ORSAY
N° d'ordre : 1448

UNIVERSITÉ DE PARIS-SUD 11

CENTRE D'ORSAY

# THÈSE

présentée
pour obtenir

L'HABILITATION À DIRIGER DES RECHERCHES
DE L'UNIVERSITÉ PARIS-SUD 11

PAR

Jean-Christophe Filliâtre

—×—

SUJET :

## Deductive Program Verification

soutenue le 2 décembre 2011 devant la commission d'examen

| | | |
|---|---|---|
| MM. | Gilles Barthe | rapporteurs |
| | Yves Bertot | |
| | K. Rustan M. Leino | |
| Mme | Christine Paulin | examinateurs |
| MM. | Olivier Danvy | |
| | Roberto di Cosmo | |
| | Gérard Huet | |
| | Xavier Leroy | |

# Deductive Program Verification

Jean-Christophe Filliâtre

à Anne, Marion,
Émilie et Juliette

# Contents

**Contents** **vii**

**Preface** **ix**

**1 Deductive Program Verification** **1**
1.1 Program Verification is Challenging . . . 3
1.2 Specifications . . . 6
1.3 Methodologies . . . 8
1.4 Proofs . . . 10

**2 A Tool for Program Verification: Why3** **13**
2.1 History . . . 13
2.2 Overview of Why3 . . . 16
2.3 Logic . . . 17
2.4 Proofs . . . 22
2.5 Programs . . . 23

**3 Case Studies in Program Verification** **37**
3.1 Termination . . . 38
3.2 Mathematics . . . 42
3.3 Data Structures . . . 49
3.4 Using Why3 as an Intermediate Language . . . 52

**4 Perspectives** **59**
4.1 Deductive Program Verification . . . 59
4.2 Why3 . . . 60

**Bibliography** **67**

**Index** **77**

# Preface

This memoir purposely focuses on my main research activity, that is deductive program verification. Even more specifically, it presents research I'm doing these days and that I'm considering for the forthcoming years. For completeness, I'm briefly listing here the works I've made that are ignored in this document.

The Coq Proof Assistant. I did my Ph.D. in the Coq team, where the Coq proof assistant had been developing for more than 10 years already at that time. The outcome of my Ph.D. was a tactic to perform verification of imperative programs in Coq [36, 37]. I used it to verify several non-trivial programs [45, 39]. Though I then gradually moved to a mere Coq user, I gained from that period a deep knowledge of that tool — not mentioning my contribution to its programming architecture. For instance, I could use Coq in 2004 to verify implementations of finite sets as AVLs and red-black trees, in a joint work with Pierre Letouzey [44]. Today, I use Coq on a daily basis to handle verification conditions that are not discharged by automated theorem provers.

Verification of C Programs. It was a wonderful experience to develop the Caduceus tool for the verification of C programs with Claude Marché in 2004 [46]. We learned a lot from that experience. In particular, the specification language we designed for Caduceus served as a basis for the design of the more ambitious specification language for Frama-C [50], namely ACSL [8]. Recently, I supervised the Ph.D. of François Bobot whose purpose was to introduce concepts of separation logic within the framework of traditional deductive verification of pointer programs [13].

Verification of Floating-Point Programs. When Sylvie Boldo joined the ProVal team a few years ago, we naturally considered extending our expertise in deductive verification to floating-point programs. She is indeed an expert in the domain. Though I'm rather ignorant with respect to floating-point arithmetic, I could provide the other ingredient to get a nice framework for deductive verification of floating-point programs [19]. Later, another specialist of floating-point arithmetic, Guil-

laume Melquiond, joined ProVal and we could go another step forward [20, 18] using his shiny tool Gappa.

Functional Programming. Those who know me can attest that I'm an enthusiastic, even fanatic, OCaml programmer. I have been lucky to be introduced to this language by Guy Cousineau, Xavier Leroy, and Damien Doligez when I was doing my undergraduate studies at École Normale Supérieure. It was difficult not to fell in love with the creation of such a dream team. Since then, I have experienced so many joyful moments of programming with OCaml that I can't remember all of them.

It explains why, since my Ph.D., I've been continuously publishing papers related to functional programming. Most of them are describing OCaml libraries [25, 26, 28, 3, 43] or neat functional programs [48, 38, 41]. A more profound paper describes a joint work with Sylvain Conchon and introduces the notion of *semi-persistent data structures* [27]. It is somewhat unfortunate that it is not covered in this memoir, as it is probably the piece of work I'm most proud of.

Obviously, OCaml had a great influence on my work related to program verification. In designing the language of Why, I mostly reused syntax, typing, and semantics of OCaml as much as possible. It appeared to be a successful inspiration. Concepts such as polymorphism, algebraic data types, pattern matching, records with mutable fields, etc., are all of genuine interest in program verification, as we show later in this document.

