.

La règle de sémantique pour les constantes, Sconst, est donnée dans la figure 8. Elle associe l'expression c à une liste d'un seul élément. Ce flot est construit avec l'opérateur `const` qui permet d'échantillonner les constantes sur un flot de base. En l'occurrence, la règle de sémantique contraint ce flot à être celui du nœud. Ainsi les constantes sont toutes émises au rythme le plus rapide du nœud.

$$\textsc{Sconst}\ \frac{s \equiv \mathsf{const}\ bs\ c}{G, H, bs \vdash c \Downarrow [s]} \qquad \begin{aligned} \mathsf{const}\ (\mathrm{T} \cdot bs)\ c &= \langle c \rangle \cdot \mathsf{const}\ bs\ c \\ \mathsf{const}\ (\mathrm{F} \cdot bs)\ c &= \langle\rangle \cdot \mathsf{const}\ bs\ c \end{aligned}$$

Figure 8 – Sémantique des constantes

Une règle plus complexe est celle du `if-then-else`. Dans la figure 9, on présente la sémantique de l'opérateur co-inductif ite. Celle-ci s'interprète facilement : si la condition et les branches sont absentes, alors le résultat est absent. Sinon, si la condition et les branches sont présentes, on choisit la branche en fonction de la valeur de la condition, qui doit être T ou F. La valeur de ite est combinatoire. Elle ne dépend que des valeurs de ses sous-expressions au même instant. On constate aussi que ite est une fonction partielle. Elle n'est pas définie si par exemple, la valeur reçue en condition n'est pas booléene, ou si la condition est présente mais l'une des deux branches est absente. Le symbole $\doteq$ ne sert qu'à faciliter la lecture : il ne s'agit ici que d'un prédicat quaternaire. La notation double barre indique que les règles sont établies par co-induction sur les flots, et non par induction sur une structure syntaxique.

$$\frac{\mathsf{ite}\ cs\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{\mathsf{ite}\ (\langle\rangle \cdot cs)\ (\langle\rangle \cdot \boldsymbol{ts})\ (\langle\rangle \cdot \boldsymbol{fs}) \doteq \langle\rangle \cdot \boldsymbol{vs}} \qquad \frac{\mathsf{ite}\ cs\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{\mathsf{ite}\ v\ (\langle \mathrm{T}\rangle \cdot cs)\ (\langle t\rangle \cdot \boldsymbol{ts})\ (\langle f\rangle \cdot \boldsymbol{fs}) \doteq \langle t\rangle \cdot \boldsymbol{vs}}$$

$$\frac{\mathsf{ite}\ cs\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{\mathsf{ite}\ v\ (\langle \mathrm{F}\rangle \cdot cs)\ (\langle t\rangle \cdot \boldsymbol{ts})\ (\langle f\rangle \cdot \boldsymbol{fs}) \doteq \langle f\rangle \cdot \boldsymbol{vs}}$$

FIGURE 9 – Relation co-inductive ite

La règle sémantique du `if-then-else` est donnée dans la figure 10. Elle prend comme hypothèse l'existence de sémantiques pour les sous-expressions. La dernière hypothèse applique l'opérateur ite sur les valeurs issues de ces expressions. La règle du `if-then-else` est stricte, c'est à dire que les deux branches doivent avoir une sémantique, quelle que soit la valeur de la condition.

$$\textsc{Site}\ \frac{G, H, bs \vdash e \Downarrow [s] \qquad G, H, bs \vdash \boldsymbol{e_t} \Downarrow \boldsymbol{ts} \qquad G, H, bs \vdash \boldsymbol{e_f} \Downarrow \boldsymbol{fs} \qquad \mathsf{ite}\ s\ \boldsymbol{ts}\ \boldsymbol{fs} \doteq \boldsymbol{vs}}{G, H, bs \vdash \texttt{if}\ e\ \texttt{then}\ \boldsymbol{e_t}\ \texttt{else}\ \boldsymbol{e_f} \Downarrow \boldsymbol{vs}}$$

FIGURE 10 – Sémantique du `if-then-else`

On présentera les autres opérateurs co-inductifs au besoin, plus loin dans ce document. On ne détaillera pas les règles sémantiques, qui sont similaires à celle du `if-then-else`. Ces règles sont présentées dans leur intégralité dans [9, p. 37].

Les équations, dont la sémantique est donnée sous la forme $G, H, bs \vdash eq$, ne produisent pas de valeurs, contrairement aux expressions. Le rôle d'une équation est de contraindre l'historique H qui paramètre le jugement. On observe en effet que la règle SEQ donnée dans la figure 11 contraint l'historique à associer les noms à gauche de l'équation aux valeurs produites par les expressions à droite de l'équation.

La sémantique d'un nœud f dans un programme G, notée $G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$ associe les flots d'entrées $\boldsymbol{xs}$ aux flots de sorties $\boldsymbol{ys}$. Elle est définie par le jugement SNODE dans la figure 11. Elle exige l'existence d'un historique H contraint par chacune des équations du nœud. On voit que H n'apparaît pas dans la conclusion de la règle qui n'expose donc pas les variables internes.

$$\textsc{Seq}\ \frac{G, H, bs \vdash \boldsymbol{e} \Downarrow H(\boldsymbol{x})}{G, H, bs \vdash \boldsymbol{x} = \boldsymbol{e}}$$

$$\textsc{Snode}\ \frac{\texttt{node}(G, f) \doteq n \qquad H(n.\mathbf{in}) = \boldsymbol{xs} \qquad H(n.\mathbf{out}) = \boldsymbol{ys} \qquad \forall eq \in n.\mathbf{eqs},\ G, H, (\texttt{base-of}\ \boldsymbol{xs}) \vdash eq}{G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}}$$

FIGURE 11 – Sémantique de l'équation et du nœud

3 Transcription et correction du système d'horloges

Dans cette section nous présentons la transcription, dernière étape du processus de normalisation du langage. Cette passe présente peu d'intérêt algorithmique car il suffit de vérifier que le programme d'entrée est normalisé puis d'appliquer une transformation directe sur chaque constructeur. Néanmoins un point délicat est soulevé lors de la preuve de préservation sémantique que nous résolvons en prouvant la correction du système d'horloges. Cette propriété s'appliquant à tout programme Lustre, même non normalisé, nous l'utilisons également dans la preuve de correction d'une des passes en amont. Il est plus facile de motiver le besoin d'une telle propriété dans le cadre de la transcription, c'est pourquoi nous préférons la décrire au préalable.

En Lustre, les deux membres (gauche et droit) d'un `fby` sont des expressions qui produisent des flots et l'élaboration garantit que ces deux expressions possèdent la même annotation d'horloge. La sémantique du `fby` est formalisée en Coq par la relation co-inductive fby présentée dans la figure 12. Celle-ci utilise un opérateur fby1 qui retarde chaque valeur présente dans le flot de droite à la prochaine présence, en la conservant dans son premier argument. L'opérateur fby permet lui d'initialiser le délai avec la première valeur présente du flot de gauche. On note que les valeurs du flot xs sont ignorées mais pas leur statut (présent/absent), ce qui force la synchronisation du statut des valeurs successives de xs et ys. La forme de ces définitions est classique, cf., par exemple, Colaço et Pouzet [11, §3.2].

$$\frac{\mathsf{fby1}\ v\ xs\ ys \doteq vs}{\mathsf{fby1}\ v\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) \doteq \langle\rangle \cdot vs} \qquad \frac{\mathsf{fby1}\ y\ xs\ ys \doteq vs}{\mathsf{fby1}\ v\ (\langle x\rangle \cdot xs)\ (\langle y\rangle \cdot ys) \doteq \langle v\rangle \cdot vs}$$

$$\frac{\mathsf{fby}\ xs\ ys \doteq vs}{\mathsf{fby}\ (\langle\rangle \cdot xs)\ (\langle\rangle \cdot ys) \doteq \langle\rangle \cdot vs} \qquad \frac{\mathsf{fby1}\ y\ xs\ ys \doteq vs}{\mathsf{fby}\ (\langle x\rangle \cdot xs)\ (\langle y\rangle \cdot ys) \doteq \langle x\rangle \cdot vs}$$

FIGURE 12 – Relation co-inductive fby

En NLustre, le membre gauche du `fby` est une constante qui n'est pas interprétée comme un flot mais comme un paramètre d'initialisation statique, ce qui facilite ensuite la génération de code impératif en fournissant directement la valeur d'initialisation pour la cellule mémoire correspondante. L'interprétation du `fby`, simplifiée, est alors donnée par une fonction totale de type `value` × `stream value` → `stream value` présentée sous forme relationnelle en figure 13.

$$\frac{\mathsf{fby}_{\mathsf{NL}}\ v\ ys = vs}{\mathsf{fby}_{\mathsf{NL}}\ v\ (\langle\rangle \cdot ys) = \langle\rangle \cdot vs} \qquad \frac{\mathsf{fby}_{\mathsf{NL}}\ y\ ys = vs}{\mathsf{fby}_{\mathsf{NL}}\ v\ (\langle y\rangle \cdot ys) = \langle v\rangle \cdot vs}$$

FIGURE 13 – Définition relationnelle de $\mathsf{fby}_{\mathsf{NL}}$

Cette distinction sémantique se propage également dans les autres constructions. Considérons l'équation `x = true fby (not x)`, valide dans les deux langages. En Lustre, l'expression `true` est une constante cadencée sur l'horloge de base et produit le flot $\langle\mathrm{T}\rangle \cdot \langle\mathrm{T}\rangle \cdot \langle\mathrm{T}\rangle \cdots$. En considérant l'interprétation du fby, on déduit que le seul flot possible pour la variable x est $\langle\mathrm{T}\rangle \cdot \langle\mathrm{F}\rangle \cdot \langle\mathrm{T}\rangle \cdot \langle\mathrm{F}\rangle \cdots$. En NLustre en revanche, la valeur `true` n'est que le paramètre initial de $\mathsf{fby}_{\mathsf{NL}}$ et n'impose pas de rythme au flot de x. Par conséquent, tout flot de la forme $(\langle\rangle^* \cdot \langle\mathrm{T}\rangle \cdot \langle\rangle^* \cdot \langle\mathrm{F}\rangle)^\omega$ est une interprétation valide de x.

Cet aspect non déterministe n'est pas souhaitable dans le cadre de la programmation de systèmes critiques car il laisse au compilateur le choix du comportement à adopter. Il est donc nécessaire d'imposer une contrainte supplémentaire sur la sémantique NLustre. La solution adoptée dans Vélus est de faire correspondre le flot d'une expression avec l'annotation d'horloge de celle-ci, déterminée lors de l'élaboration. On introduit donc une notion de sémantique pour les horloges, interprétées par des flots booléens. L'horloge de base $\bullet$ doit s'évaluer en le flot de base de l'environnement et une horloge échantillonnée `ck on x` ne doit s'évaluer en T que lorsque la sous-horloge `ck` est vraie, que la valeur de `x` est présente et égale à T ; elle s'évalue en F si `ck` est vraie et la condition fausse ou si `ck` est F et `x` absente ; elle est indéfinie sinon. On surcharge la notation $H, bs \vdash ck \Downarrow b$ pour dire que l'horloge ck produit le flot booléen b.

L'intégration de la sémantique des horloges se fait au niveau des expression en NLustre. Ici tl désigne l'un des deux destructeurs de flot, qu'on étend aux environnements par composition.

$$\frac{H, bs \vdash e \Downarrow \langle v\rangle \cdot s \qquad H, bs \vdash ck \Downarrow \mathrm{T} \cdot b \qquad \mathsf{tl}\ H, \mathsf{tl}\ bs \vdash e^{ck} \Downarrow s}{H, bs \vdash e^{ck} \Downarrow \langle v\rangle \cdot s}$$

$$\frac{H, bs \vdash e \Downarrow \langle\rangle \cdot s \qquad H, bs \vdash ck \Downarrow \mathrm{F} \cdot b \qquad \mathsf{tl}\ H, \mathsf{tl}\ bs \vdash e^{ck} \Downarrow s}{H, bs \vdash e^{ck} \Downarrow \langle\rangle \cdot s}$$

FIGURE 14 – Règles de l'alignement

Les règles données dans la figure 14 décrivent le principe d'*alignement* du flot d'une expression avec son horloge : à chaque instant, la valeur d'une expression est présente si et seulement si son horloge est vraie. Dans la suite, on notera $H, bs \vdash e^{ck} \Downarrow s$ pour signifier que l'expression e produit un flot s bien aligné avec l'horloge ck. En Lustre en revanche, les annotations d'horloges ne sont pas interprétées par le modèle sémantique, il faut démontrer que l'alignement en est une conséquence.

3.1 Correction du système d'horloges

Le prédicat d'alignement des horloges et des valeurs constitue le principal obstacle dans la preuve de préservation sémantique de la transcription. En plus d'imposer une sémantique déterministe aux programmes NLustre, l'alignement apporte une information de présence ou d'absence des valeurs à chaque instant qui s'avère importante lors de la preuve de correction du code impératif généré [7, §3.4]. Le résultat central de la transcription consiste en une preuve de cette propriété, pour tout nœud Lustre bien cadencé, ordonnançable et possédant une sémantique.

Théorème 1. *Pour tout environnement H, pour tous flots $s_1, \ldots, s_n$, $s'_1, \ldots, s'_m$, pour tout nœud Lustre de signature* `node f` $(x_1^{ck_1}, \ldots, x_n^{ck_n})$ `returns` $(y_1^{ck'_1}, \ldots, y_m^{ck'_m})$*, si*

- $bs \equiv$ `base-of`$(s_1, \ldots, s_n)$
- $H, bs \vdash x_1^{ck_1} \Downarrow s_1, \ldots, x_n^{ck_n} \Downarrow s_n$
- `f`$(s_1, \ldots, s_n) \Downarrow s'_1, \ldots, s'_m$
- $H \vdash y_1 \Downarrow s'_1, \ldots, y_m \Downarrow s'_m$

alors

$H, bs \vdash y_1^{ck'_1} \Downarrow s'_1, \ldots, y_m^{ck'_m} \Downarrow s'_m$.

On note que les flots des entrées du nœud sont supposés alignés avec leurs horloges. Par un raisonnement inductif sur l'ensemble du programme, on peut se débarrasser de cette contrainte pour tous les nœuds à l'exception du nœud principal, qui reçoit ses entrées de l'extérieur. L'ajout de cette nouvelle hypothèse au théorème principal du compilateur est justifié et n'introduit pas de faiblesse dans la preuve de correction. Il n'est pas possible de garantir une sémantique pour une exécution dans laquelle les entrées du programme ne sont pas bien formées.

On peut aussi remarquer que ce théorème s'applique à tout nœud Lustre ordonnançable (hypothèse discutée en section 3.2) possédant une sémantique, et pas uniquement aux nœuds sous forme normalisée. Ce théorème peut donc être considéré comme une propriété des programmes Lustre, indépendamment de l'étape de transcription, dénotant la correction du système d'horloges : si un nœud est bien cadencé alors les flots des variables sont alignés avec leurs horloges.

3.2 Induction et ordonnancement

La preuve du théorème 1 s'effectue à plusieurs niveaux, équations puis expressions. L'alignement du flot d'une expression avec son horloge se démontre par induction structurelle, avec les informations du système d'horloges. Par définition, les constantes sont cadencées sur l'horloge • et sont donc alignées avec le flot de base du nœud. L'autre cas de base, la variable, est plus délicat : on ne dispose de l'alignement que s'il s'agit d'une variable d'entrée du nœud ou qu'elle a déjà été traitée dans le raisonnement. Il est alors nécessaire de choisir un ordre sur les variables du programme.

L'existence d'un tel ordre est une hypothèse raisonnable faite sur les équations Lustre et correspond à l'absence de définitions cycliques [13, §III.A]. Chaque cycle de dépendances dans un nœud doit contenir au moins un opérateur `fby`, ce qui permet de briser la circularité. On dit qu'une variable est *libre à gauche* d'une expression si elle n'apparaît pas à droite d'un `fby`. Dans ce contexte, une liste d'équations $eq_1, \ldots, eq_n$ est ordonnancée si toute variable libre à gauche d'une équation eq_i est définie soit comme une entrée du nœud, soit par une équation $eq_{j<i}$. Par exemple `x = true fby not x; y = z when x` est ordonnancée si `z` est une entrée du nœud, mais `y = 2 * y + z` n'est pas ordonnançable.

En raisonnant par récurrence sur les équations ordonnancées, on peut maintenir l'invariant suivant sur les variables libres à gauche : si la variable est déclarée d'horloge ck et produit le flot de valeurs s dans l'environnement H, alors ck est alignée avec s. En effet, l'équation eq_1 ne peut contenir que les variables d'entrées pour lesquelles on a l'invariant par hypothèse ; puis par ordonnancement, une variable libre à gauche dans une équation $eq_{i>1}$ a été définie par une précédente équation $eq_{j<i}$, ou bien c'est une entrée, et l'on peut utiliser l'hypothèse de récurrence pour obtenir l'alignement.

Il est important de noter que l'absence de dépendance circulaire entre les variables d'un programme n'implique pas l'ordonnançabilité de ses équations. Par exemple `x,y = (1,x)` est une équation valide en Lustre qui ne nous permet pas d'exploiter le raisonnement décrit précédemment. Cette limitation n'est pas un obstacle à la preuve de correction de la transcription car de telles équations ne sont pas normalisées. Cependant on souhaiterait l'éliminer afin d'obtenir un résultat plus général. Une piste envisageable serait de gérer plus finement les dépendances entre les variables au sein d'une équation. Une autre serait d'appliquer une passe de désimbrication des équations avant de commencer le raisonnement inductif du paragraphe précédent.

Contrôle de dépendances. L'ordonnançabilité d'un programme est établie par une analyse de graphe. À chaque équation du nœud est associé un sommet, et des arcs sont ajoutés pour chaque équation utilisant une variable définie par une autre. Vérifier l'ordonnançabilité du programme revient alors à vérifier l'absence de cycle dans le graphe.

4 Désimbrication et distributivité

On revient maintenant sur les deux passes qui précèdent la transcription et qui mettent un programme Lustre sous forme normalisée. Il faut d'abord désimbriquer les instanciations et les `fby`s, en les plaçant dans leurs propres équations. On profite aussi de cette passe pour distribuer les opérateurs `fby`, `when`, `merge` et `if-then-else` sur leurs listes d'opérandes. Cela permet de produire un code où chaque expression, sauf une instanciation, représente exactement un flot.

4.1 Schémas de compilation

On note $\lfloor e \rfloor = ([e'_1, ..., e'_m], \boldsymbol{eqs})$ la normalisation de l'expression e. Celle-ci produit une liste d'expressions, à cause de la distribution des opérateurs. Par exemple, dans l'exemple de la figure 2, la sous-expression `(edge, n) when ck` se normalise en `(edge when ck, n when ck)`. La normalisation produit de plus des équations contenant les sous-expressions qui ont été désimbriquées. Pour simplifier, on notera $([e'_1, \ldots, e'_m], \boldsymbol{eqs'}) \leftarrow \lfloor e_1, \ldots, e_n \rfloor$ la normalisation d'une liste d'expressions $e_1, \ldots, e_n$, où $e'_1, \ldots, e'_m$ est la concaténation des listes d'expressions produites, et $\boldsymbol{eqs'}$ l'union des listes d'équations.

On présente dans la figure 15 les schémas de normalisation pour quelques cas. Les cas de la constante et de la variable ne changent pas l'expression et n'introduisent pas de nouvelle équation. Le cas des opérateurs binaires est défini par récursion, avec concaténation des équations introduites pour chaque sous-expression. Par contrainte de typage, on sait qu'une seule expression sera renvoyée par chaque appel récursif. Dans le cas du `when`, on effectue les appels récursifs, puis on distribue le `when` autour de chaque résultat de l'appel. Dans le cas du `fby`, il faut désimbriquer les expressions après distribution. On génère donc une nouvelle variable par expression et on renvoie comme expression la liste de ces variables. Pour l'instanciation, on ne distribue pas, mais on doit désimbriquer l'expression produite après les appels récursifs. On génère alors le nombre de variables correspondant au nombre de flots renvoyés par le nœud

instancié. Les cas du `merge` et du `if-then-else` ne sont pas détaillés car similaires au cas du `fby`. Cela dit, on évite de désimbriquer les expressions de contrôle directement imbriquées, comme spécifié dans la grammaire de la figure 6.

$$
\begin{aligned}
\lfloor \mathtt{c} \rfloor &= ([\mathtt{c}], [\,]) \\
\lfloor \mathtt{x} \rfloor &= ([\mathtt{x}], [\,]) \\
\lfloor e_1 \oplus e_2 \rfloor &= ([e'_1], \boldsymbol{eqs}'_1) \leftarrow \lfloor e_1 \rfloor \\
&\quad\ ([e'_2], \boldsymbol{eqs}'_2) \leftarrow \lfloor e_2 \rfloor \\
&\quad\ ([e'_1 \oplus e'_2], \boldsymbol{eqs}'_1 \cup \boldsymbol{eqs}'_2) \\
\lfloor (e_1, \ldots, e_n)\ \mathtt{when}\ b \rfloor &= ([e'_1, \ldots, e'_m], \boldsymbol{eqs}') \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([e'_1\ \mathtt{when}\ b, \ldots, e'_m\ \mathtt{when}\ b], \boldsymbol{eqs}') \\
\lfloor (e_1, \ldots, e_n)\ \mathtt{fby}\ (f_1, \ldots, f_m) \rfloor &= ([e'_1, \ldots, e'_k], \boldsymbol{eqs}'_1) \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([f'_1, \ldots, f'_k], \boldsymbol{eqs}'_2) \leftarrow \lfloor f_1, \ldots, f_m \rfloor \\
&\quad\ ([x_1, \ldots, x_k], [x_1 = e'_1\ \mathtt{fby}\ f'_1, \ldots, x_k = e'_k\ \mathtt{fby}\ f'_k] \cup \boldsymbol{eqs}'_1 \cup \boldsymbol{eqs}'_2) \\
\lfloor f(e_1, \ldots, e_n) \rfloor &= ([e'_1, \ldots, e'_m], \boldsymbol{eqs}') \leftarrow \lfloor e_1, \ldots, e_n \rfloor \\
&\quad\ ([x_1, \ldots, x_k], [(x_1, \ldots x_k) = f(e'_1, \ldots, e'_m)] \cup \boldsymbol{eqs}')
\end{aligned}
$$

FIGURE 15 – Schémas de normalisation des expressions

On ajoute à ces cas généraux des traitements particuliers, permettant de minimiser les transformations du programme. Si l'on rencontre une instanciation directement à droite d'une équation, on n'a pas besoin d'introduire de nouvelle équation, puisque l'équation est déjà sous forme normalisée. Il en va de même pour les `fby`s directement à droite d'une équation. Eviter les transformations inutiles permet de produire des programmes plus courts et lisibles. On a aussi pu prouver formellement que la fonction est idempotente : pour tout programme G, $\lfloor \lfloor G \rfloor \rfloor = \lfloor G \rfloor$.

4.2 Génération de variables

Comme on l'a vu plus haut, désimbriquer un `fby` ou une instanciation nécessite l'introduction de nouvelles variables locales dans le nœud. Dans le contexte de programmation fonctionnelle pure de Coq, on implémente la génération de variables par une monade d'état.

L'état manipulé est de type `fresh_st = (ident × list (ident × (ty × ck)))`. Celui-ci permet de conserver le prochain identifiant à générer, ainsi qu'une liste des variables générées, chacune accompagnée de son annotation de type et d'horloge. Cette liste contient les nouvelles variables locales introduites dans le nœud.

On exploite la représentation des identifiants par des entiers positifs. Chaque appel à la fonction `fresh_ident : (ty × ck) → fresh_st → (ident × fresh_st)` renvoie un nouvel identifiant, ainsi qu'un nouvel état avec l'identifiant à générer incrémenté, et l'identifiant juste généré sauvegardé dans la liste. Ainsi tous les identifiants générés par la monade sont plus grands que l'identifiant utilisé pour initialiser la monade. Il est donc important de bien choisir cet identifiant d'initialisation. La solution la plus simple est de prendre un identifiant plus grand que tout ceux apparaissant dans le nœud. Ainsi l'on sait que tous les identifiants générés sont plus grands et donc différents.

En plus de résoudre le problème de la génération de variables dans un contexte fonctionnel pur, cette manipulation explicite d'un état permet de raisonner sur la sémantique des équations.

4.3 Raisonnement sur la sémantique

On décrit maintenant la preuve de correction de cette passe de normalisation. Il s'agit de montrer que la sémantique d'un programme est préservée par la transformation.

Théorème 2 (Préservation de la sémantique)**.** *La passe de désimbrication et distributivité préserve la sémantique des programmes.*

$$\forall G\ f\ \boldsymbol{xs}\ \boldsymbol{ys},\quad G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys} \quad \Rightarrow \quad \lfloor G \rfloor \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$$

Intéressons-nous au cœur de la fonction de normalisation : le traitement des expressions. Il faut prouver que la normalisation d'une expression e ayant une sémantique $\boldsymbol{vs}$ produit des expressions $\mathbf{es}'$ ayant la même sémantique. On veut également que les équations $\boldsymbol{eqs}'$ produites aient une sémantique. Le noyau de cette propriété d'induction est donné ci-dessous.

$$\begin{aligned}&\lfloor e \rfloor = (\boldsymbol{es}', \boldsymbol{eqs}') \Rightarrow\\ &G, H, bs \vdash e \Downarrow \boldsymbol{vs} \Rightarrow\\ &\exists H',\ (H \sqsubseteq H' \ \wedge\ G, H', bs \vdash \boldsymbol{es}' \Downarrow \boldsymbol{vs} \ \wedge\ G, H', bs \vdash \boldsymbol{eqs}')\end{aligned}$$

Lors d'une opération de désimbrication, on va ajouter une nouvelle variable au nœud. Pour donner une sémantique aux nouvelles équations et expressions, il faut donc étendre l'historique H avec cette nouvelle variable. La sémantique des $\boldsymbol{es}'$ et $\boldsymbol{eqs}'$ sera donc donnée en fonction d'un nouvel historique H'. On veut que H' « raffine » H, noté $H \sqsubseteq H'$. On reviendra sur cette notion de raffinement plus loin.

La preuve est établie par induction sur la syntaxe des expressions. Les cas intéressants sont ceux du `fby` et de l'instanciation, où la désimbrication a lieu. Quand on désimbrique la sous-expression e_x de l'expression e, on introduit une nouvelle équation $x = e_x$ et on substitue x à e_x dans e. On note le résultat de cette substitution $e' = e[e_x \mapsto x]$. On veut prouver que si $G, H, bs \vdash e \Downarrow \boldsymbol{vs}$, alors il existe un nouvel historique H' étendu avec la nouvelle variable x, tel que $G, H', bs \vdash e' \Downarrow \boldsymbol{vs}$. De plus, H' doit satisfaire la sémantique imposée par la nouvelle équation produite par la désimbrication, $G, H', bs \vdash x = e_x$. Pour répondre à ces contraintes nous construisons explicitement H', une extension de H dans laquelle on associe à la variable x le flot produit par e_x. En notant ce flot v_x, on a $G, H, bs \vdash e_x \Downarrow v_x$.

Comme x est un nom frais issu de la monade de génération de noms, on peut prouver qu'il n'apparaît pas dans le domaine de H. On sait alors de H' que

— $H'(x) = v_x$ et

— $\forall y\ v, H(y) = v \rightarrow H'(y) = v \quad (H'$ raffine H, noté $H \sqsubseteq H')$.

C'est suffisant pour prouver que H' donne une sémantique à l'expression et à l'équation produite. En effet, il est facile de prouver que si H' raffine H, alors H' donne la même sémantique que H à toute expression, en particulier e_x et e. De plus, on sait que $H'(x) = v_x$, qui est bien le flot associé à e_x.

Des travaux précédents [1] avaient mis la notion de substitution au cœur du raisonnement. Au contraire, l'objet central de notre induction est l'historique H donné comme un existentiel.

Après avoir normalisé toutes les équations, on obtient un H' raffinant H et satisfaisant les contraintes des équations produites. Ce H' peut servir comme existentiel donnant la sémantique du nœud, en suivant la règle SNODE donnée en figure 11. Comme H' raffine H, on sait que les sorties du nœud normalisé sont bien les mêmes qu'avant normalisation.

On donne en figure 16 la forme désimbriquée du nœud `count_down` présenté en figure 2. On illustre en figure 17 comment la forme désimbriquée du programme se comporte. Le nouvel historique associe un flot de valeurs à la variable `$114` et toujours le même flot à `cpt`.

```
node count_down(res:bool; n:int)
returns (cpt:int)
let $114 = n fby (cpt - 1);
    cpt = if res then n else $114;
tel
```

FIGURE 16 – `count_down` désimbriqué

res	F	T	F	F	F	F	T	F	···
n	3	3	3	3	3	3	3	3	···
$114	3	2	2	1	0	-1	-2	2	···
cpt	3	3	2	1	0	-1	3	2	···

FIGURE 17 – Exécution de `count_down`

Traiter la distributivité ajoute une petite difficulté. L'une des propriétés de la fonction de normalisation est que si $\lfloor e \rfloor = (\boldsymbol{es'}, \boldsymbol{eqs'})$, alors la longueur de $\boldsymbol{es'}$ correspond au nombre de flots issus de e. Dans l'implémentation des schémas donnés dans la figure 15, les annotations de typage sont utilisées pour générer les $\boldsymbol{es'}$. Il faut donc savoir que celles-ci sont correctes, c'est-à-dire, que e est bien typée. C'est une hypothèse peu coûteuse, puisque les programmes à normaliser ont été élaborés au préalable. L'élaboration étant prouvée correcte, on sait que ces programmes sont effectivement bien typés.

5 Explicitation des initialisations

La passe de normalisation précédente met un programme sous forme désimbriquée et distribuée. Nous souhaitons aussi que les `fby`s ne soient initialisés que par des constantes et non par des expressions. C'est nécessaire pour compiler les `fby`s dans la suite de la chaine.

5.1 Schéma de compilation

Nous définissons un sous-ensemble des expressions, les « constantes lentes », composées de constantes pouvant être entourées d'opérateurs `when`. Ces `when`s permettent de maintenir le bon cadencement des équations contenant des `fby`s. Ils sont ensuite éliminés lors de la transcription.

Notons c^{ck} la constante c entourée de `when`s de manière à être cadencée sur l'horloge ck. Par exemple, $\texttt{true}^{\bullet\ \texttt{on b on c}}$ représente `true when b when c`. Notons, par ailleurs, $\texttt{def}_{ty}$ une constante « par défaut » du type ty. Notre langage manipulant des types booléens, entiers et réels, il est toujours possible de choisir une constante du type approprié (false, 0, 0.0).

On veut transformer une équation `x = (e0 fby e)`^{ck} où `e0` n'est pas déjà une constante lente. L'horloge ck de l'expression est importante, puisqu'elle permettra de générer les constantes lentes. La transformation donne alors les trois équations suivantes.

$$\lfloor \texttt{x = (e0 fby e)}^{ck} \rfloor_{fby} = \begin{cases} \texttt{xinit = true}^{ck}\ \texttt{fby false}^{ck}\texttt{;} \\ \texttt{px = def}_{ty}^{ck}\ \texttt{fby e;} \\ \texttt{x = if xinit then e0 else px;} \end{cases}$$

Le rôle de la première équation est de choisir le premier instant de présence pour le flot d'horloge de ck, l'horloge de l'équation à transformer. La deuxième équation retarde le flot de `e`. La troisième utilise le flot d'initialisation `xinit` pour choisir entre l'expression d'initialisation `e0` et le flot retardé de `e`. Ces trois équations sont effectivement normalisées, car seules des constantes lentes apparaissent à gauche des `fby`s.

Un exemple de cette transformation est celle de `count_down` désimbriquée, voir la figure 16, en forme normalisée, voir la figure 3.

Pour générer les nouvelles variables `xinit` et `px`, nous réutilisons la monade de génération de noms présentée dans la section 4.2.

Remarquons que pour deux équations contenant des `fby`s cadencées sur la même horloge, le schéma ci-dessus produirait deux équations d'initialisation identiques. Par exemple, normaliser l'équation `(x, y) = (x0, y0) fby (y, x)` causerait cette redondance. Cela est à éviter, en particulier parce que tout `fby` demandera un espace mémoire dans le programme impératif issu de la compilation. On utilise donc l'état de la monade de génération de nom pour mémoïser et réutiliser les équations d'initialisation déjà générées. Cette optimisation complexifie les preuves de correction.

5.2 Raisonnement sur la sémantique

Comme pour la passe précédente, nous montrons que la sémantique d'un programme est préservée par cette passe de normalisation.

Théorème 3 (Préservation de la sémantique)**.** *L'explicitation des initialisations préserve la sémantique des programmes.*

$$\forall G\ f\ \boldsymbol{xs}\ \boldsymbol{ys},\quad G \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys} \quad \Rightarrow \quad \lfloor G \rfloor_{fby} \vdash f(\boldsymbol{xs}) \Downarrow \boldsymbol{ys}$$

Cette fois, le cœur du problème est de prouver que la sémantique de l'équation `x = e0 fby e` est préservée par les nouvelles équations introduites. Comme dans la preuve du théorème 2, il faut étendre l'historique donnant la sémantique des expressions pour inclure les nouveaux noms. Cette transformation ajoute aussi une nouvelle difficulté. À partir de la sémantique d'un `fby`, il faut donner la sémantique pour une construction incluant des constantes lentes, deux `fby`s, et un `if-then-else`.

Constantes lentes $\mathtt{c}^{ck}$. Donner la sémantique d'une constante `c` est trivial. Comme on peut voir dans la figure 8, il s'agit du flot `const bs c`. Il est plus difficile de donner une sémantique à $\mathtt{c}^{ck}$. Prenons l'exemple de $\mathtt{c} \bullet {}^{\mathtt{on\ b1\ on\ b2}} = \mathtt{c\ when\ b1\ when\ b2}$. En se rappelant la règle du `when`, on voit qu'il est nécessaire que le flot issu de `c when b1` soit aligné avec le flot de `b1`. Par induction sur ck, on prouve que $\mathtt{c}^{ck}$ a une sémantique si l'horloge ck en a une.

Pour obtenir cette sémantique, on utilise le résultat d'alignement développé dans la section 3.1 pour prouver l'existence d'un flot d'horloge b pour l'annotation ck. Le flot de $\mathtt{c}^{ck}$ est alors $\mathsf{const}\ b\ c$, et est aligné avec les flots issus de `e0` et `e`. Utiliser cette preuve demande de nouvelles hypothèses sur notre programme. Il faut savoir qu'il est bien typé, bien cadencé et ordonnaçable.

Équations de délai. On veut maintenant donner les sémantiques de $\mathtt{true}^{ck}$ `fby` $\mathtt{false}^{ck}$ et $\mathtt{def}_{ty}^{ck}$ `fby e`. Les flots correspondant à ces expressions peuvent être explicitement construits par la fonction totale $\mathsf{fby}_{\mathsf{NL}}$. La figure 18 en présente une définition fonctionnelle qui est équivalente à la définition par prédicats co-inductifs de la figure 13.

La fonction $\mathsf{fby}_{\mathsf{NL}}$ correspond à l'opérateur fby de Lustre donné dans la figure 12. En effet, si $y0$ est un flot constant de v aligné avec y, on a `fby y0 y` $\doteq$ $\mathsf{fby}_{\mathsf{NL}}$ `v y`. En particulier, les flots $\mathtt{v}_{init}$ `=` $\mathsf{fby}_{\mathsf{NL}}$ `true (const bs false)` et $\mathtt{v}_{px}$ `=` $\mathsf{fby}_{\mathsf{NL}}\ \mathtt{def}_{ty}$ `y` (avec `y` le flot associé à `e`) sont bien les flots associés à ces deux nouvelles expressions.

Sélection de valeur avec `if-then-else`. Il reste à déduire de la sémantique de l'équation initiale `x = e0 fby e` une sémantique pour `x = if xinit then e0 else px`. Inverser l'hypothèse de sémantique établit `fby y0 y` $\doteq$ `z`, où `y0`, `y` et `z` sont les flots associés à `e0`, `e` et `x`. On peut alors prouver la relation $\mathsf{ite}\ (\mathsf{fby}_{\mathsf{NL}}\ \mathtt{true\ (const\ bs\ false)})\ \mathtt{y0}\ (\mathsf{fby}_{\mathsf{NL}}\ \mathtt{def}_{ty}\ \mathtt{y})$ avec

$$\mathsf{fby_{NL}}\ v\ (\langle\rangle \cdot xs) = \langle\rangle \cdot\ \mathsf{fby_{NL}}\ v\ xs$$
$$\mathsf{fby_{NL}}\ v\ (\langle x\rangle \cdot xs) = \langle v\rangle \cdot\ \mathsf{fby_{NL}}\ x\ xs$$

FIGURE 18 – Définition fonctionnelle de $\mathsf{fby_{NL}}$

`res`	F	T	F	F	F	F	T	F	···
`n`	3	3	3	3	3	3	3	3	···
$214	T	F	F	F	F	F	F	F	···
$215	0	2	2	1	0	-1	-2	2	···
$114	3	2	2	1	0	-1	-2	2	···
`cpt`	3	3	2	1	0	-1	3	2	···

FIGURE 19 – Exécution de `count_down` normalisé

les règles co-inductives du `ite` de la figure 9. On construit cette preuve par co-induction, en suivant les règles de fby de la figure 12.

Dans la figure 19, nous présentons comme exemple de ces nouveaux flots la trace de la forme normalisée de `count_down`.Le flot de $114, qui est maintenant construit à partir des flots de $214 et $215, est inchangé.

5.3 Préservation de l'ordonnançabilité

Comme expliqué dans la section précédente, la preuve de préservation de la sémantique lors de l'explicitation des initialisations utilise la preuve de correction du système d'horloges. Celle-ci utilise une hypothèse d'ordonnançabilité sur le programme. Il faut donc savoir que le programme G traité par cette passe est ordonnançable. Nous obtenons cette propriété en exécutant le contrôle de dépendances affiché dans la figure 1 et décrit dans la section 3.2.

Cependant, on veut également savoir que le programme $\lfloor G \rfloor_{fby}$ produit par l'explicitation des initialisations de G est ordonnançable car cette propriété est requise dans la passe suivante, la transcription.

Il faut donc prouver que la passe d'explicitation des initialisations préserve l'ordonnançabilité. L'optimisation introduite dans la section 5.1 rend la preuve plus complexe que ce à quoi on pourrait s'attendre. En effet, les variables correspondant aux initialisations introduites peuvent être utilisées par plusieurs expressions. Pour rétablir un ordre d'exécution sur le nouveau programme, il faut montrer que l'on peut placer ces équations avant toutes celles qui en dépendent. De plus, les équations d'initialisation dépendent elles-mêmes d'autres variables à cause de `when`s introduits.

Pour prouver l'existence d'un tel ordre, il faut manipuler un invariant spécifiant qu'il existe un ordonnancement pour les équations du nœud dans lequel toutes les équations d'initialisation apparaissent avant les équations de la forme `x = e0 fby e` cadencées sur la même horloge. Mécaniser et manipuler cet invariant en Coq introduit quelques technicités qu'on ne détaillera pas ici. Cependant, il permet effectivement de prouver la préservation de l'ordonnançabilité.

6 Conclusion

Nous avons présenté l'implémentation et les preuves de correction dans un assistant de preuve d'une passe de normalisation pour le langage Lustre. Cette passe a été intégrée dans notre prototype. Le résultat est une chaîne de compilation pour le langage de la figure 5 et un théorème qui lie la sémantique d'un programme source, c'est-à-dire, la relation entre les flots des entrées et des sorties, à celle du code généré. En fait, le prototype prend également en compte la réinitialisation modulaire [6], mais son traitement ne pose pas de pro-

blème particulier et nous ne présentons pas les détails ici. De même, nous traitons l'opérateur d'initialisation (−>) en adaptant directement l'approche décrite dans la section 5, puisque `e0 -> e` ≡ `if (true fby false) then e0 else e`. Cet opérateur est normalement utilisé avec le délai non initialisé, « `pre` », qui ne fait pas encore partie de notre langage. En plus des questions posées pour le `fby` de NLustre dans la section 3, l'inclusion de `pre` demanderait de considérer la sous-spécification de sa valeur initiale.

Dans la plupart des cas, l'ajout des listes au langage source ne complique pas vraiment les preuves dans Coq. On utilise quelques lemmes sur `Forall`n pour éviter des inductions imbriquées et pour se concentrer sur un unique flot arbitraire. Dans certains cas, l'absence d'un lien biunivoque entre syntaxe (expressions) et sémantique (flots correspondants) rendent les preuves encore plus fastidieuses. Des problèmes plus profonds sont posés par la possibilité qu'un nœud prenne des entrées et produise des sorties sur des cadences moins rapides que son horloge de base. Il a fallu étendre la formalisation d'horloges [8] pour incorporer des horloges « anonymes » qui, dans un programme non normalisé, représentent les flots booléens provenant d'instances de nœuds imbriquées. Après quelques itérations, nous avons fini par trouver une formalisation assez simple. Mais, à part exiger que les entrées et les sorties partagent la même cadence, on ne peut pas éviter de passer entre historiques locaux augmentés avec flots anonymes et historiques internes à une instance de nœud, tout en substituant les variables passées en argument aux paramètres formels. Et cela rend le raisonnement beaucoup plus difficile.

Travaux connexes. Nous avons implémenté une passe de normalisation directement dans Coq. Une approche différente a été poursuivie par C. Auger et ses collègues [1, 2]. Ils découpent leur algorithme en deux parties. La première introduit de nouvelles équations et la seconde élimine les tuples [1, §7.1]. Ils n'ont pas besoin, comme nous, d'expliciter des initialisations, car dans leur langage source [1, §5.1] les `fby`s ont toujours une constante à gauche. La première passe appelle une fonction externe qui calcule les équations à introduire et la substitution à appliquer. Le résultat est ensuite validé par une fonction Coq facile à implémenter [1, §7.2.1].

Boulmé et Hamon [4] montrent comment spécifier un plongement peu profond dans Coq d'un language plus riche que Lustre. Dans leur approche, la propriété d'alignement est garantie par un encodage direct dans le système de types de Coq. Nous ne savons pas exploiter cette approche dans le plongement profond nécessaire pour spécifier un compilateur.

Travail futur. Alors que l'alignement des flots avec leurs horloges doit être une propriété de tout programme Lustre qui vérifie les conditions de typage, la preuve décrite dans la section 3.1 ne la démontre que pour les programmes dans lesquels il est possible d'ordonner les équations d'un nœud d'après leurs interdépendances. De même, le contrôle de dépendances est général, il vérifie l'absence de cycles instantanés entre variables, mais son témoin est traduit dans un prédicat sur la possibilité d'ordonner strictement les équations. C'est une des raisons pour lesquelles ce contrôle est invoqué après la passe de désimbrication et distributivité (cela évite aussi de démontrer que la propriété est préservée par cette passe). Nous voudrions donc développer un principe d'induction basé sur une notion de causalité plus abstraite qui permettrait de raisonner sur les propriétés générales des programmes acceptés par notre compilateur. La préservation d'une telle notion de causalité devrait être plus facile à établir que pour celle de la section 5.3.

Remerciements. Ce travail a été soutenu par Bpifrance « Programme d'Investissements d'Avenir » dans le projet ES3CAP et le projet ANR JCJC FidelR 19-CE25-0014-01 « Fidelity in Reactive Systems Design and Compilation ».

Bibliographie

[1] Cédric Auger. *Compilation certifiée de SCADE/LUSTRE*. PhD thesis, Université Paris Sud 11, Orsay, France, April 2013.

[2] Cédric Auger, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. A formalization and proof of a modular Lustre code generator. In preparation, 2014.

[3] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the 9th ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, pages 121–130, Tucson, AZ, USA, June 2008. ACM Press.

[4] Sylvain Boulmé and Grégoire Hamon. Certifying synchrony for free. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001)*, volume 2250 of *Lecture Notes in Elecrical Engineering*, pages 495–506, Havana, Cuba, December 2001. Springer.