Finally, two of the three Ph.D. theses I supervised are related to functional programming, namely those of Julien Signoles [93] and Johannes Kanig [56, 57].

## Acknowledgments

I'm greatly honored to defend this thesis in front of such a committee, for it is exclusively composed of people I admire. If only Natarajan Shankar could have joined it, it would have been the perfect committee.

Gilles Barthe, Yves Bertot, and Rustan Leino accepted to serve as examiners, despite the extra workload it represents. I sincerely thank them. The contribution of Rustan to the domain of deductive verification is truly impressive and has influenced my work more than once. I remember when I read his PASTE 2005 paper for the first time and how many new insights on weakest preconditions I got from these eight pages. Program verification has always been central in Yves's work, as demonstrated in the *Coq'Art*, the book he co-authored with Pierre Castéran. This book was immensely useful for me when I taught program verification with Coq at the MPRI a few years ago. In 2010, Gilles kindly accepted to be our invited speaker at PLPV, a workshop I was co-organizing. It was an opportunity for me

to appreciate his impressive contribution to the domain of cryptographic software verification.

Roberto di Cosmo was my adviser at École Normale Supérieure when I was doing my undergraduate studies. I remember when I met him for the first time: he was enthusiastic about Perl at that time and even tried to convince me! More importantly, he gave me the advice I needed regarding my first internship, which led me to the Coq team and to my first meeting with Gérard Huet. It was in 1993 at INRIA Rocquencourt. What a souvenir! My first immersion into a research lab was great: computers piled in the so-called "buffers", researchers giving a tour of their work on the blackboard, hardware and software hacking all over the place, etc. I did not stay there for long, however, since I soon left to Lyon to perform my internship under the supervision of Christine Paulin. The lab was not as attractive, with its hospital-like look, but the team was great. At the monthly meetings of the Coq group, I was repeatedly impressed by stratospheric discussions related to type theory always involving Christine and Gérard. It was at one of these meetings that I picked up the PhD proposal on imperative program verification suggested by Gérard. I clearly remember this event and the way I was literally attracted by the subject. May be I was missing the very first days of my programmer's life, when I was doing imperative programming only. I am immensely grateful to Roberto, Christine, and Gérard for having guided my first steps into research.

I am a programmer, since the age of thirteen. As I explained above, I was lucky to be introduced to functional programming at some point, which changed my life as a programmer, but also influenced my research activity. That's why it makes a lot of sense for me to count two eminent members of the ML community on my committee. Xavier Leroy's contribution to the domain is beyond words, but I must confess that I'm even more impressed by his recent move to the domain of program verification with the CompCert project. I wish my own contribution to program verification will eventually be able to contribute to a similar project. Surprisingly, I met Olivier Danvy for the first time only recently. He visited me in Orsay two years ago and, almost immediately, challenged me with a programming exercise! If only all computer scientists could be in that state of mind, it would change our discipline for the better. At last, I can contribute to *Nature*'s statistical result: thanks Olivier.

## Remerciements

« Tout vient à point à qui sait attendre » dit le proverbe. Je ne sais pas si cette habilitation vient à point mais il est clair que j'ai su attendre. Je remercie très chaleureusement toutes les personnes qui m'ont encouragé dans cette voie ces dernières années, sans jamais désespérer. Cela inclut les membres de l'équipe ProVal, passés et présents, et en particulier Christine, qui a su me remotiver plusieurs fois. J'ai également reçu le chaleureux soutien de nombreux enseignants de l'École Poly-

technique, notamment Olivier Bournez, Benjamin Werner, Stéphane Lengrand et Thomas Clausen.

Je remercie le délégué aux thèses de l'école doctorale d'informatique de Paris-Sud, Dominique Gouyou-Beauchamps, pour ses conseils relatifs à l'écriture de ce mémoire, en particulier l'idée d'écrire un document que l'on puisse donner à lire à un étudiant de M2 qui souhaite comprendre notre domaine.

Un grand merci à Xavier Urbain pour son aide précieuse dans la préparation du pot ; je n'ai toutefois aucun espoir de parvenir à faire aussi bien que lui.

Enfin, je dédie ce mémoire à Anne, Marion, Émilie et Juliette qui ont supporté sans jamais se plaindre le surcroît de travail que cette habilitation a pu me donner certains week-ends, sans parler de quelques râleries supplémentaires.

# 1
# Deductive Program Verification

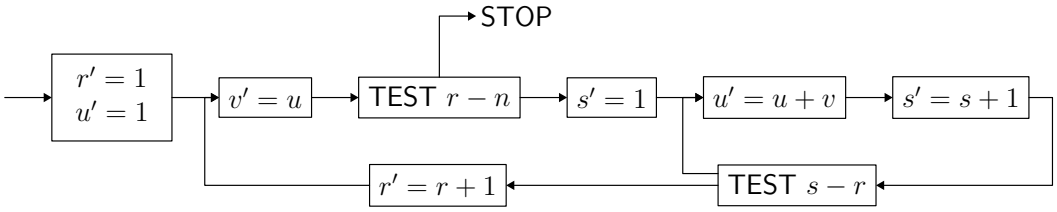
I will start with a definition:

> Deductive program verification is the art of turning the correctness of a program into a mathematical statement and then proving it.

The words are carefully chosen. The word "deductive" makes a clear distinction with other verification methods such as model checking, abstract interpretation, typing, or testing. It also emphasizes the use of a logic, to specify and to prove. I favor "program" to "software" since I am mostly interested in the verification of small and challenging programs, rather than million line long software systems. Providing methods and tools to verify textbook programs as easily as possible is one of my goals. Yet I intend to verify code that is executable and thus I favor "program" to "algorithm". Deductive program verification is an "art", exactly as programming is. It requires experience, imagination, tricks, but also taste, to figure out specifications and proof steps in the most elegant way. There is a long way to go before it becomes a technique routinely used in software production. The word "correctness" is purposely vague. It may refer to the safe execution of the program only, but it may also characterize the precise meaning of what the program is doing. We will show that deductive verification can be a challenging task whatever the notion of correctness might be. Having the correctness expressed as a "mathematical statement" has the obvious advantage that a wide range of existing tools and techniques can be used. This includes proof assistants and automated theorem provers, which were not necessarily developed with program verification in mind. Finally, it is important to embed the task of "proving" the correctness statement into the definition. This is indeed a part of the game, and a successful verification should include a proof as elegant as possible.

This chapter intends to be a survey on deductive program verification, along the lines of the definition above. My own contribution to this domain is described in the next two chapters.

The idea of deductive program verification is not new. It is actually as old as programming. One of the very first proof of program was performed by Alan Turing in 1949. His three page paper *Checking a Large Routine* [101, 79] contains a rigorous proof a of program computing the factorial by repeated additions. The program is given as the following flowchart:




Assigned variables are primed: a statement such as $r' = r + 1$ must be read as $r \leftarrow r + 1$. Variable $n$ contains the input parameter and is not modified. When the program exits, variable $u$ contains $n!$, the factorial of $n$. There are two nested loops. In the outer loop, variable $u$ contains $r!$ and its contents is copied into variable $v$. Then the inner loop computes $(r + 1)!$ in $u$, using repeated additions of $v$. Since variables $r$ and $s$ are only used for iteration, this program can profitably be written using "for" loops, in a more modern way. With a benign modification of its control flow [1], it reads as follows:

$u \leftarrow 1$
for $r = 0$ to $n - 1$ do
  $v \leftarrow u$
  for $s = 1$ to $r$ do
    $u \leftarrow u + v$

The idea defended by Turing at the very beginning of his article is that "the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows". He does so on the program above, associating mathematical assertions about variables to the various points of the code flowchart. For instance, one assertion states that, prior to the execution of $u' = u + v$, variable $v$ contains $r!$ and variable $u$ contains $s \times r!$. The paper even includes a proof of termination. Turing gives an ordinal that decreases whenever a transition in the flowchart is taken. The ordinal is $(n - r)\omega^2 + (r - s)\omega + k$, where $k$ is assigned a value from 0 to 6 according to the statement under consideration. In modern terminology, we recognize the variant $n - r$ for the outer loop and the variant $r - s$ for the inner loop. As an alternative to the ordinal, Turing proposes the natural number $(n - r)2^{80} + (r - s)2^{40} + k$, since he is assuming 40-bit integers.

1. The finding of which is left as an exercise.

Another example of early proof is Tony Hoare's proof of FIND [53]. In this 1971 paper, Hoare shows the correctness of a program that rearranges the elements of an array as follows: given an index $f$, elements are swapped in such a way that we end up with a value $v$ at position $f$, elements smaller than $v$ to the left, and elements greater than $v$ to the right. Hoare builds the program by successive refinement steps and exhibits the loop invariants in the process. He carefully states and proves the eighteen lemmas that entail their validity. For instance, one of the loops is $\mathsf{while}\ A[i] < r\ \mathsf{do}\ i := i + 1$ and has the following invariant:

$$m \leq i\ \&\ \forall p(1 \leq p < i \supset A[p] \leq r).$$

The preservation of this invariant is stated as the following lemma:

$$\begin{array}{ll} & A[i] \leq r\ \&\ m \leq i\ \&\ \forall p(1 \leq p < i \supset A[p] \leq r) \\ \supset & m \leq i + 1\ \&\ \forall p(1 \leq p < i + 1 \supset A[p] \leq r). \end{array}$$

(The loop test is purposely strict, while this not necessary from the invariant and the lemma.) Hoare's proof also includes termination and preservation of the array elements. There is even a footnote commenting on the possibility of arithmetic overflows, in statements such as $i = i + 1$ above, and on possible workarounds. Remarkably, a subsequent mechanization of this proof in 1999 [39] showed that it did not contain a single error.