[5] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for Lustre. In *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 586–601, Barcelona, Spain, June 2017. ACM Press.

[6] Timothy Bourke, Lélio Brun, and Marc Pouzet. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proceedings of the of the ACM on Programming Languages*, 4(POPL) :1–29, January 2020.

[7] Timothy Bourke, Pierre-Évariste Dagand, Marc Pouzet, and Lionel Rieg. Vérification de la génération modulaire du code impératif pour Lustre. In Julien Signoles and Sylvie Boldo, editors, *28^{ièmes} Journées Francophones des Langages Applicatifs (JFLA 2017)*, pages 165–179, Gourette, Pyrénées, France, January 2017.

[8] Timothy Bourke and Marc Pouzet. Arguments cadencés dans un compilateur Lustre vérifié. In Nicolas Magaud and Zaynah Dargaye, editors, *30^{ièmes} Journées Francophones des Langages Applicatifs (JFLA 2019)*, pages 109–124, Les Rousses, Haut-Jura, France, January 2019.

[9] Lélio Brun. *Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset*. PhD thesis, PSL Research University, June 2020.

[10] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6 : A formal language for embedded critical software development. In *Proceedings of the 11th International Symposium on Theoretical Aspects of Software Engineering (TASE 2017)*, pages 4–15, Nice, France, September 2017. IEEE Computer Society.

[11] Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In R. Alur and I. Lee, editors, *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT 2003)*, volume 2855 of *Lecture Notes in Elecrical Engineering*, pages 134–155, Philadelphia, PA, USA, October 2003. Springer.

[12] Coq Development Team. *The Coq proof assistant reference manual*. Inria, 2020.

[13] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[14] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, April 2006.

Partitionnement en régions linéaires pour la vérification formelle de réseaux de neurones

Julien Girard-Satabin[†12], Aymeric Varasse[†1], Guillaume Charpiat[2], Zakaria Chihani[1], and Marc Schoenauer[2]

[1] Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
Les auteurs marqués d'un [†] ont apporté une contibution identique
julien.girard2@cea.fr, aymeric.varasse@cea.fr, zakaria.chihani@cea.fr
[2] INRIA TAU, LRI, France
guillaume.charpiat@inria.fr, marc.schoenauer@inria.fr

Résumé

La grande polyvalence et les résultats impressionnants des réseaux de neurones modernes viennent en partie de leur non-linéarité. Cette propriété fondamentale rend malheureusement très difficile leur vérification formelle, et ce, même si on se restreint à une structure linéaire par morceaux. Cependant, chacune de ces régions linéaires prise indépendamment est simple à analyser. Nous proposons dans cet article une méthode permettant de simplifier le problème de vérification en opérant une séparation en multiples sous-problèmes linéaires. Nous présentons également des résultats concernant la structure de ces régions linéaires ainsi que leur similarité. Ce travail en cours démontre déjà la faisabilité de l'approche sur des problèmes simples ainsi que quelques expériences face à l'état de l'art.

1 Introduction

Les réseaux de neurones profonds ont fait l'objet d'intenses recherches au cours des dernières années. De premiers travaux théoriques ont prouvé leur qualité d'approximateurs universels [12], c'est-à-dire leur capacité théorique à approximer n'importe quelle fonction. Plus récemment, des résultats pratiques impressionnants sur des tâches traitant des données à haute dimension (texte, image, son) ont abouti à leur utilisation dans de nombreuses applications industrielles (p. ex. détection d'objet, modification d'image, robotique). Tout efficaces qu'ils soient, les réseaux de neurones sont des programmes, et en tant que tels ne sont pas exempts de dysfonctionnements. Des travaux de recherche ont par exemple exhibé les exemples adverses, des perturbations imperceptibles volontairement construites pour tromper le réseau. Ces exemples peuvent être transférés entre programmes [16], permettent de synthétiser des discours [17] ou des vidéos [5] à même de tromper les réseaux au niveau de l'état de l'art, et ne sont pas systématiquement détectables [24]. D'autres travaux ont démontré la possibilité de reconstruire les paramètres du réseau [23] ou les données utilisées pour son entraînement [19] uniquement en se basant sur les sorties, ce qui soulève de graves problèmes de confidentialité des données, en particulier dans le domaine médical.

Les réseaux de neurones profonds sont structurés en couches, successions d'opérations linéaires séparées par des fonctions d'activation non-linéaires, l'une des plus populaires étant la *Rectified Linear Unit* ou *ReLU* : $x \to max(x, 0)$. Cette fonction est linéaire par morceaux : ReLU a un comportement linéaire sur $\mathbb{R}_-$ et sur $\mathbb{R}_+$. Les régions de l'espace d'entrée délimitant de tels comportements sont qualifiées de *régions linéaires*, ou *facettes*. Il est commun de considérer le nombre de facettes d'un réseau comme une mesure de son expressivité [18], informellement sa capacité à approximer des fonctions de plus en plus complexes. Dans le cas d'usage classique

d'un réseau, on applique les couches linéaires et non-linéaires aux entrées. Chaque fonction d'activation pouvant produire deux cas linéaires, une exploration naïve de l'espace de recherche doit donc traiter un nombre exponentiel de cas. Cette explosion combinatoire constitue l'un des obstacles majeurs à l'application de techniques de vérifications exhaustives (*e.g.,* calcul de Satisfiabilité, SAT et Satisfiabilité Modulo Théorie, SMT).

Pour surmonter cet obstacle, nous proposons d'exploiter les résultats récents d'analyse de régions linéaires. Les auteurs de [11] avancent ainsi que le nombre effectif de facettes pour les réseaux de $\mathbb{R}$ dans $\mathbb{R}$ est borné linéairement en le nombre de neurones. Dans la suite de leurs travaux [10], ce résultat est étendu à une classe de programmes plus large et plus représentative de réseaux réels, par une borne supérieure qui n'est pas exponentielle en le nombre de neurones. Une borne plus souple est également présentée dans d'autres travaux, tels que [18]. En réduisant un réseau de neurones à une union de facettes, vérifier une propriété de sûreté sur un réseau complexe devient plus simple, car revient à vérifier un ensemble de problèmes linéaires.

Notre contribution s'établit ainsi :

1. nous proposons différentes méthodes d'énumération explicite de facettes, appliquées à différents réseaux de neurones ;
2. nous proposons un algorithme de subdivision d'un problème initial en sous-problèmes linéaires plus simples à résoudre ;
3. nous évaluons la performance de notre approche en l'appliquant sur des réseaux de neurones et en comparant par rapport à l'état de l'art ; nous analysons également les caractéristiques de ces régions linéaires en fonction de différents paramètres.

2 Travaux connexes

La vérification formelle de réseaux de neurones est un domaine naissant qui a déjà produit de nombreuses techniques et outils. Reluplex [13], son successeur Marabou [14] et le solveur Planet [9] constituent ainsi les premières tentatives de méthodes de vérification exhaustives, utilisant le calcul SMT. Dans ces travaux, les auteurs proposent une réécriture de l'algorithme du simplexe permettant une évaluation paresseuse des ReLU afin d'élaguer plus rapidement l'espace de recherche. Notre travail s'inscrit dans cette ligne de recherche ; toutefois nous proposons une technique de pré-traitement plutôt qu'une analyse totale. Notre proposition et la leur sont orthogonales, et peuvent potentiellement être combinées. D'autres techniques d'évaluation exhaustives ont été proposées, telles qu'une formulation en programmation en nombre entiers [22], et une autre utilisant le paradigme *branch and bound* [4].

Des approches non exhaustives existent également, et passent généralement à l'échelle sur des réseaux plus larges, au détriment de la précision de l'analyse. La propagation symbolique de domaines numériques et l'interprétation abstraite sont les techniques les plus courantes, utilisées par exemple dans Reluval [25], CNN-Cert [3] et ERAN [21] ; nous utilisons cette technique également. La délimitation entre ces deux familles n'est pas gravée dans le marbre : il existe par exemple des travaux combinant programmation en nombres entiers et propagation symbolique [20].

L'étude des régions linéaires des réseaux de neurones à ReLU a fait l'objet de travaux théoriques, notamment concernant leur énumération [18]. Une extension du théorème d'approximateur universel a été proposée par [1] pour les réseaux certifiés. Enfin, l'idée d'utiliser les régions linéaires pour améliorer la procédure de vérification formelle a été explorée par les auteurs de [7], où ils proposent une technique d'apprentissage qui maximise le volume des régions linéaires, ce qui résulte en une amélioration de la robustesse du réseau. Notre approche ne nécessite pas le

réentraînement du réseau et ne modifie pas les régions linéaires ; elle pourrait ainsi se combiner avec la leur.

3 Définitions

3.1 Vecteurs d'activation et facettes

Prenons comme exemple un ensemble d'entrée $\mathcal{X}$ en $\mathbb{R}^{D_{\text{in}}}$ (p. ex., pour des images RGB carrées de taille $p \times p$, $D_{\text{in}} = 3p^2$) et un espace de sortie $\mathcal{Y}$ en $\mathbb{R}^{D_{\text{out}}}$. On désigne par $f : \mathcal{X} \to \mathcal{Y}$ un réseau de neurones entraîné à L couches, de $\mathcal{X}$ vers $\mathcal{Y}$. Chaque couche a des entrées et sorties multidimensionnelles, représentées par des tableaux dont chaque case est un neurone. Une couche l_i a une entrée en $\mathbb{R}^{D_{(i-1)}}$ et une sortie en $\mathbb{R}^{D_i}$, pour $i = 2 \dots L$, avec $D_1 = D_{\text{in}}$ et $D_N = D_{\text{out}}$. On notera par la suite une couche par l pour éviter de surcharger la lecture. Nous considérons ici des réseaux pour lesquels chaque couche l est composée d'une application linéaire de paramètres θ^l, suivie d'une activation ReLU (par facilité, nous appelons ainsi la généralisation de ReLU à une entrée multidimensionnelle, opérant sur chaque coordonnée indépendamment). On note C^l la composition de ces deux opérations. Notons que θ^l résulte de l'entraînement du réseau et ne change pas durant son utilisation : les seules variables sont les vecteurs de $\mathcal{X}$. Pour une entrée quelconque, chaque neurone ReLU de chaque couche aura une sortie nulle (resp. positive) si son entrée est négative ou nulle (resp. positive), auquel cas il sera dit *inactif* (resp. *activé*).

On note l'état d'activation, ou schéma d'activation, $\mathcal{S}^l_{\mathcal{F}}$ l'état d'activation des neurones ReLU pour une couche donnée l : un neurone actif est noté 1, un neurone inactif est noté 0. À titre d'exemple, pour le réseau présenté Figure 1, $\mathcal{S}^1_{\mathcal{F}} = (0, 1, 1)$.

On appelle une *facette* le sous-ensemble $\mathcal{F}$ de l'espace d'entrée qui génère un certain vecteur d'activation $\mathcal{S}^l_{\mathcal{F}}$. Toutes les entrées dans cette facette définissent un même vecteur d'activation dans le réseau quand celui-ci les traite. Les facettes définissent donc des régions où le comportement du réseau est *linéaire*, car toutes les ReLU sont dans un état défini. Une entrée $\vec{x}$ appartient à une facette $\mathcal{F}$ si et seulement si son passage dans le réseau produit les mêmes schémas d'activation que les points inclus dans la facette.

3.2 Construction de facettes

Considérons un neurone $n_{i,l}$ de la couche l. Si ce neurone est activé, sa valeur, résultat d'une opération linéaire impliquant des neurones de la couche précédente, est positive. Par exemple, pour une multiplication matricielle d'éléments $w_{i,j}$, pour les sorties des couches précédentes $y_{j,(l-1)}$, il vient une contrainte linéaire

$$n_{i,l} = \sum_j w_{i,j} y_{j,(l-1)} > 0$$

De même, un neurone inactif amène la contrainte suivante

$$n_{i,l} = \sum_j w_{i,j} y_{j,(l-1)} \leq 0$$

Chaque neurone porte une telle contrainte sur les entrées ; une facette est ainsi la conjonction de ces contraintes. Géométriquement, on peut considérer une facette comme un polyèdre convexe décrit par l'ensemble des contraintes linéaires des activations. Voir la Figure 1 pour une illustration sur un exemple jouet.

Pour construire ces contraintes, on opère une propagation symbolique depuis les entrées. Chaque variable d'entrée est représentée par un symbole s_n. On définit une expression symbolique ϕ_e de la forme $\sum_n \alpha_{n,e} \cdot s_n + b$ où $\forall n \in [\![1, N]\!], \alpha_n \in \mathbb{R}$ et $b \in \mathbb{R}$. L'opération $\cdot$ représente la multiplication entre un réel et un symbole. Ces expressions sont construites pour chaque neurone ReLU : elles représentent l'équation de l'hyperplan séparant l'espace des entrées entre l'état activé ou désactivé. La construction de ces équations se fait par application des opérations linéaires des couches successives sur les expressions symboliques. On définit plusieurs opérations élémentaires entre opérations symboliques. Ici, $*$ (respectivement $+$) représente l'opération de multiplication (respectivement d'addition) usuelle sur les réels.

Addition d'un réel à une expression

Pour une expression ϕ_e donnée et une valeur réelle r, on a

$$\phi_e + r = \sum_n \alpha_{n,e} \cdot s_n + (b + r)$$

Multiplication d'une expression par un réel

Pour une expression ϕ_e donnée et une valeur réelle λ, on a

$$\lambda \cdot \phi_e = \sum_n (\lambda * \alpha_{n,e}) \cdot s_n + \lambda * b$$

Addition de deux expressions

Pour deux expressions ϕ_e^1 et ϕ_e^2, on note l'addition $\oplus$ et on a

$$\phi_e^1 \oplus \phi_e^2 = \sum_n \alpha_{n,e}^1 \cdot s_n + b^1 \oplus \sum_n \alpha_{n,e}^2 \cdot s_n + b^2 = \sum_n (\alpha_{n,e}^1 + \alpha_{n,e}^2) \cdot s_n + (b^1 + b^2)$$

Ces opérations linéaires sont composées au cours de la propagation des expressions symboliques dans le réseau, permettant d'obtenir les équations des hyperplans pour chaque neurone ReLU. Leur état d'activation est ensuite modifié en fonction de la facette considérée.

Dans le reste de cet article, nous cherchons à vérifier formellement un réseau de neurones. Pour un réseau f, une précondition sur l'espace d'entrée $\mathcal{D} \subset \mathcal{X}$ et une postcondition sur l'espace de sortie $\mathcal{P} \subset \mathcal{Y}$, nous souhaitons prouver formellement que

$$\forall x \in \mathcal{D} \implies f(x) \in \mathcal{P}$$

La forme exacte des pré et post conditions varie selon la propriété à vérifier. Ainsi, pour la robustesse locale face aux exemples adverses autour d'un échantillon, la propriété peut s'écrire, pour un échantillon $x \in \mathcal{X}$: $\forall \eta \in \mathbb{R}^{D_{\text{in}}}$ t.q. $\|\eta\| < \varepsilon$, $f(x + \eta) = f(x)$. Pour les propriétés de sûreté définies dans [15], les préconditions sont des contraintes linéaires.

4 Recherche de facettes pertinentes

Les sous-régions linéaires du réseau sont simples à vérifier, du fait de l'absence d'opérations produisant des disjonctions de cas (opérations dues aux ReLU). En supposant que nous disposions d'une cartographie exhaustive des facettes *effectivement atteignables* par notre réseau, il serait alors possible d'effectuer la vérification sur chaque facette. Le nombre théorique de

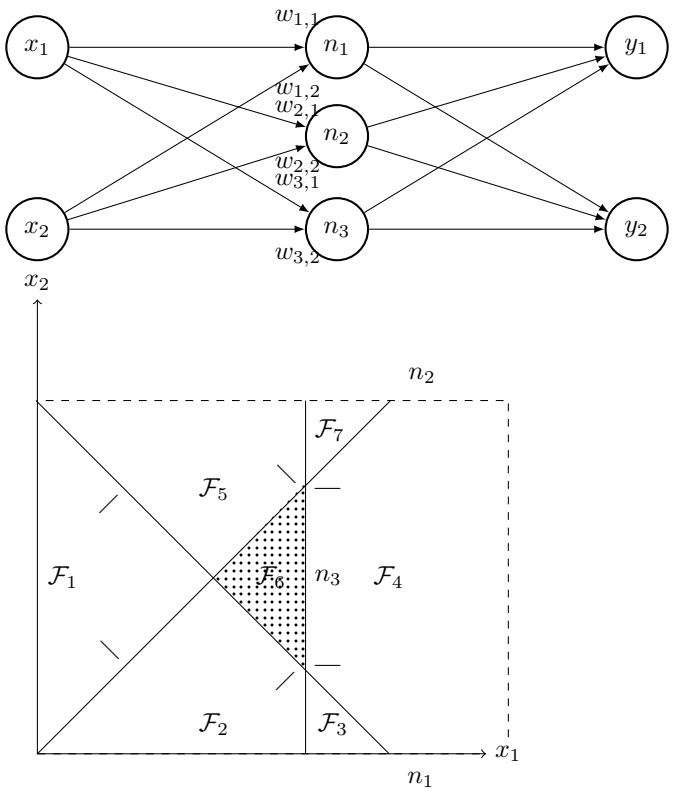


FIGURE 1 – Un réseau jouet où chaque n_i est un neurone ReLU, x_i et y_i sont des valeurs réelles, et $w_{i,j}$ des paramètres. En bas, les facettes sur l'espace d'entrée. L'état d'activation de chaque neurone génère un demi-espace. La zone en points définit la facette résultant en l'activité du neurone n_1 et l'inactivité des neurones n_2 et n_3.

facettes est exponentiel en le nombre de neurones ; toutefois un réseau n'exploite en pratique que rarement toutes ses régions linéaires. Pour une tâche simple, un réseau avec un grand nombre de neurones ne semble avoir besoin que d'une partition restreinte de l'espace d'entrée. Des facettes peuvent également disposer d'un support plus large dans l'espace d'entrée, ce qui peut par exemple indiquer que le réseau exploite beaucoup plus les informations présentes dans la région correspondante. Dans tous les cas, énumérer les facettes effectivement empruntables par notre réseau pour l'espace d'entrée considéré permettrait d'obtenir des informations intéressantes pour l'analyse. Notre procédure de subdivision se doit d'être exhaustive : si notre réseau est porteur d'un dysfonctionnement, notre procédure doit pouvoir l'exhiber. En d'autres termes, nous souhaitons trouver toutes les facettes $\mathcal{F}_i$, formant une partition $\bigcup_i \mathcal{F}_i = \mathcal{X}$.

Une première approche naïve consiste à énumérer les facettes voisines en partant d'une facette connue, déduite par exemple du calcul d'une entrée dans le réseau. Les facettes voisines peuvent alors être énumérées récursivement jusqu'à épuisement des voisins. Pour identifier les polytopes voisins, il est nécessaire d'identifier les neurones dont l'activation définit les arêtes du polytope courant. En effet, pour passer d'une facette $\mathcal{F}_i$ à une facette voisine $\mathcal{F}_j$, il suffit de changer l'activation d'un neurone, et de regarder si le polytope résultant est bien compris dans $\mathcal{X}$. En effet, deux facettes voisines sont séparées d'un hyperplan, hyperplan issu de la contrainte

linéaire d'un neurone. Il est donc nécessaire d'identifier quels hyperplans constituent les arêtes du polytope considéré, et changer l'état d'activation du neurone associé à cet hyperplan. Si le nouvel état d'activation respecte les contraintes de l'entrée, alors il constitue une nouvelle facette. L'Algorithme 1 résume cette procédure. Appliqué récursivement, il permet d'énumérer tous les voisins. Cette procédure exhaustive est cependant coûteuse en fonction de la procédure de résolution utilisée.

Data: An initial concrete facet $\mathcal{F}_{init}$, an input space domain $\mathcal{C}_{inputs}$
Result: a set of all valid hyperplanes delimiting $\mathcal{F}_{init}$
$\mathcal{F}_{current} = \mathcal{F}_{init}$;
neighbours $= \emptyset$;
// take n random constraints to build an initial polyhedron
stack = TakeRandom($\mathcal{F}_{current}$) $\bigcup \mathcal{C}_{inputs}$;
forall constraint $\in \mathcal{F}_{current}$ **do**
    // if there exists a point that respects existing constraints but does not respect the additional constraint, then it changes the form of the polyhedron
    if Solve(stack $\bigcup \neg$constraint) **then**
        // check if the obtained polyhedron is non-empty
        if Solve(stack $\bigcup$ constraint) **then**
            stack = stack $\bigcup$ constraint
        end
    end
end
return stack

Algorithme 1: Sélection d'un ensemble d'hyperplans contenant toutes les frontières d'une facette

Travail en cours

Une autre stratégie consiste à échantillonner des données issues de l'ensemble d'apprentissage du réseau, et de compter le nombre de facettes différentes auxquelles ces données appartiennent. On peut alors comparer ce nombre aux bornes théoriques proposées par exemple dans [10]. Cette méthode n'est cependant pas exhaustive et ne servirait qu'indicateur en haute dimensions : trouver une technique d'énumération plus efficace constitue un travail en cours. Une approche exploitant l'architecture en couche des réseaux est par exemple poursuivie, mais doit encore être améliorée pour des questions de performances.

5 Partitionnement en sous-problèmes linéaires

L'ensemble de toutes les facettes pertinentes $\bigcup_i \mathcal{F}_i$ obtenu constitue un partitionnement de l'espace d'entrée. Ainsi, vérifier la propriété sur chaque facette $\mathcal{F}_i$ est équivalent à vérifier la propriété sur $\mathcal{X}$ (en fait, les frontières sont vérifiées plusieurs fois). Ces problèmes linéaires étant indépendants, l'emploi de techniques de parallélisation est possible et permettrait une amélioration significative du temps de vérification. Plus formellement, considérons un ensemble de facettes $\bigcup_i \mathcal{F}_i$ pour un réseau f et un espace d'entrée $\mathcal{X}$. Le partitionnement consiste à ajouter au flot de contrôle de base du réseau f les contraintes sur les entrées $\mathcal{F}_i$, et à fixer l'état d'activation des neurones ReLU correspondant. Le flot de contrôle résultant n'est donc composé

que d'opérations linéaires : les opérations originales du flot de contrôle et les neurones ReLU activés ou désactivés en fonction du schéma d'activation $\mathcal{S}_{\mathcal{F}_i}$.

Travail en cours

Cette approche de partitionnement peut être raffinée en ciblant des régions linéaires à traiter en priorité par le solveur. Les facettes n'ont en effet pas toutes les mêmes caractéristiques : ainsi certains schémas d'activation peuvent être beaucoup plus présents que d'autres dans un ensemble de données particulier. À titre d'illustration, la figure 2 montre le nombre de schémas d'activation distincts pour un échantillonnage sur les réseaux présentés dans la Section 6.2. Nous constatons que dans certains cas, près de 40% des points de l'ensemble de données passent par la même facette. Il est dès lors envisageable de prioriser ces facettes pour obtenir le plus rapidement possible la plus large couverture des entrées, et détecter rapidement des contre-exemples.

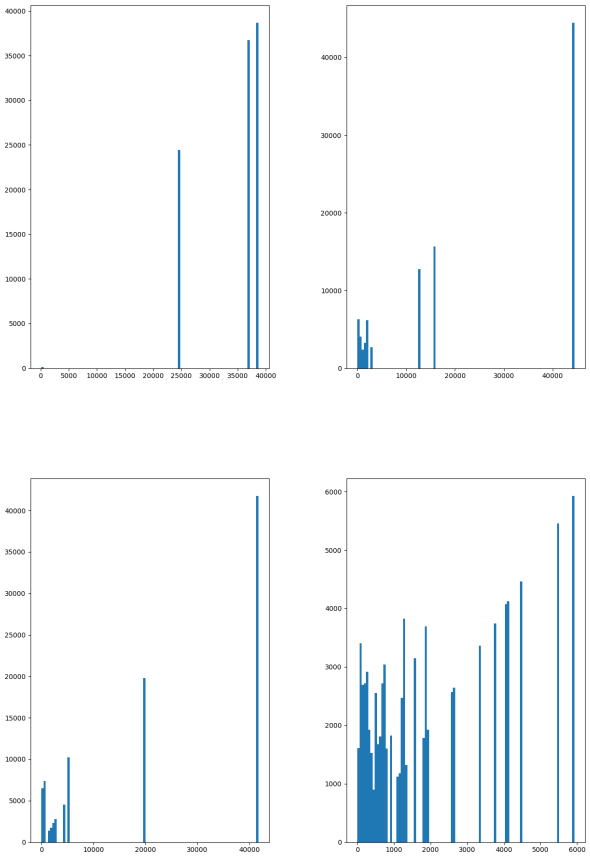


FIGURE 2 – Nombre de fois où une facette a été empruntée, pour 10000 échantillonnages. De gauche à droite et de haut en bas, réseaux de neurones différents avec dimension d'entrée croissante.

6 Évaluation

6.1 Implémentation

La technique de partitionnement est implémentée en OCaml, au sein de l'outil *Inter Standard Artificial Intelligence Encoding Hub* (ISAIEH) [1]. ISAIEH utilise le format standard de description de réseaux ONNX comme entrée principale. L'outil transcrit le flot de contrôle d'un réseau de neurones en formule SMT via le standard SMTLIB [2]. ISAIEH intègre également des techniques de réécriture pour simplifier des réseaux de neurones, dont celle que nous présentons dans cet article. La représentation intermédiaire employée par ISAIEH pour manipuler des réseaux de neurones est un graphe acyclique (N, A) orienté où les nœuds N représentent les opérations sur des tableaux multidimensionnels classiques en apprentissage profond (multiplications matricielles, convolutions, ReLU), et où les arêtes A, étiquetées avec le nom de tenseurs, définissent l'ordre de traitement des opérations. Chaque nœud N décrit l'opération courante, les entrées et les sorties ainsi que les paramètres éventuels nécessaires au calcul. Ce graphe est ensuite traité par un module d'écriture en SMT, qui se charge de convertir les opérations multidimensionnelles en format compréhensible par différentes théories SMT (QF_NRA, QF_LRA et partiellement QF_FP). L'ensemble des opérations supportées est un sous-ensemble des opérations définies par le standard ONNX [2]. Y figurent notamment la multiplication matricielle MatMul, l'application de la fonction ReLU et des opérations de *pooling* MaxPool.

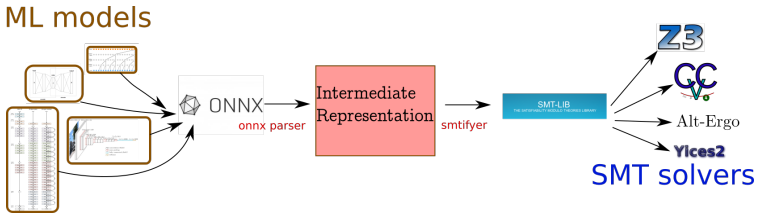


Figure 3 – Schéma de fonctionnement de ISAIEH

Les hyperplans sont obtenus par propagation symbolique au sein d'ISAIEH, ils sont ensuite convertis au format numpy array et manipulés via des scripts python. La bibliothèque python scikit-learn est utilisée pour résoudre le problème linéaire d'appartenance à une facette.

6.2 Expériences

L'hypothèse que l'on souhaite vérifier ici est la suivante : les neurones ReLU introduisent des non-linéarités qui imposent au solveur de séparer et vérifier chaque cas possible, et ceci augmente la complexité et le temps nécessaire à la résolution. En séparant le problème en différentes régions linéaires, ces non-linéarités ne sont plus présentes, et le temps de résolution s'en voit alors réduit, et inférieur au temps nécessaire à la vérification de la propriété pour la fonction non-linéaire directement sur l'ensemble de toutes les entrées possibles.

Les expériences ont été effectuées sur un ordinateur Dell Precision 5530, doté d'un processeur Intel Core i7-8850H cadencé à 2,6GHz, doté de 16 Go de mémoire RAM, avec pour système d'exploitation Ubuntu 18.04.1 LTS x86_64. Le multiprocessing est effectué grâce à la librairie multiprocess et exécutée par la librairie supbrocess, librairies standard de Python (3.8.5 64

1. https://git.frama-c.com/pub/isaieh
2. https://github.com/onnx/onnx/blob/master/docs/Operators.md

bits). La libraire Time de Python est utilisée afin de mesurer le temps nécessaire au solveur pour vérifier la propriété considérée sur chaque région linéaire du réseau de neurones. Pour la technique de partitionnement, les expériences comparent la vitesse de résolution de différents problèmes de sûreté classique pour différentes architectures.

On considère tout d'abord un réseau jouet de deux couches, avec des images en noir et blanc de taille $N \times N$ en guise d'entrée. Il y a $N/2$ neurones sur la première couche et $N/4$ sur la deuxième. Ce réseau doit déclencher une alarme quand il y a au moins un pixel blanc sur la moitié inférieure de l'image (ce qui symbolise la présence d'un obstacle) : il s'agit d'un problème de classification avec deux classes, "alarme" et "normal". Le signe de la sortie $f(x)$ du réseau désigne la classe prédite pour l'entrée x. Un maximum de trois pixels peuvent être allumés en même temps. Pour ces réseaux, le nombre de facettes peut être obtenu par énumération directe de toutes les images possibles ; les réseaux ne sont alors entraînés que sur une fraction de ce total. Le problème ϕ s'écrit alors $\forall x \in \mathcal{X}^{\text{unsafe}}, f(x) > 0$, où $\mathcal{X}^{\text{unsafe}}$ représente l'ensemble dénombrable des images contenant au moins un pixel blanc sur leur moitié inférieure. Le réseau est entraîné pour ne jamais se tromper, et le problème de vérification est UNSAT (il n'existe pas d'entrée obstacle telle que le réseau ne la détecte pas comme un obstacle). On vérifie également la propriété ϕ_4 de l'implémentation du Aircraft Collision Avoidance System, décrit dans [13]. L'objectif de ce réseau est de reconnaître les situations de risque de collisions entre deux aéronefs, et d'émettre des directives de changement de direction en cas de besoin. Le réseau compte six couches cachées pour un total de 300 neurones. Il compte cinq entrées, qui correspondent à des valeurs réelles issues de capteurs embarqués dans un avion, les capteurs sont supposés parfaits. Le réseau est entraîné pour émettre un choix de cinq directives : pas de changement, Strong Left, Strong Right, Weak Left, Weak Right. Les solveurs employés pour la vérification sont z3 [8] version 4.8.10, un solveur SMT état de l'art, Colibri [6], un solveur de programmation par contrainte et Marabou [14], un outil de vérification implémentant la procédure Reluplex.

La Table 1 synthétise les résultats de vérification. Le temps de vérification moyen d'une facette pour chaque réseau est indiqué quand il est disponible. Pour chaque réseau, la première ligne décrit les performances sans partitionnement linéaire, la deuxième (notée D&C) avec le partitionnement. La variante avec partitionnement s'exécute avec la logique QF_LRA, pour bénéficier au maximum des heuristiques de solveurs offertes par l'arithmétique linéaire. Pour les variantes sans partitionnement, on indique le temps en QF_LRA et QF_NRA pour comparaison.

Les résultats de vérification (SAT ou UNSAT) sont identiques sur les problèmes non partitionnés et tous les problèmes linéaires. La mention "multi process" indique les performances lorsque le solveur est lancé sur les différents problèmes indépendants via la bibliothèque de multiprocessing utilisée. Le timeout est à 12h. Il n'a pas été possible d'utiliser notre méthode sur Marabou à cause de problèmes de compatibilité de format. La génération des régions linéaires dure environ une dizaine de secondes pour un réseau 5×5.

La Table 2 donne des résultats préliminaires de dénombrement de facettes obtenus avec notre algorithme. Le nombre de facettes trouvé est comparé à la borne théorique proposée par [10] : $n^d/d!$ où n est le nombre de neurones et d la dimension de l'entrée.

6.3 Interprétation

Pour les variantes D&C, on observe que pour tous les réseaux jouets, la vérification de l'union des régions linéaires va systématiquement plus vite que l'application d'un solveur classique. La Table 2 montre pour les réseaux 3x3, 5x5 et 7x7 un nombre de facettes significativement plus faible que les bornes théoriques, ce qui constitue un résultat encourageant pour l'intérêt de

	z3	Marabou	Colibri
Net 5x5	34s (QF_NRA) 132s (QF_LRA)	0,05s	TIMEOUT
Net 5x5($D\&C$)	137s (single process) 23.7s (multi process) 0.05s (mean time to verify one facet)	–	1454s (multiprocess) 5.5s (mean time to verify one facet)
Net 7x7	391s (QF_NRA) TIMEOUT (QF_LRA)	0.02s	TIMEOUT
Net 7x7 (D&C)	1393s (multi process) 0.28s (mean time to verify one facet)	–	25326s (multi process) 7.3s (mean time to verify one facet)
ACAS ϕ_4	4280s (UNKNOWN, QF_NRA)	22s	TIMEOUT
ACAS ϕ_4($D\&C$)	0.35s (mean time to verify one facet)	–	2.18s (mean time to verify one facet)

Table 1 – Temps d'exécution pour différents problèmes

Réseau	nombre d'entrées	nombre de neurones	borne théorique	nombre de facettes
CAMUS 3x3	9	8	1067	4
CAMUS 5x5	25	18	$2e^9$	1007
CAMUS 7x7	49	36	$2e^{19}$	8560
CAMUS 9x9	81	62	$2.6e^{26}$	1859
ACAS	5	300	$2.02e^{10}$	–

Table 2 – Énumération de facettes

notre méthode.

On constate sur la Figure 2 que certaines mêmes facettes sont empruntées plus de 40% du temps, ce qui implique une faible diversité dans les schémas d'activation effectivement rencontrés.

La diminution du nombre de facettes pour le réseau 9x9 est un phénomène pour lequel nous n'avons pas encore d'explications : il appartiendra à la suite du travail d'élucider ce fait. Une hypothèse est que l'augmentation de la complexité de la tâche force le réseau à réorganiser ses régions linéaires durant l'entraînement, afin de traiter le problème avec un nombre réduit de facettes. Il n'a pas été possible d'obtenir le nombre exact de facettes sur un réseau ACAS, l'algorithme d'énumération ayant dysfonctionné sur ce réseau.

7 Discussion et perspectives

En nous appuyant sur des résultats théoriques récents, nous avons proposé une technique de partitionnement de problème de vérification de réseaux de neurones en sous-problèmes linéaires plus simples à vérifier. Cette technique emploie les régions linéaires de l'espace d'entrée générées par les fonctions d'activations, régions linéaires que nous dénombrons et analysons également.

Plusieurs pistes de poursuite de recherche sont envisagées. Compléter les expériences sur

un échantillon plus large de réseaux pour confirmer l'intérêt de notre technique en constitue une première. Le perfectionnement de notre algorithme d'énumération de voisins, notamment l'accélération de la procédure d'énumération de voisins en est une autre. L'approche purement géométrique que nous poursuivons pour l'instant risque de se heurter à la difficulté combinatoire observée en haute dimension ; il serait possible d'utiliser des raisonnements au niveau du réseau pour diminuer la complexité de l'algorithme. Enfin, de premiers éléments d'analyse tendent à montrer que les hyperplans varient relativement peu dans les domaines d'entrée considérés. Une possible amélioration de notre technique consisterait alors à modifier le réseau pour diminuer le nombre d'hyperplans et ainsi accélérer d'autant plus la vérification.

L'amélioration de l'outil ISAIEH, notamment par le support de plus d'opérateurs du standard ONNX et de formats de sortie, permettra d'utiliser notre technique sur plus de réseaux différents.

Remerciements

Ces travaux ont été soutenus financièrement par la Commission européenne par le biais du sous-projet SAFAIR du projet SPARTA, qui a reçu un financement du programme de recherche et d'innovation Horizon 2020 de l'Union européenne dans le cadre de la convention de subvention n° 830892. Ces travaux ont également reçu un soutien financier du projet CPS4EU, du financement de l'entreprise commune (EC) ECSEL dans le cadre de la convention de subvention n° 826276. L'EC reçoit le soutien du programme de recherche et d'innovation Horizon 2020 de l'Union européenne et de la France, de l'Espagne, de la Hongrie, de l'Italie et de l'Allemagne.

Références

[1] Maximilian Baader, Matthew Mirman, and Martin Vechev. Universal Approximation with Certified Networks. September 2019.

[2] Clark Barrett, Pascal Fontaine, and Aaron Stump. The SMT-LIB Standard. page 104.

[3] Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. CNN-Cert : An Efficient Framework for Certifying Robustness of Convolutional Neural Networks. *arXiv :1811.12395 [cs, stat]*, November 2018. arXiv : 1811.12395.

[4] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. A Unified View of Piecewise Linear Neural Network Verification. *arXiv :1711.00455 [cs]*, November 2017. arXiv : 1711.00455.

[5] Shang-Tse Chen, Cory Cornelius, Jason Martin, and Duen Horng Chau. ShapeShifter : Robust Physical Adversarial Attack on Faster R-CNN Object Detector. *arXiv :1804.05810 [cs, stat]*, April 2018. arXiv : 1804.05810.

[6] Zakaria Chihani, Bruno Marre, François Bobot, and Sébastien Bardin. Sharpening Constraint Programming Approaches for Bit-Vector Theory. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming*, volume 10335, pages 3–20. Springer International Publishing, Cham, 2017.

[7] Francesco Croce, Maksym Andriushchenko, and Matthias Hein. Provable robustness of relu networks via maximization of linear regions. In *the 22nd International Conference on Artificial Intelligence and Statistics*, pages 2057–2066, 2019.

[8] Leonardo De Moura and Nikolaj Bjørner. Z3 : An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Ruediger Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. *arXiv :1705.01320 [cs]*, May 2017. arXiv : 1705.01320.

[10] Boris Hanin and David Rolnick. Deep ReLU Networks Have Surprisingly Few Activation Patterns. page 10.

[11] Boris Hanin and David Rolnick. Complexity of Linear Regions in Deep Networks. *arXiv :1901.09021 [cs, math, stat]*, January 2019. arXiv : 1901.09021.

[12] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2) :251–257, 1991.

[13] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex : An Efficient SMT Solver for Verifying Deep Neural Networks. *arXiv :1702.01135 [cs]*, February 2017. arXiv : 1702.01135.

[14] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer, and Clark Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, volume 11561, pages 443–452. Springer International Publishing, Cham, 2019.

[15] Guido Manfredi and Yannick Jestin. An introduction to acas xu and the challenges ahead. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–9. IEEE, 2016.

[16] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in Machine Learning : from Phenomena to Black-Box Attacks using Adversarial Samples. *arXiv :1605.07277 [cs]*, May 2016. arXiv : 1605.07277.

[17] Yao Qin, Nicholas Carlini, Ian Goodfellow, Garrison Cottrell, and Colin Raffel. Imperceptible, Robust, and Targeted Adversarial Examples for Automatic Speech Recognition. *arXiv :1903.10346 [cs, eess, stat]*, March 2019. arXiv : 1903.10346.

[18] Thiago Serra, Christian Tjandraatmadja, and Srikumar Ramalingam. Bounding and Counting Linear Regions of Deep Neural Networks. *arXiv :1711.02114 [cs, math, stat]*, September 2018. arXiv : 1711.02114.

[19] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks Against Machine Learning Models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18, San Jose, CA, USA, May 2017. IEEE.

[20] Gagandeep Singh and Timon Gehr. BOOSTING ROBUSTNESS CERTIFICATION OF NEURAL NETWORKS. *ICLR 2019*, page 12, 2019.

[21] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and Effective Robustness Certification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10825–10836. Curran Associates, Inc., 2018.

[22] Vincent Tjeng, Kai Y Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2018.

[23] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. pages 601–618, 2016.

[24] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness May Be at Odds with Accuracy. *arXiv :1805.12152 [cs, stat]*, May 2018. arXiv : 1805.12152.

[25] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. *arXiv :1804.10829 [cs]*, April 2018. arXiv : 1804.10829.

Lavez vos graphes plus blanc avec HoCL

Type de soumission : article court

Jocelyn Sérot[1,2]

[1] Institut Pascal, UMR 6602 UCA/CNRS/SIGMA
[2] IETR, UMR 6164, U. Rennes 1, INSA Renne
jocelyn.serot@uca.fr

Résumé

On introduit HoCL, un petit DSL purement fonctionnel pour la spécification de graphes flots de données hiérarchiques paramétrés. HoCL s'appuie sur une sémantique purement fonctionnelle pour faciliter la description de tels graphes en autorisant notamment l'encodage de schémas de graphes sous la forme de fonctions d'ordre supérieur. Le compilateur fournit des *backends* spécifiques permettant l'interfaçage à des outils existants de visualisation, d'analyse et optimisation et d'implémentation. HoCL est écrit en OCaml et disponible en *open source*.

1 Introduction - Contexte et motivations

Les graphes flots de données (GFD) sont un formalisme très utilisé pour la spécification et l'implémentation d'applications de traitement du signal. Dans ce cadre, les applications sont décrites sous la forme de graphes constitués d'unités de calcul appelées *acteurs* reliées par des canaux de communication de type FIFO et s'exécutant, dès que les canaux connectés en entrée contiennent suffisamment de jetons de données (*tokens*), conformément à un ensemble de *règles d'activation*. Le principal intérêt de ce modèle est sa capacité à exprimer le parallélisme intrinsèque des applications. La possibilité de faire varier la sémantique précise des règles d'activations, autrement dit de préciser le modèle de calcul (*Model of Computation*) associé au GFD, permet par ailleurs d'ajuster le compromis expressivité-prédictabilité du formalisme.

Il n'est donc pas surprenant qu'un grand nombre d'outils exploitant ce type de formalisme et dédiés à la spécification et à l'implémentation, sous forme logicielle ou matérielle, aient été développés. Dans le domaine du traitement du signal, on peut notamment citer la plate-forme Ptolemy [3] issues des travaux de Lee *et al.* à Berkeley, l'outil Preesm [12] développé à Rennes ou, dans le domaine commercial, le logiciel LabView [11].

Avec ces outils, la spécification du GFD se fait typiquement de manière textuelle, via un langage *ad-hoc*, ou graphiquement, via une interface graphique dédiée. Dans les deux cas, la définition de graphes comportant un grand nombre de nœuds et/ou d'arcs peut rapidement devenir fastidieuse et source d'erreurs.

Pour pallier ce problème, on peut s'appuyer sur la dualité naturelle existant entre les GFD et les langages de programmation fonctionnels. La mise en évidence de cette dualité remonte aux premiers langages dits « flot de données », comme VAL [10] et SISAL [5]. Depuis, elle a été exploitée dans de nombreux langages, comme Lava [2] ou CLaSH [6] (dans le cadre de la conception de circuits numériques) ou Fran [4] (dans le cadre de la programmation fonctionnelle réactive). On la trouve aussi au cœur de langages synchrones comme Lustre [7] ou Lucid Synchrone [13].

Ces langages offrent notamment la possibilité d'encoder des schémas de graphe génériques sous la forme de fonctions d'ordre supérieur, ce qui permet d'élever significativement le niveau d'abstraction par rapport à une description purement graphique. L'usage d'un langage

fortement typé permet par ailleurs de détecter au plus tôt les erreurs de « câblage » au sein des graphes. Mais, parce qu'ils visent à donner une sémantique non seulement statique mais aussi dynamique aux programmes, c.à.d. à spécifier non seulement la *topologie* du graphe décrit mais aussi le modèle de calcul sous-jacent, ces langages ne répondent pas forcément aux besoins des utilisateurs, lorsque l'objectif est simplement, et très pragmatiquement, d'éviter la saisie « à la main », d'un graphe contenant plusieurs dizaines voire centaines d'acteurs et ce indépendamment du modèle de calcul sous-jacent. Dans ce contexte, il peut être souhaitable de disposer d'un langage permettant de spécifier simplement et uniquement la topologie du graphe à décrire. Un tel langage, parfois appelé *langage de coordination*, peut alors être utilisé en amont d'outils existants pour générer des représentations de graphes complexes destinées à être traitées par ces outils.