From a theoretical point of view, a manual proof such as in the examples above can be carried out for any code and any specification, the only limit being the mathematical skills of the user. In practice, however, this is not that simple. Proving a program requires to check the validity of many conditions, most of them being tedious and error-prone statements. Fortunately, the situation has evolved a lot in the past two decades. Languages, techniques, and tools have flourished, which allow the mechanization of the verification process. Still, before we describe them in detail, let us take some time to examine other examples. Our purpose is to show why deductive software verification can be truly difficult.

## 1.1 Program Verification is Challenging

A program specification can be limited to the mere *safety*: the program will execute without ever running into a fatal error, such as dividing by zero or accessing an invalid memory location. Proving safety can already be quite challenging. A few years ago, a bug was found in the Java standard library which was related to the computation of the average of two integers in some binary search [12]. With array indices `l` and `u` large enough, the computation of `(l + u) / 2` may return a negative value if the addition overflows, resulting in some fatal access out of array

bounds[2]. On a 32-bit machine with enough memory, it is indeed possible to have arrays with more than a billion elements, and thus to trigger this bug. A possible fix is to compute the mean value using `l + (u-l)/2` instead. This sole example shows that proving safety may require to prove the absence of arithmetic overflows, which in turn may require a lot of other properties to be checked. This can be even more difficult. Consider another fix using a logical shift right instead of a division, that is `(l + u) >>> 2` in Java. Then *there is* an overflow in some cases, but it is a harmless one. Thus proving this code correct would require an even more complex model of machine arithmetic.

Another example is termination. Most of the time, termination is a desired property and should naturally be associated to safety. It is worth pointing out that early papers on program verification, in particular the ones cited above [101, 53], already included proving termination as part of the proof process. In principle, proving termination has an easy rule of thumb: each loop or recursive function is associated with some entity, called a *variant*, which decreases at each step and cannot decrease indefinitely [49]. Typically, a natural number will do. But exactly as for safety above, proving termination can be challenging anyway. A historical example is John McCarthy's "91 function" [71]. It is a function over integers defined as follows:

$$f(n) = \begin{cases} f(f(n+11)) & \text{if } n \leq 100, \\ n - 10 & \text{otherwise.} \end{cases}$$

Figuring out a variant for this function is not that difficult: it is as simple as $101 - n$. But proving that it is indeed a variant, that is proving that it is non-negative and decreases for each recursive call, is another story. Indeed, it requires to establish a postcondition for the function, relating its output value to its parameter $n$ — namely, that $f(n)$ returns 91 whenever $n \leq 100$, and $n - 10$ otherwise. As for binary search above, proving a property as simple as termination actually requires us to prove much more about the program.

Another example of program the sole termination of which is difficult to establish is Robert Floyd's cycle detection algorithm, also known as "tortoise and hare" algorithm [61, ex. 6 p. 7]. To check whether a singly linked list contains a cycle, this algorithm traverses it at speeds 1 and 2 at the same time, using only two pointers. Whenever a pointer reaches the end of the list, the outcome is negative. Whenever the two pointers are equal, the outcome is positive. The beauty of this algorithm is that it necessarily terminates, the hare not being able to overtake the tortoise within the cycle, if any. Turning that into a proof is non-trivial. One has to state the finiteness of the memory cells of the list, to introduce the inductive notion of

2. Interestingly, binary search was precisely taken as an example of program verification in Jon Bentley's *Programming Pearls* [10] more than 25 years ago. Unfortunately, his proof does not consider the possibility of an arithmetic overflow.

path from one list cell to another, and finally to come up with a variant which is quite involved.

Here is yet another example where proving safety is truly difficult. Let us consider computing the first $M$ prime numbers, along the lines of Don Knuth's `MIX` program from *The Art of Computer Programming* [60, p. 147]. We are going to fill a table with the first $M$ primes, in increasing order. We store 2 in the first slot and then we consider odd numbers starting from 3, performing divisibility tests using the prime numbers computed so far [3]. Say we have already found the first $m$ prime numbers, for $m \geq 2$. They are stored in $p[1], \ldots, p[m]$. Then we look for the next prime, examining odd numbers starting from $p[m] + 2$. If $n$ is our candidate, the loop checking for the primality of $n$ can be sketched as follows:

$$\begin{array}{l} \mathsf{while}\ p[i]^2 < n \wedge n \mod p[i] \neq 0\ \mathsf{do} \\ \quad i \leftarrow i + 1 \end{array}$$

In practice, it is written differently, as we need to distinguish between the two possible reasons for a loop exit. However, the loop above suffices to illustrate the idea. Indeed, proving the safety of this code requires to prove that variable $i$ will not exceed the bounds of array $p$. More precisely, we are going to prove that $i$ will not exceed $m$, that is, we will not run out of prime numbers. Additionally, it ensures that dividing by $p[i]$ is safe, since $p[i] \geq 2$ for $i \leq m$. Last, it ensures the termination of the loop in an obvious way. To prove that $i$ does not exceed $m$, it suffices to prove that $p[m]$ is large enough for the test $p[m]^2 < n$ to fail. Fortunately, this is ensured by Bertrand's postulate: for $k > 1$, there is always a prime $p$ such that $k < p < 2k$. Therefore, we have $n < 2p[m] \leq p[m]^2$. Said otherwise, verifying the safety of this program requires not less than Bertrand's postulate. In 2003, Laurent Théry proved Knuth's program for prime numbers using Why and Coq [99]. Truly a tour de force, this proof includes the full proof of Bertrand's postulate in Coq.

As a last example, let us consider the verification of a sorting algorithm. The algorithmic literature contains so many sorting algorithms that it is natural to consider proving them as well, at least as an exercise. Expressing that a routine sorting an array ends up with the array being sorted is quite easy. It amounts to providing a postcondition such as

$$\forall ij,\ 0 \leq i \leq j < n \Rightarrow a[i] \leq a[j].$$

But this is only half of the specification, and this is the easy one. We also need to state that the final contents of the array is a rearrangement of its initial contents. Otherwise, a code that simply fills the array with a constant value would obviously fulfill the specification above. When it comes to specify the rearrangement property,

3. There are more efficient ways to compute prime numbers, but that is not the point here: the purpose of this program is to illustrate the `MIX` assembly language.

there are several solutions but none is easy to define. In a higher-order logic, one can state the existence of a bijection. Though elegant, it is not amenable to proof automation. Another solution is to introduce the notion of multiset, to turn the contents of an array into a multiset, and then to state the equality of the multisets for the initial and final contents of the array. A third possibility is to define the notion of permutation as the smallest equivalence relation containing transpositions, which is adapted to sorting algorithms that swap elements [45]. Whatever the solution, it will be quite difficult to get a fully automated proof, even for simple sorting algorithms such as insertion sort or selection sort.

All these verification examples show the necessity of expressive specification languages, even to prove simple properties such as safety. In particular, we cannot expect deductive program verification to be conducted within a decidable logic.

## 1.2 Specifications

Roughly speaking, a specification language is made of two parts: first, a mathematical language in which components of the specification (pre- and postconditions, invariants, etc.) are written, as well as verification conditions *in fine*; second, its integration with a programming language. Obviously, the choice of a specification language is tied to both the methodology used to generate the verification conditions and the technique used to prove them. These two aspects will be discussed in the next two sections.

As far as the mathematical language is concerned, first-order predicate calculus with equality is a natural choice, immediately picked up without discussion in pioneer work [49, p. 21]. Today, many systems use a flavor of first-order logic [9, 5, 30, 16, 65, 8]. Beside equality, a handful of built-in theories are typically considered. Linear arithmetic, purely applicative arrays, and bit vectors are the common ones; non-linear or real arithmetic may be available as well. Other extensions to first-order logic include recursive functions and algebraic data types — in Dafny [65] and Why3 [16] for instance — as well as polymorphism [17] and inductive predicates [16]. Set theory is an alternative, considered for instance in the B method [1].

An alternate route is to consider using the logic of an existing, general-purpose proof assistant. Examples of such systems successfully used in program verification include Coq [98], PVS [83], Isabelle [85], and ACL2 [58]. This approach has obvious benefits. First, it provides a powerful specification language, as these proof assistants typically provide a rich and higher-order logic. Second, the logic is already implemented and, in particular, interactive proof tools are ready to be used. Finally, proof assistants come with libraries developed over years, which eases the specification process. Anyhow, this approach has drawbacks as well. One is that a richer logic comes at a cost: proof automation is typically more difficult to achieve.

This is discussed later in Section 1.4.

One can also introduce a new logic, dedicated for program verification. Undoubtedly the best example is separation logic [89]. Among other things, it allows local reasoning on heap fragments, being a tool of choice for proving programs involving pointer data structures. Similar proofs with traditional first-order predicate calculus require cumbersome and explicit manipulations of sets of pointers.