On décrit dans cet article un langage, HoCL [1], développé, à titre expérimental, en réponse à cet objectif. HoCL est purement fonctionnel, fortement typé et supporte la définition de schéma de graphes génériques sous la forme de fonctions d'ordre supérieur. HoCL a vocation à être utilisé comme frontal à des outils existants de manipulation et d'implémentation d'applications exprimées sous forme de GFD, comme DIF [8] ou Preesm [12] [2].

Compte-tenu du format de l'article, on se contente, dans la suite, de présenter les principaux traits du langage, de manière informelle et à l'aide d'exemples simples, l'objectif étant de donner une idée de son expressivité. Les aspects plus techniques – la sémantique formelle du langage en particulier –, sont décrits dans un ensemble de documents disponibles sur le dépôt GitHub [15]. Un certain nombre d'exemples, y compris plusieurs applications complètes de traitement du signal et des images, sont par ailleurs disponibles sur ce dépôt [3].

2 Le langage HoCL

Une version simplifiée de la syntaxe abstraite du langage est donnée sur la figure 1 [4]. La méta-syntaxe est classique : la notation $[e]$ désigne un élément e optionnel ; la notation e_s^* (resp. e_s^+) désigne 0 (resp. 1) ou plus répétitions de l'élément e, séparées par l'élément s. Les non-terminaux sont notés entre $\langle\rangle$, les terminaux en italique (sauf les mots-clés, en gras). Un exemple de programme, avec le GFD correspondant, est donné sur la figure 2.

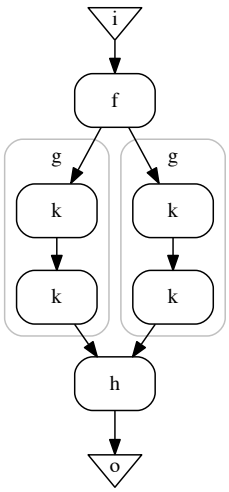
Les expressions de types dénotent les types pouvant être attachés aux ports d'entrée-sortie des acteurs [5]. Un type est soit un type de base, soit le type τ `param`, où τ est un type de base. Les types de base sont attachés aux flots de données, le type `param` aux paramètres. Les types de bases sont limités aux types prédéfinis `int` et `bool`, aux types abstraits (introduits par une déclaration de type explicite) où une variable de type. Ces dernières sont utilisées pour déclarer des acteurs polymorphiques (cf. par exemple Sec. 2.4).

Les déclarations de nœuds, introduites par le mot-clé `node`, se composent d'une interface et d'une description. L'interface donne le nom du nœud et liste le nom et le type des ports

1. HoCL est l'acronyme de *Higher Order Coordination Language*. Les lettres C et L correspondent aux termes *Coordination Language*. Les lettres H et O font référence à l'ordre supérieur (*Higher Order*). HoCL est aussi la formule chimique de l'acide hypochloreux, traditionnellement utilisé comme détachant ou décapant, d'où le titre.
2. A l'origine, HoCL a été développé en réponse à des besoins exprimés par les utilisateurs de l'outil Preesm.
3. Le code des exemples décrits dans cet article, en particulier, est disponible à l'adresse `https://github.com/jserot/hocl/tree/master/examples/working/jfla`.
4. Les principales omissions concernent les expressions et les motifs de manipulation de listes.
5. Qui sont propagés aux arcs du graphe de la phase de synthèse-vérification de types.

$$
\begin{array}{rcl}
\langle \text{program} \rangle & ::= & \langle \text{type\_decl} \rangle^* \ \langle \text{defn} \rangle^* \ \langle \text{node\_decl} \rangle^+ \\
\langle \text{type\_decl} \rangle & ::= & \textbf{type}\ \textit{ident} \\
\langle \text{node\_decl} \rangle & ::= & \textbf{node}\ \textit{ident}\ (\ \langle \text{io\_decl} \rangle^*_{,}\ )\ (\ \langle \text{io\_decl} \rangle^*_{,}\ )\ [\langle \text{node\_impl} \rangle] \\
 & | & \textbf{graph}\ \textit{ident}\ (\ \langle \text{gio\_decl} \rangle^*_{,}\ )\ (\ \langle \text{io\_decl} \rangle^*_{,}\ )\ \langle \text{node\_impl} \rangle \\
\langle \text{io\_decl} \rangle & ::= & \textit{ident}\ \textbf{:}\ \langle \text{type\_expr} \rangle \\
\langle \text{gio\_decl} \rangle & ::= & \textit{ident}\ \textbf{:}\ \langle \text{type\_expr} \rangle\ [= \langle \text{const\_expr} \rangle] \\
\langle \text{node\_impl} \rangle & ::= & \langle \text{defn} \rangle^* \\
\langle \text{defn} \rangle & ::= & \textbf{val}\ [\textbf{rec}]\ \langle \text{binding} \rangle^+_{\textbf{and}} \\
\langle \text{binding} \rangle & ::= & \langle \text{pattern} \rangle = \langle \text{expr} \rangle \\
\langle \text{pattern} \rangle & ::= & \textit{ident} \quad | \quad (\ \langle \text{pattern} \rangle^+_{,}\ ) \quad | \quad () \\
\langle \text{expr} \rangle & ::= & \langle \text{const\_expr} \rangle \quad | \quad \textit{ident} \quad | \quad \langle \text{expr} \rangle\ \langle \text{expr} \rangle \quad | \quad (\ \langle \text{expr} \rangle^+_{,}\ ) \\
 & | & \textbf{fun}\ \langle \text{funpat} \rangle \rightarrow \langle \text{expr} \rangle \quad | \quad \textbf{if}\ \langle \text{expr} \rangle\ \textbf{then}\ \langle \text{expr} \rangle\ \textbf{else}\ \langle \text{expr} \rangle \\
 & | & \textbf{let}\ [\textbf{rec}]\ \langle \text{binding} \rangle^+_{\textbf{and}}\ \textbf{in}\ \langle \text{expr} \rangle \quad | \quad () \\
\langle \text{funpat} \rangle & ::= & \textit{ident} \\
\langle \text{const\_expr} \rangle & ::= & \textit{int} \quad | \quad \textbf{true} \quad | \quad \textbf{false} \\
\langle \text{type\_expr} \rangle & ::= & \langle \text{base\_type} \rangle \quad | \quad \langle \text{base\_type} \rangle\ \textbf{param} \\
\langle \text{base\_type} \rangle & ::= & \mathsf{int} \quad | \quad \mathsf{bool} \quad | \quad \textit{ident} \quad | \quad \textit{tyvar}
\end{array}
$$

FIGURE 1 – Syntaxe abstraite (simplifiée) du langage HoCL




```
type t;

node f in (i:t) out (o1:t, o2:t);
node k in (i:t) out (o:t);
node h in (i1:t, i2:t) out (o:t);

node g in (i: t) out (o: t)
fun
  val o = i ▷ k ▷ k
end;

graph top in (i: t) out (o: t)
fun
  val (x1,x2) = f i
  val o = h (g x1) (g x2)
end;
```

FIGURE 2 – Un exemple de GFD et son codage en HoCL

d'entrée et de sortie. Dans l'exemple de la figure 2, pour simplifier, toutes les entrées et sorties ont pour type `t`. La description d'un nœud peut être soit vide (comme pour les nœuds `f`, `k` et `h` dans l'exemple), soit constituée d'un ensemble de définitions (comme pour le nœud `g`). Dans le premier cas, le nœud décrit un acteur dit *atomique*. Les instances de ces nœuds sont vues comme des boîtes noires [6]. Dans le second cas, les définitions décrivent le *sous-graphe* associé

6. Il est toutefois possible – bien que ce ne soit pas illustré ici – d'associer aux acteurs atomiques des

au nœud [7].

Chaque définition, introduite par le mot-clé `val` lie un ensemble de noms à un ensemble de valeurs (comme le `let` en Caml), ces valeurs étant dénotées par des expressions dont la syntaxe est précisée sur la figure 1.

La sémantique de ces expressions est classique, à l'exception de l'application. En HoCL, les nœuds sont en effet vus comme des fonctions, un nœud déclaré ainsi :

$$\texttt{node f in } (i_1 : \tau_1,\ \ldots,\ i_m : \tau_m) \texttt{ out } (o_1 : \tau'_1,\ \ldots,\ o_n : \tau'_n)$$

étant interprété comme une fonction ayant pour type :

$$i_1 : \tau_1 \rightarrow \ldots \rightarrow\ i_m : \tau_m \rightarrow \tau'_1 \times \ldots \times \tau'_n \tag{1}$$

L'application complète de la fonction associée à un nœud correspond alors à l'*instanciation* du nœud correspondant. L'application partielle, par contre, construit, classiquement, une fermeture. Dans l'exemple de la figure 2, la définition du nœud `g` indique ainsi que le sous-graphe correspondant est construit en instanciant deux fois le nœud `k` et en reliant l'entrée de la première instance à l'entrée dudit sous-graphe, sa sortie à l'entrée de la seconde instance et la sortie de la seconde instance à la sortie du sous-graphe [8].

Les déclarations de graphes, introduites par le mot-clé `graph`, sont constituées, comme pour les nœuds, d'une interface et d'une définition. Mais, à la différence d'un nœud, la définition est ici obligatoirement donnée sous la forme d'un sous-graphe et cette définition est automatiquement instanciée. Une spécification valide en HoCL est donc constituée d'au moins une déclaration `graph` [9].

Ici, le graphe `top` est défini en

— instanciant le nœud `f`, reliant son entrée à l'entrée du graphe,
— instanciant deux fois le sous-graphe `g`, reliant la première (resp. seconde) sortie de nœud `f` [10] à l'entrée de la première (resp. seconde) instance,
— instanciant le nœud `h`, reliant sa première (resp. seconde) entrée à la sortie de la première (resp. seconde) instance de `g`.

L'exemple de la figure 2 montre que sémantique d'une définition introduite par le mot-clé `val` dépend du ou des noms lié(s) à gauche. Si un nom désigne une sortie du (sous-)graphe, ladite sortie est reliée (connectée) à la valeur correspondante à droite (qui doit désigner une entrée ou le port de sortie d'une instance de nœud, condition vérifiable par typage). C'est le cas, par exemple, pour la sortie `o` du sous-graphe `g`. Sinon, le mot-clé `val` sert simplement à nommer des valeurs intermédiaires, à l'instar du `let` de Caml par exemple. C'est le cas, par exemple, dans la définition du graphe `top` qui nomme explicitement les deux sorties du nœud `f` afin de les utiliser séparément.

annotations dédiées aux *backends*.

7. Le mot-clé `fun`, utilisé pour introduire ces définitions, se justifie par le fait qu'il est aussi possible, en HoCL, de décrire des sous-graphes de manière *structurelle*, c.à.d. en listant explicitement les instances de nœuds et leurs interconnexions. Cette possibilité, introduite dans le but de faciliter l'apprentissage du langage à des programmeurs peu familiers du style fonctionnel, n'est pas détaillée ici.

8. L'opérateur $\triangleright$ étant, classiquement, défini par $x \ \triangleright f = f\ x$

9. Il peut en y avoir plusieurs.

10. La sortie de l'*instance* du nœud `f` devrait-on dire, *stricto sensu*. Dans la suite, par souci de concision, et sauf confusion possible, on confonra le nom d'un avec celui de ses instances.

2.1 Arguments nommés

Comme attesté par le type de la fonction associée à un nœud (cf. 1), HoCL supporte le passage d'argument nommés (*labeled arguments*). Ainsi, le dernière ligne de la définition du graphe `top`, sur la figure 2, aurait pu s'écrire :

```
val o = h i1:(g x1) i2:(g x2)
```

Comme en OCaml, cette caractéristique permet d'une part de documenter l'instanciation d'un nœud (en particulier si le nom des ports d'entrée à été choisi judicieusement [11]) et d'autre part de relier ces ports indépendamment de l'ordre dans lequel ils ont été spécifiés (en écrivant par exemple ici : **val** o = h i2:(g x2) i1:(g x1)). Cette deuxième possibilité trouve notamment son intérêt en conjonction avec l'application partielle pour la spécification des *paramètres* d'un nœud (cf. Sec. 2.4).

2.2 Fonctions et schémas de graphe

Comme indiqué sur la figure 1, le mot-clé `val` peut aussi servir à définir des fonctions. La définition du sous-graphe `g`, sur la figure 2, aurait ainsi pu s'écrire de la manière suivante [12] :

```
val twice f x = f (f x)
val o = twice k x
```

ou encore, et manière plus générale :

```
val o = iter 2 k x
```

où `iter` est la fonction classiquement définie ainsi :

```
val rec iter n f x =
  if n=0 then x
  else iter (n-1) f (f x)
```

Des fonctions comme `iter` peuvent être utilisées pour décrire des « schémas de graphes », c.à.d. des patrons génériques fréquemment rencontrées dans les graphes d'applications. La distribution HoCL comprend une « bibliothèque standard » qui intègre plusieurs fonctions de ce type. Parmi celles-ci, on peut citer, par exemple, `pipe` (une variante de `iter` qui applique une liste de n fonctions distinctes en séquence) ou `map` (qui applique une même fonction en parallèle à une liste d'arguments). Un point important est que toutes ces fonctions sont définies au sein du langage lui-même, sous la forme de simples déclarations en HoCL [13]. Le jeu de fonctions utilisables n'est donc pas limité à un sous-ensemble prédéfini mais peut être étendu ou modifié librement par le programmeur d'applications en fonction de ses besoins. Cette possibilité, qui découle du caractère pleinement fonctionnel du langage, est à mettre en regard de celles offertes par les outils de conception flot de données cités en introduction [3, 11, 14], pour lesquels la notion de schéma de graphe, quand elle est supportée, se limite à un ensemble fixe, prédéfini et non extensible.

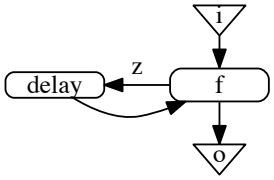
2.3 Graphes cycliques

Dans la plupart des modèles flot de données, la présence de cycles au sein d'un graphe – c.à.d. d'arcs reliant la sortie d'un acteur à une de ses entrées – sert à exprimer les dépendance

11. Ce qui, nous en convenons, n'est pas le cas de l'exemple utilisé ici.
12. Où, comme en Caml, `val f x = e` est une abréviation pour `val f = fun x -> e`.
13. Dans le fichier `lib/hocl/stdlib.hcl`, en pratique.

entre deux activations successives de ce graphe. Par exemple, le graphe de la figure 3-a décrit un filtre récursif, prenant en entrée une séquence $x_1, x_2, \ldots$ de jetons et produisant en sortie la séquence $y_1, y_2, \ldots$, où $(y_i, z_i) = f(x_i, z_{i-1}) \forall\ i > 0$ et z_0 est une constante. Le nœud nommé `delay` sert ici à exprimer le dépendance inter-activation et à fournir la première valeur z_0. Ce type de structure se décrit comme indiqué sur la figure 3-b en HoCL. Comme en Caml, le mot-clé `rec` exprime la nature récursive de la définition, le symbole `z`, lié à la seconde sortie du nœud `f`, apparaissant à la fois à droite et à gauche du signe égal.

Contrairement aux langages flot de données associant une sémantique *dynamique* aux graphes décrits (les formalismes synchrones, en particulier), HoCL ne vérifie pas la causalité du graphe. La définition **val rec** (o,z) = f i z, par exemple, est parfaitement acceptée, et produit un graphe au sein duquel, la seconde sortie du nœud `f` est directement rebouclée sur sa seconde entrée. En tant que pur langage de coordination, ce n'est en effet pas son rôle [14] et il délègue cette tâche aux outils qu'il alimente en aval.



(a)

```
graph top
   in (i: int)
  out (o: int)
fun
   val rec (o,z) = f i (delay z)
end;
```

(b)

FIGURE 3 – Un exemple de graphe *cyclique* et son codage en HoCL

2.4 Paramètres

Les modèles flot de données *paramétrés* ont été introduits dans [1]. Ils se caractérisent par la présence d'acteurs *dynamiquement reconfigurables*, c.à.d. d'acteurs dont le comportement peut être modifié à des instants particulier de l'exécution du graphe. La nature précise des modifications induites par ces reconfigurations – par exemple, la fonction calculée et/ou les taux de production-consommation sur les ports d'entrée-sortie notamment – et les instants auxquels elles ont lieu dépendent du modèle de calcul.

D'un point de vue strictement topologique, la notion de graphe paramétré conduit simplement à distinguer deux types d'arcs dans le graphe : ceux véhiculant des jetons de données proprement dits et ceux matérialisant les dépendances des acteurs aux paramètres. En HoCL, cette distinction est reflétée par le type des acteurs concernés : les premiers auront pour type `t`, où `t` est un type de base [15], les seconds `t param`, où `param` un constructeur de type prédéfini.

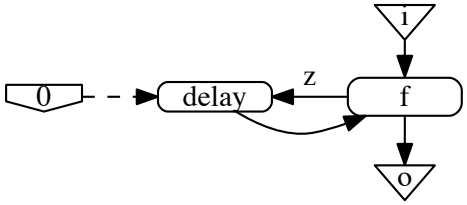
Considérons par exemple l'acteur `delay` utilisé dans l'exemple de la figure 3. Il est naturel d'associer à cet acteur un paramètre permettant de spécifier la valeur initiale produite sur sa sortie. Pour cela, on le déclarera de la manière suivante :

```
node delay in (init: α param, i: α) out (o: α);
```

où α désigne une variable de type [16].

14. Dans l'exemple de la figure 3, on pourrait imaginer que le compilateur vérifie la présence d'un acteur de délai dans le cycle ; mais, sauf à user d'annotations spécifiques, il n'a pas le moyen d'identifier sûrement de tels acteurs.
15. Soit `int`, `bool` où n'importe quel type abstrait introduit par une déclaration de type.
16. Notée `'a` dans le code source, comme en Caml.

La reformulation de l'exemple de la figure 3 avec cette version reconfigurable de l'acteur `delay` est donnée sur la figure 4. La valeur du paramètre `init` est ici fixée à 0. L'opérateur de mise entre quote (`''`) agit comme un constructeur transformant une valeur de type `t` en une valeur de type `t param`. Cet opérateur est nécessaire car une valeur littérale comme `0` peut tout aussi bien désigner un constante entière, de type `int` donc [17], qu'une valeur de *paramètre*, de type `int param`. Sur la figure 4 les arcs en trait continu dénotent les dépendances de données, ceux en trait pointillé les dépendances de paramètres et la boîte en forme de maison inversée une valeur de paramètre.



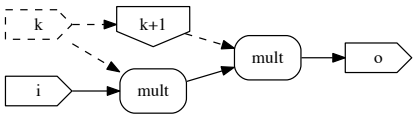

```
graph top
   in (i: int)
  out (o: int)
fun
  val rec (o,z) = f i (delay '0' z)
end;
```

FIGURE 4 – Un exemple de graphe *paramétré* et son codage en HoCL

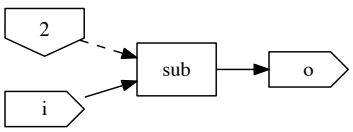
On peut noter qu'avec cette approche, la (re)configuration d'un acteur s'interprète simplement comme l'*application partielle* de la fonction correspondante. La définition

```
val delay0 = delay init:'0'
```

par exemple, définit un acteur `delay0`, de type `int` → `int`, opérant sur des flots d'entiers et produisant initialement la valeur 0. La possibilité de nommer les arguments, comme indiqué à la section 2.1, permet de procéder de la sorte même lorsque les paramètres ne sont pas spécifiés en premier dans la liste des entrées du nœud correspondant.




```
node sub
  in (k:int param, i:int) out (o:int)
fun
  val o = i ▷ mult k ▷ mult (k+1)
end;
```




```
graph top
  in (n:int param=2, i:int) out (o:int)
fun
  val o = i ▷ sub n
end;
```

FIGURE 5 – Un exemple de graphe hiérarchique paramétré

Paramètres et hiérarchie. Lorsque un nœud paramétré est défini par un sous-graphe, la valeur des paramètres peut être utilisée pour paramétrer des nœuds de ce sous-graphe, soit directement, soit indirectement, via des expressions faisant référence à ces paramètres. De cette

17. Comme illustré dans la définition de la fonction `iter` à la section 2.2.

manière, les paramètres peuvent être propagés d'un niveau de hiérarchie à un autre, comme illustré sur la figure 5.

Au sein du graphe `sub`, `k` est vu comme un paramètre d'entrée (le port correspondant est dessiné en pointillé) et ce paramètre est utilisé pour paramétrer les deux instances de l'acteur `mult`, directement pour la première, via l'expression `k+1` pour la seconde. Les expressions mettant en jeu des paramètres sont, dans la version actuelle du langage, limitées aux expressions arithmétiques et booléennes (les paramètres étant eux-mêmes limités à des valeurs entières ou booléennes). Un point important est que ces expressions ne sont *pas* évaluées statiquement (à la compilation) car leur interprétation dépend *in fine* de la sémantique du modèle de calcul associé au graphe [18].

3 Conclusion - Etat du projet et perspectives

Un prototype de compilateur, écrit en OCaml, est disponible sur la plate-forme Github [15]. La distribution fournit en fait deux outils : un compilateur en ligne de commande, générant une représentation du ou des graphes flot de données décrits le programme HoCL dans plusieurs formats, et un interpréteur, autorisant la construction interactive de tels graphes.

Les formats de sortie actuellement supportés – outre le format `DOT`, utilisé pour la visualisation des graphes via les outils de la chaîne Graphviz [19] – sont DIF, Preesm et SystemC. DIF (*Dataflow Interchange Format*, [8]) est une représentation intermédiaire standardisée pour la description de graphes flot de données paramétrés multi-modèles. Le *backend* DIF permet d'utiliser HoCL comme formalisme de spécification de haut niveau pour un grand nombre d'outils existants dédiés à l'analyse, l'optimisation et l'implémentation de graphes flot de données. Preesm [12] est un outil dédié à l'implémentation optimisée d'applications de traitement du signal sur plate-forme hétérogène multi-cœurs embarquée. Le format SystemC permet, sous certaines hypothèses [20] d'utiliser HoCL pour simuler le comportement des programmes.

Concernant l'interpréteur, une courte vidéo illustrant son fonctionnement est disponible en ligne [21].

Comme indiqué en introduction, HoCL est un projet expérimental, développé en réponse à des besoins opérationnels très concrets. A ce titre il est loin d'être figé et susceptible d'évolutions en fonction des besoins réels rencontrés par les usagers des domaines d'applications concernés [22]. Nous travaillons actuellement sur la reformulation en HoCL de plusieurs applications de complexité réaliste issues du domaine du traitement du signal et des images et formulées via des outils de plus bas niveau (comme DIF ou Preesm), afin d'une part de juger de l'intérêt de tel ou tel trait et d'autre part d'évaluer le gain apporté, en termes de productivité, par le langage dans son ensemble. Parmi les modifications et extensions suggérées à ce stade se trouve notamment la possibilité de passer des informations spécifiques à tel ou tel modèle de calcul – les taux de production-consommation sur les ports d'entrée-sortie dans le modèle SDF par exemple – sous la forme d'annotations et ce sans compromettre le caractère générique du langage.

18. Sémantique qui définit en particulier *quand* les paramètres sont évalués pour déclencher la reconfiguration des acteurs concernés.
19. www.graphviz.org
20. Le modèle de calcul doit être de type DDF (Dynamic DataFlow) ou SDF (Synchronous DataFlow) et le code des acteurs atomiques doit être fourni sous la forme de fonctions écrites en C ou C++.
21. https://www.youtube.com/watch?v=WQSacRxYgGM
22. Mais aussi des avis des concepteurs et implémenteurs de langages en général, et fonctionnels en particulier ; c'est d'ailleurs une des motivations de cet article.

Références

[1] B. Bhattacharya and S. Bhattacharyya, "Parameterized dataflow modeling for dsp systems," *Signal Processing, IEEE Transactions on*, vol. 49, pp. 2408 – 2421, 11 2001.

[2] P. Bjesse, K. Claessen, M. Sheeran, S. Singh. Lava : hardware design in Haskell, SIGPLAN Not., 34 :174–84, 1998.

[3] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[4] C. Elliott and P. Hudak. Functional Reactive Animation. International Conference on Functional Programming, 1997.

[5] J. T. Feo A. P. W. Böhm, D. C. Cann and R. R. Oldehoeft. SISAL 2.0 reference manual. Technical Report CS-91-118, Colorado State University, 1991.

[6] M. Gerards, C. Baaij, J. Kuper, M. Kooijman. Higher-order abstraction in hardware descriptions with ClaSH. *Proceedings of the 14th Euromicro Conference on Digital System Design, DSD 2011*, pp. 495-502, 2011. IEEE Computer Society.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[8] C.-J. Hsu, F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya, "Dif : An interchange format for dataflow-based design tools," in *Computer Systems : Architectures, Modeling, and Simulation*, A. D. Pimentel and S. Vassiliadis, Eds. Springer, 2004, pp. 423–432.

[9] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[10] J.R. Mc Graw. The val language : description and analysis. *ACM Trans PLS*, 4 :44–82, 1982.

[11] National Instruments. www.ni.com/labview

[12] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. F. Nezan, and S. Aridhi, "PREESM : A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming," in *EDERC*, Italy, Sep. 2014, p. 36.

[13] M. Pouzet Lucid Synchrone, version 3. Tutorial and reference manual. www.di.ens.fr/~pouzet/lucid-synchrone

[14] N. Sane, H. Kee, G. Seetharaman, and S. Bhattacharyya, "Scalable representation of dataflow graph structures using topological patterns," in *IEEE Workshop On Signal Processing Systems*, 11, 2010, pp. 13–18.

[15] J. Sérot, The HoCL project. Source code and documentation available online at `github.com/jserot/hocl`.

MLANG: an Open-Source Toolchain for the Income Tax Computation*

Denis Merigoux[1] and Raphaël Monat[2]

[1] Inria Paris
denis.merigoux@inria.fr
[2] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
raphael.monat@lip6.fr

Abstract

Once per year, the French Directorate for Public Finances (DGFiP) computes for each citizen the amount of income tax owed to the State, depending on their earnings reported on their tax return. This computation is largely automated and performed by a computer program. The source code is written partly in a custom domain specific language called M (used only by the DGFiP), and published yearly. The other part is written in C, and has never been released. Thanks to a mix of retro-engineering and code refactoring, we are able to present MLANG: an open-source toolchain allowing to reproduce the French income tax computation.

Introduction Each year, French citizens declare their earnings to the Directorate of Public Finances (DGFiP). The DGFiP then computes the amount of income tax due from each fiscal household to the State, according to the rules enacted into law by the French Parliament. The computation of the income tax is complex, both in terms of the core calculation (tax brackets and family quotient) and of the astronomical number of exceptions and tax credit that have to be dealt with. The "simplified earnings declaration"[1] has more than 250 inputs. The implementation of the computation has only been partially published[2] since 2016. The published code consists of approximately 92,000 lines of code, and is written in a custom Domain Specific Language (DSL) called "M", created and managed internally by the DGFiP.

Previous work Since the DGFiP did not publish their internal compiler for M, we have retro-engineered the semantics of M[1] with some input from the DGFiP, paving the way for an open-source compiler, MLANG. However, testing the MLANG-compiled published code on private DGFiP test cases yielded a high error rate, with 438 over 542 tests failing. After careful inspection, we concluded we were missing data. In 2020, we were able to sign an agreement with the DGFiP to privately access the source code of the M compiler. It turned out that once the M files have been compiled into C, the compiler inserts those files into a pipeline, which calls the compiled codebase multiple times, and changes the values of some variables between the calls. From a technical point of view, this pipeline is needed because M does not support user-defined functions. This unpublished pipeline, written in C and large of about 33,000 lines of code, accounts for the French tax computation mechanism called *multiples liquidations*. The DGFiP refuses to publish any of its C code, including the pipeline, arguing security concerns.

However, thanks to our private access to the sources, we were able to fix our compiler through the introduction of a new DSL replacing the pipeline, described below. We believe our compiler is now correct for the 2018-income tax computation (except in the case of litigations, happening for 2% of French households[3]).

*This work is partially supported by the E.R.C. under Consolidator Grant Agreement 681393 — MOPSA.

[1] https://www3.impots.gouv.fr/simulateur/calcul_impot/2020/simplifie/index.htm
[2] https://framagit.org/dgfip/ir-calcul/
[3] https://www.economie.gouv.fr/files/files/directions_services/dgfip/Rapport/2019/ra2019.pdf

Introducing a new DSL We created a second DSL, called M++, that we used to concisely refactor the unpublished C pipeline into a 100-lines-long new source. Since we considered only the case of tax computations for years after 2018, and where litigations are not supported, the M++ code replaces 6,500 lines of unpublished C code. The downscaling factor for the code size can be explained by writing the code in an appropriate high-level DSL.

Compilation Several transformations (shown in Fig. 1) are applied inside Mlang to the M and M++ code until reaching a common intermediate representation called BIR (Backend Intermediate Representation), which is a simple imperative, function-less language with arithmetic computations and conditional statements. Since our compiler manipulates the whole codebase, it is able to perform optimizations and target multiple backends. Given a restricted set of input and output variables, the generated code is optimized through dead code removal and partial evaluation. These simple optimizations allow us to remove at least 58.5% of the BIR program, and even 97.4% of the program when restricting inputs to the "simplified earnings declaration". We can then translate BIR to Python (our only backend to date) or interpret it.

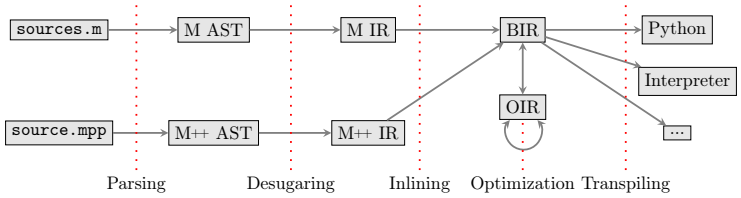


Figure 1: Mlang compilation passes

Correctness of Mlang We evaluated Mlang on the private test set of the DGFiP and passed all of them. We also created our custom test cases using random sampling, and compared the results of Mlang with the internal DGFiP compiler. We did not observe any difference during interpretation of the new test cases.

Conclusion Mlang is an open-source toolchain[4], allowing to reproduce the income tax computation. The whole toolchain is written in OCaml, and is 8,000 LOC, compared to 65,000 LOC for the DGFiP's compiler written in C. We believe that Mlang will be able to replace the ageing compiler used by the DGFiP, which was written in the 1990s. This will allow easier code maintenance, better code analysis support, and the distribution of specialized versions of the income tax computation program in multiple languages. The last point is particularly interesting, as it could increase the level of assurance for a wide range of applications that currently use custom, simplified implementations of the income tax computation like OpenFisca.

References

[1] Denis Merigoux, Raphaël Monat, and Christophe Gaie. Étude formelle de l'implémentation du code des impôts. In *31ème Journées Francophones des Langages Applicatifs*, Gruissan, France, January 2020.

[4] https://github.com/MLanguage/mlang

Cap' ou pas cap' ?

Preuve de programmes pour une machine à capacités en présence de code inconnu

Aïna Linn Georges[1], Armaël Guéneau[1], Thomas Van Strydonck[2], Amin Timany[1], Alix Trieu[1], Dominique Devriese[3], and Lars Birkedal[1]

[1] Aarhus University, Danemark [2] KU Leuven, Belgique [3] Vrije Universiteit Brussel, Belgique

Résumé

Une machine à capacités est un type de microprocesseur permettant une séparation des permissions précise grâce à l'utilisation de *capacités*, mots machine porteurs d'une certaine autorité. Dans cet article, nous présentons une méthode permettant de vérifier la correction fonctionnelle de programmes exécutés par la machine alors même que ceux-ci appellent ou sont appelés par du code inconnu (et potentiellement malveillant). Le bon fonctionnement de tels programmes repose sur leur utilisation judicieuse des capacités. Du point de vue logique, notre approche permet donc de tirer parti des garanties fournies par la machine pour raisonner formellement sur des programmes. Les éléments clefs de cette approche sont la définition d'une logique de programmes puis d'une relation logique dont on démontre qu'elle fournit une spécification pour du code inconnu, le tout étant formalisé en Coq.

La méthodologie en question sous-tend le travail précédent des auteurs lié à la formalisation d'une convention d'appel sûre en présence d'un nouveau type de capacités [GGVS+21], mais n'est pas détaillée dans l'article en question. L'article présent se veut être une introduction pédagogique à cette méthodologie, dans un cadre plus simple (sans nouvelles capacités exotiques), et sur un exemple minimal.

1 Introduction

Une machine à capacités ("*capability machine*") est un certain type de microprocesseur fournissant, en sus des fonctionnalités usuelles, des mécanismes de compartimentalisation mémoire et de séparation des privilèges, à grain fin, sous la formes de capacités matérielles. Ce type d'architecture matérielle est étudié depuis les années 1960 [DVH66, Lev84], et en particulier plus récemment au sein du projet CHERI [WNW+19]. La machine à capacités considérée dans cet article se veut être un modèle simplifié d'une machine de la famille CHERI [1].

Une capacité est une valeur associée à une certaine autorité, permettant par exemple d'accéder à une zone de la mémoire ou d'interagir avec un autre composant du système. Dans une machine à capacités, une capacité est représentable par un mot machine, pouvant être stocké dans les registres ou la mémoire, et dont la machine garantit l'intégrité (il est impossible de contrefaire des capacités). Indépendamment de la représentation en machine, on modèle ici une capacité comme étant un 4-uplet (p, b, e, a), avec p une permission et b, e et a des adresses mémoire, $[b, e[$ correspondant à l'intervalle d'autorité de la capacité, et a étant dans l'intervalle.

Un mot machine est donc soit une capacité, soit un entier. Dans CHERI, un bit supplémentaire est associé à chaque mot afin de distinguer ces deux possibilités. Celui-ci est vérifié et maintenu par la machine, et n'est pas directement accessible par le code exécuté.

1. Une différence notable concerne notre utilisation de capacités "enter" au lieu de paires de capacités "scellées".

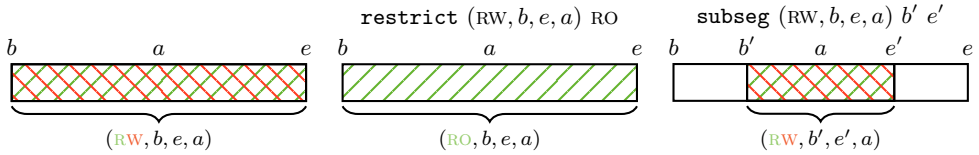


FIGURE 1 – Illustration du comportement de `restrict` et `subseg`.

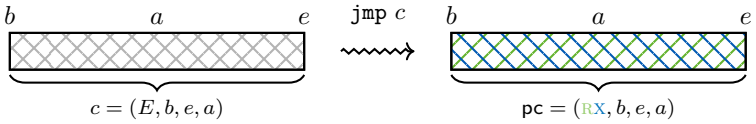


FIGURE 2 – Illustration du comportement de `jmp`.

On peut distinguer différents types de capacités ; on s'intéresse ici aux deux types les plus communs à CHERI. Les *capacités mémoire* donnent l'autorité d'accéder à la plage de mémoire $[b, e[$, avec la permission p (par exemple RW ou RX). Celles-ci sont utilisables comme un pointeur dont l'inclusion entre les bornes et la permission sont directement vérifiés par le matériel. Par ailleurs, étant donné une capacité mémoire, il est possible d'en dériver une nouvelle capacité avec une autorité plus restreinte, en restreignant la permission (avec l'instruction `restrict`) ou en restreignant la plage d'autorité (avec l'instruction `subseg`) comme illustré en Figure 1.

Les *capacités objet* fournissent un mécanisme similaire aux clôtures des langages de haut niveau. Une capacité objet (associée à la permission "enter" ou E) détient l'autorité permettant d'invoquer un certain composant, sans toutefois donner accès aux capacités privées dont celui-ci a besoin pour fonctionner. Invoquer la capacité (via l'instruction `jmp`) exécute le composant en question et change la permission de la capacité de E à RX (Figure 2), donnant par là accès au code et capacités dans sa plage d'autorité, qui étaient auparavant inaccessibles. Dans CHERI, les capacités objet prennent la forme de paires de capacités code et données, qui sont ensuite "scellées" ensemble [WNW+16] ; la formulation que l'on considère ici provient du M-Machine [CKD94] et est légèrement plus simple mais similaire conceptuellement.

Une intuition clef pour comprendre le fonctionnement du système est que l'on utilise une capacité pour accéder à un bloc de mémoire qui peut, lui aussi, contenir d'autres capacités et ainsi de suite. L'autorité d'un programme en train de s'exécuter vient donc des capacités atteignables transitivement à partir du contenu des registres.

Les capacités permettent alors d'interagir de manière sûre avec du code auquel on ne fait pas confiance, en restreignant les capacités auxquelles celui-ci a transitivement accès. Étant donné un système dont les composants ne se font pas tous mutuellement confiance, certains pouvant inclure du code inconnu ou malveillant, les capacités fournissent un moyen de garantir malgré tout que le système obéit à certaines propriétés de sécurité – en initialisant avec soin les composants auxquels on fait confiance, la manière dont ils interagissent avec ceux auxquels on ne fait pas confiance, et en tirant parti des vérifications (dues au capacités) effectuées par la machine à tout instant.

La question est alors : quelles propriétés formelles peut on effectivement faire respecter grâce à l'utilisation des capacités ? Et comment peut on démontrer rigoureusement qu'elles le sont, autrement dit, comment tirer parti des propriétés de la machine à capacités pour raisonner formellement sur l'interaction d'un programme connu avec du code inconnu ?

La réponse proposée ici est la suivante. Tout programme étant appelé par – ou appelant –

du code inconnu peut protéger l'accès à certaines capacités et régions mémoires, en utilisant notamment des capacités objet. On dit alors que ces données protégées constituent son "état privé", sur lequel il peut librement établir et maintenir certaines propriétés, à condition d'avoir correctement restreint l'accès du code inconnu aux données privées. Pour toute propriété de l'état privé qui nous intéresse, il suffit ensuite de vérifier : 1) que cette propriété est un *invariant* du code connu (elle est vraie initialement et préservée par son exécution) et 2) que le code connu satisfait dans son ensemble une spécification de "bonne encapsulation". Alors, par propriété de la machine à capacités, on obtient que cet invariant est préservé lors de l'exécution du système entier, quel que soit le code inconnu interagissant avec le programme connu qui a été vérifié.

Plus précisément, les éléments clefs de cette méthodologie sont les suivants :

— Nous définissons une logique de programme permettant de formellement vérifier la correction de programmes s'exécutant sur notre machine à capacités. Celle-ci est définie à l'aide d'Iris [JKJ+18], une logique de séparation nous fournissant de puissants principes de raisonnement dont notamment la notion d'invariant logique (Section 4).

— Nous définissons, à l'aide de la logique de programme, la spécification de ce que sont une capacité et un programme "sans risque" : une capacité (ou un programme, respectivement) est "sans risque" si elle ne peut pas être utilisée pour invalider un invariant établi précédemment dans la logique. Une capacité sans risque peut donc être partagée librement avec du code inconnu. Cette définition peut être vue comme une relation logique unaire caractérisant notre notion de "sûreté des capacités" (Section 5).

— Nous démontrons (et c'est notre théorème principal) que pour un programme arbitraire, si celui ci n'a accès qu'à des valeurs "sans risque", alors l'exécution du programme lui-même est "sans risque". Ceci est une propriété globale de la machine, exprimant que celle-ci "fonctionne bien" : il n'est pas possible pour un programme d'outrepasser l'autorité reçue initialement, quelque soit la séquence d'instructions qu'il exécute (Section 5).

— La dernière pièce du puzzle est le théorème reliant les invariants établis dans la logique de programme à la sémantique opérationnelle de la machine (Section 4). Étant donné un scénario concret (typiquement, un système mélangeant du code connu et vérifié avec du code inconnu et arbitraire), ceci nous permet d'obtenir *in fine* un théorème élémentaire décrivant son exécution en terme de la sémantique opérationnelle de la machine.

L'objectif de cet article est enfin d'illustrer cette méthodologie en la déployant sur un exemple simple. On introduit l'exemple en Section 2, et on détaille sa preuve en Section 6, après avoir introduit les principes de raisonnement nécessaires. Les résultats et exemples présentés ici ont été intégralement formalisés en Coq, et sont disponibles en ligne : `https://github.com/logsem/cerise`.

2 Motivation

On considère dans cet article un scénario simple, dans lequel on vérifie la correction d'un composant connu interagissant avec un composant "adversaire" composé de code inconnu et auquel on ne fait pas confiance.

Commençons par raisonner dans le cadre d'un langage de haut niveau avec références et fonctions de première classe (on utilise ici une syntaxe OCaml, mais le même exemple s'appliquerait aussi en Javascript entre autres).

$$\text{let } x = \text{ref } 0 \text{ in } (\lambda n.\ \text{if } n \geq 0 \text{ then } x := !x + n)$$

Que dire du programme ci-dessus ? Celui-ci alloue une nouvelle référence x initialisée à 0, et crée une clôture permettant d'augmenter la valeur de la référence en y ajoutant un entier n

$$
\begin{array}{lll}
r \in \mathrm{RegName} ::= \mathsf{pc} \mid \mathsf{r}_0 \mid \mathsf{r}_1 \mid \ldots & & \rho \in \mathbb{Z} + \mathrm{RegName} \\
i & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{move}\ r\ \rho \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ \rho \mid \texttt{add}\ r\ \rho\ \rho \mid \texttt{sub}\ r\ \rho\ \rho \mid \\
 & & \texttt{lt}\ r\ \rho\ \rho \mid \texttt{lea}\ r\ \rho \mid \texttt{restrict}\ r\ \rho \mid \texttt{subseg}\ r\ \rho\ \rho \mid \texttt{isptr}\ r\ r \mid \texttt{getp}\ r\ r \mid \\
 & & \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}
$$

FIGURE 3 – Instructions de notre machine à capacités.

passé en argument, à condition que celui-ci soit positif. Cette clôture est renvoyée au contexte environnant le programme. On s'attend alors que, quelle que soit l'utilisation qui est faite de cette clôture par le contexte environnant (même mal intentionné), la valeur de x sera toujours positive. Et en effet, c'est une propriété qu'il est possible de formuler et prouver formellement [SGD17].

Essentiellement, cette propriété est vraie car une implémentation raisonnable d'un langage de haut niveau (par exemple, OCaml ou Javascript) ne permet pas d'inspecter l'environnement d'une clôture. Donc, avoir accès à la clôture produite par le programme ci-dessus ne donne pas d'accès (direct) à x. Il est seulement possible d'appeler la clôture en lui donnant un argument : celle-ci "encapsule" donc correctement l'état local du programme (la référence x).

La même propriété ("x contient un entier positif à tout instant") est-elle vraie si, après avoir construit la clôture, notre programme passe la main à un contexte adversaire implémenté, non pas en OCaml, mais en assembleur ? La réponse est non : celui-ci peut simplement forger un pointeur vers x, et y écrire un entier négatif, invalidant ainsi la propriété.

Sur une machine à capacités, on peut toutefois préserver cette propriété ! À condition qu'un programme tel que ci-dessus fasse une utilisation judicieuse des capacités objet (une fois compilé ou traduit pour la machine à capacités), alors celui-ci peut passer la main à un contexte arbitraire, écrit en assembleur, sans que celui-ci puisse modifier sa référence privée.

Dans la suite de cet article, on détaille comment la propriété "x est toujours positif" peut être vérifiée formellement, pour une version du programme ci-dessus implémentée en langage machine, et interagissant avec un composant inconnu, également en langage machine. L'implémentation précise de notre programme vérifié apparaît en Figure 6, et sa vérification sera détaillée en Section 6. Dans les sections qui suivent, on présente d'abord brièvement la sémantique de notre machine, puis l'on définit les deux principes de raisonnement clefs nécessaires à la vérification : une logique de programme pour la machine, permettant de vérifier la correction de code connu, et une relation logique et son théorème fondamental, qui donnent une "spécification universelle" pour du code inconnu.

3 Sémantique opérationnelle de la machine

La liste des instructions de la machine est donnée en Figure 3. Les instructions `jmp` et `jnz` correspondent à un saut inconditionnel et un saut conditionnel respectivement ; `mov` copie un mot d'un registre à un autre ; `load` et `store` permettent de lire et d'écrire en mémoire. Les opérations arithmétique sont fournies par `add`, `sub` et `lt`. L'instruction `lea` modifie l'adresse courante d'une capacité en lui ajoutant un entier ; `restrict` permet de restreindre la permission d'une capacité, et `subseg` de restreindre ses bornes. Finalement, `isptr` permet de savoir si un mot est un entier ou une capacité, et `getb`, `getb`, `gete`, `geta` renvoient les différents composants d'une capacité (permission, bornes, et addresse). Les instruction `fail` et `halt` stoppent l'exécution de la machine, dans un état d'erreur ou de succès, respectivement.

La Figure 4 donne un extrait de la sémantique opérationnelle de la machine. L'état de la machine φ correspond à l'état actuel de la mémoire (φ.mem) et des registres (φ.reg). La règle

$$\text{ExecSingle}\quad \varphi \to \begin{cases} [\![\mathit{decode}(z)]\!](\varphi) & \text{si } \varphi.\mathrm{reg}(\mathsf{pc}) = (p,b,e,a) \wedge b \le a < e \wedge \\ & \quad p \in \{\textsc{rx}, \textsc{rwx}\} \wedge \varphi.\mathrm{mem}(a) = z \\ (\mathsf{Failed}, \varphi) & \text{sinon} \end{cases}$$

i	$[\![i]\!](\varphi)$	Conditions
`fail`	$(\mathsf{Failed}, \varphi)$	
`halt`	$(\mathsf{Halted}, \varphi)$	
`move` r ρ	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$w = \mathrm{getWord}(\varphi, \rho)$
`load` r_1 r_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto w])$	$\varphi.\mathrm{reg}(r_2) = (p,b,e,a)$ et $w = \varphi.\mathrm{mem}(a)$ et $b \le a < e$ et $p \in \{\textsc{ro}, \textsc{rx}, \textsc{rw}, \textsc{rwx}\}$
`store` r ρ	$\mathrm{updPC}(\varphi[\mathrm{mem}.a \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,b,e,a)$ et $b \le a < e$ et $p \in \{\textsc{rw}, \textsc{rwx}\}$ et $w = \mathrm{getWord}(\varphi, \rho)$
`jmp` r	$(\mathsf{Standby}, \varphi[\mathrm{reg}.\mathsf{pc} \mapsto \mathit{newPc}])$	si $\varphi.\mathrm{reg}(r) = (\textsc{e}, b, e, a)$, alors $\mathit{newPc} = (\textsc{rx}, b, e, a)$ sinon $\mathit{newPc} = \varphi.\mathrm{reg}(r)$
`restrict` r ρ	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,b,e,a)$ et $p' = \mathrm{decodePerm}(\mathrm{getWord}(\varphi, \rho))$ et $p' \preccurlyeq p$ et $w = (p', b, e, a)$
`subseg` r ρ_1 ρ_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,b,e,a)$ et pour $i \in \{1,2\}$, $z_i = \mathrm{getWord}(\varphi, \rho_i)$ et $z_i \in \mathbb{Z}$ et $b \le z_1$ et $0 \le z_2 \le e$ et $p \neq \textsc{e}$ et $w = (p, z_1, z_2, a)$
`lea` r ρ	$\mathrm{updPC}(\varphi[\mathrm{reg}.r \mapsto w])$	$\varphi.\mathrm{reg}(r) = (p,b,e,a)$ et $z = \mathrm{getWord}(\varphi, \rho)$ et $p \neq \textsc{e}$ et $w = (p, b, e, a + z)$
`geta` r_1 r_2	$\mathrm{updPC}(\varphi[\mathrm{reg}.r_1 \mapsto a])$	$\varphi.\mathrm{reg}(r_2) = (\_, \_, \_, \_, a)$
...		
_	$(\mathsf{Failed}, \varphi)$	sinon

$$\mathrm{updPC}(\varphi) = \begin{cases} (\mathsf{Standby}, \varphi[\mathrm{reg}.\mathsf{pc} \mapsto (p,b,e,a+1)]) & \text{si } \varphi.\mathrm{reg}(\mathsf{pc}) = (p,b,e,a) \\ (\mathsf{Failed}, \varphi) & \text{sinon} \end{cases}$$

$$\mathrm{getWord}(\varphi, \rho) = \begin{cases} \rho & \text{si } \rho \in \mathbb{Z} \\ \varphi.\mathrm{reg}(\rho) & \text{si } \rho \in \mathrm{RegName} \end{cases}$$

Figure 4 – Sémantique opérationnelle : exécution d'une instruction.

ExecSingle décrit un unique pas d'exécution, et, en plus du nouvel état de la machine, indique si celle-ci s'arrête (Halted), échoue (Failed) ou peut continuer son exécution (Standby). La suite de la figure détaille la sémantique d'une partie des instructions, illustrant les vérifications effectuées par la machine, notamment lors des accès à la mémoire (`load`, `store`) ou de la manipulation de capacités (`restrict`, `subseg`, `lea`). Par ailleurs, on peut observer qu'un saut (`jmp`) transforme une capacité avec permission E en une capacité RX lorsque celle-ci est chargée dans `pc`.

Un aspect important de la sémantique opérationnelle concerne donc la gestion des erreurs : lorsqu'une vérification liée aux capacités échoue (par exemple lorsqu'un programme essaye d'utiliser une capacité hors de sa plage d'autorité), la sémantique n'est pas "bloquée" ; à la place, la machine évolue explicitement vers un état "échec".

4 Logique de programme

On définit une logique de programme permettant de raisonner de façon modulaire sur les programmes exécutés par la machine. Il s'agit en particulier d'une logique de séparation, définie comme une instanciation d'Iris [JKJ+18] avec la sémantique opérationnelle de notre machine à capacités.

Dans la logique de programme, on établira des spécifications "modulo échec", qui autorisent le programme à échouer en cours de route. En effet, pour la propriété de sécurité recherchée ici, faire échouer la machine est en fait sûr ! Ce qu'on cherche à garantir est que des invariants du code connu sont préservés lors de l'exécution de code inconnu. Dans cette situation, que la machine échoue n'est à la fois pas un problème (les invariants sont bien préservés si la machine est dans l'état d'échec), et également une possibilité qu'on ne peut exclure (on ne peut empêcher le code inconnu d'échouer un test de capacités).

La logique de programme inclut les connecteurs standard de logique de séparation, dont la conjonction séparante ($*$) et la baguette magique ($-\!\!*$, à lire comme une implication). L'assertion $\mathtt{a} \mapsto w$ exprime la possession d'une case mémoire d'adresse $\mathtt{a}$ contenant le mot machine w, et l'assertion $r \Mapsto w$ exprime la possession d'un registre r contenant w. Un mot machine w est soit un entier, soit une capacité. On note également $\vec{a} \mapsto \vec{l}$ la possession de plusieurs cellules mémoires d'adresses $\vec{a}$ et contenant $\vec{l}$. Le compteur de programme (registre contenant la capacité pointant vers le code actuellement exécuté) est noté $\mathtt{pc}$.

Un élément clef, hérité d'Iris, est la notion d'invariant logique. L'assertion $\boxed{P}$ (duplicable et persistante) exprime que l'assertion P est satisfaite, et continuera de l'être à chaque étape future de l'exécution. Les règles de preuve associées sont standards et héritées d'Iris.

On distingue trois formes différentes de spécifications de programmes :

$\{w; P\} \rightsquigarrow \bullet$	exécution complète
$\{w_0; P\} \rightsquigarrow \{w_1; Q\}$	fragment de code
$\langle w_0; P\rangle \to \langle w_1; Q\rangle$	unique instruction

Dans chaque cas, w, w_0 ou w_1 dénote la valeur du compteur de programme ($\mathtt{pc}$), et P ou Q est une assertion décrivant l'état de la machine. Typiquement, le compteur de programme contient une capacité avec permission RX, pointant vers une zone de mémoire contenant des entiers correspondant au code du programme, et qui sont décodés en des instructions lors de l'exécution (à noter, tenter de décoder une capacité échoue toujours).

On a $\{w; P\} \rightsquigarrow \bullet$ si, partant d'un état machine satisfaisant P et avec $\mathtt{pc}$ égal à w, alors la machine peut s'exécuter jusqu'à s'arrêter (possiblement dans l'état "échec"), ou boucler sans terminer. On a $\{w_0; P\} \rightsquigarrow \{w_1; Q\}$ si, partant d'un état satisfaisant P et avec $\mathtt{pc}$ égal w_0, alors on peut arriver à un état satisfaisant Q avec $\mathtt{pc}$ égal à w_1. On a $\langle w_0; P\rangle \to \langle w_1; Q\rangle$ lorsque ceci résulte de l'exécution d'une unique instruction. (On a alors typiquement $w_1 = w_0 + 1$, sauf dans le cas des instructions `jmp` et `jnz`). Ces trois spécifications requièrent de plus (ceci est implicite au fonctionnement d'Iris mais crucial) *que les invariants de la logique soient préservés à chaque étape de l'exécution.*

À titre d'exemple, on montre ci-dessous un exemple de spécification pour l'instruction `subseg` (SubsegSuccess). Celle-ci n'est en fait pas la spécification la plus générale pour cette instruction : elle correspond au cas où l'exécution n'échoue pas, et demande de prouver les conditions ValidPC et ValidSubseg. Notre logique de programme fournit également des règles (qu'on ne montre pas ici) pour raisonner sur les cas où l'exécution d'une instruction échoue.

SubsegSuccess
$$\frac{\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}, p') \qquad \text{ValidSubseg}(p, b, e, z_1, z_2) \qquad \text{decode}(n) = \texttt{subseg}\ r\ z_1\ z_2}{\begin{array}{l}\langle (p_{pc}, b_{pc}, e_{pc}, a_{pc});\quad a_{pc} \mapsto n * r \Rrightarrow (p, b, e, a)\rangle \to \\ \langle (p_{pc}, b_{pc}, e_{pc}, a_{pc}+1);\ a_{pc} \mapsto n * r \Rrightarrow (p, z_1, z_2, a)\rangle\end{array}}$$

$$\begin{array}{lcl}\text{ValidPC}(p_{pc}, b_{pc}, e_{pc}, a_{pc}, p') & \triangleq & p_{pc} \preccurlyeq p' \wedge p' \in \{\text{RX}, \text{RWX}\} \wedge b_{pc} \le a_{pc} < e_{pc} \\ \text{ValidSubseg}(p, b, e, z_1, z_2) & \triangleq & b \le z_1 \wedge 0 \le z_2 \le e\end{array}$$

Prouver une spécification de la forme $\{w_0; P\} \rightsquigarrow \{w_1; Q\}$ revient à utiliser en séquence une série de règles de la forme $\langle w_0; R\rangle \to \langle w_1; S\rangle$, une pour chaque instruction du bloc de code considéré. Plus généralement, ces trois notions de spécification de programmes se composent des façons auxquelles on peut s'attendre ; par exemple, on a les propriétés suivantes :

SeqFrag
$$\frac{\{w_0; P\} \rightsquigarrow \{w_1; Q\} \qquad \{w_1; Q\} \rightsquigarrow \{w_2; R\}}{\{w_0; P\} \rightsquigarrow \{w_2; R\}}$$

SeqFull
$$\frac{\{w_0; P\} \rightsquigarrow \{w_1; Q\} \qquad \{w_1; Q\} \rightsquigarrow \bullet}{\{w_0; P\} \rightsquigarrow \bullet}$$

StepFull
$$\frac{\langle w_0; P\rangle \to \langle w_1; Q\rangle \qquad \{w_1; Q\} \rightsquigarrow \bullet}{\{w_0; P\} \rightsquigarrow \bullet}$$

StepFrag
$$\frac{\langle w_0; P\rangle \to \langle w_1; Q\rangle \qquad \{w_1; Q\} \rightsquigarrow \{w_2; R\}}{\{w_0; P\} \rightsquigarrow \{w_2; R\}}$$

La dernière pièce du puzzle est le théorème d'adéquation reliant une spécification établie dans la logique de programme à la sémantique opérationnelle de la machine. Son énoncé (ci-dessous) est ici légèrement informel faute d'avoir défini la sémantique de la machine. On le lit de la manière suivante : "si une spécification de la forme $\{w; P\} \rightsquigarrow \bullet$ est établie dans la logique de programme, sous des invariants $\boxed{I_0}$, ..., $\boxed{I_n}$, et pour un état initial de la machine $(regs_0, mem_0)$ qui satisfait P et les invariants, alors ces invariants sont préservés pour tout état ultérieur lors de l'exécution de la machine".

Adéquation
$$\frac{\begin{array}{c}\boxed{I_0}, \ldots, \boxed{I_n} \vdash \{w; P\} \rightsquigarrow \bullet \\ (regs_0, mem_0) \vDash I_0 * \ldots * I_n * \mathsf{pc} \Rrightarrow w * P \qquad (regs_0, mem_0) \longrightarrow^* (regs, mem)\end{array}}{(regs, mem) \vDash I_0 * \ldots * I_n}$$

Pour le lecteur familier avec Iris, un point plus technique mais intéressant est que nos trois notions de spécification sont en fait définies à partir de la notion plus primitive de *plus faible précondition* (wp), fournie par Iris, et qui est définie directement en fonction de la sémantique de la machine. Dans un langage de haut niveau, wp est typiquement paramétré par une expression du langage. Dans le cadre de notre machine à capacités, cette notion d'expression n'existe pas : les programmes sont des données ordinaires en mémoire. À la place, notre notion de wp est paramétrée par des "modes d'exécution", dont SingleStep, correspondant à l'exécution d'une unique instruction, et RepeatSingleStep, correspondant à une exécution complète de la machine.

Les définitions (ci-dessous) montrent que les spécifications pour une unique instruction ($\langle w_0; P\rangle \to \langle w_1; Q\rangle$) et pour une exécution complète ($\{w; P\} \rightsquigarrow \bullet$) correspondent alors à ces deux modes d'exécution de wp. Finalement, la spécification d'un fragment de code ($\{w_0; P\} \rightsquigarrow \{w_1; Q\}$) est définie en style "passage de continuation", d'après la spécification pour une exécution complète.

$$\begin{array}{lcl}\langle w_0; P\rangle \to \langle w_1; Q\rangle & \triangleq & \mathsf{pc} \Rrightarrow w_0 * P \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{SingleStep}\ \{\mathsf{pc} \Rrightarrow w_1 * Q\} \\ \{w; P\} \rightsquigarrow \bullet & \triangleq & \mathsf{pc} \Rrightarrow w * P \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{Repeat}\ \mathsf{SingleStep}\ \{\mathsf{True}\} \\ \{w_0; P\} \rightsquigarrow \{w_1; Q\} & \triangleq & \{w_0; P * \{w_1; Q\} \rightsquigarrow \bullet\} \rightsquigarrow \bullet\end{array}$$

$$\boxed{\mathcal{V}(w)} \begin{cases} \mathcal{V}(z) & \triangleq \mathsf{True} \\ \mathcal{V}(\text{E}, b, e, a) & \triangleq \triangleright\, \Box\, \mathcal{E}(\text{RX}, b, e, a) \\ \mathcal{V}(\text{RO}/\text{RX}, b, e, -) & \triangleq \mathop{\Large *}_{a \in [b,e[} \exists P, \boxed{\exists w,\, a \mapsto w * P(w)} * \triangleright \Box\, (\forall w,\, P(w) \mathrel{-\!\!*} \mathcal{V}(w)) \\ \mathcal{V}(\text{RW}/\text{RWX}, b, e, -) & \triangleq \mathop{\Large *}_{a \in [b,e[} \boxed{\exists w,\, a \mapsto w * \mathcal{V}(w)} \end{cases}$$

$$\boxed{\mathcal{E}(w)} \quad \triangleq\ \forall reg,\ \left\{ w; \mathop{\Large *}_{(r,v) \in reg, r \neq \mathsf{pc}} r \Rightarrow v * \mathcal{V}(v) \right\} \rightsquigarrow \bullet$$

FIGURE 5 – Relation logique définissant la notion de "valeur sûre à partager".

5 Relation logique et théorème fondamental

Notre logique de programmes permet non seulement d'établir la correction fonctionnelle de code connu, mais est également utile pour définir ce qui est notre principe de raisonnement clef pour raisonner à propos de code inconnu.

On donne ainsi une définition de ce qui rend une valeur (un mot machine) "sûre à partager avec du code inconnu". Intuitivement, une valeur est sûre à partager avec un adversaire si elle se conforme à un contrat de sûreté des capacités : elle donne accès exactement à la mémoire sur laquelle elle possède une autorité (défini par son intervalle d'autorité et sa permission), et elle ne peut pas être utilisée pour augmenter cette autorité ou invalider un invariant de la logique.

La définition formelle est donnée en Figure 5. On définit simultanément la notion de "valeur sûre à partager" ($\mathcal{V}$) et "valeur sûre à exécuter" ($\mathcal{E}$).

— *Une valeur sûre à partager* ne donne accès transitivement qu'à des valeurs sûres à partager, ou du code sûr à exécuter (dans le cas d'une clôture).

— *Une valeur sûre à exécuter*, étant donné des valeurs sûres dans les registres, s'exécute sur la machine tout en préservant les invariants Iris (par définition de $\{\cdot;\cdot\} \rightsquigarrow \bullet$).

À proprement parler, cette définition est circulaire. Il est possible de l'énoncer en Iris car il s'agit d'une logique step-indexée, d'où l'utilisation de la modalité $\triangleright$ dans la définition, que le lecteur peut essentiellement ignorer ici.

Un entier (z) est toujours sûr à partager, car il ne peut être utilisé pour accéder à la mémoire.

Une capacité objet E est sûre à partager si le code qu'elle encapsule est sûr à exécuter. Celle-ci peut également être exécutée à tout instant : cette contrainte est représentée par la modalité $\Box$, qui dans Iris s'interprète comme restreignant la définition de $\mathcal{V}(\text{E}, -)$ à être "toujours vraie", et ne pouvant donc être prouvée qu'en fonction d'invariants logiques.

Une capacité RW- donne accès en lecture et écriture à son intervalle : on ne peut que définir son contenu comme étant lui-même sûr à partager. En revanche, une capacité avec une permission RO/RX ne peut pas être utilisée par du code inconnu pour changer les mots dans son intervalle ; dans ce cas, ceux-ci peuvent obéir à tout invariant (P) *au moins aussi restrictif* que $\mathcal{V}$. Intuitivement, les mots dans l'intervalle doivent être sûrs, car ils peuvent être lus par l'adversaire, mais celui-ci ne peut les modifier, donc il est possible de garantir un invariant plus fort. Par exemple, $P(w)$ pourrait être le prédicat "$w = 42$" pour décrire qu'une valeur stockée en mémoire reste toujours égale à l'entier 42.

Cette définition de sûreté est-elle triviale ? Autrement dit, la définition de sûreté donnée en Figure 5 ne serait-elle pas toujours vraie ? La réponse est non ! Toutefois, il n'est pas complètement évident de s'en convaincre.

En première approche, la définition de $\mathcal{E}(w)$ n'est pas triviale car elle nécessite de prouver que, partant de w, une exécution complète de la machine *préserve les invariants de la logique*. Cette contrainte n'est pas visible dans la définition, car implicite à la définition de la logique

de programmes et au fonctionnement d'Iris, mais elle est cruciale. Par ailleurs, la définition de $\mathcal{V}(w)$ n'est pas non plus triviale, car, par exemple dans le cas d'une capacité RW, elle impose que la permission $a \mapsto -$ pour chaque cellule mémoire a soit gouvernée par un invariant spécifique ($\boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)}$). Or une permission "$a \mapsto -$" n'est pas duplicable. Toute cellule mémoire qui est donc gouvernée par un invariant plus spécifique ne peut donc pas être associée à une capacité sûre au sens de $\mathcal{V}$: on ne peut avoir qu'un seul exemplaire de $a \mapsto -$, qui ne peut donc pas faire partie de deux invariants différents.

Quel est donc un exemple de capacité qui n'est *pas* sûre ? Considérons une case mémoire d'adresse x initialisée à 0. Supposons alors qu'on alloue l'invariant Iris suivant : $\boxed{x \mapsto 0}$. Celui-ci exprime que x contiendra l'entier 0 pour le reste de l'exécution. Alors, intuitivement, une capacité $(\text{RW}, x, x+1, x)$ n'est pas sûre à partager avec un adversaire ! En effet, celui-ci pourrait l'utiliser pour écrire une valeur arbitraire à l'adresse x, invalidant l'invariant. Formellement, on ne peut prouver $\mathcal{V}(\text{RW}, x, x+1, x)$, car il n'est pas possible de créer l'invariant $\boxed{\exists w,\ x \mapsto w * \mathcal{V}(w)}$: la ressource pour la case mémoire x fait déjà partie de l'invariant $\boxed{x \mapsto 0}$. De même, on ne peut prouver $\mathcal{E}$ pour tout fragment de code qui écrit une valeur différente de 0 à l'adresse x, car on ne peut justifier dans la preuve la préservation de l'invariant lié à x.

Théorème fondamental. Le théorème fondamental (Théorème 1) est le résultat principal de ce travail : c'est un théorème non trivial, dont la preuve exige d'examiner tous les cas possibles de la sémantique de chaque instruction de la machine. Celui-ci établit que, selon notre définition, tout code "sûr à partager" est en fait également "sûr à exécuter". Ce théorème nous donne donc une spécification pour le comportement de code arbitraire. Tant qu'elle ne donne accès qu'à des mots mémoire sûrs (en particulier, des instructions arbitraires correspondent à des entiers et sont donc toujours sûres), alors une capacité est sûre à exécuter.

Théorème 1 (TFRL)**.** *Soient* $p \in \text{Perm}, b, e, a \in \text{Addr}$. *Si* $\mathcal{V}(p, b, e, a)$, *alors* $\mathcal{E}(p, b, e, a)$.

Une autre interprétation du théorème fondamental est qu'il exprime que la machine "fonctionne bien". Exécuter les instructions ne peut créer plus d'autorité que ce qui était déjà disponible initialement ; le cas contraire serait un bogue de conception de la machine à capacités.

Le lecteur attentif aura remarqué que notre modèle ne distingue pas les capacités -X et celles sans X. Ceci est une conséquence directe du théorème fondamental ! Notre modèle exprime "l'autorité" qu'une portion de code a sur la mémoire. D'après le Théorème 1, être capable d'exécuter une portion de code ne donne pas d'autorité supplémentaire par rapport à seulement savoir le lire.

Pour récapituler, notre relation logique caractérise l'interface entre du code vérifié qui veut préserver des invariants sur un état interne ; et du code "externe" arbitraire dont on a suffisamment restreint les capacités. Le théorème fondamental nous donne une propriété de sûreté pour du code inconnu, et nous permet de vérifier du code connu qui appelle du code adversaire et potentiellement malveillant.

Il est important de noter que la distinction entre code connu et code adversaire est purement logique : elle n'existe pas à l'exécution. On peut avoir deux composants vérifiés séparément et qui ne se font mutuellement pas confiance : dans ce cas, du point de vue de la preuve de chaque composant, l'autre composant sera alors considéré comme le code adversaire.

6 Raisonner en présence de code inconnu : un exemple

(Le code et preuve présentés dans cette section correspondent aux fichiers `adder.v` et `adder_adequacy.v` dans le dossier `theories/examples/` de la formalisation Coq.)

```
;; r1 : capacité vers une zone mémoire où
;;  écrire le code d'activation de la clôture
;; r3 : capacité vers l'adresse x
g: move r2 pc
   lea r2 23 ;; offset vers f
   subseg r2 f f_end ;; restreint au code de f
   ;; crtcls (emplacement) (code) (data)
   crtcls r1 r2 r3
   ;; r1 = clôture (capacité E), r2, r3 = 0
   jmp r0
g_end:

a: move r1 pc
   lea r1 7
   load renv r1
   lea r1 -1
   load r1 r1
   jmp r1
   data 0 ;; sera : capacité de code
   data 0 ;; sera : capacité de données
a_end:
```

```
;; renv : capacité vers l'adresse x
;; r2 : argument passé par l'appelant
;;     (censé être un entier)
f: move r1 pc ;; / r1: addresse de la fin
   lea r1 7   ;; \     du programme
   lt r3 r2 0 ;; a-t-on r2 ≥ 0?
   jnz r1 r3 ;; si non : quitter
   load r3 renv   ;; / si oui : l'ajouter
   add r3 r3 r2  ;; | à la mémoire privée
   store renv r3 ;; \ ...
   move renv 0 ;; / nettoyer les capacités
   move r1 0   ;; \ vers l'état privé
   jmp r0
f_end:
```

FIGURE 6 – Implémentation de notre composant vérifié.
g : code de création de la clôture ; f : corps de la clôture ; a : code d'activation de la clôture. Pour simplifier, on suppose que le code pour g et f se suivent en mémoire, i.e. g_end = f.

Revenons sur l'exemple introduit en Section 2. Pour se simplifier légèrement la tâche et se passer d'une routine auxiliaire d'allocation mémoire, on suppose que notre composant vérifié est donné accès (exclusif) à une cellule mémoire x, et construit alors une clôture qui encapsule l'accès à x. À haut niveau, l'implémentation du composant est donc équivalente à :

$$(\lambda n.\ \text{if } n \geq 0 \text{ then } x := !x + n)$$

Alors, on veut vérifier la propriété suivante : *quelle que soit l'implémentation du composant adversaire, si x contient initialement un entier positif, à tout moment de l'exécution, la valeur de x reste positive*. En effet, l'adversaire n'a pas d'accès direct à x mais seulement à la clôture ci-dessus : si celle-ci est correctement implémentée à l'aide de capacités objet, alors l'adversaire ne peut modifier x que via des appels à la clôture, qui ne peut qu'augmenter la valeur de x.

Le code que l'on vérifie effectivement est une séquence d'instructions machine. Il apparaît en Figure 6, en syntaxe pseudo-assembleur. Le code spécifique au composant est composé de deux routines : g, exécutée initialement et créant la clôture, et f implémentant la clôture elle-même.

La Figure 7 illustre l'agencement de la mémoire avant et après l'exécution de g. Le code de g reçoit dans le registre r_3 une capacité vers x, crée la clôture encapsulant cette capacité, et passe le contrôle à l'adversaire en sautant à la capacité dans la registre r_0, sur laquelle on ne sait rien. Pour créer la clôture, g utilise la macro-instruction `crtcls` (un alias pour une séquence d'instructions qu'on ne détaille pas ici). Celle-ci écrit en mémoire le code d'activation de la clôture, la capacité vers x, et la capacité vers le code de f, et crée une capacité objet (avec permission E) encapsulant le tout. Le code d'activation (qui est exécuté à chaque invocation de la clôture et passe le contrôle à f) est reproduit en Figure 6 (en a). Il n'est pas spécifique à l'exemple considéré ici, et correspond seulement à la manière dont on implémente ici des clôtures.

Lorsque la clôture est invoquée par l'adversaire, le code d'activation (a) est exécuté. Celui-ci se contente de copier dans les registres les capacités pour x et f (stockées à la suite du code).

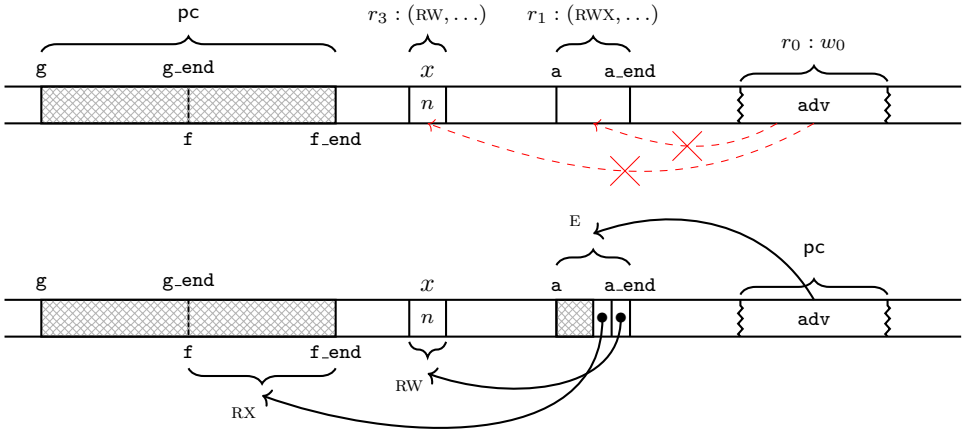


FIGURE 7 – Agencement de la mémoire : 1) initialement, et 2) après l'exécution de g.

$$\boxed{[\mathsf{g}, \mathsf{g\_end}[\mapsto g_{instrs}}$$

$$\vdash \left\{ (\text{RX}, \mathsf{g}, \mathsf{f\_end}, \mathsf{g}); \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (\text{RWX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto - * \\ r_3 \Mapsto (\text{RW}, x, x+1, x) * [\mathsf{a}..\mathsf{a\_end}] \mapsto [-] \end{array} \right\} \rightsquigarrow$$
$$\left\{ w_0; \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (\text{E}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto 0 * r_3 \Mapsto 0 * \\ [\mathsf{a}..\mathsf{a\_end}] \mapsto (a_{instrs} +\!\!+ [(\text{RX}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f})] +\!\!+ [(\text{RW}, x, x+1, x)]) \end{array} \right\}$$

$$\boxed{[\mathsf{f}, \mathsf{f\_end}[\mapsto f_{instrs}}, \boxed{\exists n,\, x \mapsto n \wedge n \geq 0}$$

$$\vdash \left\{ (\text{RX}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f}); \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto - * r_2 \Mapsto k * r_3 \Mapsto - * \\ r_{\text{env}} \Mapsto (\text{RW}, x, x+1, x) \end{array} \right\} \rightsquigarrow$$
$$\left\{ w_0;\ \exists k'\, n', \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto 0 * r_2 \Mapsto k' * r_3 \Mapsto n' * \\ r_{\text{env}} \Mapsto 0 \end{array} \right\}$$

$$\boxed{[\mathsf{a}, \mathsf{a\_end}[\mapsto (a_{instrs} +\!\!+ c_{code} +\!\!+ c_{data})}$$

$$\vdash \quad \left\{ (\text{RX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a});\ r_1 \Mapsto - * r_{\text{env}} \Mapsto - \right\} \rightsquigarrow \left\{ c_{code};\ r_1 \Mapsto c_{code} * r_{\text{env}} \Mapsto c_{data} \right\}$$

FIGURE 8 – Spécifications pour les routines g, f et a.
On note g_{instrs}, f_{instrs} et a_{instrs} les listes de leurs instructions encodées en mots machine.

Il invoque ensuite f, qui utilise la capacité vers x pour modifier sa valeur (le cas échéant), et prend ensuite soin de nettoyer les registres des capacités temporaires avant de rendre le contrôle à l'adversaire (par convention, r_0 contient le pointeur de retour).

L'étape suivante est d'énoncer et prouver une spécification pour chaque routine (Figure 8). On ne détaille pas les preuves : il s'agit ici d'un exercice standard de preuves de programmes. Les spécifications sont vérifiées en parcourant le code de f, g et a, et en utilisant successivement la spécification de chaque instruction machine rencontrée. Notons l'utilisation de plusieurs invariants : un pour le code de chaque programme (qui doit être en mémoire et accessible), et, plus important, un invariant contraignant la valeur stockée à l'adresse x à être un entier positif.

Il reste alors la partie la plus intéressante de la preuve : combiner les spécifications des routines individuelles avec la spécification du code inconnu (donnée par le Théorème 1), afin

d'obtenir une spécification pour une exécution complète du système et finalement appliquer le théorème d'adéquation, obtenant ainsi que x contient bien un entier positif à tout instant, par préservation de l'invariant idoine. Les étapes principales du raisonnement sont comme suit.

Le but est de montrer la spécification suivante pour une exécution complète de la machine. Cette spécification devant être établie sous l'invariant logique $\boxed{\exists n, x \mapsto n \wedge n \geq 0}$, par le théorème d'adéquation, cela implique alors directement la propriété sur x voulue.

$$\boxed{[\mathsf{g}, \mathsf{g\_end}[\mapsto g_{instrs}]}, \boxed{[\mathsf{f}, \mathsf{f\_end}[\mapsto f_{instrs}]}, \boxed{\exists n, x \mapsto n \wedge n \geq 0}$$
$$\vdash \quad \left\{ (\mathrm{RX}, \mathsf{g}, \mathsf{f\_end}, \mathsf{g}); \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (\mathrm{RWX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto -* \\ r_3 \Mapsto (\mathrm{RW}, x, x+1, x) * [\mathsf{a}, \mathsf{a\_end}[\mapsto [-] * \\ \mathcal{V}(w_0) * \ast_{(r,v) \in reg, r \notin \{\mathsf{pc}, r_0 .. r_3\}} r \Mapsto v * \mathcal{V}(v) \end{array} \right\} \rightsquigarrow \bullet$$

Cette spécification requiert les ressources nécessaires à l'exécution de g (cf. la précondition de g en Figure 8), mais aussi que w_0 (le pointeur vers le code inconnu) et les valeurs contenues dans le reste des registres soient des mots machines sûrs. Par exemple, on peut préalablement vérifier que w_0 est une capacité vers une zone de mémoire ne contenant que du code et des données (donc pas d'autres capacités), et initialiser le reste des registres à zéro : d'après la définition de $\mathcal{V}$, un entier est toujours sûr, de même qu'une capacité pointant sur une région contenant des entiers. En fait, par composition avec la spécification de g (Figure 8), il suffit de raisonner sur la continuation de g. Il suffit donc de montrer :

$$\boxed{[\mathsf{f}, \mathsf{f\_end}[\mapsto f_{instrs}]}, \boxed{\exists n, x \mapsto n \wedge n \geq 0}$$
$$\vdash \quad \left\{ w_0; \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (E, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto 0 * r_3 \Mapsto 0 * \\ [\mathsf{a}, \mathsf{a\_end}[\mapsto (a_{instrs} +\!\!+ [(\mathrm{RX}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f})] +\!\!+ [(\mathrm{RW}, x, x+1, x)]) * \\ \mathcal{V}(w_0) * \ast_{(r,v) \in reg, r \notin \{\mathsf{pc}, r_0 .. r_3\}} r \Mapsto v * \mathcal{V}(v) \end{array} \right\} \rightsquigarrow \bullet$$

Or, puisque w_0 est sûr (on a $\mathcal{V}(w_0)$), d'après le Théorème 1, il est sûr à exécuter (donc on a $\mathcal{E}(w_0)$). En dépliant la définition de $\mathcal{E}$, on obtient le but voulu (la spécification ci-dessus), à condition de pouvoir prouver que la valeur de chaque registre est elle-même sûre. La preuve est immédiate pour tous les registres sauf r_1, qui pointe sur notre clôture : puisque l'on partage celle-ci avec le code inconnu, il nous incombe de prouver qu'elle est sûre.

Il reste donc à prouver $\mathcal{V}(\mathrm{E}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$. On prend soin de placer les ressources correspondant au code d'activation de la clôture (entre a et a_end) dans un nouvel invariant. Alors, sans rentrer dans les détails liés aux modalités $\Box$ et $\triangleright$, il suffit d'établir $\mathcal{E}(\mathrm{RX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$.

En composant les spécifications de a et f, on peut vérifier que celles-ci ne supposent rien à propos des valeurs contenues initialement dans les registres (dont on sait seulement qu'elles sont sûres) ; et que de même, les valeurs contenues dans les registres après l'exécution de f sont sûres (soit ce sont des entiers, soit elles n'ont pas été modifiées). On a donc :

$$\boxed{[\mathsf{f}, \mathsf{f\_end}[\mapsto f_{instrs}]}, \boxed{\exists n,\ x \mapsto n \wedge n \geq 0},$$
$$\boxed{[\mathsf{a}, \mathsf{a\_end}[\mapsto (a_{instrs} +\!\!+ (\mathrm{RX}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f}) +\!\!+ (\mathrm{RW}, x, x+1, x))}$$
$$\vdash \quad \begin{array}{l} \{(\mathrm{RX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a});\ r_0 \Mapsto w_0 * \mathcal{V}(w_0) * \ast_{(r,v) \in reg, r \neq \mathsf{pc}, r_0} r \Mapsto v * \mathcal{V}(v)\ \} \rightsquigarrow \\ \{w_0;\ \ast_{(r,v) \in reg, r \neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v)\ \} \end{array}$$

Pour établir $\mathcal{E}(\mathrm{RX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$, il suffit alors de raisonner sur la continuation de f (f retournant à l'adversaire en exécutant le pointeur de retour w_0) ; c'est à dire, il suffit d'établir : $\left\{ w_0; \ast_{(r,v) \in reg, r \neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v) \right\} \rightsquigarrow \bullet$. Or, w_0 est supposé sûr (par définition de $\mathcal{E}$) : on a $\mathcal{V}(w_0)$. Le Théorème 1 donne alors $\mathcal{E}(w_0)$, ce qui est exactement ce qu'il restait à prouver. Qed.

Théorème final. Via le théorème d'adéquation, on obtient alors le théorème suivant pour l'exécution de la machine dans notre scénario, exprimé en fonction de la sémantique opérationnelle :

Théorème 2 (Exécution correcte de la machine)**.** *En partant d'un état initial de la machine* (reg, mem) *où :*

- *mem a été initialisée avec le code de* g *et* f*, et du code inconnu pour l'adversaire (entre les adresses* adv *et* adv_end*) (Figure 7) ;*
- $reg(\mathsf{pc}) = (\text{RX}, \mathsf{g}, \mathsf{f\_end}, \mathsf{g})$, $reg(r_0) = (\text{RWX}, \mathsf{adv}, \mathsf{adv\_end}, \mathsf{adv})$, $reg(r_1) = (\text{RWX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$, $reg(r_3) = (\text{RW}, x, x+1, x)$*, et* $reg(r) \in \mathbb{Z}$ *dans les autres cas ;*
- $mem(x)$ *est un entier positif.*

Alors, pour tous reg', mem'*, si* $(reg, mem) \longrightarrow^* (reg', mem')$ *alors* $mem'(x)$ *est un entier positif.*

En d'autres termes, pour une machine correctement initialisée, alors l'invariant établi dans la logique à propos de x est vrai à chaque étape de l'exécution : à tout instant, la valeur stockée en mémoire à l'adresse x est un entier positif.

Ce théorème est-il satisfaisant ? On peut anticiper deux commentaires possibles à propos du théorème ci-dessus, qui suggéreraient que celui-ci n'est pas aussi général que l'on pourrait vouloir, et auxquelles on répond ci-dessous.

Le théorème fait-il des hypothèses trop spécifiques à propos de l'état initial de la machine ? Le Théorème 2 ne détaille en effet pas la manière dont la mémoire est initialisée avec le code pour g et f, ou le code adversaire. De même, on peut se demander d'où viennent les capacités que l'on suppose ici être présentes dans les registres pc, r_0, r_1 et r_3. Quel est l'état initial d'une machine à capacités, immédiatement après sa mise sous tension ?

Les détails précis dépendent de l'implémentation matérielle ; mais invariablement, une machine à capacités doit initialement fournir une "capacité omnipotente" donnant autorité sur l'ensemble de la mémoire (ici, ce serait $(\text{RWX}, 0, \mathsf{addr\_max}, 0)$). C'est alors le rôle du code de démarrage de la machine que de restreindre et diviser cette capacité et de la distribuer aux différents composants du programme. Le Théorème 2 suppose que l'on se place immédiatement après l'exécution du code de démarrage de la machine, afin de s'abstraire des détails d'implémentation liés à l'initialisation de la machine. Étant donné une implémentation concrète du code de démarrage, ce serait un exercice standard de vérification de programmes que de vérifier sa correction et le connecter au théorème établi ici.

Ne serait-il pas plus général de commencer par l'exécution du code inconnu plutôt que du code connu ? Pour les raisons détaillées précédemment, il est nécessaire de faire confiance au code exécuté au démarrage de la machine. Si celui-ci est considéré inconnu (ou un adversaire), alors il est impossible de garantir quoi que ce soit, puisque celui-ci a accès à une capacité omnipotente. Il est donc nécessaire d'exécuter une partie du code d'initialisation au démarrage de la machine (ici, la création des clôtures), avant de passer le contrôle à l'adversaire. Le scénario détaillé ici requiert du code de démarrage de la machine que celui-ci ait déjà préparé un certain nombre de régions mémoires pour son fonctionnement (la cellule à l'adresse x et la région pour le code d'activation). L'exemple du compteur présenté ensuite en Section 7 a des prérequis différents pour le code de démarrage : l'implémentation du compteur (code d'initialisation et clôtures) alloue elle même la mémoire nécessaire à la demande, mais nécessite que le système inclue une routine supplémentaire (`malloc`) implémentant un allocateur mémoire : le travail du code de démarrage est alors de fournir au compteur un pointeur vers cette routine.

On peut donc imaginer divers scénarios distribuant différemment les responsabilités entre code exécuté initialement ou invoqué via le code inconnu, mais il est nécessaire dans tous les cas d'exécuter une section de code d'initialisation connu (et vérifié) au démarrage de la machine.

7 Études de cas

En plus de l'exemple détaillé dans la section précédente, nous avons implémenté et vérifié les exemples suivants. Pour simplifier, on se contente ici de présenter une version haut niveau de leur implémentation, la formalisation Coq contenant les détails. Dans chaque cas, on vérifie que les assertions n'échouent pas. Plus précisément, chaque exemple est muni d'une routine auxiliaire "assert", maintenant une cellule mémoire interne, initialisée à 0 et mise à 1 en cas d'appel à assert avec une condition fausse. On prouve alors que cette cellule contient 0 à tout moment de l'exécution.

Compteur pouvant être incrémenté, lu et réinitialisé On vérifie un compteur comportant une référence privée (la valeur actuelle du compteur), et exposant trois clôtures, pour respectivement incrémenter le compteur, lire sa valeur, et la réinitialiser à zéro.

$$\begin{array}{l}\text{let } x = \text{alloc } 0 \text{ in} \\ \quad (\lambda().\ x := !x + 1),\ (\lambda().\ \text{assert } (!x \geq 0);\ !x),\ (\lambda().\ x := 0)\end{array}$$

Les trois points d'entrée sont vérifiés indépendamment, et utilisent le même invariant à propos de x que dans l'exemple précédent. Contrairement à l'exemple précédent, la mémoire pour x est ici allouée dynamiquement, en faisant appel à une routine auxiliaire "alloc". Le théorème final requiert donc seulement que la routine d'allocation soit initialisée en mémoire, et ne demande pas au code de démarrage de réserver de la mémoire pour le fonctionnement du compteur. Comme précédemment, les clôtures sont passées à un contexte adversaire composé d'instructions arbitraire, et on peut montrer que l'assertion n'échoue pas. Pour plus de généralité, on prouve également que le point d'entrée de la routine "alloc" est sûr, et on donne accès à "alloc" à l'adversaire.

Partage d'une capacité read-only (RO) On vérifie un exemple illustrant l'utilisation d'une capacité RO (en lecture seule).

$$\begin{array}{l}\text{let } x = \text{alloc } 1 \text{ in} \\ \text{let } y = \text{restrict } x \text{ RO in} \\ \text{unknown\_code}(y); \\ \text{assert } (!x = 1); \\ \text{halt}()\end{array}$$

Dans cet exemple, "unknown_code" est une fonction inconnue, et "restrict x RO" restreint la capacité x (qui a la permission RWX car renvoyée par alloc) à une permission RO. Ici, l'adversaire correspond à la fonction unknown_code. Selon le modèle, on peut raisonner sur l'exécution de foo tant que l'on sait que unknown_code est une capacité sûre à partager (concrètement : elle n'a pas d'accès direct à x). Dans ce cas, on sait (soit par la définition de validité d'une capacité E, soit par le théorème fondamental) que unknown_code est sûr à exécuter, tant que les valeurs partagées avec celui-ci sont sûres. On doit donc montrer que la capacité y est sûre. D'après la définition de sûreté (Figure 5), on doit donc montrer :

$$\exists P,\ \boxed{\exists w,\ y \mapsto w * P(w)} * \triangleright \square\, (\forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w))$$

Autrement dit, on peut décrire la mémoire pointée par y en choisissant un prédicat P *au moins aussi restrictif* que $\mathcal{V}$. Pour montrer que l'assertion n'échoue pas, on choisit ici $P(w) \triangleq w = 1$.