Once the mathematical language for specification and verification conditions is set, we are left with its integration into a programming language. Pioneer work intends to bind logical assertions to the program flowchart [101, 49]. A key step was then Hoare's axiomatic basis [52] introducing the concept known nowadays as *Hoare triple*. In modern notation, such a triple $\{P\}s\{Q\}$ binds together a precondition $P$, a program statement $s$, and a postcondition $Q$. Its validity means that execution of $s$ in any state satisfying $P$ must result, on completion, in a state satisfying $Q$. Requiring $s$ to be terminating defines total correctness, as opposed to partial correctness otherwise. The beauty of Hoare logic is not really that it proposes a methodology to derive the correctness of a program — weakest preconditions are more efficient, as we will see in next section — but that it opens the way to a *modular* reasoning framework about programs. Software components can be enclosed in Hoare triples, with preconditions that are proof requirements for the caller and, conversely, postconditions that are ensured to be valid by the callee. In some hypothetical language, such a triple for a function $f$ could look like

**requires** $P$
**ensures** $Q$
**function** $f(x_1, \ldots, x_n)$

where $P$ may refer to the current state and to function parameters $x_i$, and $Q$ may refer to the state prior to the call, to the final state, to function parameters, and to the returned value, if any. This idea has been popularized by Meyer's *design by contract*, a key feature of the Eiffel programming language [76], and later by specification languages such as JML [63] and its derivatives [47, 8, 5]. Modern specification languages include some variations, such as the ability to have several contracts for a given function, to distinguish side effects from the postcondition, to allows exceptional behaviors, etc. Anyway, they all clearly derive from Hoare logic.

This is not the only way to go, though. A specification language can be tied to a programming language within a dedicated *program logic*, which mixes programs and logical statements in a more intricate way. The aforementioned B method is built on top of a notion of generalized substitution, which infuses programming constructs among logical ones within a single language. Similarly, when program verification is conducted within a general-purpose proof assistant, there is only one language, that is the logic language. A program $f$ is simply a purely applicative

and terminating function, already part of the logic. Proving it correct amounts to establishing a statement like

$$\forall x,\ P(x) \Rightarrow Q(x, f(x)). \tag{1.1}$$

The user is then left with a traditional proof, or is assisted with some "tactic" dedicated to program verification.

So far we have discussed the specification language and its integration with the programming language. We now turn to methodologies to extract verification conditions from a program and its associated specification.

## 1.3 Methodologies

Hoare logic is undoubtedly the most famous of all techniques, Hoare's paper [52] being the most cited among pioneer work. It proposes rules to establish the validity of Hoare triples. The rule for assignment, for instance, reads

$$\{P[x \leftarrow E]\}\ x := E\ \{P\}.$$

It assumes $E$ to be an expression without side effect, thus safely shared between program and logic, and also $x$ not to be aliased with another program variable. More generally, each program construct is given a deductive rule. Some additional rule allows to strengthen a precondition and to weaken a postcondition. It is a neat system, from which you can derive the validity of arbitrarily complex programs. In practice, however, one quickly finds it painful to figure out intermediate assertions for Hoare rules to compose nicely. The natural approach is rather to assign meaningful assertions at key places in the code, and to let the system figure out the intermediate assertions. This is exactly what Dijkstra's calculus of weakest precondition [35] is doing. Given a program statement $s$ and a postcondition $Q$, it computes a precondition $wp(s, Q)$, recursively over the structure of $s$, that precisely captures the weakest requirement over the initial state such that the execution of $s$ will end up in a final state satisfying $Q$. As a consequence, the validity of the Hoare triple $\{P\}s\{Q\}$ is equivalent to

$$P \Rightarrow wp(s, Q).$$

Computing weakest preconditions is the approach taken in most modern verification condition generators. Recent work has even improved over Dijkstra's calculus to provide greater efficiency [64]. There is still the limitation of programs being free of aliasing, though. As a consequence, the semantics of the program under consideration must be encoded as a set of types, symbols, and axioms, known as the *memory model*, together with an alias-free version of this program performing actions on

this memory model. Within the last decade, two intermediate languages appeared, the sole purpose of which is to perform the computation of weakest preconditions, independently of the memory model: Why [47, 15] and Boogie [4]. Many tools now employ this route which consists in building a memory model and then reusing an existing intermediate language (Krakatoa [74], SPEC# [6], VCC [30], Frama-C [50], Dafny, among others).

Building memory models is an activity in itself. Some key ideas come from the 70s [23]. They could be successfully applied at the turn of the century when practice finally caught up with theory [21], and then even improved [54]. Other models include ideas from state-of-the-art recent research. A nice example is the model used in the L4verified project [59]. This model [100] combines low-level details allowing the verification of high-performance C code with higher-level concepts of separation logic.