Ce prédicat satisfait la condition $\triangleright \Box (\forall w,\, P(w) \mathrel{-\!\!*} \mathcal{V}(w))$ (1 est toujours sûr à partager), et nous permet de montrer que x pointe vers l'entier 1 après l'appel au code inconnu.

8 Travaux connexes

Cet article présente une version simplifiée de la méthodologie précédemment mise en place par les auteurs pour raisonner à l'aide d'Iris à propos d'une convention d'appel sûre [GGVS+21]. L'utilisation seule des capacités objets ne permet en effet pas d'implémenter au niveau assembleur des appels de fonction (vers un adversaire) qui soient fidèles à la notion d'appel de fonction d'un langage de haut niveau. Si les capacités objets permettent d'implémenter une forme d'encapsulation d'état local, elles ne garantissent pas que l'ordre des appels et retours de fonction soit bien parenthésé. Notamment, elles n'empêchent pas un adversaire d'invoquer plusieurs fois un pointeur de retour fourni par un appelant (chose impossible dans un langage de haut niveau sans opérateurs de contrôle).

Skorstengaard et al. [SDB19] montrent qu'il est possible d'implémenter une convention d'appel fidèle en utilisant un type additionnel de capacités "locales". L'article en question suit une méthodologie similaire à celle décrite ici, définissant une relation logique caractérisant une certaine notion de sûreté. Les preuves ne sont toutefois pas mécanisées et les détails de la relation logique sont relativement difficile à suivre.

L'article ultérieur de Georges et al. [GGVS+21] introduit d'une part un nouveau type de capacités ("non-initialisées") pour améliorer l'efficacité de la convention d'appel de Skorstengaard et al., et utilise Iris pour formuler une définition de sûreté comme une relation logique et mécaniser les preuves correspondantes. L'utilisation d'Iris permet la relation logique d'être exprimée de façon plus concise et à un plus haut niveau que dans le travail de Skorstengaard et al. En comparaison avec l'article présent, celle-ci est toutefois significativement plus compliquée que celle présentée ici, car plus expressive : elle permet de raisonner sur des propriétés de "bon parenthésage" des appels de fonction.

Certains langages de haut niveau permettent également l'utilisation de "capacités objet" pour protéger un état privé lors de l'interaction avec du code arbitraire. (Typiquement implémentées grâce à l'encapsulation fournie par les clôtures du langage.) Devriese et al. [DBP16] définissent une notion de "sûreté des capacités" pour un sous-ensemble de Javascript (incluant une notion d'effets observables) à l'aide d'une relation logique, et montrent qu'elle permet de raisonner sur différents exemples concrets. En terme d'expressivité, leur relation logique est plus proche de celle pour une convention d'appel sûre [GGVS+21] que celle présentée ici. Elle est toutefois énoncée sur papier et n'a pas été mécanisée.

Plus récemment, Swasey et collaborateurs [SGD17] présentent une logique de programme permettant de raisonner sur une forme de "capacités objets" dans un langage de haut niveau. Leur méthodologie est extrêmement similaire à la notre : les principes de raisonnement logiques sont essentiellement les mêmes, mais ils se placent dans le cadre d'un langage de haut niveau, là où nous utilisons des capacités objet sur une machine à capacités.

Par exemple, Swasey et al. définissent deux prédicats pour décrire une référence : un prédicat pour des références "haute intégrité" ($\ell \hookrightarrow v$) et un pour des références "faible intégrité" ($\mathsf{lowloc}\ v$). Les premières donnent accès exclusif à la référence et ne sont pas partageables avec un adversaire ; les deuxièmes sont partageables avec un adversaire mais ne peuvent être utilisées que pour lire et écrire des valeurs "faible intégrité". Dans notre formalisation, une référence "haute intégrité" correspond alors à une ressource mémoire $a \mapsto w$, et une référence "faible intégrité" correspond à l'invariant utilisé dans la définition de $\mathcal{V}$: $\boxed{\exists w, a \mapsto w * \mathcal{V}(w)}$. Via cette correspondance, nos définitions satisfont les mêmes règles de raisonnement que celles

énoncées par Swasey et al. ; en particulier, les différents "object capability patterns" qu'ils vérifient seraient également implémentables et vérifiables de manière similaire dans le cadre d'une machine à capacités.

Nienhuis et al. [NJB+20] vérifient formellement un certain nombre de propriétés "architecturales" des machines à capacités CHERI. Il s'agit d'un effort de formalisation conséquent : les auteurs considèrent une sémantique détaillée et réaliste de CHERI, significativement plus complexe que le modèle minimal que l'on utilise ici. L'approche de Nienhuis et al. est différente de la notre : ceux-ci énoncent les propriétés de sécurité qu'ils vérifient comme des propriétés de trace, en fonction d'une trace d'"actions abstraites" décrivant les capacités transitant dans la machine. Cette approche permet de formuler les propriétés voulues de façon très concrète et explicite. Par exemple, ceux-ci énoncent et prouvent une propriété de "monotonicité des capacités" : lors de l'exécution, l'autorité des capacités accessibles ne peut pas augmenter (autrement dit, la machine n'autorise pas à forger des capacités). Cela semble être une propriété raisonnable et nécessaire au bon fonctionnement de la machine à capacités. Pourtant, formellement, cette propriété est invalidée par les appels entre composants (dans notre cas, sauter à une capacité E). La propriété prouvée par Nienhuis et al. est donc restreinte à un fragment de trace ne comportant pas d'appel à un composant séparé. Notre méthodologie est moins explicite, mais plus expressive. L'énoncé (très extensionnel) de notre théorème fondamental, qui peut être vu comme un théorème de "bon fonctionnement de la machine", est difficile à comprendre en terme de la sémantique opérationnelle de la machine. Malgré tout, celui-ci permet *in fine* de raisonner sur une exécution complète de la machine avec un nombre arbitraire d'appels entre composants différents.

9 Conclusion

Nous avons présenté une méthodologie pour la vérification de certaines propriétés de sûreté pour du code machine s'exécutant sur une machine à capacités. Déjà mentionné précédemment, une extension possible de ce travail est la prise en compte de modes d'intéractions plus riches entre code connu et code adversaire, avec par exemple notre étude d'une convention d'appel sûre mettant en jeu des capacités supplémentaires et un modèle logique plus élaboré. Même en se contenant des capacités considérées ici, une extension possible serait de considérer une version binaire de la relation logique présentée ici. Une relation logique binaire permettrait de prouver des propriétés de "confidentialité" (du type "l'adversaire ne peut pas lire la valeur à l'adresse x") que l'on ne peut directement obtenir avec le modèle présenté ici.

Finalement, on peut difficilement imaginer vérifier la correction d'un programme de taille conséquente implémenté en code machine, pour la même raison qu'en pratique, peu de programmes sont directement implémentés en assembleur. Il semble donc désirable de tenter d'appliquer notre méthodologie à la vérification d'un compilateur d'un langage de plus haut niveau vers une machine à capacités—compilateur qui devrait alors préserver les propriétés de sécurité établies à propos du programme source.

Remerciements Merci à Léon Gondelman et Pierre Pradic pour les commentaires et remarques sur des versions précédentes de ce document. Ce travail a été soutenu en partie par une subvention Villum Investigator (no. 25804), Center for Basic Research in Program Verification (CPV), de la VILLUM Foundation ; par le Fonds de la Recherche Scientifique - Flandre (FWO) sous numéro G0G0519N ; et par le projet DFF 6108-00363 du Danish Council for Independent Research for the Natural Sciences (FNU). Thomas Van Strydonck est titulaire d'une bourse de recherche du Fonds de la Recherche Scientifique - Flandre (FWO). Amin Timany a

été titulaire d'une bourse postdoctorale du Fonds de la Recherche Scientifique - Flandre (FWO) pendant certaines parties de ce projet.

Références

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–327. ACM, 1994.

[DBP16] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about Object Capabilities Using Logical Relations and Effect Parametricity. In *European Symposium on Security and Privacy*. IEEE, 2016.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3) :143–155, March 1966.

[GGVS+21] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and Provable Local Capability Revocation using Uninitialized Capabilities. In *POPL*, 2021. (Conditionally accepted).

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up : A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28 :e20, 2018.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, 1984.

[NJB+20] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security : Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*, May 2020.

[SDB19] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a Machine with Local Capabilities : Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems*, 42(1) :5 :1–5 :53, December 2019.

[SGD17] David Swasey, Deepak Garg, and Derek Dreyer. Robust and Compositional Verification of Object Capability Patterns. In *OOPSLA*. ACM, 2017.

[WNW+16] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro*, 36(5) :38–49, September 2016.

[WNW+19] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions : CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 2019.

Mastering Program Verification using Possession and Prophecies

Xavier Denis

Université Paris-Saclay, CNRS, Inria, Lab. de Recherche en Informatique, 91405, Orsay, France.

Abstract

Deductive program verification seeks to eliminate bugs in software by translating programs annotated with specifications into logical formulas which are then solved using semi-automated tools. When verifying programs using a mutable heap, it is often required to show that pointers do not alias each other, ensuring there is only one way to modify structures in memory. This leads to cumbersome proof obligations and makes verification much more challenging. Newer languages like Rust feature pointers as well but prevent aliasing through the type system. This opens the door to simpler approaches to verification, free of tedious proof obligations.

We propose a technique for the verification of Rust programs by translation to a functional language. The challenge of this translation is the handling of mutable borrows, pointers with control of aliasing in a region of memory. To overcome this, we used a technique inspired by prophecy variables to predict the final values of borrows. The main contribution of this work is to prove this translation correct. We developed a proof-of-concept tool to show the viability of this approach.

1 Introduction

Over the past 50 years, programming languages have made major strides, allowing programmers to reason about and abstract ever larger software projects. Yet, when performance becomes a concerns, they resort to low-level programming languages like C/C++ or even assembly. These languages offer control over memory and in particular allow unrestricted usage of *pointers* to build complex data structures and efficient algorithms. This power comes at a cost. Reasoning about the correctness of pointer programs is very tricky.

Pointers introduce new challenges, they can be *invalid*, pointing to uninitialized memory and they can *alias* causing variables to observe each other's writes. When two variables are aliased, changing either also changes the other. This makes it difficult to reason about code as programmers must keep in mind all potential aliases to understand the state of their programs. For example, Figure 1 performs a swap using XOR to avoid allocating a temporary variable[3]. But there's a bug! If the user provides the same pointer to both arguments, this function will write 0 to it instead.

```
void XorSwap( int* x, int* y ) { *x ^= *y;
*y ^= *x;  *x ^= *y; }

fn xor_swap(x: &mut i32, y: &mut i32) {
  *x ^= *y; *y ^= *x; *x ^= *y; }
```

Figure 1: C Swap implemented with XOR which assumes arguments are non-aliased (top) and its Rust translation (bottom)

```
fn main() {
    let mut x = 0;
    let y = &mut x;
    inc(y); inc(y);
    assert_eq!(*y, 2);
}
```

```
fn inc(x: &mut i32) {
    *x += 1;
}
```

Figure 2: Incrementing a reference twice

In many programs, the full flexibility of C pointers isn't required or even desirable, there is no need to allow aliasing or invalid pointers. This insight is one of the motivating reasons behind Rust, which makes systems programming simpler by restricting the kinds of pointers we can use.

1.1 The Rust Programming Language

The Rust programming language is a recent entry in the field of systems programming. Its 1.0 release, published in 2015[2] aims to "empower everyone to build reliable and efficient software"[1]. The designers of the language identified memory safety as a key difficulty of systems programming. To solve this, Rust uses a type system with *ownership* to guarantee memory safety and data-race freedom without a garbage collector[10]. This type system eliminates common sources of errors like double-freeing, invalid and null pointers and mutable aliasing.

When the `xorSwap` program of Figure 1 is translated to Rust, as shown in Figure 1, the bug is no longer possible: a call to that function using the same argument twice will trigger a type checking error.

Ownership in Rust In Rust every memory cell has a unique *owner*, which has exclusive read and write permissions. To perform memory mutation, Rust uses safe pointers called *references* that can *borrow* these permissions from their owner. In Figure 2 we create a mutable borrow y which we use to increment the value of the memory cell x. Once y is created, it becomes an error to use x for the duration of the borrow's *lifetime*. After the lifetime ends we can begin using x once again, for example to assert it has been incremented. These borrows can come in one of two forms: either unique and mutable or shareable and immutable.

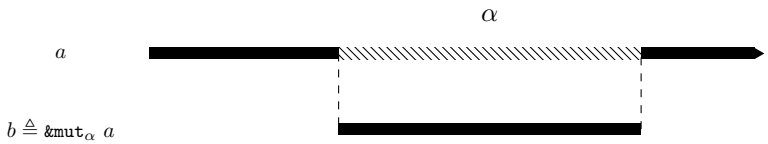


Figure 3: A view of lifetimes

Lifetimes The type of each borrow is annotated with a *lifetime*, a *static* range in the program execution for which permissions have been granted. When the borrow b is created in Figure 3, a transfers its permissions into b and cannot be accessed for the entirety of the lifetime α. However, the lifetime α of b is not the same as the *liveness* of b. By *liveness* we mean in the sense of *live variable analysis*[12]. In this sense a variable is *live* if it may be used in the future. However, a borrow's lifetime can outlive the variable it was stored in.

```
fn proj_togle<'a>(toggle: bool, a: &'a mut u32, b: &'a mut u32) -> &'a mut u32 {
  if toggle {
    a
  } else {
    b
  }
}
```

Figure 4: Selecting a borrow in a non-statically introspectable way

To illustrate this difference, consider Figure 4. We are given a function that takes as arguments two borrows for a lifetime `'a`, returning one of them based on an unknown toggle value. In either branch of the if expression it is evident that the *liveness* of `a` and `b` are different, but their *lifetimes* are the same. Though in each branch one of the variables *dies*, the *ownership* that was borrowed will only be restored at the end of `'a`.

1.2 Verification

Deductive verification of programs works by translating a program and its specification into a collection of *verification conditions* such that their truth implies the program follows its specification. These verification conditions can be discharged using a variety of tools, though commonly, automated SMT solvers are used. This highly automated approach to program verification is appealing because it allows engineers to focus their attention on developing a specification rather than proving logical formulas.

```
/*@ requires \valid(x) && \valid(y) && \seperated(x, y)
  @ ensures \old(*x) == *y \old(*y) == x */
void XorSwap( int* x, int* y ) { *x ^= *y;  *y ^= *x;  *x ^= *y; }
```

Figure 5: XOR Swap with its ACSL[5] specification

When we verify C programs, these verification conditions must include checking that pointers point to valid memory and often require proving that two pointers don't alias. If we consider Figure 1, we could use a specification language like ACSL[5] to describe the desired swapping behavior. When we do that in Figure 5, we must also assume that our input values are valid and non-aliasing or our specification would be false! In Rust we get these hypotheses for free from the type system.

1.3 Contributions

In the rest of this paper we will show how we can leverage the properties of Rust's type system to give a simpler method of deductive verification for Rust programs. We will do this by translating Rust programs into a functional language with non-determinism. This translation handles mutable, immutable borrows and owned pointers. We first formalize and present MiniMir in Section 2, a kernel for languages with borrows and lifetimes. Then in Section 3 we present a translation of MiniMir to a functional language with non-determinism and present a sketch of the safety proof of the translation. In Section 4, we quickly discuss a proof-of-concept implementation and showing our approach is capable of verifying real Rust programs.

2 The MiniMir language

Rust is a complex and modern programming language, incorporating many advances and ideas from the past decades of language design. These features make the compilation of Rust quite complicated, so MIR was introduced. MIR simplifies Rust programs by breaking down expressions into primitive statements that resemble a typed assembly, and by representing programs using their control flow graph. This desugaring is not purely for syntactic reasons, since the 2018 introduction of *non-lexical lifetimes*, the Rust borrow-checker is formulated most naturally on the MIR graph.

The small size of the MIR language and its relatively simple type system make it an ideal input language for verification. In this paper we will first describe a large, representative subset of MIR which we call MiniMir that includes support for borrows, sum and product types.

2.1 Syntax

$\langle \textit{Instruction} \rangle ::= \langle x \rangle$ ‘`:=`’ ‘`box`’ $\langle x \rangle$
| $\langle x \rangle$ ‘`:=`’ ‘`unbox`’ $\langle x \rangle$ | $\langle x \rangle$ ‘`:=`’ ‘`copy`’ $\langle x \rangle$
| $\langle x \rangle$ ‘`:=`’ ‘`copy`’ ‘`*`’ $\langle x \rangle$
| ‘`drop`’ $\langle x \rangle$ | ‘`swap`’ ‘`(`’ $\langle x \rangle$ ‘`,`’ $\langle x \rangle$ ‘`)`’
| $\langle x \rangle$ ‘`:=`’ $\langle x \rangle$ $\langle op \rangle$ $\langle x \rangle$ | $\langle x \rangle$ ‘`:=`’ $\langle const \rangle$
| ‘`(`’ $\langle x \rangle$ ‘`,`’ $\langle x \rangle$ ‘`)`’ ‘`:=`’ $\langle x \rangle$ | $\langle x \rangle$ ‘`:=`’ ‘`(`’ $\langle x \rangle$ ‘`,`’ $\langle x \rangle$ ‘`)`’ | $\langle x \rangle$ ‘`:=`’ ‘`inj`’$_i$ $\langle x \rangle$
| ‘`call`’ f ($\langle x \rangle$, .., $\langle x \rangle$) | ‘`assert`’ $\langle x \rangle$
| $\langle x \rangle$ ‘`:=`’ ‘`&mut`’$_\alpha$ $\langle x \rangle$ | ‘`immut`’ $\langle x \rangle$ | $\langle x \rangle$ ‘`:=`’ ‘`unnest`’ $\langle x \rangle$
| ‘`(`’ ‘`ref`’ $\langle x \rangle$ ‘`,`’ ‘`ref`’ $\langle x \rangle$ ‘`)`’ ‘`:=`’ *$\langle x \rangle$ | ‘`thaw`’ α | $\alpha \leq \alpha$

$\langle \textit{Statement} \rangle ::= \langle \textit{Instruction} \rangle$ ‘`;`’ ‘`goto`’ ℓ | ‘`return`’ $\langle x \rangle$
| ‘`match`’ $\langle x \rangle$ ‘`{`’ inj$_0$ $\langle x \rangle \to$ ‘`goto`’ ℓ, inj$_1$ $\langle x \rangle \to$ ‘`goto`’ ℓ ‘`}`’
| ‘`match`’ *$\langle x \rangle$ ‘`{`’ inj$_0$ ‘`ref`’ $\langle y_0 \rangle \to$ ‘`goto`’ ℓ_0, inj$_1$ ‘`ref`’ $\langle y_1 \rangle \to$ ‘`goto`’ ℓ_1, ‘`}`’

$\langle \textit{Type, } T \rangle ::=$ ‘`&mut`’$_\alpha$ $\langle \textit{Type} \rangle$ | ‘`&`’$_\alpha$ $\langle \textit{Type} \rangle$ | $\langle \textit{Type} \rangle \times \langle \textit{Type} \rangle$ | $\langle \textit{Type} \rangle + \langle \textit{Type} \rangle$ | ‘`box`’ $\langle \textit{Type} \rangle$
| unit | int

$\langle \textit{Signature} \rangle ::=$ ‘`fn`’ name ‘`<`’ α, ... | $\alpha \leq \alpha$, ... ‘`>`’ ‘`(`’ $\langle v \rangle$ ‘`,`’ ... ‘`)`’ ‘`->`’ $\langle \textit{Type} \rangle$

Figure 6: The syntax of MiniMir

A MiniMir program $\mathcal{P}$ is a collection of *function declarations*, and must contain a function `main` which acts as the entry point. Function declarations are in turn composed of a triple (Δ, ℓ, σ) consisting of a set of *labeled statements* Δ, an entry label ℓ, and a *function signature* σ. In MiniMir functions can be parametrized over lifetimes α and constraints $\alpha \leq \beta$. A function signature consists of the name, any lifetime parameters, the argument names and types and the return type.

Statements come in four forms: an instruction followed by a goto (`I; goto` ℓ’), a return (`return` x), a match on a variable (`match` $x^{T_0+T_1}$ `{ ... }`), or a match on a borrow (`match` $*x^{P\,(T_0+T_1)}$ `{ ... }`) . Matching on a borrow allows programs to turn a borrow on a sum into a borrow on the value contained in the sum. When this occurs, programs are free to

modify the value held in the sum but cannot change the constructor of the sum. We write $\mathcal{P}_{f,\ell}$ to refer to the statement associated with the label ℓ of the function f in program $\mathcal{P}$.

Instructions perform the basic operations of the language. In each instruction the variables are annotated with their types. We use P to denote a pointer which is either a box or a borrow. In MiniMir a `box` type is an owned pointer to a value, granting the owner exclusive access to the pointed region of memory. There are instructions to allocate ($x^{\texttt{box}\,T}$`:= box` y^T), dereference and remove an owned pointer from the heap (x^T`:= unbox` $y^{\texttt{box}\,T}$). Unboxing *consumes* the box provided as argument, removing the box from memory. Products ($x^{T_0 \times T_1}$`:= (`y^{T_0}`,` z^{T_1}`)`) and sums ($x^{T_0+T_1}$`:= inj`$_i$ y^T) of values can be constructed, taking ownership of their contents. One of the distinguishing features of MiniMir is that we can destruct products and sums in two manners based on whether we *own* the value (`let (`x^{T_0}`,` y^{T_1}`) :=` $z^{T_0 \times T_1}$) or whether we have a *borrow* of the value (`let (ref` $x^{P\,T_0}$`, ref` $y^{P\,T_1}$`) := *`$z^{P\,(T_0 \times T_1)}$). The equivalent operations for sums are presented above as they are *statements* and not instructions. Mutable borrows can be created from other values ($x^{\&\texttt{mut}_\alpha\,T}$`:= &mut`$_\alpha y^T$), weakened into an immutable borrow (`immut` $y^{\&\texttt{mut}_\alpha\,T}$) and *unnested* ($x^{\&\texttt{mut}_\alpha\,T}$`:= unnest` $y^{\&\texttt{mut}_\alpha\,P\,T}$). Unnesting is an essential operation when working with borrows. It collapses a layer of indirection between pointers, giving us access to the internal pointer. This allows us to transform a `&mut`$_\alpha$`&mut`$_\beta$ T into a `&mut`$_\alpha$ T. We can use this to perform a *reborrow*, where we borrow the pointee of a borrow with a lifetime α for a lifetime β. To verify the safety of borrows, we also add several ghost instructions, to end a lifetime (`thaw` α) and to impose an ordering over lifetimes ($\alpha \leq \beta$). Simple values constructed exclusively from constants, sums, products and immutable references can be *copied*[6, Annex F] (x^T`:= copy` y^T). These values can also be copied from behind a box or borrow (x^T`:= copy *`$y^{P\,T}$). In MiniMir we must explicitly *kill* our variables, the typing context at each label must contain only the variables that are *live*. We call this killing operation *drop*, and place a restriction on its use: it can only be used when its argument is an integer, a unit or an immutable reference (`drop` x^T). To drop a more complex structure composed of sums, products and mutable references, the structure must be recursively decomposed and freed piece by piece. Writing to memory is done by swapping (`swap(`$x^{P\,T}$`,` $y^{P\,T}$`)`). Primitive operations like assigning literals (x`:= C`), arithmetic and comparisons are supported (x^T`:=` $y^{\texttt{int}}$`op` $z^{\texttt{int}}$).

2.1.1 Example: Double-increments in MiniMir

```
ℓ0: x := 0; y := &mut_α x; goto ℓ2

ℓ2: t3 := copy *y; t4 := t3 + 1; drop t3; t5 := &mut_β t4; β ≤ α; goto ℓ7
ℓ7: t6 := &mut_β y; t7 := unnest t6; swap(t7, t5); goto ℓ9
ℓ9: immut t5; drop t5; immut t7; drop t7; thaw β; drop t4; goto ℓ15

ℓ15: t8 := copy *y; t9 := t8 + 1; drop t8; t10 := &mut_β t9; β ≤
α; goto ℓ20
ℓ20: t11 := &mut_β y; t12 := unnest t11; swap(t12, t10); goto ℓ24
ℓ24: immut t10; drop t10; immut t12; drop t12; thaw β; drop t9; goto ℓ30

ℓ30: immut y; drop y; thaw α; assert { x = 2 }; goto ℓ34
ℓ34 drop x; t0 := (); return t0
```

Figure 7: Translation of Figure 2, with the function `inc` inlined.

We can see this translation applied to Figure 2 in Figure 7. We chose to inline the definition of `inc` to make this translation more compact. We also exclude the `goto` of instructions that are on the same line.

In the first group of lines, we allocate a constant `x` and create a borrow `y` for some lifetime α. The following paragraph of code corresponds to the execution of the first call to `inc`. As integers are copy types, we can perform a copy out of y, and add one to it. To write the increased value back to `y`, we must perform a `swap`. A swap can only be performed between either two boxes or two mutable borrows with the same lifetimes. We begin by creating a borrow for a lifetime β of the incremented value. The next three instructions implement a *reborrow*, where we borrow the pointee of `y` for a shorter lifetime than α, in this case β. At the end of β, we recover access to `y`, and can perform further calculations using it. The result of this reborrow is a pointer `t7` to x for β. This allows us to swap `t7` and `t5`. The following line cleans up the no-longer needed temporaries and ends our sub-lifetime β.

The third paragraph is identical in structure to the second as it corresponds to the second call of `inc`. Since the borrow `t6` of `y` expired, we can call this function again using `y`. In the final paragraph, we drop `y`, and end its lifetime α, restoring ownership to `x`. This allows us to observe that `x` has been mutated twice.

2.2 Type System

Because of the imperative, graph structure of MiniMir, the typing judgements take a different form from most languages. The presence of cycles within a graph make traditional natural deduction trees impossible to use. Instead, each program point is assigned a typing context, and typing relates the contexts before and after the evaluation of a statement.

The type system of MiniMir enforces ownership of values in MiniMir and the safety of borrows. By creating pointers to values, borrows introduce aliasing. Allowing multiple mutable pointers to a value would cause all the aliasing problems of C. To prevent this issue, in MiniMir when a borrow is created, it must be associated to a lifetime α. For the duration of α the lender will be made inaccessible or *frozen*. Values which are frozen cannot appear in instructions of MiniMir programs until they are unfrozen through a `thaw` α. At the moment of thawing, the type system ensures that there are no outstanding borrows for α or any lifetime $\beta \leq \alpha$. To verify this safety property, we track a partial order on the lifetimes of a program in the type system.

Each label ℓ of a function is associated with a *partial variable context* $\mathbf{\Gamma}$. The partial variable context is a collection of items of the form $x :^{\dagger\alpha} T$ or $x :^{\bullet} T$, the first denoting a variable x is frozen for lifetime α with type T while the second denotes it is *active*. We also associate a *partial lifetime context* $\mathbf{L}$ with each label ℓ of a function, which consists of elements $\alpha \leq \beta$, establishing an order between two lifetimes.

In a MiniMir program $\mathcal{P}$, each function f is given a *whole context* $(\mathbf{\Xi}, \mathbf{\Lambda})$ which is the collection of the partial variable and lifetime contexts for all labels in f. We write $\mathbf{\Xi}_\ell$ or $\mathbf{\Lambda}_\ell$ to refer to the partial contexts of the label ℓ. Additionally, we use $\mathbf{\Sigma}$ to refer to the collection of signatures found in a program.

Typing judgements for instructions have the form $\mathbf{\Sigma}; \mathbf{\Gamma}; \mathbf{L} \vdash_f I \dashv \mathbf{\Gamma}; \mathbf{L}$, which relate the partial contexts before and after executing the instruction I in the function f. The typing judgement for $x^{\texttt{\&mut}_\alpha\, T} \texttt{:= \&mut}_\alpha y^T$ (Figure 8) expresses how the original value y is frozen until α expires while granting x access for that lifetime. No modifications are made to the partial lifetime context $\mathbf{L}$ as creating a borrow does not immediately place it in an ordered relation.

When a lifetime is ended using a `thaw` α, the type system ensures that everything borrowed

$$\frac{}{\mathbf{\Sigma}; y :^\bullet T, \mathbf{\Gamma}; \mathbf{L} \vdash_f x^{\texttt{\&mut}_\alpha\, T} \texttt{:= \&mut}_\alpha y^T \dashv \mathbf{\Gamma}, x :^\bullet \texttt{\&mut}_\alpha\, T, y :^{\dagger\alpha} T; \mathbf{L}} \text{ (BORROW-MUT)}$$

$$\frac{}{\mathbf{\Sigma}; x : \texttt{\&mut}_\alpha\, T, \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{immut x} \dashv \mathbf{\Gamma}, x : \texttt{\&}_\alpha\, T; \mathbf{L}} \text{ (IMMUT)}$$

$$\frac{T \in \{\texttt{int}, \texttt{unit}, \texttt{\&}_\alpha\}}{\mathbf{\Sigma}; x : T, \mathbf{\Gamma}; \mathbf{L} \vdash_f \texttt{drop}\ x^T \dashv \mathbf{\Gamma}; \mathbf{L}} \text{ (DROP)}$$

Figure 8: Excerpt of the instruction typing rules of MiniMir

for α has been dropped and that all lifetimes $\beta \leq \alpha$ have already been thawed. After a lifetime is thawed, all variables in the context which were frozen for α are unfrozen, restoring access.

Typing judgements for statements have the form $\mathbf{\Sigma}; \mathbf{\Xi}; \mathcal{P} \vdash_{f,a} S$ and verify that the statement with label a in function f, types with the expected partial context.

$$\frac{\mathbf{\Sigma}; \mathbf{\Xi}_a; \mathbf{\Lambda}_a \vdash_f I \dashv \mathbf{\Sigma}; \mathbf{\Xi}_\ell; \mathbf{\Lambda}_\ell}{\mathbf{\Sigma}; \mathbf{\Xi}; \mathbf{\Lambda} \vdash_{f,a} I\texttt{; goto } \ell}$$

The rule for (GOTO) checks that the instruction I can be typed using $\mathbf{\Xi}_a$, producing $\mathbf{\Xi}_\ell$. In this approach, each statement checks an edge of the MiniMir graph. The complete type system for MiniMir is included in [6, Annex F].

2.3 Operational Semantics of MiniMir

The operational semantics of MiniMir is given by a reduction relation over an abstract heap machine. In MiniMir, we allocate all values on the heap, to simplify the operational semantics of our language. Our abstract machine employs a *frame* which resembles a stack but serves only to translate a variable name into a heap address, for example, we cannot store an integer value directly in the frame. The configurations of the MiniMir machine are of the form $\langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H} \rangle$, where f is the name of the function being executed and ℓ is a label in f, $\mathbf{S}$ is the *call stack*, $\mathbf{F}$ is the *frame*, and $\mathbf{H}$ is the *heap*. The stack is composed of triples $[f; \ell, x, \mathbf{F}]$ of a return label, a variable name for the result and a frame. A frame $\mathbf{F}$ is a partial function from variable names to *addresses*. We support pointer arithmetic: adding an address and an integer to obtain an address. The heap maps those addresses to a value which is either: an address, an integer, or unit. Complex values such as pairs or sums are stored in contiguous regions of the heap. Operations which must move or copy these values calculate the *size* of the values from their types:

$$\begin{aligned} |T_1 \times T_2| &= |T_1| + |T_2| \\ |T_1 + T_2| &= 1 + max(|T_1|, |T_2|) \\ |\texttt{box}\ T| &= |int| = |unit| = 1 \end{aligned}$$

Definition 2.1 (Notations)**.** *If F is a partial function, and A is a subset of its domain, then $A \triangleleft F$ denotes the* domain restriction *removing A from the domain of F, and is defined as*

$$A \triangleleft F = \{(x, v) \mid (x, v) \in F, x \notin A\}$$

Below, we include the rule for `drop`, which kills a variable. When a variable is dropped, we lookup its address in the current frame and remove it from the heap. The remaining reductions are given in [6, Annex J].

$$\frac{\mathcal{P}_{f;\ell} = \texttt{drop } x^T\texttt{; goto } \ell\texttt{'}}{\langle f;\ell \mid \mathbf{S} \mid \mathbf{F} \oplus \{(x,a)\} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle f;\ell' \mid \mathbf{S} \mid \mathbf{F} \mid \{a\} \lhd \mathbf{H}\rangle} \text{ (DROP)}$$

Certain instructions like `thaw` are *ghost*, meaning they disappear during computation. These semantics are effectively no-ops in the semantics of MiniMir. The rule for mutable borrows, simply creates a pointer to a value, showing how borrows are nothing more than well-behaved pointers.

$$\frac{\mathcal{P}_{f;\ell} = x^{\texttt{\&mut}_\alpha\, T}\texttt{:= \&mut}_\alpha y^T\texttt{; goto } \ell\texttt{'} \qquad \mathbf{F}(y) = a \qquad b \notin dom(\mathbf{H})}{\langle \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H}\rangle \to_{\mathcal{P}} \langle \ell' \mid \mathbf{S} \mid \mathbf{F} + \{(x,b)\} \mid \mathbf{H} + \{(b,a)\}\rangle} \text{ (BORROW-MUT)}$$

The full semantics for MiniMir is included in [6, Annex J].

2.4 Type preservation by reduction

Simplification In the following sections, we only consider programs without function calls, and therefore we ignore the stacks of the MiniMir machine. The problems raised by function calls are orthogonal to the primary challenge of this proof. As a result, programs only consists in the body of the `main` function.

We would like to ensure that the only way for a well-typed MiniMir program to get stuck is by failing an assertion. This property allows us to reduce the safety of a program to the validity of its assertions. But if we look at a reduction like (BORROW-MUT), the machine would get stuck if $y \notin dom(\mathbf{F})$.

To resolve this problem we define a notion *well-typed configurations* which restricts us to only the configurations that could appear when evaluating a well typed program. This allows us to prove a standard preservation of typing lemma showing that in well-typed programs only failed assertions can block evaluation.

To define well-typed configurations we make use of *heap typing* to show a heap is composed only of well-formed values. The heap typing must ensure the existence of at most one active mutable borrow of an address. To establish this property we use a *borrow store*, which holds tokens for each lender and borrower in our heap. By consuming the borrow store, the heap typing can ensure each borrow is used only once.

Definition 2.2 (Borrow Store)**.** *A* borrow store *is a finite set of elements of the form* $\mathbf{take}^r_\alpha(a,T)$ *or* $\mathbf{give}^r_\alpha(a,T)$*, where* a *is an address,* T *is a type and* r *is either* i *for immutable or* m *for mutable.*

A borrow store $\mathcal{B}$ *is* safe *if for every address* a *appearing in* $\mathcal{B}$ *it contains exactly the tokens* $\mathbf{take}^m_\alpha(a,T)$ *and* $\mathbf{give}^m_\alpha(a,T)$ *or contains* $\mathbf{give}^i_\alpha(a,T)$ *and zero or more* $\mathbf{take}^i_\alpha(a,T)$*.*

When we write $\mathcal{B}; \mathbf{H} \models a :^{\mathbf{n}} T$, we mean to say that the heap fragment $\mathbf{H}$ starting at location a corresponds to a value of type T with $\mathbf{n} \in \{\dagger\alpha; \bullet\}$, under the borrow store $\mathcal{B}$.

Definition 2.3 (Heap Fragment Type for MiniMir)**.** *Below are the heap typing rules for mutable and immutable references. The borrow store checks that both the lender and borrower agree on the addresses and types borrowed. The full rules are found in [6, Annex I].*

$$\frac{}{\mathbf{give}^m_\alpha(a,T);\varnothing \models a :^{\dagger\alpha} T} \qquad \frac{\mathcal{B};\mathbf{H} \models a :^{\dagger\alpha} T}{\mathbf{give}^i_\alpha(a,T),\mathcal{B};\mathbf{H} \models a :^{\dagger\alpha} T}$$

$$\frac{\mathcal{B};\mathbf{H} \models a :^{\mathbf{n}} T}{\mathbf{take}^m_\alpha(a,T),\mathcal{B};\{(p,a)\} \oplus \mathbf{H} \models p :^{\mathbf{n}} \texttt{\&mut}_\alpha T} \qquad \frac{}{\mathbf{take}^i_\alpha(a,T);\{(p,a)\} \models p :^{\mathbf{n}} \texttt{\&}_\alpha T}$$

A well-typed configuration is one in which the heap consists exactly of the memory for each variable in our frame. The heap is divided up into disjoint segments for each variable, giving each cell of memory an *owner*. When a mutable borrow is created, the borrow takes ownership of the borrowed memory while the lender no longer has any owned memory. As borrows are turned immutable, they relinquish control over the memory they control, giving it back to the lender. In any given configuration, a cell of memory has a single, predictable owner.

Definition 2.4. *A configuration* $\langle f;\ell \mid - \mid \mathbf{F} \mid \mathbf{H}\rangle$ *of a well-typed program* $\mathcal{P}$ *is* well-typed *if it satisfies the following conditions:*

$$dom(\Xi_\ell) = dom(\mathbf{F}) \qquad \mathbf{H} = \bigoplus_{x\in\mathbf{F}} \mathbf{H}_x \qquad \mathcal{B} = \bigoplus_{x\in\mathbf{F}} \mathcal{B}_x \qquad safe(\mathcal{B})$$

$$\mathcal{B}_x;\mathbf{H}_x \models x :^{\mathbf{n}} \Xi_\ell(x,\mathbf{n})$$

Finally, type preservation is presented over well-typed configurations of MiniMir programs.

Lemma 2.5 (Type Preservation)**.** *Given a well-typed MiniMir configuration* C*, if* $C \to_{\mathcal{P}} C'$*, then* C' *is well-typed.*

The interesting cases of this proof are discussed in [6, Annex I].

3 Verification by translation

With Lemma 2.5, we've shown that if a MiniMir program gets stuck it must be due to a failed assertion. To show that a program doesn't get stuck at all we must show that all assertions pass. The approach we explored in this work is the verification of MiniMir by translation to a functional language. Verifying functional languages is generally simpler and proceeding by translation allows us to leverage pre-existing tools for our target language.

As our target language we use anyML, a simple ML family language with *any/assume non-determinism*. With this form of non-determinism, we can pick a value using the `any` expression, and place constraints on *past* choices using `assume` at a later point. In Figure 9 we non-deterministically choose a value for `x` while later assuming that `1 <= y`. This places a constraint on the *past* choice that we made for `x`.

The key insight of this translation was first given in RustHorn[11], the idea is to represent mutable borrows as a pair of a current and a *final* value. The moment we mutably borrow a variable, we assign the lender the *final* value of the borrow which we guess non-determinstcally using `any`. We can then perform operations using our newly created borrow, when we're done performing mutations, we use an assume to say that the final value is equal to the current value at that moment. By doing this we force our non-deterministic choice to take into account every mutation that happened during the borrow's life.

```
let x = any in
let y = x + 1 in
assume { 1 <= y };
let z = y + x + 2 in
assert { z >= 3 }
```

Figure 9: any/assume non-determinism in action

The intuition for this representation comes from Figure 3, when we borrow a into b, not only can we not use a until α is over, but b has exclusive permissions to the value of a. This means that any changes to that value must be done through b, when we drop b we release all permissions, effectively freezing the value until the end of α.

3.1 The anyML language

The anyML language is an untyped, ML family language with mutual recursion, assertions and any/assume non-determinism. It is meant to simplify the WhyML language of Why3 which we use in our implementation. The semantics of anyML is described by an abstract machine with environment[6, Annex J].

The `any` expression picks an arbitrary value, and a `assume { e }` evaluates an assumption `e`, if it doesn't reduce to `true`, then it diverges.

⟨*Expression, e*⟩ ::= '`let`' x '`=`' ⟨*e'*⟩ '`in`' ⟨*e*⟩ | ⟨*e*⟩ ⟨*e'*⟩ | x | ⟨*v*⟩ | ⟨*e*⟩ ⟨*op*⟩ ⟨*e'*⟩ | '`match`' ⟨*e*⟩ '`with`' '`|`' inj_0 x_0 '`->`' ⟨*e*⟩ '`|`' inj_1 x_1 '`->`' ⟨*e*⟩ '`end`' | '`let`' '`(`' x, y '`)`' = ⟨*e*⟩ | '`assert`' '`{`' ⟨*e*⟩ '`}`' | '`(`' ⟨*e*⟩ '`,`' ⟨*e*⟩ '`)`' | inj_i ⟨*e*⟩ | '`rec`' ⟨*FunDef*⟩ '`and`' ... '`and`' ⟨*FunDef*⟩ | '`any`' | '`assume`' '`{`' ⟨*e*⟩ '`}`'

⟨*FunDef*⟩ ::= '`fun`' f x ... x = ⟨*Expr*⟩

⟨*Values, v*⟩ ::= '`(`' ⟨*v*⟩ '`,`' ⟨*v*⟩ '`)`' | inj_i ⟨*v*⟩ | *n* | '`rec`' ⟨*FunDef*⟩ '`and`' ... '`and`' ⟨*FunDef*⟩

3.2 Translating from MiniMir to anyML

The translation takes a well-typed MiniMir program and produces a set of mutually recursive anyML functions with the same entrypoint as MiniMir. A labeled statement becomes a function where the arguments are the entire domain of the partial variable context associated with the label. Each `goto` ℓ is compiled to a function call, and the arguments are the variables in the context of the target label, Ξ_ℓ. Most instructions from MiniMir are translated to their direct counterparts in anyML. For example, here is the translation of the construction of a pair:

$$[\![\ell\texttt{: } x \texttt{ := } (y, z)\texttt{; goto } \ell'\texttt{}]\!] \triangleq \texttt{fun } \ell\ \vec{a} = \texttt{let } x = (y, z) \texttt{ in } \ell'\ \vec{a'}$$

where $\vec{a} = dom(\Xi_\ell)$ and $\vec{a'} = dom(\Xi_{\ell'}) = x, \vec{a} \setminus \{y, z\}$

When a borrow is created, it is assigned a pair of the value being borrowed and a non-deterministic choice for the *final* value. We use two operators to access borrows defined as: $*x \triangleq \texttt{fst } x$ and $\hat{}x \triangleq \texttt{snd } x$.

$$[\![\ell\texttt{: } x^{\texttt{\&mut}_\alpha\, T} \texttt{:= \&mut}_\alpha y^T\texttt{; goto } \ell'\texttt{}]\!] \triangleq \begin{array}{l}\texttt{fun } \ell\ \vec{a}\ \texttt{= let } x\ \texttt{= (}y\texttt{, any) in}\\ \texttt{let } y\ \texttt{= }\hat{}x\ \texttt{in } \ell'\ \vec{a'}\end{array}$$

When a mutable borrow is made immutable, we use the `assume` expression to equate its current value and final value. The intuition is that since we are forbidding further changes to a borrow so we must have its final value.

$$[\![\ell\texttt{: immut } y^{\texttt{\&mut}_\alpha\, T}\texttt{; goto } \ell'\texttt{'}]\!] \triangleq \texttt{fun } \ell\ \vec{a}\ \texttt{= assume \{ *}y\texttt{ = \^{}}y\texttt{\}; let } y\texttt{ = *}y\texttt{ in } \ell'\ \vec{a'}$$

Ghost instructions like `thaw` α are elided entirely during translation. The full translation is presented in [6, Annex H]. In Figure 10, the program of Figure 7 is translated to anyML.

3.2.1 Example: Translating double increment to anyML

In Figure 10, we can find the translation of the program given in Figure 7. Each statement is turned into a mutually recursive function, with as arguments the domain of its typing context. In the translated figure, we have combined certain of these statement functions to correspond with the syntactic sugar used in Figure 7.

```
fun l0 () = let x = 0 in let y = (x, any) in let x = ^y in l2 x y
and fun l2 x y = let t3 = *y in let t4 = t3 + 1 in
  let t5 = (t4, any) in let t4 = ^t5 in l7 x y t4 t5
and fun l7 x y t4 t5 = let t6 = (y, any) in let y = ^t6 in
  let t7 = ( **t6, *^t6) in assume { ^*t6 = ^^ t6 };
  let temp = *t7 in let t7 = ( *t5, ^t7) in let t5 = (temp, ^t5) in
  l9 x y t4 t5 t7
and fun l9 x y t4 t5 t7 = assume { *t5 = ^t5}; let t5 = *t5 in
  assume {*t7 = ^t7}; let t7 = *t7 in l15 x y
and fun l15 x y = let t8 = *y in let t9 = t8 + 1 in
  let t10 = (t9, any) in let t9 = ^t10 in l20 x y t9 t10
and fun l20 x y t9 t10 = let t11 = (y, any) in let y = ^t11 in
  let t12 = ( **t11, *^t11) in assume { ^* t11 = ^^ t11} in
  let temp = *t12 in let t12 = ( *t10, ^t12) in let t10 = (temp, ^t10) in
  l24 x y t9 t10 t12
and fun l24 x y t9 t10 t12 = assume { *t10 = ^t10 }; let t10 = *t10 in
  assume { *t12 = ^t12 }; let t12 = *t12 in l30 x y
and fun l30 x y = assume { *y = ^y}; assert { x = 2}; let t0 = () in t0
```

Figure 10: Translation of Figure 7 to anyML

When in `l0` we create the borrow `y`, we make our non-deterministic guess of the final value through `any`. We then immediately *shadow* the binding for `x`, using the final value of `y` in its stead. Things get interesting when we reach `l7`, where we borrow `y` and `unnest` the result. This translates to the following fragment of anyML:

```
let t6 = (y, any) in let y = ^t6 in
let t7 = ( **t6, *^t6) in assume { ^*t6 = ^^ t6 }; ..
```

When we perform an unnesting, we collapse a layer of indirection, recovering the internal borrow, for a shorter lifetime. To understand this line recall that `t6` is a borrow of a borrow, and that by unnesting we consume `t6`, and so we should emit an *assumption* about its final value. The following line executes the `swap` instruction, exchanging the *current* values of both borrows. The function `l9` shows the translation for `immut`, when we make a borrow immutable,

we *assume* it has reached its final value. The drop instructions become nothing more than `()`, and we stop passing the dropped variable to the next labeled function. Finally, note that the instruction for *thawing* β is omitted in the resulting anyML, it exists purely for safety of the MiniMir program. We will leverage its existence to justify the safety of this translation.

3.3 Correctness of translation

We now examine the correctness of the translation presented in the previous section. Our safety theorem states that for a program $\mathcal{P}$, if none of the executions of the translation $[\![\mathcal{P}]\!]$ block then neither does the original program.

Theorem 3.1 (Safety)**.** *Given a well-typed MiniMir program* $\vdash \mathcal{P}$*, if* $[\![\mathcal{P}]\!]$ *is safe, then* $\mathcal{P}$ *is safe.*

The proof of this theorem will rely on a simulation, $\sim_{\mathcal{P}}$, established between *MiniMir traces* and anyML configurations. We will detail the simulation invariant in Section 3.3.1. We prove Theorem 3.1 using the following three lemmas.

Lemma 3.2 (Preservation of Simulation)**.** *Given a MiniMir trace* $\Theta = C \to^*_{\mathcal{P}} C'$ *and a anyML configuration* K *such that* $\Theta \sim_{\mathcal{P}} K$*, if* $C \to_{\mathcal{P}} C''$*, there exists a* K' *such that* $K \to K'$ *and* $C'' \to^*_{\mathcal{P}} C' \sim_{\mathcal{P}} K'$*.*

Lemma 3.3 (Progress)**.** *Given a MiniMir configuration* C *and a anyML configuration* K *such that* $C \sim_{\mathcal{P}} K$*, if* K *is not stuck then* C *is not stuck.*

Lemma 3.4 (Terminal Configurations)**.** *Given a terminal anyML configuration* K*, for any MiniMir configurations* C *such that* $C \sim_{\mathcal{P}} K$*,* C *is terminal.*

We don't present the proofs of these theorems here, but they are discussed in more detail in [6]. In Section 3.3.2, we will give an outline of the proof for the most complex of the lemmas. With these lemmas in hand, the proof of safety is rather simple:

Proof of Theorem 3.1. Suppose that C is not safe, therefore there exists a trace $C \to^*_{\mathcal{P}} C'$ which gets stuck. By iterating Lemma 3.2, there exists K', such that $C' \sim_{\mathcal{P}} K'$. Because K' is safe, it cannot be stuck and must either be terminal or continue reducing. If K' is a terminal configuration then by Lemma 3.4 there is a contradiction since C' is not terminal. If K' is not terminal then there exists K'' such that $K' \to_K K''$ and by Lemma 3.3 there exists C'' such that $C' \to_{\mathcal{P}} C''$, therefore C' is not stuck. □

3.3.1 Simulation Invariant

In our simulation we wish to relate MiniMir configurations with their anyML translations. The relation must translate both the code *and* environments. Translating environments poses a problem however, how do we translate a mutable borrow? As presented in Section 3, a mutable borrow in MiniMir is translated to a pair of a current and final value. It is essential that the simulation relation includes only anyML configurations that have made the *correct* choice for the final value of our borrow because only those configurations will satisfy later assumptions. That is why we establish our simulation between MiniMir *traces* and anyML configurations, doing so allows us to look into the future of a MiniMir trace and find the final value of any borrow.

Using our MiniMir trace we construct a *prophecy map* containing the final values of every mutable borrow. In our simulation the prophecy map represents the *correct* non-deterministic

choices we must make for each mutable borrow. We can use this prophecy map to translate fragments of the MiniMir heap into anyML values, performing a *readback*. The readback is defined by extending the heap typing previously defined to construct values along the tree. Below we present the readback of mutable and immutable borrows. The full rules can be found in [6, Annex I] During the readback we use the prophecy map at two points, whenever we translate the loaner, and when we translate the borrow itself.

Definition 3.5 (Readback)**.** *The readback* $\mathcal{R}^{\mathbf{n}}(\mathcal{B}, A, \mathbf{H}, a, T, V)$ *of a region of memory* $\mathbf{H}$ *at a type* T *and activeness* $\mathbf{n}$ *with prophecy map* A *and borrow store* $\mathcal{B}$ *produces a mapping* V *from an address* a *and type* T *to a anyML value* v*. The entries of this mapping represent all sub-values contained in* $\mathbf{H}$*.*

$$\frac{A(a,T) = v}{\mathcal{R}^{\dagger\alpha}(\{\mathbf{give}^m_\alpha(a,T)\}, A, \varnothing, a, T, (a,T,v))} \qquad \frac{\mathcal{R}^{\dagger\alpha}(\mathcal{B}, A, \mathbf{H}, a, T, V)}{\mathcal{R}^{\dagger\alpha}(\{\mathbf{give}^i_\alpha(a,T)\} \cup \mathcal{B}, A, \mathbf{H}, a, T, V)}$$

$$\frac{A(a,T) = v}{\mathcal{R}^{\mathbf{n}}(\{\mathbf{take}^i_\alpha(a,T)\}, A, \{(p,a)\}, p, \&_\alpha\ T, (p,T,v))}$$

$$\frac{\mathcal{R}^{\mathbf{n}}(\mathcal{B}, A, \{a\} \lhd \mathbf{H}, \mathbf{H}(a), T, E) \qquad A(\mathbf{H}(a), T) = v}{\mathcal{R}^{\mathbf{n}}(\mathbf{take}^m_\alpha(a,T) \cup \mathcal{B}, A, \{(p,a)\} \oplus \mathbf{H}, p, \texttt{\&mut}_\alpha\, T, E + (a, T \times T, (E(\mathbf{H}(a)), v)))}$$

The readback can then be applied to an entire heap through HeapEnv, producing a anyML environment.

Definition 3.6 (HeapEnv)**.** *The relation* HeapEnv *shows how to construct a anyML environment from a MiniMir configuration. It uses the heap types of a configuration to readback memory cells as anyML values.*

Formally, HeapEnv $(\mathcal{B}, A, \mathbf{F}, \mathbf{H}, \mathbf{\Gamma}, E)$ *is a 6-place relation between an activeness, a borrow store, a prophecy map, a MiniMir frame and heap, a partial typing environment and a anyML environment where:*

$$dom(\mathbf{F}) = dom(E) \qquad \mathbf{H} = \bigoplus_{x \in dom(\mathbf{F})} \mathbf{H}_x \qquad E = \{(x, V_x(F(x), T)) \mid x :^{\mathbf{n}} T \in \mathbf{\Gamma}\}$$

$$\forall x \in dom(\mathbf{F}), \mathcal{R}^{\mathbf{n}}(\mathcal{B}_x, A, \mathbf{H}_x, \mathbf{F}(x), \mathbf{\Gamma}(x, \mathbf{n}), V_x)$$

To calculate a prophecy map we walk backwards along the MiniMir trace, each time a `thaw` α instruction is encountered, we perform a readback for all values that were frozen for the lifetime α. The relation McFly performs this walk, given an initial set of values for prophecies at the end of the trace (usually $\varnothing$), it constructs the prophecies at the start of the trace.

$$\frac{\mathcal{P}_{f;l} = \texttt{thaw}\ \alpha \qquad A' = A \oplus \bigoplus_{\mathbf{give}^r_\alpha(a,T) \in \mathcal{B}} \{E \mid \text{HeapEnv}\ (A, \mathcal{B}, \mathbf{F}', \mathbf{H}', \mathbf{\Xi}_{f;\ell}, E)\}\, \mathcal{R}^{\dagger\alpha}(\mathcal{B}'_a, A, \mathbf{H}'_a, a, \mathbf{\Xi}_{f';\ell'}(a, \bullet), E)\}}{\text{McFly}^*(A', \langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H} \rangle \to_{\mathcal{P}} \langle f'; \ell' \mid - \mid \mathbf{F}' \mid \mathbf{H}' \mid \mathcal{B}' \rangle, A)}$$

Finally, the relation $\sim_{\mathcal{P}}$ packages the calculation of a prophecy map and the translation of the environments and code together.

Definition 3.7 (Simulation Relation)**.** *The relation $\sim_{\mathcal{P}}$ between a well-typed MiniMir trace Θ and a anyML configuration is defined by the following conditions:*

$$\begin{aligned}\Theta = \langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H} \mid \mathcal{B} \rangle \to_{\mathcal{P}}^{*} C \sim_{\mathcal{P}} \langle [\![\mathcal{P}_l]\!] \mid E \mid K \rangle \\ \text{HeapEnv}\ (A, \mathcal{B}, \mathbf{F}, \mathbf{H}, \Xi_\ell, E) \\ K = ret\ E' \cdot \ldots \cdot ret\ E'' \\ \text{McFly}(A, \langle f; \ell \mid \mathbf{S} \mid \mathbf{F} \mid \mathbf{H} \mid \mathcal{B} \rangle \to_{\mathcal{P}}^{*} C)\end{aligned}$$

3.3.2 Preservation of Simulation

We've given a description of the simulation relation $\sim_{\mathcal{P}}$, and we claim that it is sufficient to prove the three lemmas required for Theorem 3.1. Now we will give the intuition behind the proof of Lemma 2.5, though a more complete discussion will be left to [6, Annex I].

In our translation, a $\texttt{immut}\ y^{\texttt{\&mut}_\alpha\ T}$ is translated into:

$$\texttt{assume \{ *}y\texttt{ = \^{}}y\texttt{\}; let y = *y in }\ell\ \vec{a}$$

But in our simulation, we choose the final value from a $\texttt{thaw}\ \alpha$ using McFly. These two points can be arbitrarily far apart in a MiniMir trace, and we must show that the values haven't changed in the meantime. Intuitively, this makes sense since by turning a reference immutable we would expect there to be no further changes to the referred value. This can be observed in the heap typing for immutable references, which have no ownership of the referred memory, which instead belongs to the *frozen* lender. This is the key idea of this proof, that we can find out the value of a dynamic point in the program ($\texttt{immut}\ y^{\texttt{\&mut}_\alpha\ T}$) by looking at a further, static point ($\texttt{thaw}\ \alpha$).

Actually proving the case of $\texttt{immut}\ y^{\texttt{\&mut}_\alpha\ T}$ is more challenging as we must show that the prophecy for x comes from the correct thaw. Then since memory associated with x cannot have changed in the interim as we previously discussed, we can show that the value readback at the moment of thawing must be the same as the current value at the moment of weakening.

The details of this proof are technical and syntactically heavy, if readers would like more a more detailed view consult [6, Annex I].

4 Experimentation

As further validation of the translation presented in Section 2, we implemented a proof-of-concept tool as an extension to the Rust compiler. This tool takes Rust programs in their MIR representation and translates them to WhyML, the specification and programming language of the Why3 verification suite. The translated programs can then be checked using the Why3 prover interface. Using this tool we were able to verify simple programs working on structures such as linked lists.

One of the primary implementation challenges is determining where the $\texttt{drop}\ x^T$ should be in MIR code. The borrow checker of Rust infers a position where it should be inserted but that information is hidden from other passes. Extracting that information is essential as the safety of this tool relies on correct placement of thaws to mark the end of lifetimes.

We also took advantage of this tool to test several simple extensions, such as support for pre-conditions, post-conditions and invariants in code being verified. Each of these is lowered, nearly directly to the equivalent Why3 constructs. This enabled us to verify the functional correctness of several programs. A few of these programs are included in [6, Annex K].

5 Conclusion

In this paper, we presented a schema to verify Rust-like programs by translation to a functional language. Our source language MiniMir, is translated to anyML, an ML-like language with non-determinism. In our translation we represent mutable borrows as pairs of their current and future values. To prove that our translation is correct, we developed an original simulation technique between an execution trace and a functional configuration. The proof requires us to show that we can predict the future values of each borrow, and we show that we can statically identify points in the program which can be used to predict those values. Finally, we validated this technique experimentally by implementing this translation as a proof-of-concept tool. Our tool was able to verify safety properties for simple programs using list operations.

Related Work. Tools like Frama-C[9] allow the verification of pointer programs written in C, including in the face of aliasing. However, reasoning about aliasing increases the difficulty of proofs both for authors and automated tools used to discharge proofs. To alleviate this, Frama-C offers non-aliasing predicates to require the separation of variables. Asserting that all variables are non-aliasing requires quadratic amounts of hypotheses, which overwhelms automation like SMT solvers. Other languages like Spark/Ada heavily restricted pointers or *access types* to avoid aliasing. Now, they have adopted a form of ownership and borrowing[7] inspired from Rust to gain flexibility while preserving reasoning abilities.

For the verification of Rust programs, RustBelt[8] provides a near complete formalization of Rust in the Coq proof assistant. This provides a high assurance environment to reason about Rust code but requires complex, manual proofs. RustHorn[11], which inspired this work takes another approach, translating programs to Constrained Horn Clauses. The difference in target languages leads to different proofs and approaches to resolving prophecies. Perhaps more importantly, by choosing to translate to a functional language we provide a more convenient base for extensions commonly found in semi-automatic verification approaches. By choosing CHCs, RustHorn limits itself to fully automatic verification which leads to a restriction on the properties that can actually be proven.

Another approach for deductive verification of Rust is explored by Prusti[4] which translates programs to a separation logic with fractional permissions. While the Prusti team has a more complete and developed tool, it captures a measurably smaller fragment of Rust programs than our approach. It is not yet possible to perform operations like reborrowing in loop bodies, a pattern which frequently and naturally appears in real Rust code. Additionally, no complete formalization of their approach has been presented which makes it difficult to perform a thorough comparison.

Future Work. The approach presented in Section 3 allows us to verify Rust-style programs using both mutable and immutable references. However, the proof of correctness is complex using an original form of simulation. Formalizing this development in Coq would give a much firmer base for further extensions to the language. The work on RustBelt[8], could serve a starting point for a Coq formalization, since key lemmas like type safety are already proven on the core language λRust.

Currently, we have not considered the questions of specification languages for Rust, the encoding for datatype invariants and ghost code more generally remains an open question. To put this translation into application and incorporate extensions, our tool needs to be extended to more gracefully handle real-world Rust programs.

6 Acknowledgements

I would like to thank my Master's (and now PhD) thesis directors Jacques-Henri Jourdan and Claude Marché for their support and pedagogy. I've learned a lot already from both of you and I'm looking forward to the next three years. Of course, this paper would not be what it is without the insight and patience of its reviewers and guide, thank you for your efforts. Finally, I would like to thank the VALS team at the LRI for welcoming me despite the challenging sanitary situation and my professors at the MPRI for a stimulating and challenging final year of Master's studies.

References

[1] Rust Programming Language. https://www.rust-lang.org/.

[2] Announcing Rust 1.0. https://blog.rust-lang.org/2015/05/15/Rust-1.0.html, May 2015.

[3] Aliasing (computing). https://en.wikipedia.org/w/index.php?title=Aliasing_(computing)&oldid=959584508, May 2020.

[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, October 2019.

[5] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI C Specification Language. page 81.

[6] Xavier Denis. Deductive program verification for a language with a Rust-like typing discipline. Master's thesis, Université de Paris, September 2020.

[7] Claire Dross and Johannes Kanig. Recursive Data Structures in SPARK. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 178–189, Cham, 2020. Springer International Publishing.

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, January 2018.

[9] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.

[10] Nicholas D. Matsakis and Felix S. Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, October 2014.

[11] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. RustHorn: CHC-based Verification for Rust Programs (full version). In *ESOP*, page 49, 2020.

[12] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, January 1995. Association for Computing Machinery.

Mettons de l'Ordre dans CIC !

Théo Laurent[1] and Kenji Maillard[2]

[1] Inria Paris, Équipe Prosecco, Paris
[2] Inria Rennes, Équipe Gallinette, Nantes

Résumé

Les langages à types dépendant souffrent régulièrement de problèmes de modularité. L'ajout de sous-typage permet d'améliorer l'interopérabilité entre composants. Cependant, l'étude de la métathéorie des langages résultants, cruciale dans le cadre des assistants à la preuve, dépend souvent de propriétés syntaxiques délicates à établir [Luo88 ; Miq01]. Nous proposons une approche à base de modèles syntaxiques [BPT17b] pour capturer différentes notions de sous-typage en théorie des types et en particulier pour le Calcul des Constructions Inductives (CIC). Ces modèles servent de prototypes et de guides pour la conception d'extensions de CIC munies de sous-typage, de types à raffinement ou de cumulativité. Cette approche systématique vise à combler l'écart existant entre les différents systèmes de sous-typage présentés dans la littérature et à proposer des solutions aux limitations parfois artificielles de leurs implémentations dans des assistants à la preuve.

1 Introduction

Les langages de programmation comme les assistants de preuve souffrent parfois de problèmes d'interopérabilité entre composants. Ceux-ci se manifestent sous la forme d'une rigidité importante du système : par exemple une fonction définie sur les entiers naturels $\mathbb{N}$ ne s'applique *a priori* pas au type des entiers naturels pairs Even, bien qu'un utilisateur puisse légitimement considérer qu'il existe une coercion canonique entre les deux types. Les disciplines de sous-typage proposent de pallier ce problème en introduisant dans le système de type de telles approximations. Ces disciplines peuvent prendre des formes variées : cumulativité, sous-typage structurel, types à raffinement, degrés d'effacement (irrelevance, de tailles, ...).

En présence de types dépendants, les coercions $\vdash \mathtt{coe}_X^Y\, t : Y$ sur un terme $\vdash t : X$ induites par l'ordre de sous-typage $X <: Y$ apparaissent aussi dans les types. Pour obtenir une discipline efficace de sous-typage, les différentes constructions du langage doivent interagir avec cet ordre (monotonicité, irrelevance). Les problématiques engendrées par ces interactions non-triviales se retrouvent sous des formes similaires quelle que soit la discipline de sous-typage envisagée.

Dans ce document nous proposons une feuille de route pour établir de manière solide les propriétés métathéoriques de calculs à types dépendants munis de sous-typage. Nous établissons un panorama succinct des nombreuses contributions à ce domaine dans la Section 2 et décrivons certains problèmes ouverts motivant nos travaux. Notre méthodologie, présentée en Section 3, s'appuie sur l'étude de modèles syntaxiques de CIC [BPT17a] équipant les types avec une relation de sous-typage (Section 4). Les pistes de réflexion présentées dans ce papier constituent des travaux de recherche en cours.

2 État de l'art & Applications Envisagées

La littérature autour de CIC abonde d'extensions munissant CIC d'une forme d'ordre : types à raffinement [Soz07], sous-typage structurel [LA08], cumulativité [Ass14 ; TS18], types munis de tailles pour la récursion gardée [Abe08b ; Abe08a ; AP13].

```
Inductive vect A : ℕ → □:=
| nil : vect A 0
| cons : A → vect A m → vect A (S m).
```

```
Inductive vect A (n : ℕ) : □:=
| nil : ∀ n, n = 0 → vect A n
| cons : ∀ n m, n = S m → A → vect A m → vect A n.
```

FIGURE 1 – Vecteurs (à gauche) et leur traduction de fordisme (à droite).

Sous-typages subsumptif et coercif. L'approche *subsumptive* au sous-typage part du postulat que les coercions sont entièrement implicites et transparentes : aucune information n'apparaît dans les termes. Le Calcul des Constructions Implicite [Miq01 ; Ber15] présente une forme radicale de sous-typage subsumptif où $T <: U$ dès que tout terme $\vdash t : T$ vérifie aussi $\vdash t : U$. Par opposition, l'approche *coercive* repose sur l'utilisation explicite de fonctions de coercion, éventuellement inférées durant une phase d'élaboration [Luo96 ; LS99 ; LL16].

Il est à noter que les approches subsumptives posent souvent des problèmes de décidabilité puisque moins d'information est conservée dans les termes. D'autre part, ces systèmes requièrent une conversion non typée ce qui nécessiterait une adaptation importante de la méthode à base de modèles syntaxiques. Nous prenons donc ici une approche coercive tout en notant que des modèles validant la transparence des coercions dans le cadre subsumptif pourraient permettre de réconcilier ces deux mondes.

Sous-typage via raffinement. Le langage Russel [Soz07] emploie du sous-typage par raffinement pour attacher des invariants à des programmes Coq et générer des conditions de vérification. Il est validé par une transformation de programme vers CIC enrichie de types sous-ensembles suffisamment extensionnels.

Le langage F⋆ propose une approche similaire a la preuve de programme, mais à une échelle plus importante. Il implémente une théorie des types extensionnelle avec des types à raffinement, du sous-typage et du sous-effet. La métathéorie du langage à été étudiée sur des fragments parfois conséquents [Swa+16 ; Ahm+17 ; Ahm+18], mais une approche englobant toutes les différentes facettes du système manque encore à ce jour.

Sous-typage structurel. Le sous-typage structurel concerne l'aspect congruent des constructions du langage vis-à-vis de l'ordre de sous-typage. Un modèle où toutes les constructions sont monotones validerait donc immédiatement cette discipline de sous-typage. La monotonicité des paramètres des inductifs, c'est à dire le fait que $\text{list}\, A <: \text{list}\, B$ quand $A <: B$, est un cas particulier de sous-typage structurel. C'est d'ailleurs une extension régulièrement sollicitée par les utilisateurs de F⋆ [Hri14], mais dont l'adoption se heurte à l'absence de fondations solides.

Une seconde application potentielle d'un modèle rigoureux serait la validation de la traduction de *fordisme*[1] pour Coq en présence de cumulativité. Celle-ci consiste à transformer un inductif à indices en un inductif ne prenant plus que des paramètres mais en contraignant chaque constructeur avec des preuves d'égalité supplémentaires (voir Fig. 1 pour l'exemple des vecteurs). En présence de cumulativité, cette traduction n'est valide que si l'on internalise la monotonie de l'égalité vis-à-vis de son premier paramètre.

Sous-typage polarisé Le sous-typage ne considère pas uniquement des constructions monotones : dans la plupart des disciplines de sous-typage pour des types simples, le type des fonctions

1. Ce nom semble être une allusion à la citation d'Henri Ford « Le client peut choisir la couleur de sa voiture, pourvu que ce soit noir. » [McB00]

est notoirement contravariant dans le domaine. Le sous-typage polarisé [Abe08a] ajoute des informations de monotonicité aux types de fonctions telles que leur caractère monotone/positif/-covariant, antitone/négatif/contravariant, invariant ou indépendant. Nuyts [Nuy15] esquisse un cadre extrêmement riche à base de modalité dépassant de loin le cadre du sous-typage et servant d'inspiration aux modèles que l'on présente ici.

Le modèle ensembliste de Timany et Sozeau [TS18] ne valide pas la contravariance des produits dépendants vis-à-vis de la cumulativité. Un modèle cumulatif de CIC admettant la contravariance des produits dépendants validerait l'η-réduction dans Coq en permettant de prouver la préservation du typage pour la réduction étendue avec $\lambda x : A.\, t\, x \rightsquigarrow t$.

3 Méthodologie proposée

Nous proposons une méthodologie employant un prototypage à l'aide de modèles de CIC appelés *syntaxiques* car ils vérifient certaines "bonnes propriétés" favorisant une implémentation rapide. Nous expliquons ensuite comment ces modèles peuvent nous aiguiller dans la preuve de propriétés fondamentales que notre langage à types dépendants et sous-typage devrait vérifier.

3.1 Modèles Syntaxiques

Un modèle d'un langage est une interprétation des termes et des types du langage dans un domaine cible fixé. Boulier, Pédrot et Tabareau qualifient de *syntaxique* un modèle de la théorie des types dont la cible est aussi une théorie des types et préservant la présentation syntaxique du langage et notamment la conversion entre termes. Afin de travailler avec plusieurs langages à types dépendants, nous adoptons une présentation basée sur les jugements suivants :

$\Gamma \vdash$	Γ est un contexte bien formé
$\Gamma \vdash A$	A est un type bien formé dans le contexte Γ
$\Gamma \vdash t : A$	t est un terme de type A dans le contexte Γ
$\Gamma \vdash A \equiv B$	A et B sont des types convertibles
$\Gamma \vdash t \equiv u : A$	t et u sont des termes convertibles de type A

Cette présentation des jugements est fondée sur les théories algébriques généralisées de [Car86] et correspond assez directement aux composantes d'une catégorie avec famille (CwF) [Dyb95 ; AK16]. Une théorie des types concrète étend ces CwF avec de nouveaux types, termes et équations. La théorie des types de Martin-Löf requiert des produits Π et sommes Σ dépendants, un type identité Id ainsi qu'un univers $\square$. Pour CIC, on considérera en plus une hiérarchie d'univers $\square_i : \square_{i+1}$, et une signature d'inductifs contenant en particulier $\mathbb{N}, \mathbb{B}, \texttt{list}$ ainsi que leurs éliminateurs.

Dans cette présentation, un modèle est la donnée d'une fonction $[\![-]\!]$ interprétant un type $\Gamma \vdash_{\texttt{src}} A$ de la source en un type $[\![\Gamma]\!] \vdash_{\texttt{tgt}} [\![A]\!]$ de la cible (puis étendu aux contextes), ainsi que d'une fonction $[-]$ interprétant un terme $\Gamma \vdash_{\texttt{src}} t : A$ de la source en un terme $[\![\Gamma]\!] \vdash_{\texttt{tgt}} [t] : [\![A]\!]$ de la cible. On dira que ce modèle est *syntaxique* lorsque cette traduction des contextes, types et termes peut se décrire effectivement sur la syntaxe, et telle que la conversion soit préservée. En pratique, on définit d'abord les deux fonctions de traduction $[\![-]\!]$ et $[-]$ sur la syntaxe des types et des termes, puis on montre par induction sur la dérivation de typage que

$$\Gamma \vdash_{\texttt{src}} t : A \qquad \Rightarrow \qquad [\![\Gamma]\!] \vdash_{\texttt{tgt}} [t] : [\![A]\!]$$

ce qui est grandement facilité si la conversion est préservée $\vdash_{\texttt{src}} A \equiv B \Rightarrow \vdash_{\texttt{tgt}} [\![A]\!] \equiv [\![B]\!]$.

Un certain nombre de modèles et traductions peuvent être raisonnablement considérés comme syntaxiques. Citons à titre d'exemples :

le modèle exceptionel [PT18 ; Péd+19] ou modèle des types pointés ;

les modèles de paramétricité [BL11 ; KL12 ; AGJ14 ; Bou18] équipant les termes avec une preuve de paramétricité ;

le modèle Setoid [Alt+19] équipant chaque type avec une relation d'équivalence ;

les modèles monadiques [PT17] interprétant un type comme une algèbre d'une monade.

La traduction de la théorie des types extensionnelle vers la théorie des types intensionnelle munie de l'extensionalité des fonctions (`funext`) et de l'irrélévance des preuves d'égalité (`UIP`) fournit un contre-exemple de modèle non-syntaxique [Hof95 ; WST19].

Alternativement, un modèle syntaxique peut être vu comme une phase de compilation préservant le typage. Cependant l'emploi du vocable *modèle* correspond mieux à notre programme : une traduction explique la théorie source via son encodage dans la théorie cible, mais surtout propose des extensions potentielles de la théorie source. Cette démarche n'est pas récente : la logique linéaire [Gir87] et le λ-calcul différentiel [ER03] proviennent de l'analyse des modèles d'espaces cohérents et de sémantiques quantitatives ; l'introduction de monades [Mog89] ou d'adjonctions [Lev03] pour manipuler des effets dérive directement de l'analyse de modèles catégoriques. L'aspect novateur est de contraindre les modèles à être présentés effectivement afin de guider plus précisément les extensions du langage source. Cette approche permet notamment :

— De valider rapidement certaines idées de nature sémantique sur l'objet d'étude, ici des ordres partiels ;

— De réutiliser les propriétés de l'assistant de preuve dans lequel on travaille (type LF), sans se préoccuper dans un premier temps des aspects purement syntaxiques et algorithmiques ;

— De restreindre l'espace de conception du langage source, en particulier celui des règles de réductions.

Plus particulièrement, les modèles syntaxiques peuvent aider à établir les propriétés métathéoriques clés suivantes :

la consistance relative vis-à-vis de la cible peut s'obtenir en étudiant la traduction du type vide $[\![\bot]\!]$ en supposant la théorie cible consistante ;

la normalisation forte en montrant que chaque réduction dans la source se traduit en une ou plusieurs réductions dans la cible et que celle-ci est fortement normalisante ;

la décidabilité de la conversion, du sous-typage et du typage en réemployant des résultats idoines dans la théorie cible ;

la préservation du typage en proposant des règles de réduction à ajouter à la théorie source validées par la traduction ;

la canonicité en montrant que toutes les conversions entre termes traduits dans la cible – vérifiant elle-même la canonicité – ont déjà lieu dans la source.

3.2 Vers une Implémentation de Référence

L'aspect *traduction de programme* des modèles syntaxiques est particulièrement attrayant pour la formalisation, puisque cette tâche peut être vue comme une phase de compilation. Dans le cadre de l'assistant de preuve Coq, le projet MetaCoq [Soz+20] permet par ailleurs de réifier et manipuler des termes de Gallina, le langage de programmation de Coq. Le processus envisagé est le suivant :

$$[M] := ([M]_0, [M]_1) \qquad [\![A]\!] := \Sigma(x : [A]_0).[A]_1 x$$

$$\begin{array}{rl} [x]_0 := \pi_1\ x & [x]_1 := \pi_2\ x \\ [\lambda(x:A).M]_0 := \lambda(x : [\![A]\!]).[M]_0 & [\lambda(x:A).M]_1 := \lambda(x : [\![A]\!]).[M]_1 \\ [M\ N]_0 := [M]_0\ [N] & [M\ N]_1 := [M]_1\ [N] \\ [\Pi(x:A).B]_0 := \Pi(x : [\![A]\!]).[B]_0 & [\Pi(x:A).B]_1 := \lambda f.\Pi(x : [\![A]\!]).[B]_1\ (f\ (\pi_1\ x)) \\ [\Box_i]_0 := \Box_i & [\Box_i]_1 := \lambda(A : \Box_i).A \to \Box_i \end{array}$$

FIGURE 2 – Transformation de paramétricité généreuse

1. implémenter les modèles considérés composante par composante dans l'assistant de preuve ;
2. étudier les éléments du modèle que l'on peut rajouter à un langage source, syntaxiquement proche de Gallina ;
3. implémenter la traduction de la source vers le modèle sur l'AST fourni par MetaCoq en réifiant les composantes ;
4. enfin, écrire des termes dans le langage source pour expérimenter avec les fonctionnalités.

Dans le cadre du sous-typage, la dernière étape pourrait être étendue avec une phase d'élaboration : MetaCoq serait employé pour récupérer des termes Coq sans coercion avant typage, et l'inférence de types serait instrumentée pour insérer les coercions à la volée. Cette infrastructure permettrait de réutiliser de nombreuses composantes de Coq dont notamment l'algorithme de conversion.

4 Ébauches de Modèles pour le Sous-Typage

Dans cette section, nous proposons deux ébauches de modèles inspirés par la littérature sur le sous-typage qui ont pour vocation de guider la résolution des problèmes présentés en Section 2.

4.1 Modèle de Sous-Ensembles

Notre premier modèle s'inspire des types à raffinement : on souhaite interpréter un type par une paire dépendante $\{C \mid P\}$ d'un type sous-jacent C et d'un prédicat P sur ce type. Ainsi, $\{C \mid P\}$ peut être vu comme un raffinement de C. La paramétricité *généreuse* de [Bou18], présentée en Fig. 2, interprète un type $A : \Box_i$ par une paire $[A] := ([A]_0, [A]_1)$ constituée d'un type $[A]_0 : \Box_i$ et d'une famille $[A]_1 : [A]_0 \to \Box_i$, et fournit donc un modèle proche. Dans l'esprit des systèmes de types à raffinement, on pourrait être naïvement amené à définir une relation $\subset$ sur les types par :

$$A \subset B \qquad \Longleftrightarrow \qquad \Sigma(h : [A]_0 = [B]_0) \quad \forall a : [A]_0, [A]_1\, a \to [B]_1(\texttt{cast}\, h\, a) \tag{1}$$

où $\texttt{cast} : \forall\{X\}\{P : X \to \Box\}\{x\ y : X\}, x = y \to P\,x \to P\,y$.

Cependant, quand $[B]_1$ n'est pas à valeur dans des propositions, par exemple $B = \Box_i$, cette définition accepte plusieurs coercions extensionnellement distinctes là où notre intuition de sous-typage en réclame une seule. Plusieurs réponses sont possibles face à cette difficulté. On pourrait être tentés de quotienter les familles de types afin qu'elles prennent leurs valeurs dans des propositions $P \mapsto \| P \|$. KELLER et LASSON [KL12] expliquent en détail en quoi cette option est incompatible avec les univers – on ne peut pas extraire un prédicat sur un type A à partir

de la proposition $\| A \to \square_i \|$ – et résolvent ce problème en modifiant la hiérarchie d'univers. À défaut, on pourrait essayer de quotienter la relation $\subset$, mais on perd alors l'interprétation des coercions comme fonctions.

Ne pouvant pas écraser brutalement cette structure, on peut essayer de l'apprivoiser : à la place d'une structure d'ordre, nous avons une structure plus riche de catégorie. Ce chemin mène tout droit vers l'algèbre supérieure, sujet riche et complexe que nous laisseront ici de côté (voir [LH11 ; Nuy15 ; RS17]).

Prenant en compte ces difficultés, on modifie donc le modèle afin que le prédicat propositionnel [2] équipant chaque type soit internalisé dans l'univers, c'est à dire $[\![\square_i]\!] \cong \Sigma(C : \square_i) C \to \mathrm{Prop}$ (voir Fig. 3). On équipe l'univers du prédicat $\lambda X.\top$ toujours valide. Un terme arbitraire $u : A$ est alors traduit en une paire $[u] := ([u]_0 : [A]_C, [u]_1 : [A]_P\, [u]_0)$ d'un élément du type sous-jacent $[A]_C$ et d'une preuve que le prédicat $[A]_P$ tient pour cet élément.

Dans la traduction des produits dépendants, le codomaine n'est pas raffiné dans le type sous-jacent ce qui permet au produit dépendant d'être covariant vis-à-vis du codomaine. Par contre, le domaine est nécessairement raffiné dans le type sous-jacent – comme dans la paramétricité généreuse – car on ne veut considérer que les éléments du domaine qui valident le prédicat considéré. Par conséquent ce modèle ne valide pas la contravariance dans le domaine.

Théorème 1. *La traduction sous-ensemble est un modèle syntaxique de CIC dans CIC :*

$$\begin{aligned} \Gamma \vdash_{CIC} t : A \quad &\Longrightarrow \quad [\![\Gamma]\!] \vdash_{CIC} [t] : [\![A]\!] \\ \Gamma \vdash_{CIC} A \equiv B \quad &\Longrightarrow \quad [\![\Gamma]\!] \vdash_{CIC} [\![A]\!] \equiv [\![B]\!] \end{aligned}$$

Ce théorème fournit la première étape : on souhaite ensuite étendre la théorie source, c'est à dire CIC, avec un jugement de sous-typage décidable et des types à raffinement. Ce jugement de sous-typage se traduit par la formule (1). Pour un exemple concret de type à raffinement, on peut étendre la source avec un type primitif Even, traduit par $[\mathsf{Even}] := (([\mathbb{N}]_C, \lambda n.\ \texttt{is\_even}\ n), \sharp\sharp)$, ainsi qu'une relation $\Gamma \vdash \mathsf{Even} <: \mathbb{N}$ puisque cette relation est validée par le modèle. Cette théorie étendue serait toujours consistante, puisque $[\![\bot]\!]$ est vide et on conjecture que l'on peut préserver la canonicité tout en obtenant des règles de sous-typage intéressantes. Par exemple, la traduction des listes montre que l'on pourrait valider la monotonicité de `list` vis-à-vis du sous-typage, fournissant ainsi une réponse concrète à [Hri14].

4.2 Modèle de Préordres

Le modèle de préordre équipe chaque type avec une relation de préordre. Il permet d'internaliser la notion de constructions monotones et antitones en théorie des types. Formellement, on interprète un type A par un préordre, c'est à dire un quadruplet $[\![A]\!] = (|A|, \leq^A, \texttt{refl}_A, \texttt{trans}_A)$ constitué d'un type sous-jacent $|A|$, équipé d'une relation $\leq^A : |A| \to |A| \to \mathrm{Prop}$ réflexive et transitive

$$\texttt{refl}_A : \forall (a : |A|), a \leq^A a \qquad \texttt{trans}_A : a_0 \leq^A a_1 \wedge a_1 \leq^A a_2 \to a_0 \leq^A a_2$$

Une famille de type $x : A \vdash B$ s'interprète comme un foncteur de A vu comme une catégorie vers la catégorie des préordres et des applications monotones.

Étant donnés un préordre A et une famille $x : A \vdash B$, on peut construire deux produits dépendants, le produit monotone $\Pi^{\mathrm{mon}}(x : A).B$ et le produit antitone $\Pi^{\mathrm{anti}}(x : A).B$ qui

2. Dans ce contexte, une proposition $p : \mathrm{Prop}$ est considérée comme un type avec au plus un habitant, un sous-singleton. Afin d'obtenir des modèles syntaxiques, on demandera parfois qu'il y ait au plus un habitant *définitionnellement*, c'est à dire que la proposition soit stricte [Gil+19].

$$
\begin{array}{rlrl}
[\Pi(a:A).\ B]_C &:= \Pi(a:[\![A]\!]).\ \pi_1\ [B]_0 & [\Pi(a:A).\ B]_P &:= \lambda f.\ \Pi(a:[\![A]\!]).\ \pi_2\ [B]_0\ (f\ a) \\
[\square_i]_C &:= \Sigma(X:\square_i).\ X \to \mathrm{Prop} & [\square_i]_P &:= \lambda X.\ \top \\
[\mathbb{N}]_C &:= \mathbb{N} & [\mathbb{N}]_P &:= \ \ldots\ \equiv \lambda X.\ \top
\end{array}
$$

$$
\begin{array}{rlrl}
[\Pi(a:A).\ B]_0 &:= ([\Pi(a:A).\ B]_C, [\Pi(a:A).\ B]_P) & [\Pi(a:A).\ B]_1 &:= \not\Vdash \\
[\square_i]_0 &:= ([\square_i]_C, [\square_i]_P) & [\square_i]_1 &:= \not\Vdash \\
[\mathbb{N}]_0 &:= ([\mathbb{N}]_C, [\mathbb{N}]_P) & [\mathbb{N}]_1 &:= \not\Vdash \\
[M\ N]_0 &:= [M]_0\ [N] & [M\ N]_1 &:= [M]_1\ [N] \\
[\lambda(a:A).\ M]_0 &:= \lambda(a:[\![A]\!]).\ [M]_0 & [\lambda(a:A).\ M]_1 &:= \lambda(a:[\![A]\!]).\ [M]_1 \\
[x]_0 &:= \pi_1\ x & [x]_1 &:= \pi_2\ x
\end{array}
$$

FIGURE 3 – Traduction sous-ensemble

contiennent les fonctions $\Pi(x : A).B$ respectivement monotones et antitones. Équipés de la relation induite par l'ordre points-à-points, ces types forment des préordres. Ces constructions s'étendent fonctoriellement : étant données une application monotone $f_A : A_1 \to A_0$ et une transformation naturelle $f_B : B_0 \circ f_A \dot{\to} B_1$, on construit des applications monotones $\Pi^\star(x : A_0).B_0 \to \Pi^\star(x : A_1).B_1$. Nous laissons le détail de ces constructions à l'ébauche de formalisation accompagnant cette note [LM20], et notons que cette construction est contravariante vis-à-vis du domaine et covariante vis-à-vis du codomaine.

Les types inductifs [3] s'interprètent munis de leur relation de paramétricité [BL11]. Sur des inductifs sans paramètre tels que $\mathbb{N}$ ou $\mathbb{B}$, cela revient à considérer l'ordre discret, c'est à dire l'égalité intensionnelle. Sur des listes list A, l'ordre de A est étendu composantes à composantes entre listes de même taille. Sur des sommes dépendantes $\Sigma(x : A).B$, on obtient naturellement l'ordre lexicographique.

La traduction doit équiper les univers $[\square_i] := \mathbb{U}_i$ avec un ordre de manière compatible avec les familles de types : si $X \leq^{\mathbb{U}_i} Y$ on doit pouvoir construire une fonction monotone *canonique* de X dans Y. En particulier, on ne peut pas refléter toutes les fonctions (monotones) entre les types de notre univers : si $\mathbb{B}$ représente les booléens munis de l'ordre discret, il y a 4 applications $\mathbb{B} \to \mathbb{B}$ dont deux bijections, l'identité et la négation, que l'on ne peut pas distinguer à priori dans CIC – cela fournirait une réfutation de l'univalence [Uni13]. Il est donc nécessaire de rajouter une composante intensionnelle à nos types afin de pouvoir trier entre les "bons" morphismes – qui seront capturés par l'ordre sur l'univers – et les autres.

Techniquement, il s'agit d'interpréter un type par un *code*, un élément d'un inductif conservant les "plans de constructions" du type. Les codes sont ensuite interprétés comme des préordres à l'aide d'une fonction de décodage, voir Fig. 4. De manière similaire, une relation est définie inductivement sur la structure des codes puis décodée en un morphisme entre le décodage respectif des codes.

Extension du sous-typage Si le modèle permet de parler de monotonie, les types introduits jusqu'ici ne font que propager un ordre. En enrichissant l'univers de types avec des relations particulières, on peut tenter de modéliser plusieurs disciplines de sous-typage de la Section 2 :

Cumulativité explicite on ajoute des relations $\mathfrak{u}_i \leq^{\mathbb{U}} \mathfrak{u}_j$ dès que $i \leq j$, et on interprète ces relations par l'application monotone plongeant $\mathbb{U}_i$ dans $\mathbb{U}_j$.