We already explained that proof assistants such as Coq, PVS, Isabelle, or ACL2, are tools of choice to write program specifications, and even to write programs when they happen to be purely applicative. Then, proving a program $f$ to be correct, that is establishing a statement such as (1.1), is a matter of following the definition of $f$, splitting the goal into smaller verification conditions. This may be automated, using a dedicated tactic (either built-in or developed for the purpose of program verification). The user may have to provide some intermediate assertions, such as a generalized specification for a recursive function. This can be compared to the computation of weakest preconditions. When done, the user is left with a bunch of purely logical statements and all the power of the proof assistant can be exploited to discharge them. An impressive example of such a program verification is the verified C compiler CompCert [69], developed and proved correct within the Coq proof assistant by Xavier Leroy and his team.

But the use of general-purpose proof assistants is not limited to the verification of purely applicative programs. One can use the logic of a proof assistant to define an imperative programming language and its semantics. Then a proof methodology can be carried out within the proof assistant and particular programs can be verified. This is called *deep embedding*. A successful example is Schirmer's work in Isabelle/HOL [90]. It provides a Hoare logic-like language with procedures, operating over a polymorphic state space. It was used in the L4verified project. An additional benefit of deep embedding is that it allows meta-theoretical proofs, such as soundness of the proof method w.r.t. the semantics of the language. An alternative to deep embedding is *shallow embedding*. Concepts related to programs are mere notations for semantic definitions, which can be tackled by unfolding or, better, by relevant axioms and theorems. An example is Ynot [80], a Coq-embedded system to verify higher-order and effectful programs using a combination of monads, separation logic, and Hoare triples. Another example is Arthur Charguéraud's characteristic formulae [24].

When a program is given a specification, and is processed through a suitable deductive verification method, one is left with a set of mathematical statements, the so-called *verification conditions*. The last step is to prove them.

## 1.4 Proofs

The L4verified and CompCert projects are two examples of large program verification conducted within a proof assistant (Isabelle/HOL and Coq, respectively). At least they show the relevance of using a general-purpose proof assistant to discharge verification conditions. Even when those are computed completely independently, for instance using a weakest precondition calculus, it is still possible to translate them to the native format of a proof assistant. This is always possible, as the logic of the proof assistant is typically richer than the one used to express the verification conditions.

However, there are several reasons why we may want to avoid this path, at least in a first step. One reason is that deductive verification typically generates numerous, yet simple, goals. These include array bounds checking, variant decrease, absence of arithmetic overflows, non-nullity of pointers, etc. They are better handled directly by automated theorem provers, as discussed below. Processing them within a proof assistant would incur an extra cost, such as one manipulation at least per verification condition, if not more. Another reason to avoid interactive proof in the first step is proof maintenance. In the process of figuring out the right specification, or even the right code, it is likely that verification conditions will change a lot before stabilizing. Any manipulation within the proof assistant, as small as it is, will slow the whole specify/prove cycle.

It is better to turn to automated theorem provers when it comes to the task of discharging verification conditions. Automated theorem provers have a long history, as demonstrated by the CASC competition [94] and the huge corresponding library of problems TPTP [95]. However, all the automated theorem provers taking part in this competition were never really involved in deductive program verification. The main reason must be the lack of support for arithmetic, a must-have in program verification [4]. Independently, another line of automated theorem provers emerged, called *SMT solvers* [5]. Initiated by Shostak's and Nelson-Oppen's decision procedures [92, 81] in the early 80s, it focuses on the combination of first-order logic, congruence closure, and built-in theories. The latter typically involve linear arithmetic, at least. One of the earliest provers in that line was Nelson's Simplify [34]. Developed in the 90s, in the context of program verification, it still performs impressively. Other SMT solvers followed, such as CVC3 [7], Yices [33], Z3 [32], and

4. Things may change, as there are rumors that next versions of Vampire, the state-of-the-art TPTP prover, will have support for arithmetic.

5. We only consider here SMT solvers with support for quantifiers.

Alt-Ergo [14]. Today, they have reached such a level of efficiency that a program implementing binary search can be proved correct fully automatically within a second. This includes proving safety, termination, absence of arithmetic overflow and full behavioral correctness.

Obviously, we are also left from time to time with verification conditions that cannot be proved automatically. A possibility is then to insert more intermediate assertions in the code, to reduce the cost of each individual deduction. In other words, the user inserts logical *cuts* to ease the proof. If this is unsuccessful, or if this is not even an option, then there is no other choice than turning to a proof assistant and starting an interactive proof. A definitive improvement in this process is the recent introduction of automated theorem provers within proof assistants. Isabelle/HOL is leading the way with its Sledgehammer tool [75].