3. Dans un premier temps, on ne considère pas les inductifs avec indices.

Univers $\mathbb{U}_i$ et fonction de décodage $\mathrm{El} : \mathbb{U}_i \to \Box$

$$\frac{}{\mathfrak{nat} \in \mathbb{U}} \qquad \frac{}{\mathfrak{bool} \in \mathbb{U}} \qquad \frac{A \in \mathbb{U}}{\mathfrak{list}\, A \in \mathbb{U}} \qquad \frac{i < j}{\mathfrak{u}_i \in \mathbb{U}_j}$$

$$\frac{A \in \mathbb{U} \qquad B \in \Pi^{\mathrm{mon}}(a : \mathrm{El}\, A)\, \mathbb{U}}{\mathfrak{s}\, A\, B \in \mathbb{U}} \qquad \frac{\star \in \{\mathtt{mon}, \mathtt{anti}\} \qquad A \in \mathbb{U}_i \qquad B \in \Pi^{\mathrm{mon}}(a : \mathrm{El}\, A)\, \mathbb{U}}{\pi\, \star\, A\, B \in \mathbb{U}}$$

$$\begin{aligned} \mathrm{El}\, \mathfrak{nat} &= \mathbb{N} & \mathrm{El}\, \mathfrak{bool} &= \mathbb{B} \\ \mathrm{El}\, (\mathfrak{list}\, A) &= \mathrm{list}\, (\mathrm{El}\, A) & \mathrm{El}\, \mathfrak{u}_k &= \mathbb{U}_k \\ \mathrm{El}\, (\mathfrak{s}\, A\, B) &= \Sigma(x : A).B & \mathrm{El}\, (\pi\, \star\, A\, B) &= \Pi^{\star}(a : \mathrm{El}\, A)\ \mathrm{El}(B\, a) \end{aligned}$$

Ordre $\leq^{\mathbb{U}}$ sur les codes

$$\frac{}{\mathfrak{nat} \leq^{\mathbb{U}} \mathfrak{nat}}\ \mathfrak{nat}\text{-}\leq^{\mathbb{U}} \qquad \frac{}{\mathfrak{u}_k \leq^{\mathbb{U}} \mathfrak{u}_k}\ \mathfrak{u}\text{-}\leq^{\mathbb{U}} \qquad \frac{A \leq^{\mathbb{U}} A'}{\mathfrak{list}\, A \leq^{\mathbb{U}} \mathfrak{list}\, A'}\ \mathfrak{list}\text{-}\leq^{\mathbb{U}}$$

$$\frac{A_\varepsilon : A' \leq^{\mathbb{U}} A \qquad a : \mathrm{El}\, A, a' : \mathrm{El}\, A', a_\epsilon : \mathrm{El}_\varepsilon\, A_\varepsilon\ a \leq^{A'} a' \vdash B\, a \leq^{\mathbb{U}} B'\, a'}{\pi\, \star\, A\, B \leq^{\mathbb{U}} \pi\, \star\, A'\, B'}\ \pi\text{-}\leq^{\mathbb{U}}$$

FIGURE 4 – Univers de code, relation d'ordre et décodage

Sous-typage en largeur étant fixé un type $\mathcal{L}$ d'étiquettes muni d'une égalité décidable, on étend l'univers avec des types sommes étiquetés et des types enregistrements avec champs étiquetés. La relation de sous-typage entre ces types est induite par l'inclusion des ensembles d'étiquettes combinées au sous-typage structurel, par exemple :

$$A <: B \qquad \Longrightarrow \qquad \{\ \ell_1 : A;\ \ell_0 : X\ \} <: \{\ \ell_1 : B\ \}$$

Comme pour le modèle des sous-ensembles présenté en Section 4.1, les éléments ci-dessus permettent d'assembler un modèle syntaxique de CIC. Cependant, afin de tirer pleinement parti de l'aspect polarisé du modèle de préordre, la théorie source devrait être largement étendue. NUYTS [Nuy15] propose d'employer 3 modalités `discr`/`op`/`codiscr` pour manipuler l'ordre sur les types, néanmoins une expérience préliminaire en Agda laisse penser que la théorie résultante semble difficile à utiliser en l'état et sera au cœur des recherches ultérieures sur ce sujet.

5 Conclusion

Nous avons présenté et motivé une approche à base de modèles syntaxiques pour étudier plusieurs disciplines de sous-typage en présence de types dépendants. Des formalisation en Coq et en Agda des deux modèles ébauchés dans la section précédente sont en cours de développement, première étape concrète de cette étude. Nous espérons que notre démarche pourra aider à consolider la littérature déjà conséquente mais morcelée du sous-typage en présence de types dépendants, et plus généralement qu'elle montrera l'intérêt d'employer des assistants à la preuve tout au long de la conception de langages de programmation.

Références

[Abe08a] Andreas Abel. « Polarised subtyping for sized types ». In : *Math. Struct. Comput. Sci.* 18.5 (2008), p. 797-822. DOI : 10.1017/S0960129508006853. URL : https://doi.org/10.1017/S0960129508006853.

[Abe08b] Andreas Abel. « Semi-continuous Sized Types and Termination ». In : *Logical Methods in Computer Science* Volume 4, Issue 2 (avr. 2008). DOI : 10.2168/LMCS-4(2:3)2008. URL : https://lmcs.episciences.org/1236.

[AGJ14] Robert Atkey, Neil Ghani et Patricia Johann. « A relationally parametric model of dependent type theory ». In : *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 2014, p. 503-516. DOI : 10.1145/2535838.2535852. URL : https://doi.org/10.1145/2535838.2535852.

[Ahm+17] Danel Ahman et al. « Dijkstra Monads for Free ». In : *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, jan. 2017, p. 515-529. DOI : 10.1145/3009837.3009878. URL : https://www.fstar-lang.org/papers/dm4free/.

[Ahm+18] Danel Ahman et al. « Recalling a Witness : Foundations and Applications of Monotonic State ». In : *PACMPL* 2.POPL (jan. 2018), 65 :1-65 :30. URL : https://arxiv.org/abs/1707.02466.

[AK16] Thorsten Altenkirch et Ambrus Kaposi. « Type theory in type theory using quotient inductive types ». In : *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, p. 18-29. DOI : 10.1145/2837614.2837638. URL : https://doi.org/10.1145/2837614.2837638.

[Alt+19] Thorsten Altenkirch et al. « Setoid Type Theory - A Syntactic Translation ». In : *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*. 2019, p. 155-196. DOI : 10.1007/978-3-030-33636-3_7. URL : https://doi.org/10.1007/978-3-030-33636-3_7.

[AP13] Andreas M. Abel et Brigitte Pientka. « Wellfounded Recursion with Copatterns : A Unified Approach to Termination and Productivity ». In : *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA : Association for Computing Machinery, 2013, 185–196. ISBN : 9781450323260. DOI : 10.1145/2500365.2500591. URL : https://doi.org/10.1145/2500365.2500591.

[Ass14] Ali Assaf. « A Calculus of Constructions with Explicit Subtyping ». In : *20th International Conference on Types for Proofs and Programs, TYPES 2014, May 12-15, 2014, Paris, France*. 2014, p. 27-46. DOI : 10.4230/LIPIcs.TYPES.2014.27. URL : https://doi.org/10.4230/LIPIcs.TYPES.2014.27.

[Ber15] Bruno Bernardo. « Un Calcul des Constructions implicite avec sommes dépendantes et à inférence de type décidable. » Version soutenance. Theses. École polytechnique, oct. 2015. URL : https://hal.inria.fr/tel-01197380.

[BL11] Jean-Philippe Bernardy et Marc Lasson. « Realizability and Parametricity in Pure Type Systems ». In : *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. 2011, p. 108-122. DOI : 10.1007/978-3-642-19805-2_8. URL : https://doi.org/10.1007/978-3-642-19805-2_8.

[Bou18] Simon Boulier. « Extending type theory with syntactic models ». Theses. Ecole nationale supérieure Mines-Télécom Atlantique, nov. 2018. URL : https://tel.archives-ouvertes.fr/tel-02007839.

[BPT17a] Simon Boulier, Pierre-Marie Pédrot et Nicolas Tabareau. « Modèles de la théorie des types donnés par traduction de programme ». In : *28ièmes Journées Francophones des Langages Applicatifs*. Gourette, France, jan. 2017. URL : https://hal.archives-ouvertes.fr/hal-01503089.

[BPT17b] Simon Boulier, Pierre-Marie Pédrot et Nicolas Tabareau. « The next 700 syntactical models of type theory ». In : *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. 2017, p. 182-194. DOI : 10.1145/3018610.3018620. URL : https://doi.org/10.1145/3018610.3018620.

[Car86] John Cartmell. « Generalised algebraic theories and contextual categories ». In : *Annals of Pure and Applied Logic* 32 (1986), p. 209 -243. ISSN : 0168-0072. DOI : https://doi.org/10.1016/0168-0072(86)90053-9. URL : http://www.sciencedirect.com/science/article/pii/0168007286900539.

[Dyb95] Peter Dybjer. « Internal Type Theory ». In : *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*. 1995, p. 120-134. DOI : 10.1007/3-540-61780-9_66. URL : https://doi.org/10.1007/3-540-61780-9_66.

[ER03] Thomas Ehrhard et Laurent Regnier. « The differential lambda-calculus ». In : *Theor. Comput. Sci.* 309.1-3 (2003), p. 1-41. DOI : 10.1016/S0304-3975(03)00392-X. URL : https://doi.org/10.1016/S0304-3975(03)00392-X.

[Gil+19] Gaëtan Gilbert et al. « Definitional proof-irrelevance without K ». In : *Proc. ACM Program. Lang.* 3.POPL (2019), 3 :1-3 :28. DOI : 10.1145/3290316. URL : https://doi.org/10.1145/3290316.

[Gir87] Jean-Yves Girard. « Linear logic ». In : *Theoretical Computer Science* 50.1 (1987), p. 1 -101. ISSN : 0304-3975. DOI : https://doi.org/10.1016/0304-3975(87)90045-4. URL : http://www.sciencedirect.com/science/article/pii/0304397587900454.

[Hof95] Martin Hofmann. « Conservativity of Equality Reflection over Intensional Type Theory ». In : *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*. 1995, p. 153-164. DOI : 10.1007/3-540-61780-9_68. URL : https://doi.org/10.1007/3-540-61780-9_68.

[Hri14] Catalin Hritcu. *Issue #65 on Fstar's bug tracker "Polarities : subtyping for datatypes"*. https://github.com/FStarLang/FStar/issues/65. 2014. URL : https://github.com/FStarLang/FStar/issues/65.

[KL12] Chantal Keller et Marc Lasson. « Parametricity in an Impredicative Sort ». In : *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*. 2012, p. 381-395. DOI : 10.4230/LIPIcs.CSL.2012.381. URL : https://doi.org/10.4230/LIPIcs.CSL.2012.381.

[LA08] Zhaohui Luo et Robin Adams. « Structural subtyping for inductive types with functorial equality rules ». In : *Math. Struct. Comput. Sci.* 18.5 (2008), p. 931-972. DOI : 10.1017/S0960129508006956. URL : https://doi.org/10.1017/S0960129508006956.

[Lev03] Paul Blain Levy. *Call-By-Push-Value*. en. Dordrecht : Springer Netherlands, 2003. ISBN : 978-94-010-3752-5 978-94-007-0954-6. DOI : 10.1007/978-94-007-0954-6. URL : http://link.springer.com/10.1007/978-94-007-0954-6 (visité le 15/07/2020).

[LH11] Daniel R. Licata et Robert Harper. « 2-Dimensional Directed Type Theory ». In : *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 2011, Pittsburgh, PA, USA, May 25-28, 2011*. 2011, p. 263-289. DOI : 10.1016/j.entcs.2011.09.026. URL : https://doi.org/10.1016/j.entcs.2011.09.026.

[LL16] Georgiana E. Lungu et Zhaohui Luo. « On Subtyping in Type Theories with Canonical Objects ». In : *22nd International Conference on Types for Proofs and Programs, TYPES 2016, May 23-26, 2016, Novi Sad, Serbia*. 2016, 13 :1-13 :31. DOI : 10.4230/LIPIcs.TYPES.2016.13. URL : https://doi.org/10.4230/LIPIcs.TYPES.2016.13.

[LM20] Théo Laurent et Kenji Maillard. *Développement Coq accompagnant cet article*. 2020. URL : https://gitlab.inria.fr/kmaillar/subcic.

[LS99] Zhaohui Luo et Sergei Soloviev. « Dependent Coercions ». In : *Conference on Category Theory and Computer Science, CTCS 1999, Edinburgh, UK, December 10-12, 1999*. 1999, p. 152-168. DOI : 10.1016/S1571-0661(05)80314-7. URL : https://doi.org/10.1016/S1571-0661(05)80314-7.

[Luo88] Zhaohui Luo. *CCômega_subset and its metatheory*. https://www.lfcs.inf.ed.ac.uk/reports/88/ECS-LFCS-88-58/. 1988.

[Luo96] Zhaohui Luo. « Coercive Subtyping in Type Theory ». In : *Computer Science Logic, 10th International Workshop, CSL '96, Annual Conference of the EACSL, Utrecht, The Netherlands, September 21-27, 1996, Selected Papers*. 1996, p. 276-296. DOI : 10.1007/3-540-63172-0_45. URL : https://doi.org/10.1007/3-540-63172-0_45.

[McB00] Conor McBride. « Dependently typed functional programs and their proofs ». Thèse de doct. University of Edinburgh, UK, 2000. URL : http://hdl.handle.net/1842/374.

[Miq01] Alexandre Miquel. « Le Calcul des Constructions implicite : syntaxe et sémantique ». Theses. Université Paris 7, déc. 2001. URL : https://www.fing.edu.uy/~amiquel/publis/these.pdf.

[Mog89] Eugenio Moggi. « Computational Lambda-Calculus and Monads ». In : *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. 1989, p. 14-23. DOI : 10.1109/LICS.1989.39155. URL : https://doi.org/10.1109/LICS.1989.39155.

[Nuy15] Andreas Nuyts. « Toward a Directed Homotopy Type Theory based on 4 Kinds of Variance ». Mém. de mast. Katholieke Universiteit Leuven, 2015. URL : https://anuyts.github.io/files/mathesis.pdf.

[Péd+19] Pierre-Marie Pédrot et al. « A reasonably exceptional type theory ». In : *Proc. ACM Program. Lang.* 3.ICFP (2019), 108 :1-108 :29. DOI : 10.1145/3341712. URL : https://doi.org/10.1145/3341712.

[PT17] Pierre-Marie Pédrot et Nicolas Tabareau. « An effectful way to eliminate addiction to dependence ». In : *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. 2017, p. 1-12. DOI : 10.1109/LICS.2017.8005113. URL : https://doi.org/10.1109/LICS.2017.8005113.

[PT18] Pierre-Marie Pédrot et Nicolas Tabareau. « Failure is Not an Option - An Exceptional Type Theory ». In : *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 2018, p. 245-271. DOI : 10.1007/978-3-319-89884-1_9. URL : https://doi.org/10.1007/978-3-319-89884-1_9.

[RS17] Emily Riehl et Michael Shulman. « A type theory for synthetic ∞-categories ». en. In : *Higher Structures* 1.1 (2017), p. 78. URL : https://journals.mq.edu.au/index.php/higher_structures/article/view/36.

[Soz07] Matthieu Sozeau. « Subset coercions in Coq ». In : *Types for Proofs and Programs*. T. 4502. 2007, p. 237-252.

[Soz+20] Matthieu Sozeau et al. « Coq Coq correct ! verification of type checking and erasure for Coq, in Coq ». In : *Proc. ACM Program. Lang.* 4.POPL (2020), 8 :1-8 :28. DOI : 10.1145/3371076. URL : https://doi.org/10.1145/3371076.

[Swa+16] Nikhil Swamy et al. « Dependent Types and Multi-Monadic Effects in F* ». In : *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, jan. 2016, p. 256-270. ISBN : 978-1-4503-3549-2. URL : https://www.fstar-lang.org/papers/mumon/.

[TS18] Amin Timany et Matthieu Sozeau. « Cumulative Inductive Types In Coq ». In : *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*. 2018, 29 :1-29 :16. DOI : 10.4230/LIPIcs.FSCD.2018.29. URL : https://doi.org/10.4230/LIPIcs.FSCD.2018.29.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory : Univalent Foundations of Mathematics*. Institute for Advanced Study : https://homotopytypetheory.org/book, 2013.

[WST19] Théo Winterhalter, Matthieu Sozeau et Nicolas Tabareau. « Eliminating reflection from type theory ». In : *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. 2019, p. 91-103. DOI : 10.1145/3293880.3294095. URL : https://doi.org/10.1145/3293880.3294095.

Théories de permutations avec Why3

Alain Giorgetti

Institut FEMTO-ST, UMR CNRS 6174
Univ. of Bourgogne Franche-Comté
16 route de Gray, F-25030 Besançon Cedex, France
alain.giorgetti@femto-st.fr

Résumé

Ce travail s'inscrit dans le cadre d'une étude générale de la pertinence de l'application des méthodes formelles du génie logiciel à la recherche en combinatoire. Dans cet article, nous expérimentons l'adéquation de l'outil de preuve formelle Why3 pour vérifier mécaniquement le contenu d'un article de combinatoire récent, sur des théories du premier ordre pour la notion de permutation sur un ensemble fini. La première théorie voit une permutation comme une fonction bijective, la seconde comme un couple d'ordres totaux stricts. Nous démontrons formellement avec Why3 certaines propriétés des modèles de ces deux théories. Nous introduisons ensuite une troisième théorie réduite à un ordre total strict sur un intervalle fini d'entiers. Nous démontrons formellement la propriété caractéristique de ses modèles, modulo un lemme que nous testons avec l'outil AutoCheck de test aléatoire et énumératif de propriétés OCaml et Why3.

1 Introduction

Les permutations sont des objets mathématiques fondamentaux, abondamment étudiés, notamment en combinatoire. En informatique, une permutation permet par exemple de spécifier qu'un programme qui trie une collection de données préserve son contenu. Très récemment, Michael Albert, Mathilde Bouvel et Valentin Féray ont étudié deux théories des permutations sur un ensemble fini, dans le cadre de la logique du premier ordre [ABF20]. Une permutation y est d'abord vue comme une bijection, puis comme un ordre total sur un ensemble fini lui-même totalement ordonné. Nous souhaitons étudier l'adéquation de l'environnement de vérification déductive Why3 [BFM+20] à la formalisation de ces théories et au raisonnement formel sur ses modèles.

Nous avons formalisé ces deux théories de permutations dans le langage WhyML de Why3, puis nous avons démontré formellement des propriétés de leurs modèles finis, avec Why3 et Coq [Coq20], comme résumé dans les parties 2 et 3. La partie 4 résume notre proposition de simplification de la seconde théorie, dans laquelle l'ensemble fini permuté est un intervalle fini d'entiers, naturellement ordonné, ce qui permet de ne considérer qu'un ordre total au lieu de deux. Comme modèles possibles de cette troisième théorie, nous avons défini deux fonctions qui devraient être des permutations mutuellement inverses. Dans la démonstration formelle de la propriété caractéristique des modèles de cette théorie, il ne reste plus qu'à démontrer qu'une de ces deux fonctions est inverse de l'autre. Nous avons testé cette conjecture à l'aide d'AutoCheck [EG20], un nouvel outil de test aléatoire et énumératif de propriétés OCaml et Why3.

Cette étude de cas de combinatoire certifiée [GM21] est détaillée dans un rapport de recherche de 17 pages [Gio20] qui rend la démarche et le code accessibles à tout étudiant ou combinatoricien, même peu familier avec la spécification et la preuve formelle, afin qu'il puisse s'en inspirer pour pratiquer ces méthodes. Peu de connaissances de Why3 sont requises pour comprendre le code présenté. En effet, chaque extrait de code est commenté, et la formalisation

est proche du texte mathématique initial. En complément du manuel de Why3, deux articles récents détaillent davantage le mécanisme de clonage de modules [FP20] et la notion de type avec invariant [Fil20] utilisés. Pour l'audience des JFLA, nous résumons ce travail (parties 2, 3 et 4), puis nous situons nos contributions par rapport à l'état de l'art (partie 5) et nous en présentons quelques perspectives (partie 6).

Le code est développé et vérifié avec Why3 1.3.3, Coq 8.9.0, AutoCheck 0.1.1 et les solveurs SMT Alt-Ergo 2.2.0, CVC4 1.6 et Z3 4.7.1. Le code, sa documentation, les démonstrations formelles et leurs statistiques sont accessibles à partir de la page `https://alaingiorgetti.github.io/autocheck/`.

La plateforme de vérification déductive Why3 [BFM+20] utilisée est fondée sur une logique classique du premier ordre avec types. Elle propose le langage WhyML pour la spécification fonctionnelle de programmes impératifs et fonctionnels, avec des annotations formelles (préconditions, postconditions, assertions intermédiaires, invariants et variants de boucle, etc.). Le système utilise toutes ces annotations pour générer des *conditions de vérification*, aussi appelées *buts*, qui sont des formules WhyML. Ensuite, des prouveurs automatiques (comme les solveurs SMT) et des assistants de preuve (comme Coq) peuvent être utilisés pour prouver ces buts. De plus, Why3 propose de nombreuses transformations logiques pour simplifier ces buts, jusqu'à obtenir soit une preuve 100% interactive, soit des sous-buts assez simples pour être déchargés automatiquement. Why3 propose quatre *stratégies* de recherche de preuve, nommées Auto level 0 à 3, qui sont des combinaisons de transformations logiques et d'appels aux prouveurs automatiques, qui donnent souvent de bons résultats, dans une limite de temps croissante selon leur numéro. Nous dirons qu'une preuve avec Why3 est *automatique* si elle est obtenue par l'une de ces stratégies, et qu'elle est *interactive* sinon, lorsqu'au moins une transformation Why3 a été appliquée à la main. Enfin, Why3 offre un mécanisme d'extraction automatique des programmes vers divers langages cibles, dont OCaml, pour les exécuter.

2 Une permutation est une bijection

Dans [ABF20], la première théorie, nommée TOOB (T*heory* O*f* O*ne* B*ijection*), voit une permutation comme un symbole relationnel R binaire bijectif. Les axiomes de cette théorie ne sont pas précisés.

La librairie standard de Why3 ne contient pas de tels axiomes. Nous avons donc axiomatisé en WhyML qu'un prédicat binaire `rel` quelconque est fonctionnel, injectif, surjectif et bijectif, au sens des définitions suivantes : le *graphe* d'une fonction f de A dans B est l'ensemble $\{(x, f(x)) \mid x \in A\}$ des couples composés d'un élément x du *domaine* A de la fonction et de son *image* $f(x)$ par la fonction. Une relation binaire est dite *fonctionnelle* si elle est le graphe d'une fonction, et *injective*, *surjective* ou *bijective* si cette fonction l'est.

Ainsi, le code WhyML

```
module TOOB
  type t
  predicate rel t t
  clone export Bijectivity with type t = t, type u = t, predicate rel = rel, axiom .
end
```

formalise la théorie TOOB. Sa signature est réduite au prédicat binaire `rel`. Ses axiomes sont obtenus par instanciation de notre module `Bijectivity` qui caractérise une relation bijective entre un type `t` et un type `u`. La notation `axiom .` signifie que tous les axiomes de ce module sont admis.

Les auteurs de [ABF20] démontrent que les fonctions bijectives σ sur l'ensemble $[1..n]$ des n premiers entiers naturels non nuls sont des modèles de la théorie TOOB. Nous formalisons tout intervalle d'entiers relatifs $[l..u] = \{l, \ldots, u\}$ non vide ($l \leq u$) par un type WhyML d'entiers bornés `bint` (pour b*ounded* int*eger*) défini comme un enregistrement à un seul champ `to_int` du type `int` des entiers relatifs de Why3, soumis à l'invariant de type ($l \leq$ `to_int` $\leq u$). [Gio20, partie 2.3]. Nous formalisons ensuite σ par

```
val constant sigma : map bint bint
axiom Sigma_bij : bijective sigma
```

où `map` est le type polymorphe des applications, défini dans la librairie standard par

```
type map 'a 'b = 'a → 'b
```

et `bijective` est le prédicat que nous avons défini par

```
predicate injective (m: map 'a 'b) = ∀ i j: 'a. m[i] = m[j] → i = j
predicate surjective (m: map 'a 'b) = ∀ j: 'b. ∃ i: 'a. m[i] = j
predicate bijective (m: map 'a 'b) = injective m ∧ surjective m
```

Le modèle de TOOB associé à la permutation σ de $[1..n]$ est le couple (A^σ, R^σ) où $A^\sigma = [1..n]$ et R^σ est tel que $i\ R^\sigma\ j$ si et seulement si $\sigma(i) = j$ [ABF20, partie 2.2]. Cette correspondance entre σ et R^σ est formalisée par le prédicat

```
predicate rel_sigma (i j: bint) = map_rel sigma i j
```

où le prédicat `map_rel` défini par

```
predicate map_rel (m: map 'a 'b) (x: 'a) (y: 'b) = (m[x] = y)
```

associe son graphe à toute application. L'instanciation

```
clone TOOB with type t = bint, predicate rel = rel_sigma
```

du module `TOOB` formalise que le couple (A^σ, R^σ) est un modèle de la théorie TOOB. Ce clonage sans le mot-clé WhyML `axiom` exige la démonstration des instances correspondantes des axiomes du module `TOOB`. Ils sont prouvés automatiquement à l'aide d'Alt-Ergo, CVC4 ou Z3, sauf l'axiome de surjectivité, qui requiert une preuve interactive composée de deux transformations Why3.

Enfin, 17 lignes de code WhyML [Gio20, Listing 3] formalisent que tout modèle (A, R) de TOOB est isomorphe à un modèle de permutation (A^σ, R^σ) [ABF20, Proposition 2]. Cette propriété est démontrée interactivement avec Why3, à l'aide d'un lemme démontré en Coq.

3 Une permutation est un couple d'ordres totaux stricts

La théorie TOTO (T*heory* O*f* T*wo* O*rders*) de [ABF20] définit une permutation p de taille n comme un couple $(<_P, <_V)$ d'ordres totaux stricts sur le même ensemble $\{e_1, \ldots, e_n\}$ de n éléments, appelés *positions* (resp. *valeurs*) lorsqu'ils sont comparés avec $<_P$ (resp. $<_V$). L'ordre $<_P$ sur les positions est défini par $e_1 <_P \ldots <_P e_n$. L'ordre $<_V$ sur les valeurs est défini par $p(e_1) <_V \ldots <_V p(e_n)$.

Nous formalisons la théorie TOTO par

```
module TOTO
  type t               (* type for positions and values *)
  predicate ltP t t    (* strict total order on positions *)
  predicate ltV t t    (* strict total order on values *)
  clone relations.TotalStrictOrder with type t = t, predicate rel = ltP, axiom .
  clone relations.TotalStrictOrder as V with type t = t, predicate rel = ltV, axiom .
end
```

où les prédicats `ltP` et `ltV` formalisent respectivement les ordres totaux stricts $<_P$ et $<_V$. Le module `TotalStrictOrder` du fichier `relations.mlw` de la librairie standard de Why3 définit qu'une endorelation `rel` sur un type `t` est un ordre total strict si elle est transitive, asymétrique, et totale au sens de l'axiome suivant :

```
axiom Trichotomy : ∀ x y:t. rel x y ∨ rel y x ∨ x = y
```

Le modèle de la théorie TOTO associé à la permutation σ sur l'intervalle d'entiers $[l..u]$ est le triplet $(A^\sigma, <_P^\sigma, <_V^\sigma)$ où $A^\sigma = \{(i, \sigma(i)) \mid i \in [l..u]\}$ est l'ensemble des arcs du graphe de la fonction σ, et $<_P^\sigma$ (resp. $<_V^\sigma$) est l'ordre induit sur la première (resp. seconde) composante des éléments de A^σ par l'ordre naturel strict $<$ sur les entiers [ABF20]. Nous formalisons σ par

```
val constant m : map bint bint
```

et A^σ par un type `arrow` dont les habitants sont les arcs `(i, m[i])` du graphe de l'application `m`, pour tout habitant `i` du type `bint`.

Six lignes de code WhyML [Gio20, partie 4.2] suffisent pour formaliser la propriété que le triplet $(A^\sigma, <_P^\sigma, <_V^\sigma)$ est un modèle de la théorie TOTO. Les prédicats qui formalisent $<_P^\sigma$ et $<_V^\sigma$ doivent satisfaire les axiomes d'un ordre total strict. Ces démonstrations sont automatiques.

Pour tout modèle $(A, <_P, <_V)$ de TOTO, il existe une permutation σ telle que $(A, <_P, <_V)$ et $(A^\sigma, <_P^\sigma, <_V^\sigma)$ sont isomorphes [ABF20]. Construire une démonstration formelle de cette propriété est un travail conséquent et délicat, notamment en raison des lourdeurs imposées par la co-existence de deux ordres totaux. Cet objectif est laissé en perspective et le problème est simplifié en limitant A à un intervalle d'entiers et l'ordre $<_P$ à l'ordre naturel sur les entiers, comme détaillé dans la partie suivante.

4 Une permutation est un ordre total strict

Considérons les modèles $(A, <_P, <_V)$ de la théorie TOTO dans lesquels A est un intervalle d'entiers I et $<_P$ est la restriction à I de l'ordre naturel $<$ sur les entiers, notée $<_I$. Dans ces modèles, l'ordre total strict $<_V$ suffit pour décrire une permutation sur I. Pour étudier ces modèles, nous considérons une théorie de permutations composée d'un unique ordre total strict sur un intervalle d'entiers, nommée STOI (pour *Strict Total Order on one integer Interval*) [1].

Le modèle de la théorie STOI associé à la permutation σ sur l'intervalle d'entiers $[l..u]$ est le couple $([l..u], <^\sigma)$ tel que $i <^\sigma j$ si et seulement si $\sigma(i) < \sigma(j)$. La définition

```
let predicate lt_map (m: map bint bint) (i j: bint) = lt_bint m[i] m[j]
```

du prédicat et de la fonction booléenne `lt_map` formalise cette correspondance entre une application `m`, de type `(map bint bint)`, et une relation binaire, ici formalisée par une fonction booléenne de type `(bint → bint → bool)`. Le prédicat et la fonction booléenne `lt_bint`, de type `(bint → bint → bool)`, formalise l'ordre $<_{[l..u]}$. Dans cette partie toutes les fonctions sont implémentées (et certains prédicats, sous forme de fonctions booléennes), afin de pouvoir tester certains lemmes avant de chercher à les démontrer formellement.

Le code WhyML

```
val constant sigma : map bint bint
axiom sigma_bij : bijective sigma
let predicate lt_sigma (i j: bint) = lt_map sigma i j
clone export relations.TotalStrictOrder with type t = bint, predicate rel = lt_sigma
```

1. Cette terminologie de "théorie" n'est pas tout à fait correcte, car en logique mathématique le domaine des symboles relationnels et fonctionnels d'une théorie n'est pas fixé dans la théorie, mais dans ses modèles. Nous commettons néanmoins cet abus de langage par souci d'homogénéité avec les deux parties précédentes.

formalise la définition de $<^\sigma$ et la propriété que $([l..u], <^\sigma)$ est un modèle de la théorie STOI, pour toute application bijective σ sur $[l..u]$. Why3 démontre automatiquement que le prédicat `lt_sigma` satisfait les axiomes d'un ordre total strict.

Réciproquement, une permutation peut être associée à tout modèle $([l..u], <)$ de la théorie STOI, selon la propriété caractéristique suivante des modèles de cette théorie.

Proposition 1. *Pour tout ordre total strict $<$ sur l'intervalle d'entiers $[l..u]$, il existe une permutation σ sur $[l..u]$ telle que $<=<^\sigma$.*

Pour caractériser cette permutation σ, nous définissons d'abord une fonction `rank` qui associe une application sur $[l..u]$ à toute relation binaire sur $[l..u]$. Ensuite, nous justifions la proposition 1 par la conjonction des deux propriétés suivantes : (P_1) l'image (`rank` $<$) d'un ordre total strict $<$ par la fonction `rank` est une application bijective – c'est la permutation σ de la proposition 1 ; (P_2) la fonction `lt_map` est un inverse à gauche de la fonction `rank` – c'est une autre formulation de l'égalité $<=<^\sigma$ de la proposition 1.

Pour tout ordre total strict $<$ sur un ensemble fini A à n éléments, le *rang* de $a \in A$, défini par $rank(a) = card(\{b \in A \mid b < a\}) + 1$, est le nombre d'éléments de A inférieurs à a selon $<$, augmenté de 1 pour que le rang soit toujours dans l'intervalle $[1..n]$ [ABF20, page 9]. Nous généralisons cette définition à tout intervalle d'entiers $[l..u]$, par une fonction WhyML `rank` telle que `(rank lt)`$(a) = card(\{b \in [l..u] \mid$ `lt` $b\ a\}) + l$, pour toute fonction booléenne binaire `(lt : bint → bint → bool)` [Gio20, partie 5.3].

Pour tout ordre total strict `lt`, nous avons démontré formellement que la fonction `(rank lt)` est à valeurs dans l'intervalle $[l..u]$. L'ajout de trois lemmes rend cette preuve automatique avec Why3. Nous avons aussi élaboré une preuve interactive Coq pour démontrer que la fonction `(rank lt)` est injective. Pour établir qu'elle est surjective, donc bijective (propriété P_1), nous introduisons une fonction `unrank` [Gio20, partie 5.4] de telle sorte que les fonctions `(rank lt)` et `(unrank lt)` soient mutuellement inverses, au sens des deux lemmes d'inverse à gauche

```
lemma   rank_ltK: ∀ lt. totalStrictOrder lt → ∀ i. rank lt (unrank lt i) = i
lemma unrank_ltK: ∀ lt. totalStrictOrder lt → ∀ i. unrank lt (rank lt i) = i
```

où le prédicat `totalStrictOrder` spécifie que son paramètre, de type `('a → 'a → bool)`, implémente un ordre total strict. Le second lemme se démontre automatiquement avec Why3 ou interactivement avec Coq, à partir du premier lemme et de l'injectivité de `(rank lt)`, prouvée avec Coq. La démonstration du premier lemme est laissée en perspective. Ce lemme est testé avec AutoCheck [EG20], par énumération de tous les ordres totaux stricts sur l'intervalle [1..6] [Gio20, partie 5.5]. AutoCheck est un prototype d'outil de test automatique de propriétés OCaml et WhyML, par génération aléatoire et énumérative de données de test.

La propriété P_2 est formalisée par le lemme

```
lemma lt_mapK: ∀ lt. totalStrictOrder lt → lt_map (rank lt) = lt
```

qui est prouvé avec Why3, par la transformation split_vc suivie d'un appel au prouveur Z3. Ensuite, la proposition 1 est prouvée interactivement par 8 transformations Why3.

5 Travaux connexes

Dans un environnement de preuve formelle, la notion de permutation peut apparaître dans une spécification de l'existence d'une permutation entre deux structures (partie 5.1), dans des structures représentant une permutation (partie 5.2) ou dans une axiomatisation de la notion de permutation (partie 5.3). Nous détaillons ces occurrences de la notion de permutation dans

les environnements Why3 et Coq que nous utilisons, tout en positionnant nos contributions dans cette classification.

5.1 Existence d'une permutation entre deux structures

Dans la librairie standard de Coq, les prédicats `Sorting.PermutSetoid.permutation` et `Sorting.Permutation` caractérisent l'existence d'une permutation entre deux listes. Le premier est fondé sur l'égalité des contenus des listes, vus comme des multi-ensembles, pour toute égalité décidable entre leurs éléments. Le second est fondé sur la composition de transpositions.

Dans la librairie standard de Why3, les prédicats `list.Permut.permut`, `map.MapPermut.permut` et `seq.Permut.permut` caractérisent l'existence d'une permutation, respectivement entre deux listes, applications et séquences. Ils sont définis indépendamment les uns des autres et sont utilisés pour d'autres structures, par exemple dans le prédicat `array.ArrayPermut.permut` sur les tableaux mutables. Tous ces prédicats sont fondés sur l'égalité du nombre d'occurrences de chaque élément dans les deux structures.

Ces prédicats permettent par exemple de spécifier que le résultat du tri d'une structure est une permutation de la structure initiale. En complément, la librairie Coccinelle de Coq introduit une notion de permutation relative à une endorelation binaire quelconque, puis caractérise par un prédicat inductif simple l'existence d'une telle permutation entre deux listes, pour faciliter la modélisation de certains ordres pour la réécriture de termes [Con07].

Notre contexte et nos objectifs n'étant ni la vérification de tris ni la modélisation de la réécriture, mais la vérification formelle de théorèmes et de programmes de la combinatoire, nous n'apportons aucune contribution à ces caractérisations.

5.2 Représentations formelles d'une permutation

Dans un outil de vérification de théorèmes et de programmes, une *représentation formelle* d'une permutation doit permettre de spécifier des propriétés universelles et existentielles des permutations, voire d'implémenter des programmes qui agissent sur des permutations. Dans un environnement typé comme Why3 ou Coq, chaque représentation peut être définie par un type de données, éventuellement obtenu par sous-typage d'un type prédéfini, à l'aide d'un prédicat caractéristique.

Par exemple, le prédicat `math_permut` de [Con07] et le type `permut` de [DG18] formalisent la définition d'une permutation sur $[0..n-1]$ comme une endofonction bijective, à partir du type des fonctions sur les entiers naturels de Coq. En WhyML, les études de cas `inverse_in_place`[2] et [GDL19] définissent un prédicat qui caractérise les tableaux d'entiers (type prédéfini `array int`) de longueur n dont les valeurs sont dans l'intervalle $[0..n-1]$ de leurs indices et deux à deux distinctes. Quoique moins concret, le modèle d'application bijective de la présente étude – composé du type `(map bint bint)` et du prédicat caractéristique `bijective` des applications injectives et surjectives, défini dans la partie 2 – est aussi une représentation formelle possible des permutations en Why3.

5.3 Axiomatisations formelles de la notion de permutation

Dans une logique adaptée, toute caractérisation ou représentation formelle d'une permutation, dont celles des parties 5.1 et 5.2, peut inspirer une théorie de permutations, voire plusieurs.

2. http://toccata.lri.fr/gallery/inverse_in_place.en.html

Cependant, à notre connaissance, aucun travail antérieur n'aborde cette question selon la perspective des théories et de leurs modèles. De plus, aucun code public Why3 ne formalise de théorie de permutations comme nous le faisons, uniquement avec un ou deux symboles relationnels binaires.

6 Conclusion et perspectives

Nous avons présenté une formalisation originale de théories de permutations dans le langage WhyML de la plateforme de vérification déductive Why3. Les deux premières théories sont issues d'un article de combinatoire, qui étudie ensuite leur expressivité. La troisième théorie est une restriction calculatoire de la deuxième théorie, créée directement en WhyML par l'auteur. Des propriétés des modèles de ces théories sont énoncées et démontrées formellement, souvent de manière automatique, sinon de manière interactive, par des transformations Why3 ou des tactiques Coq. Un outil de test automatique est appliqué à une propriété dont la démonstration est moins immédiate.

Le code produit est proche du texte mathématique. Peu d'éléments techniques l'alourdissent, en partie grâce à la plasticité du langage WhyML. Si le code est parfois plus long que le texte mathématique, c'est surtout parce qu'il explicite des détails laissés implicites dans ce dernier, un phénomène bien connu en formalisation des mathématiques. En levant toute ambiguïté sur chaque notion, le code peut même faciliter la compréhension du texte par un non-spécialiste. Les premières propositions, considérées comme élémentaires dans le texte mathématique, ont été démontrées facilement avec Why3, automatiquement ou après une courte interaction. Les propositions suivantes, plus profondes, requièrent des efforts de spécification et de preuve formelle plus conséquents. Globalement, cette expérimentation encourage l'étude directe de certains sujets combinatoires de manière formelle, en recourant au test automatique quand la preuve devient trop chronophage, afin de ne pas trop freiner l'élan créatif.

Ce travail a également des retombées positives sur la plateforme Why3. C'est d'abord un exemple supplémentaire d'utilisation de fonctionnalités de Why3, comme son mécanisme de clonage de modules, sa notion de type avec invariant et son langage de preuve interactive. L'étude a aussi donné lieu à des développements plus généraux que son propre cadre, ré-utilisables dans d'autres applications. Enfin, l'étude contribue à l'amélioration de Why3, en identifiant des limitations et en suggérant des extensions de sa librairie standard.

Ces premiers résultats nous encouragent à poursuivre ce travail de recherche dans diverses directions. Nous envisageons d'abord de formaliser une plus grande proportion du texte mathématique de référence. Nous souhaitons aussi étendre l'étude à d'autres points de vue sur la notion de permutation, dont ceux suggérés dans la partie 5 ou ceux des produits de cycles disjoints et des nombres factoriels, aussi appelés codes de permutations [GDL19, partie 4]. Il s'agit non seulement de formaliser plusieurs théories et représentations de permutations, mais également de les relier entre elles, d'étudier leurs combinaisons et de démontrer formellement leurs propriétés. Ceci fait, il serait utile d'évaluer l'impact de chaque théorie et représentation sur le degré d'automaticité qu'elle procure pour démontrer des théorèmes ou des propriétés de programmes sur les permutations.

Une perspective majeure est d'élargir l'étude aux théories de deux permutations, voire d'ensembles de permutations, en démontrant le cas échéant leur structure de groupe. Jusqu'ici, nous n'avons considéré que des théories pour **une** permutation. Or, il est fréquemment utile de considérer conjointement une bijection et son inverse. Nous avons conçu et utilisé des variantes de modules de la librairie standard de Why3 qui axiomatisent qu'une fonction f est un inverse à gauche d'une fonction g, en déduisent que g est injective et f est surjective, puis que les

deux fonctions sont bijectives et mutuellement inverses si de plus g est un inverse à gauche de f [Gio20, partie 2.4]. Nous proposons de considérer ce code comme une théorie de deux fonctions mutuellement inverses, nommée TOTIF (T*heory* O*f* T*wo* I*nverse* F*unctions*), puis de spécialiser TOTIF en une théorie de **deux** permutations inverses.

Remerciements. L'équipe de développement de Why3 et les relecteurs anonymes sont chaleureusement remerciés pour leurs précieux conseils. Ce travail a été soutenu par l'EIPHI Graduate School (contrat ANR-17-EURE-0002).

Références

[ABF20] Michael Albert, Mathilde Bouvel, and Valentin Féray. Two first-order logics of permutations. *Journal of Combinatorial Theory, Series A*, 171 :105158, April 2020. https://arxiv.org/abs/1808.05459v2.

[BFM+20] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 Platform*, 2020. http://why3.lri.fr/manual.pdf.

[Con07] Evelyne Contejean. Modeling Permutations in Coq for Coccinelle. In *Rewriting, Computation and Proof : Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, pages 259–269. Springer, 2007.

[Coq20] The Coq Proof Assistant, 2020. http://coq.inria.fr.

[DG18] Catherine Dubois and Alain Giorgetti. Tests and proofs for custom data generators. *Formal Aspects of Computing*, 30 :659–684, Jul 2018.

[EG20] Clotilde Erard and Alain Giorgetti. AutoCheck, 2020. https://alaingiorgetti.github.io/autocheck/.

[Fil20] Jean-Christophe Filliâtre. Simpler proofs with decentralized invariants. *Journal of Logical and Algebraic Methods in Programming*, March 2020. To appear. See http://why3.lri.fr/spdi/.

[FP20] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in Why3. In *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 12476 of *Lecture Notes in Computer Science*. Springer, October 2020. http://why3.lri.fr/isola-2020/.

[GDL19] Alain Giorgetti, Catherine Dubois, and Rémi Lazarini. Combinatoire formelle avec Why3 et Coq. In Nicolas Magaud and Zaynah Dargaye, editors, *Journées Francophones des Langages Applicatifs. JFLA 2019*, pages 139–154, 2019. https://hal.inria.fr/hal-01985195.

[Gio20] Alain Giorgetti. Formalisation et vérification de théories de permutations. Rapport de recherche RR-1715, UBFC (Université de Bourgogne Franche-Comté) and FEMTO-ST, December 2020. https://hal.archives-ouvertes.fr/hal-03033416.

[GM21] Alain Giorgetti and Nicolas Magaud. Combinatoire certifiée. In Mireille Blay-Fornarino, Catherine Dubois, and Pierre-Etienne Moreau, editors, *GdR Génie de la Programmation et du Logiciel, Défis 2030*, pages 61–65, 2021. https://hal.archives-ouvertes.fr/hal-03097727.

Les nombres premiers au crible de la preuve formelle

Josué Moreau*

Université Paris-Saclay

Résumé

Dans cet article, nous nous intéressons à la vérification formelle du crible d'Euler, un algorithme permettant d'énumérer tous les nombres premiers inférieurs à une certaine borne. Après avoir décrit cet algorithme ainsi qu'une implémentation efficace de celui-ci en OCaml, nous présentons la preuve de celle-ci, réalisée à l'aide de Why3. Nous examinons également l'efficacité du code prouvé par rapport à celle d'autres cribles, tel que celui d'Ératosthène.

1 Introduction

Le crible d'Euler, tout comme le crible d'Ératosthène, permet d'énumérer l'ensemble des nombres premiers compris entre 2 et une limite donnée N. Le crible d'Euler a une complexité temporelle en $O(N)$ pour énumérer tous les nombres premiers dans l'ensemble $\{2, ..., N\}$ [8] alors que celui d'Ératosthène a une complexité en $O(N \log \log N)$.

Le crible d'Euler est plus difficile à implémenter que le crible d'Ératosthène. C'est d'autant plus vrai si on réalise quelques optimisations pour rendre le crible d'Euler plus efficace en temps et en mémoire. Ces subtilités d'implémentation sont autant d'occasions de faire des erreurs, d'où la nécessité de vérifier formellement l'algorithme.

Dans cet article, on décrit une telle vérification réalisée à l'aide de l'outil Why3 [2]. La preuve a été réalisée uniquement à l'aide de démonstrateurs automatiques. Cependant, une partie de cette preuve a nécessité l'utilisation de différents outils fournis par Why3 permettant d'aider le raisonnement des démonstrateurs automatiques. L'intégralité de cette preuve est disponible en ligne [1].

Cet article est organisé de la façon suivante. La section 2 décrit l'algorithme du crible d'Euler, ainsi que son implémentation dans le langage OCaml. Puis la section 3 décrit les grandes lignes de la preuve qui a été réalisée à l'aide de Why3. Enfin, la section 4 présente le code OCaml automatiquement extrait par Why3 à partir de la preuve qui a été réalisée. En particulier, nous comparons l'efficacité de ce programme extrait avec des implémentations du crible d'Ératosthène et du crible d'Ératosthène segmenté.

2 Le crible d'Euler

Le code de l'implémentation, dans le langage OCaml, de l'algorithme que nous allons prouver se trouve dans la figure 2. L'algorithme du crible d'Euler prend comme unique entrée la limite $N \geq 2$ et va renvoyer tous les nombres premiers compris entre 2 et N. Il se divise en deux parties. La première partie prend un nombre n non marqué tel que $2 \leq n \leq N$ et va marquer tous les multiples np tels que $2 \leq np \leq N$ et $p \geq n$ n'est pas marqué. Elle correspond à la fonction `remove_products` du code OCaml. La deuxième partie de l'algorithme boucle sur un entier n et appelle à chaque tour de boucle la fonction `remove_products` puis affecte n au plus

*Ce travail a été effectué dans le cadre d'un stage de L3 à l'Université Paris-Saclay, du 1er au 30 juin 2020, encadré à distance par Jean-Christophe Filliâtre.

petit entier inférieur à la limite, non marqué et strictement supérieur à n et recommence tant que $n \leq N$.

À la fin de chaque étape de la boucle principale, tous les produits de n sont marqués. En effet, en supposant que pour tout $2 \leq k < n$, les multiples de k sont déjà marqués, l'appel de `remove_products` marque tous les produits np tels que $p \geq n$ n'est pas marqué. Si un $p \geq n$ était déjà marqué, alors, par définition du marquage, il existe un i non marqué et j tel que $2 \leq i < n$, $2 \leq j < p$ et $ij = p$. Or, comme tous les multiples de i étaient déjà marqués avant l'appel de `remove_products`, $ijn = pn$ a déjà été marqué, au plus tard lors de l'appel de `remove_products` sur l'entier i. Ainsi, l'algorithme ne marquant que les produits de nombres non marqués, il ne marque qu'une seule fois chaque entier non premier, comme multiple de son plus petit facteur premier. C'est là la clé de la complexité linéaire du crible d'Euler.

L'implémentation de l'algorithme que nous allons prouver utilise une liste chaînée croissante pour représenter les nombres de l'ensemble $\{2, ..., N\}$. Cette liste est simplement chaînée et est implémentée dans un tableau des suivants `arr`. Une liste chaînée est importante pour l'efficacité car elle permet d'obtenir le prochain entier à parcourir en temps constant.

Voici un schéma décrivant l'état de la liste chaînée dans le tableau `arr` à un moment donné pendant une exécution du crible d'Euler. Dans celle-ci l'algorithme vient de marquer (en gris) tous les multiples de 2 et de 3. Il va commencer à marquer les multiples de 5, en marquant les produits de 5 avec 5, 7, 11, 13, ...

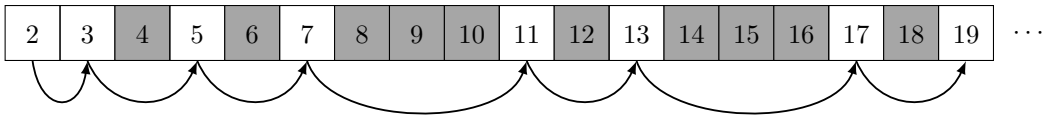


FIGURE 1 – Illustration de la liste chaînée.

Lors du marquage d'un entier i, on ne le supprime pas de la liste chaînée car cela coûterait trop cher en temps, ou nécessiterait une liste doublement chaînée qui coûterait trop cher en espace. Au lieu de cela, on se contente de changer le signe de `arr[`i`]` pour signifier le marquage de i. Lorsque la fonction `remove_products` parcourt la liste pour barrer tous les multiples de n, elle en profite pour éliminer les entiers qui sont marqués. Pour cela, elle maintient l'élément précédent dans la liste dans une variable `p`. Il est donc important de remarquer qu'un entier i, au moment où la boucle de `remove_products` arrive sur lui, peut très bien être déjà marqué. Si c'est le cas, alors il aura nécessairement été marqué par une précédente itération de la boucle. Lorsque la boucle de `remove_products` arrive sur l'entier $i =$ `arr[p]`, elle marque ni. Puis, si i était marqué, elle modifie `arr[p]` en lui donnant pour valeur `arr[`i`]` puis elle recommence en regardant l'entier `arr[p]`. Si i n'était pas marqué, elle recommence en regardant l'entier `arr[`i`]`. Cette opération permet donc à la liste chaînée de "sauter" par dessus les entier déjà marqués, et donc d'examiner au plus une fois chaque entier marqué.

Avec ce procédé, il y a un nombre important d'entiers qui sont marqués et qui ne seront pas réexaminés par la fonction `remove_products` et qui, par conséquent, seront encore dans la liste chaînée à la fin de l'algorithme. Il s'agit des entiers qui ont été marqués comme étant multiples d'un certain n, mais dont le produit avec tous les entiers suivant n dans la liste chaînée est supérieur à la limite du crible. Ces entiers seront supprimés lors de l'extraction des nombres premiers de la liste chaînée.

```
(* on représente une liste chaînée par un tableau, chaque indice du tableau *)
(* contient l'indice de la case suivante dans la liste chaînée. *)
type t = { arr: int array;  max: int;  max_arr: int; }

(* marque tous les multiples de n encore non marqués et saute par dessus les *)
(* nombres déjà marqués qu'il croise *)
let remove_products (t: t) (n: int) : unit =
  let d = get_max t / n in
  let rec loop (p: int) : unit =
    let next = get_next t p in
    if 0 <= next && next <= get_max t then
      if next <= d then begin
        set_mark t (n * next);
        if get_mark t next then begin
          (* si next était déjà marqué, alors on le retire de la liste chaînée *)
          (* en sautant par dessus *)
          set_next t p (get_next t next);
          loop p
        end else loop next (* sinon on passe à l'entier suivant *)
      end in
  set_mark t (n * n); (* tous les n * i pour i < n sont déjà marqués *)
  loop n

let euler_sieve (max: int) : int array =
  (* initialement, la liste chaînée contient tous les nombres impairs et pour *)
  (* marquer la fin de la liste chaînée, son dernier entier a pour valeur max + 1 *)
  let t = create max in
  let rec loop (n: int) : unit =
    (* n prend successivement les valeurs des nombres premiers *)
    remove_products t n;
    let nn = get_next t n in
    if nn <= max / nn then loop1 nn in
  if max >= 9 then loop 3;
  ... extraction des nombres premiers ...
```

FIGURE 2 – Code du crible d'Euler en OCaml.

Afin de diviser l'espace utilisé par deux, le tableau `arr` ne représente que les nombres impairs. Ainsi, la case d'index i du tableau représente l'entier $2i+1$. Les fonctions `get_next` et `set_next` utilisées dans le code précédent sont donc les suivantes :

```
let set_next (t: t) (i: int) (v: int) : unit = t.arr.(i / 2) <- v
let get_next (t: t) (i: int) : int =
  if t.arr.(i / 2) < 0 then - t.arr.(i / 2)
  else t.arr.(i / 2)
```

De plus, pour chaque indice i, `arr[`i`]` contient un entier positif si et seulement si l'entier i est non marqué. Dans ce cas, `arr[`i`]` est l'entier suivant i dans la liste chaînée. L'opération de marquage d'un entier i consiste à remplacer le contenu de la case i par son opposé. On a donc dans le code les fonctions `get_mark` et `set_mark` suivantes :

```
let set_mark (t: t) (i: int) : unit =
  if t.arr.(i / 2) >= 0 then t.arr.(i / 2) <- - t.arr.(i / 2)
let get_mark (t: t) (i: int) : bool = t.arr.(i / 2) < 0
```

Enfin, la fonction `create` ci-dessous crée l'état initial de la structure de donnée maintenue par l'algorithme. La liste chaînée décrite précédemment contient uniquement les entiers impairs. Il y a donc dans chaque case i du tableau `arr` l'entier $2i + 3$, à l'exception de la dernière case du tableau, qui représente le dernier entier de la liste chaînée, qui contient l'entier `max` $+ 1$.

```
let create (max: int) : t =
  let len_arr = (max - 1) / 2 + 1 in
  let arr = Array.make len_arr (-2) in
  for i = 1 to len_arr - 1 do
    arr.(i) <- if i = len_arr - 1 then max + 1 else 2 * i + 3
  done;
  { arr = arr; max = max; max_arr = (max - 1) / 2 }
```

Voici une autre figure décrivant l'implémentation du tableau d'entiers, représentant uniquement les entiers impairs (notés entre parenthèses, en dessous de leurs indices respectifs dans le tableau) et le marquage avec les entiers négatifs. Les flèches sont dessinées ici uniquement pour montrer les liaisons réalisées par ce tableau. Les multiples de 3 ont déjà été marqués et ceux qui se trouvent dans la portion du tableau représentée ont été éliminés de la liste chaînée. Les multiples de 5 sont en cours de marquage. On peut en effet constater que 25 a été marqué mais pas encore éliminé de la liste chaînée. Il le sera lorsque l'algorithme marquera 5×25.

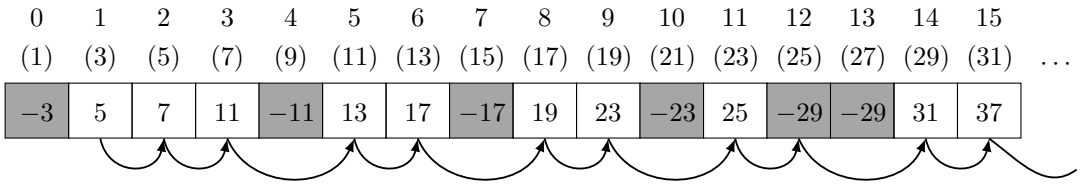


FIGURE 3 – Illustration de l'implémentation de la liste chaînée.

Enfin, à la toute fin de l'algorithme, on souhaite renvoyer un tableau ne contenant que les nombres premiers. Pour cela, on pourrait commencer par compter les nombres premiers de la liste chaînée, puis créer un tableau de cette taille et enfin y copier les nombres premiers. Cependant, on peut faire un peu mieux, en écrasant le début du tableau contenant la liste chaînée avec les nombres premiers, puis en extrayant avec `Array.sub` le préfixe contenant tous les nombres premiers. Le code de la figure 4, qui correspond à la fin du code de la figure 2, représente cette extraction des nombres premiers.

Dans l'implémentation OCaml que nous allons prouver, on manipule à plusieurs reprises un type `t`. Ce type contient comme premier champ le tableau `arr` décrit précédemment. Le champ `max` correspond à la limite donnée en entrée au crible et le champ `max_arr` correspond au plus grand indice du tableau `arr`, c'est-à-dire `max_arr` $\stackrel{\text{def}}{=} \frac{\texttt{max}-1}{2}$. La fonction `remove_products` du code OCaml correspond à la fonction du même nom décrite précédemment. La fonction `euler_sieve` est la fonction principale du crible. Elle appelle `remove_products` dans sa boucle principale et extrait ensuite les nombres premiers de la liste chaînée pour les renvoyer dans un tableau.

```
let cnt = ref 1 in
let p = ref 1 in t.arr.(0) <- 2;
while 2 * !p + 1 <= max do
  let next = t.arr.(!p) / 2 in
  (* on élimine les derniers multiples et on écrase le début de la liste chaînée *)
  if next <= t.max_arr then
    if t.arr.(next) < 0 then t.arr.(!p) <- - t.arr.(next)
    else begin
      t.arr.(!cnt) <- 2 * !p + 1;
      cnt := !cnt + 1;
      p := next end
  else begin
    t.arr.(!cnt) <- 2 * !p + 1;
    cnt := !cnt + 1;
    p := t.max_arr + 1 end
done;
Array.sub t.arr 0 !cnt (* extrait les nombres premiers écrits au début de t.arr *)
```

FIGURE 4 – Code OCaml correspondant à l'extraction des nombres premiers.

3 Une vérification formelle avec Why3

La preuve qui est décrite ici a été intégralement réalisée avec l'outil Why3 [2] et les démonstrateurs automatiques Alt-Ergo, CVC4, Z3, E et Vampire.

Spécification. La spécification de la fonction principale du crible, `euler_sieve`, est la suivante :

```
let euler_sieve (max: int63) : array63
  requires { max_int > max ≥ 3 }
  ensures  { ∀ i j. 0 ≤ i < j < result.length → result[i] < result[j] }
  ensures  { ∀ i. 0 ≤ i < result.length → 2 ≤ result[i] ≤ max }
  ensures  { ∀ i. 0 ≤ i < result.length → prime result[i] }
  ensures  { ∀ i. 2 ≤ i ≤ max → prime i →
                ∃ j. 0 ≤ j < result.length ∧ result[j] = i }
= ...
```

Le type `int63` est une modélisation Why3 des entiers OCaml qui sont des entiers 63 bits signés. De même, le type `array63` est une modélisation Why3 d'un tableau OCaml contenant des entiers OCaml. La précondition requiert que `max` ne soit pas égal au plus grand entier. En effet, le code a besoin de stocker la valeur `max+1` dans la dernière case du tableau pour signaler la fin.

La première postcondition énonce que les nombres du tableau renvoyé en résultat sont écrits dans l'ordre strictement croissant. La postcondition suivante énonce que les nombres renvoyés dans le tableau sont tous dans l'ensemble $\{2, ..., \texttt{max}\}$. Enfin, les deux dernières postconditions énoncent que tous les éléments du tableau renvoyé sont premiers et que tous les nombres premiers compris entre 2 et `max` sont dans le tableau. Le prédicat `prime` utilisé dans ces deux dernières postconditions provient de la bibliothèque `number.Prime` de Why3. Il a déjà été utilisé dans les preuves d'autres programmes manipulant des nombres premiers, comme le crible d'Ératosthène [4] ou celui de l'algorithme des nombres premiers de Knuth [9, 7].

Vérification. La preuve du crible d'Euler a été faite par raffinement [6], en deux temps. La première partie de la preuve a été faite, dans un module `EulerSieve`, à l'aide d'un type `t` représentant une structure abstraite. Ce type modélise la liste chaînée et le marquage, sans en connaître l'implémentation, de la manière suivante :

```
type t = private {
  mutable ghost nexts: seq int;
  mutable ghost marked: seq bool;
  max: int63;
}
```

Le champ `nexts` modélise la liste chaînée implémentée par le tableau `arr`. À la différence de ce dernier, cependant, tous les nombres y sont représentés et le marquage en est absent. Ce dernier est modélisé par le champ `marked`. Comme pour `nexts`, tous les nombres y sont représentés. Le type `seq` utilisé pour décrire la liste chaînée et le marquage est ici un type modélisant des séquences. Il s'agit de tableaux immuables fournis par la bibliothèque standard de Why3. Pour pouvoir les manipuler, il est donc nécessaire que les champs correspondants soient mutables. De plus, comme on peut le voir ci-dessus, les champs modélisant la liste chaînée et le marquage sont des champs fantômes [5]. Les champs fantômes n'existent que dans la preuve et pas dans le programme. Ils seront automatiquement supprimés par Why3 lors de l'extraction du programme, une fois démontré, vers du code OCaml. Ces champs fantômes sont ici très utiles puisqu'ils permettent de représenter des structures de données simples, ce qui a pour conséquence de faciliter la manipulation des structures de données dans la logique et, par conséquent, de faciliter la preuve.

Des fonctions permettant de modifier ces champs ont été déclarées. Elles sont définies de manière abstraites dans le module `EulerSieve`. Voici un exemple, extrait de la preuve, d'une telle fonction abstraite :

```
val set_mark (t: t) (i: int63) : unit
  requires { 0 ≤ i ≤ t.max }
  requires { mod i 2 = 1 }
  writes   { t.marked }
  ensures  { t.marked = (old t.marked)[i ← true] }
```

La deuxième partie de la preuve a consisté en le raffinement du type `t` précédemment défini. Ce raffinement se trouve dans le module `EulerSieveImpl` qui est une implémentation du module `EulerSieve`. La nouvelle définition du type `t` est la suivante :

```
type t = {
  mutable ghost nexts: seq int;
  mutable ghost marked: seq bool;
  arr: array63;
  max: int63;
  max_arr: int63
}
```

On observe l'ajout des champs `arr` et `max_arr` correspondant aux champs de mêmes noms définis dans la section 2. Il s'agit donc de l'implémentation de la liste chaînée et du marquage. Le champ `arr` est lié aux champs fantômes `nexts` et `marked` par des invariants de liaison dont voici un extrait :

```
invariant { ∀ i. 0 ≤ i ≤ max_arr →
                 Seq.get marked (2 * i + 1) ↔ arr[i] < 0 }
```

Les fonctions permettant de manipuler la liste chaînée et le marquage ont été ensuite implémentées dans le module `EulerSieveImpl`. Il a été nécessaire, après implémentation, de montrer que celles-ci satisfont les spécifications de leurs fonctions abstraites respectives et conservent les invariants du type `t`, tel que l'invariant de liaison évoqué ci-dessus.

Ainsi, la preuve du crible d'Euler a été faite uniquement dans la structure `t` abstraite, définie précédemment. Puis cette structure `t` a été réalisée avec les optimisations décrites en section 2 et nous avons montré que cette structure concrète respectait sa spécification et celle de la structure abstraite `t`.

La preuve a nécessité l'écriture de 775 lignes dans le langage WhyML : 517 lignes pour la spécification et 258 lignes de code. Au total, 840 buts ont été prouvés. Afin de démontrer le crible d'Euler, de nombreuses interactions ont été nécessaires pour aider les démonstrateurs automatiques. Parmi ces interactions, de nombreux lemmes ont été énoncés et prouvés. Des assertions ont également été écrites dans le programme. Enfin, des transformations logiques ont été appliquées à des buts à de nombreuses reprises. La répartition des buts démontrés par les différents démonstrateurs automatiques est la suivante :

démonstrateur	nombre de buts prouvés	temps maximum sur un but
Eprover 2.4	7	0,74 s
Vampire 4.4.0	6	8,22 s
Alt-Ergo 2.3.2	517	7,85 s
Alt-Ergo 2.0.0	89	8,28 s
Z3 4.8.6	137	1,84 s
CVC4 1.7	39	0,34 s
CVC4 1.6	45	4,05 s

4 Le code OCaml extrait de la preuve

Une fois la démonstration terminée, il est maintenant possible d'extraire le code OCaml du programme démontré. Ceci est fait par la commande `extract` de Why3. Le code OCaml obtenu après extraction est présenté à la figure 5. Seules quelques modifications de mise en page ont été effectuées entre le code extrait et le code présenté. Voici quelques mesures du temps de calcul et de la mémoire utilisée par le code extrait :

N	temps de calcul	mémoire utilisée
10^6	0,022 s	8 Mo
10^7	0,140 s	73 Mo
10^8	1,350 s	721 Mo
10^9	14,500 s	7 200 Mo

Afin de pouvoir évaluer l'efficacité de ce code, nous comparons maintenant les temps de calcul du code extrait de notre crible d'Euler avec des implémentations du crible d'Ératosthène et du crible d'Ératosthène segmenté [1]. Concernant ce dernier, la comparaison est fournie pour permettre de donner une idée de la différence d'efficacité, mais elle n'est pas totalement équitable. En effet, contrairement aux cribles d'Ératosthène et d'Euler, qui renvoie un tableau contenant tous les nombres premiers, le crible d'Ératosthène segmenté ne renvoie rien mais applique une fonction `int -> unit` donnée en argument sur tous les nombres premiers compris entre 2 et la limite.

1. Merci à Jean-Christophe Filliâtre pour son implémentation du crible d'Ératosthène segmenté.

```
type t = { arr: int array; max: int; max_arr: int }

let create max =
  let len_arr = (max - 1) / 2 + 1 in
  let arr = Array.make len_arr (-2) in
  for i = 1 to len_arr - 1 do
    arr.(i) <- if i = len_arr - 1 then max + 1 else 2 * i + 3
  done;
  { arr = arr; max = max; max_arr = (max - 1) / 2 }

let set_next t i v = t.arr.(i / 2) <- v
let get_next t i = if t.arr.(i / 2) < 0 then - t.arr.(i / 2) else t.arr.(i / 2)
let set_mark t i = if t.arr.(i / 2) >= 0 then t.arr.(i / 2) <- - t.arr.(i / 2)
let get_mark t i = t.arr.(i / 2) < 0
let get_max t = t.max

let remove_products t n =
  let d = get_max t / n in
  let rec loop (p: int) : unit =
    let next = get_next t p in
    if 0 <= next && next <= get_max t then begin
      if next <= d then begin
        set_mark t (n * next);
        if get_mark t next then begin
          set_next t p (get_next t next); loop p
        end else loop next
      end end in
  set_mark t (n * n); loop n

let euler_sieve max =
  let t = create max in
  let rec loop n =
    remove_products t n;
    let nn = get_next t n in
    if nn <= max / nn then loop nn in
  if max >= 9 then loop 3;
  let cnt = ref 1 in
  let p = ref 1 in t.arr.(0) <- 2;
  while 2 * !p + 1 <= max do
    let next = t.arr.(!p) / 2 in
    if next <= t.max_arr then
      if t.arr.(next) < 0 then t.arr.(!p) <- - t.arr.(next)
      else begin t.arr.(!cnt) <- 2 * !p + 1; cnt := !cnt + 1; p := next end
    else begin t.arr.(!cnt) <- 2 * !p + 1; cnt := !cnt + 1; p := t.max_arr + 1 end
  done;
  Array.sub t.arr 0 !cnt
```

FIGURE 5 – Le code extrait en OCaml du crible d'Euler.

N	crible d'Ératosthène	crible d'Euler	crible d'Ératosthène segmenté
10^6	0,033 s	0,022 s	0,013 s
10^7	0,186 s	0,140 s	0,039 s
10^8	1,970 s	1,350 s	0,312 s
10^9	21,780 s	14,500 s	3,390 s

On peut observer ici que le code du crible d'Euler est plus rapide que l'implémentation du crible d'Ératosthène que nous utilisons. Cependant, les performances sont encore éloignées de celles du crible d'Ératosthène segmenté, qui tire parti de la rapidité offerte par la mémoire cache du processeur.

5 Conclusion

Dans cet article, nous avons présenté une implémentation du crible d'Euler. Nous en avons montré la spécification et nous avons prouvé que l'implémentation fournie respectait cette spécification. La preuve a été réalisée par raffinement, à l'aide de différents outils fournis par Why3. Le code extrait de la preuve par Why3 possède de bonnes performances, meilleures que celles fournies par un crible d'Ératosthène. Cependant, elles sont encore loin des performances du crible d'Ératosthène segmenté, bien que celui-ci ne réponde pas exactement au même problème. Il serait ainsi intéressant, dans le même but que pour le crible d'Euler, de vérifier formellement une implémentation du crible d'Ératosthène segmenté. Une autre perspective intéressante serait de prouver formellement que l'implémentation du crible d'Euler que nous avons présentée est bien de complexité linéaire, par exemple à l'aide de crédits temps [3].

Remerciements. Je remercie chaleureusement Jean-Christophe Filliâtre pour le stage effectué sous sa direction, ainsi que pour le temps qu'il m'a consacré. Merci aussi pour toutes les choses passionnantes qu'il m'a fait découvrir dans les domaines des méthodes formelles, de la compilation et de la programmation en particulier, ainsi que pour ses nombreux conseils et encouragements durant la rédaction de cet article et pour sa relecture. Je remercie également tous ceux qui ont relu cet article pour leurs corrections et conseils.

Références

[1] Preuve en Why3 du crible d'Euler. https://github.com/aistun/EulerSieve/tree/master/.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. The Why3 platform. http://why3.lri.fr/.

[3] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, 62(3) :331–365, March 2019.

[4] Martin Clochard. Sieve of Eratosthenes. http://toccata.lri.fr/gallery/sieve.en.html.

[5] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3) :152–174, 2016.

[6] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in Why3. In Tiziana Margaria and Bernhard Steffen, editors, *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, Lecture Notes in Computer Science, Rhodes, Greece, October 2020. http://why3.lri.fr/isola-2020/.

[7] Jean-Christophe Filliâtre. Knuth's prime numbers. http://toccata.lri.fr/gallery/knuth_prime_numbers.en.html.

[8] David Gries and Jayadev Misra. A linear sieve algorithm for finding prime numbers. *Commun. ACM*, 21(12) :999–1003, December 1978.

[9] Donald E. Knuth. *The Art of Computer Programming*, volume 1, page 147. Addison Wesley Professional, 1997.

Flottants primitifs dans Coq

Érik Martin-Dorel[1], Pierre Roux[2]

[1] Lab. IRIT, Université de Toulouse, CNRS, Université Paul Sabatier, Toulouse, France
[2] ONERA / DTIS, Université de Toulouse, Toulouse, France

Résumé

Certaines preuves mathématiques font intervenir d'important calculs, par exemple le théorème des quatre couleurs, le théorème de Hales sur l'empilement optimal des sphères (ex-conjecture de Kepler) ou l'arithmétique d'intervalles. Pour les calculs numériques, l'arithmétique à virgule flottante est très appréciée pour son efficacité malgré l'introduction d'erreurs d'arrondi. Des garanties formelles peuvent toutefois être obtenues en se basant sur le standard IEEE 754, qui spécifie précisément l'arithmétique à virgule flottante, et un assistant de preuve tel que Coq, qui offre des fonctionnalités de calcul efficace.

Coq propose de longue date une interface aux entiers machine mais jusqu'à récemment, l'arithmétique flottante était au mieux émulée grâce à ces entiers plutôt que d'utiliser les flottants machine, au prix d'un ralentissement de deux à trois ordres de grandeur. Cette proposition de démonstration vise à présenter l'interface vers les flottants machines disponibles dans Coq depuis sa version 8.11. Après une rapide présentation des choix techniques effectués, on présentera l'interface retenue, et en particulier les questions de saisie et d'affichage de constantes. Enfin sera évoquée l'utilisation de cette fonctionnalité dans les bibliothèques ValidSDP et Coq.Interval.

1 Motivation

La preuve de certains résultats mathématiques peut faire appel à des calculs numériques de telle sorte que la confiance en ces preuves nécessite une confiance en les calculs numériques eux-mêmes. Ainsi, pour être capable d'effectuer efficacement ce genre de preuves dans un assistant de preuve comme Coq, l'outil doit avoir des capacités de calcul numérique efficace.

L'arithmétique à virgule flottante est largement utilisée, en particulier pour son efficacité grâce à son implémentation matérielle. Toutefois, elle ne donne généralement pas des résultats exacts puisqu'elle introduit des erreurs d'arrondi. Des preuves rigoureuses peuvent néanmoins être obtenues en bornant ces erreurs d'arrondi. Il y a ainsi un intérêt évident à disposer d'un accès correct et efficace aux opérations flottantes du processeur au sein de Coq.

On donne ci-dessous quelques exemples de preuves mettant en jeu des calculs flottants. Considérons la preuve qu'un nombre réel donné $a \in \mathbb{R}$ est positif. On peut exhiber un autre nombre réel r tel que $a = r^2$ et appliquer le lemme garantissant que tout carré d'un nombre réel est positif. Typiquement, on peut utiliser la racine carré $\sqrt{a}$. Une méthode similaire peut être appliquée pour prouver qu'une matrice $A \in \mathbb{R}^{n\times n}$ est semi-définie positive [1] puisqu'on peut exhiber une matrice R tel que [2] $A = R^T R$. Une telle matrice peut être calculée en utilisant un algorithme appelé la décomposition de Cholesky, présenté sur la Figure 1. L'algorithme réussit, en n'effectuant ni une racine négative ni une division par zéro, dès que A est définie positive [3].

Quand elle est évaluée en arithmétique à virgule flottante, l'égalité exacte $A = R^T R$ est perdue, mais il reste possible de borner l'accumulation des erreurs d'arrondi lors de la décomposition de Cholesky de telle sorte qu'on obtient le Théorème 1.1 suivant, modulo quelques préconditions qui sont typiquement vérifiées en pratique, mais omises ici par souci de concision (ce résultat ayant été formellement prouvé en Coq dans un travail antérieur [Rou16], et disponible dans la bibliothèque libValidSDP [4]).

$R := 0$;
for j **from** 1 **to** n **do**
  for i **from** 1 **to** $j-1$ **do**
    $R_{i,j} := \left(A_{i,j} - \Sigma_{k=1}^{i-1} R_{k,i} R_{k,j}\right) / R_{i,i}$;
  end for
  $R_{j,j} := \sqrt{M_{j,j} - \Sigma_{k=1}^{j-1} {R_{k,j}}^2}$;
end for

Figure 1: Décomposition de Cholesky : pour $A \in \mathbb{R}^{n\times n}$, tente de calculer R tel que $A = R^T R$.

1. Une matrice $A \in \mathbb{R}^{n\times n}$ est dite semi-définie positive si pour tout $x \in \mathbb{R}^n$, $x^T A\, x \geq 0$.
2. Puisque, si $A = R^T R$, on a $x^T A\, x = x^T \left(R^T R\right) x = (Rx)^T (Rx) = \|Rx\|^2 \geq 0$.
3. Une matrice $A \in \mathbb{R}^{n\times n}$ est dite définie positive si pour tout $x \in \mathbb{R}^n \setminus \{0\}$, $x^T A\, x > 0$.
4. https://github.com/validsdp/validsdp/blob/v0.7.0/libvalidsdp/cholesky.v#L646

Theorem 1.1 (Corollaire 2.4 in [Rum06]). *Pour $A \in \mathbb{R}^{n\times n}$, en définissant $c := \frac{(n+1)\epsilon}{1-2(n+1)\epsilon}\,\mathrm{tr}(A) + 4n\,(2(n+1) + \max_i A_{i,i})\,\eta$, si la décomposition de Cholesky flottante réussit sur $A - c\,I$, alors A est définie positive. ϵ et η sont des constantes dépendantes du format flottant utilisé.*

Ainsi, une implémentation efficace de l'arithmétique à virgule flottante dans un assistant de preuve mène à des preuves efficaces de positivité de matrices. Cela peut avoir de multiples applications, comme prouver que des polynômes sont positifs en les exprimant comme des sommes de carrés [MR17], ce qui peut être utilisé dans une preuve de la conjecture de Kepler [MAGW15].

L'arithmétique d'intervalles constitue un autre exemple de preuve mettant en jeu des calculs numériques. Des intervalles enveloppant peuvent être facilement calculés en arithmétique à vigule flottante en utilisant les arrondis dirigés, vers $\pm\infty$. La bibliothèque Coq.Interval [MDM16] implémente l'arithmétique d'intervalles et peut bénéficier de calculs flottants efficaces.

Plus généralement, de nombreux résultats sur les méthodes numériques rigoureuses [Rum10] pourraient avoir des implémentations formelles efficaces si une arithmétique flottante efficace est disponible dans les assistants de preuve.

Coq dispose d'un support intégré pour le calcul, qui peut être utilisé dans les preuves et de récents progrès ont été faits pour fournir des entiers machines 63 bits. Coq dispose maintenant depuis sa version 8.11 d'un interface similaire vers les flottants machine « double-précision ». [5] Pour de plus amples détails sur l'implémentation, l'extension de la base de confiance et une évaluation des performances, on pourra se référer au papier original [BMR19].

2 Démonstration

On effectuera une démonstration de cette nouvelle fonctionnalité en commençant par présenter la lecture et l'affichage de constantes (qui ont été améliorés depuis [BMR19]) et les compromis associés entre lisibilité de l'écriture décimale pour l'utilisateur humain et exactitude de l'écriture binaire [6]. On présentera ensuite les différentes opérations disponibles, tant les opérations de calcul flottant que de conversion avec les entiers, puis on étudiera la spécification de ces opérations, en particulier le lien avec la bibliothèque Flocq [BM11]. Enfin, deux exemples d'utilisation seront brièvement présentés : les bibliothèques Coq.Interval et ValidSDP [7].

Références

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq : A unified library for proving floating-point algorithms in coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pages 243–252. IEEE Computer Society, 2011. doi:10.1109/ARITH.2011.40.

[BMR19] Guillaume Bertholon, Érik Martin-Dorel, and Pierre Roux. Primitive Floats in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 7 :1–7 :20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.7.

[MAGW15] Victor Magron, Xavier Allamigeon, Stéphane Gaubert, and Benjamin Werner. Formal proofs for nonlinear optimization. *Journal of Formalized Reasoning*, 8(1) :1–24, 2015. URL : http://jfr.unibo.it/article/view/4319.

[MDM16] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3) :187–217, October 2016. doi:10.1007/s10817-015-9350-4.

[MR17] Érik Martin-Dorel and Pierre Roux. A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. In Yves Bertot and Viktor Vafeiadis, editors, *CPP 2017, Paris, France, January 16-17, 2017*, pages 90–99. ACM, 2017. doi:10.1145/3018610.3018622.

[Rou16] Pierre Roux. Formal Proofs of Rounding Error Bounds - With Application to an Automatic Positive Definiteness Check. *J. Autom. Reasoning*, 57(2) :135–156, 2016. doi:10.1007/s10817-015-9339-z.

[Rum06] Siegfried M. Rump. Verification of positive definiteness. *BIT Numerical Mathematics*, 46 :433–452, 2006.

[Rum10] Siegfried M. Rump. Verification methods : Rigorous results using floating-point arithmetic. *Acta Numerica*, 19 :287–449, 2010.

5. https://github.com/coq/coq/pull/9867

6. Certains nombres décimaux n'ont pas de représentation binaire finie et si toute valeur binaire à une représentation décimale, elle peut être trop grande pour être utilisable en pratique.

7. Preuves de positivité de matrice suivant le principe exposé ci dessus.

A Fichier Coq de la démonstration

```
(** Démonstration des flottants primitifs

   (nécessite Coq >= 8.11 (démo réalisée avec master) *)

Require Import ZArith Int63.

(** * Un module de la librairie standard *)

(** On importe le module Floats de la librairie standard de Coq *)
Require Import Floats.

(** on obtient un type des flottants primitifs *)
Check float.

(** ** Constantes *)

(** les constantes sont interprétées comme
    le flottant le plus proche *)
Open Scope float_scope.

Check 0.5.

(** avec warning si un arrondi est nécessaire *)
Check 0.1.

(** l'impression se fait avec 17 décimales, ce qui garantit
    que (parse o print) est injectif *)

(** néanmoins, ça peut donner des choses un peu étranges *)
Goal 0.99999999999999995 = 1.
Proof. reflexivity. Qed.

(** on peut voir la valeur exacte en hexadécimal *)
Set Printing All.
Check 0.1.
Unset Printing All.

(** on a trois valeurs spéciales *)
Check nan.
Check infinity.
Check neg_infinity.

(** et le 0 est signé *)
Check 0.
Check -0.

Goal 0 = -0.
Proof. Fail reflexivity. Abort.

(** mais pas les NaNs (nan est unique) *)
Goal nan = -nan.
Proof. reflexivity. Qed.

(** ** Opérations primitives *)

(** *** Opérations arithmétiques *)

(** on dispose des opération arithmétiques standard
    (toutes en arrondi au plus proche (tie to even)) *)
```

```
Eval compute in - 0.5.
Eval compute in abs (- 0.5).
Eval compute in 1 + 0.5.
Eval compute in 1 - 0.5.
Eval compute in 0.5 * 3.
Eval compute in 3 / 2.
Eval compute in 1 / 0.
Eval compute in 1 / (-0).
Eval compute in sqrt 2.
Eval compute in sqrt (-2).

(** *** Comparaison *)

Eval compute in 0.5 ?= 1.
Eval compute in 0.5 ?= 0.5.
Eval compute in infinity ?= 0.5.
Eval compute in 0 ?= (-0).
Eval compute in 1 ?= nan.

(** et les fonctions booléennes *)
Eval compute in 0.5 <=? 1.
Eval compute in 1 <? 1.
Eval compute in 1 =? 1.

(** *** "Destructeurs" *)

(** - classe *)
Eval compute in classify 0.5.
Print float_class.

(** - mantisse, exposant (int63 non signé, donc décalé de 2101) *)
Check frshiftexp.
Definition m_e := Eval compute in frshiftexp 15.
Print m_e.
Eval compute in 0.9375 * 16. (** 16 = 2^4 = 2^(2105 - 2101) *)

(** - mantisse comme un int63 *)
Eval compute in normfr_mantissa 0.9375.

(** - quelques fonctions pratiques (implémentées en Coq) *)
Eval compute in is_nan 0.
Eval compute in is_zero 0.
Eval compute in is_infinity 0.
Eval compute in get_sign (-0).
Eval compute in frexp 15. (** frshiftexp dans Z au lieu de int63,
                              sans décalage *)
Eval compute in ulp 15. (** Unit in Last Place *)

(** *** "Constructeurs" *)

(** - à partir d'un entier *)
Eval compute in of_int63 15.

(** - décalage d'exposant (décalé de 2101) *)
Check ldshiftexp.
Eval compute in ldshiftexp 10 2103. (** 10 * 2^(2103 - 2101) = 10 * 2^2 *)
Eval compute in ldshiftexp (fst m_e) (snd m_e).

(** - ldexp : ldexp sur Z au lieu de int63, sans décalage *)
Eval compute in ldexp 10 2.
```

```
(** *** Prédécesseur et successeur *)

(** pour implémenter l'arithmétique d'intervalle *)
Eval compute in next_up 1.
Eval compute in next_down 1.
Eval compute in next_up neg_infinity.
Eval compute in next_down infinity.

(* et si le temps le permet, remarque sur
   quelques valeurs flottantes remarquables *)
Eval compute in next_up 0.
Eval compute in classify (next_up 0).
Eval compute in next_up 1 - 1.
Eval compute in classify (next_down infinity).

(** tout est implémenté aussi avec vm_compute et native_compute *)
Eval vm_compute in 1 + 0.5.
Eval native_compute in 1 + 0.5.

(** ** Spécification *)

(** 1. On dispose d'un type spec_float *)
Print spec_float.

(** 2. de fonctions float <-> spec_float *)
Check Prim2SF.
Check SF2Prim.

(** 3. d'une implémentation de chaque fonction *)
Print SFopp.

(** 4. et d'un axiome de correction *)
Check opp_spec.
Check add_spec.

(** Cette spécification est extraite de la librairie Flocq.
    C'est un extrait minimal (seulement 400 lignes)
    mais peu commode. En pratique on utilisera Flocq (>= 3.3)
    qui fait le lien avec les réels de la librairie standard. *)
Require Import Reals Flocq.IEEE754.BinarySingleNaN Flocq.IEEE754.PrimFloat.

Check Prim2B.
Check B2R.
Check add_equiv.
Check Bplus_correct.

(** * Exemples d'utilisation *)

(** ** Librairie Coq.Interval: https://gitlab.inria.fr/coqinterval/interval *)

Require Import Coquelicot.Coquelicot Interval.Tactic.

Lemma Rump_Tucker :
  Rabs (RInt (fun x => sin (x + exp x)) 0 8 - 0.3474) <= 0.1.
Proof.
(** flottants primitifs : 2.3 s *)
Time integral with (i_degree 6, i_fuel 1000).
Undo.
(** flottants émulés (avec bigZ) : 36 s *)
Time integral with (i_degree 6, i_fuel 1000, i_prec 53).
Qed.
```

```
(** ** Librairie ValidSDP: https://github.com/validsdp/validsdp *)

(** une "grosse" matrice *)
Require Import matrice.
Check A.

(** de taille 120x120 *)
Eval compute in length A.

(** on prouve qu'elle est semi-définie positive *)

Require Import ValidSDP.posdef_check.

(** d'abord avec des flottants émulés (bigZ) *)
Lemma with_bigint : posdef_seqF A.
Time posdef_check.
Qed.

Print Assumptions with_bigint.

(* Int63.mul : Int63.int -> Int63.int -> Int63.int *)
(* Int63.mul_spec : ∀x y : Int63.int, Int63.to_Z (Int63.mul x y)
   = BinInt.Z.modulo (BinInt.Z.mul (Int63.to_Z x) (Int63.to_Z y)) Int63.wB *)

(** puis avec des flottants primitifs *)
Lemma with_prim_float : posdef_seqF A.
Time primitive_posdef_check.
Qed.

Print Assumptions with_prim_float.

(* mul_spec : ∀x y : float, Prim2SF (x * y)
   = SF64mul (Prim2SF x) (Prim2SF y) *)
```

Auteurs

Birkedal, Lars 157
Bourke, Timothy 117
Chailloux, Emmanuel 72
Charpiat, Guillaume 134
Chihani, Zakaria 134
Clément, Basile 48
Denis, Xavier 174
Devriese, Dominique 157
Frédéric, Bour 48
Georges, Aïna Linn 157
Giorgetti, Alain 202
Girard-Satabin, Julien 134
Guéneau, Armaël 174
Jeanmaire, Paul 117
Journault, Matthieu 45
Khayam, Adam 95
Ladeveze, Quentin 78
Laurent, Théo 190
Letan, Thomas 75
Maillard, Kenji 190
Martin, Érik 220
Martinez, Thierry 69
Merigoux, Denis 155
Miné, Antoine 45
Monat, Raphaël 45, 155
Moreau, Josué 210
Noizet, Louis 95
Ouadjaout, Abdelraouf 45
Pereira, Mário 21
Pesin, Basile 117
Pottier, François 3
Pouzet, Marc 117
Ravara, António 21
Roux, Pierre 220
Scherer, Gabriel 48
Schmitt, Alan 95
Schoenauer, Marc 134
Sérot, Jocelyn 72, 146
Timany, Amin 157
Trieu, Alix 157
Van Strydonck, Thomas 157
Varasse, Aymeric 134
Xia, Li-Yao 75