Finally, another approach is to use a dedicated prover. It can be motivated by a need for a close connection between verification conditions as displayed during an interactive proof process and the input code and specification. Examples include the system KeY [9] and the B method, for instance. Regarding the latter, it may also be motivated by the poor support for set theory in existing automated theorem provers. One obvious drawback of a dedicated prover is that it will hardly benefit from state-of-the-art results in automated or interactive theorem proving, at least not for free.

# 2
# A Tool for Program Verification: Why3

Since my PhD, I have been continuously developing a tool for program verification, with other members of ProVal. This chapter describes this tool. It does not intend to be exhaustive — more details about Why3 are available in papers related to Why3 [16, 17], in its manual [15], and from its website `why3.lri.fr`. This chapter rather discusses design choices and technical solutions.

## 2.1 History

I started the development of a tool for deductive program verification during my PhD [36, 37]. At the time, it was a tactic for the Coq proof assistant, called `Correctness`. Given a Hoare triple $\{P\}e\{Q\}$, this tactic was building a proof of

$$\forall \vec{x},\ P \Rightarrow \exists \vec{y}, \exists r,\ Q \tag{2.1}$$

where $\vec{x}$ is the set of all variables occurring in the triple, $\vec{y}$ the set of variables possibly modified by expression $e$, and $r$ is the result of $e$. Actually, only the computational part of this proof was built, as a purely applicative interpretation of $e$ in a monadic style. When fed to Coq's `refine` tactic, it resulted in proof obligations, exactly as if Hoare logic rules had been applied. Programs included a standard WHILE language with local variables and (possibly recursive) functions. It was using a shallow embedding of the logic: data types were those of Coq, as were formulas in annotations. Using that tactic, I could prove a few non-trivial programs such as Hoare's FIND [39], sorting algorithms quicksort and heapsort [45], and Knuth-Morris-Pratt string searching algorithm. All proofs were conducted in Coq, with no proof automation beside the `omega` tactic for Presburger arithmetic. A few others used the `Correctness` tactic. Nicolas Magaud contributed the verification of insertion sort as an undergraduate internship [45]. Worth mentioning is Laurent Théry's proof of Knuth's algorithm for prime numbers [99], which includes the full proof of Bertrand's postulate in Coq (see Section 3.2).

In 2001, while visiting Cesar Muñoz at NASA Langley, I turned my implementation into a standalone tool, for Cesar to perform proofs using PVS instead of Coq. I called this tool Why, for it is a way to tell the machine *why* a computation is sound, instead of simply telling it *how* to compute. The technology was still based on a purely applicative translation of imperative programs, but proof obligations were now extracted from it by Why itself, no more by Coq's `refine` tactic. Yet it was still possible to build the Coq proof for (2.1) when the proof obligations were discharged using Coq. The tool Why was first released in July 2002.

The idea of translating imperative programs into purely applicative ones to derive proof obligations is elegant. In particular, it provides strong guarantees since, once proof obligations are discharged, a proof of (2.1) can be built automatically and checked by Coq [1]. Unfortunately, it also has the drawback of traditional Hoare logic: the user has to provide adequate intermediate assertions. At some point, it becomes a bottleneck, as we figured out when we started using Why for deductive verification of Java programs. It was in 2002 and Christine Paulin, Claude Marché, and Xavier Urbain were developing the tool Krakatoa [74], which translates Java programs into Why programs. The problem of intermediate assertions being a showstopper, we had to go for the obvious solution: the computation of weakest preconditions. We quickly modified the Why tool accordingly, weakest preconditions being straightforward to implement. There was one big side effect, though: we lost the Coq proof of (2.1) in the process, our weakest precondition calculus not being proof-producing.

The divorce from Coq was on its way, anyhow. At the same period, we were already experimenting with automated theorem provers, the first ones being haRVey [87] and Simplify [34]. Corresponding back-ends were added to Why. As a consequence, proof obligations were not all discharged with Coq anymore, ruling out the possibility of a complete Coq proof of (2.1). Other back-ends followed, for other automated theorem provers (CVC Lite in 2004, SMT-lib syntax in 2005, Zenon in 2006, Alt-Ergo in 2008) as well as other proof assistants (HOL Light and Mizar in 2003, Isabelle/HOL and HOL4 in 2005). All these back-ends were added "on demand", from our own experiments or from user requests. They were maintained to various degrees, depending on their actual use. Some of the back-ends have been used for no more than a single program verification.

Supporting many theorem provers is challenging. This is definitely more subtle than writing several pretty-printers for the corresponding input languages. The main reason is that theorem provers implement different type systems: some are untyped, some have simple types, and others have richer type systems with polymorphic types, dependent types, etc. The logic of Why happens to be a compromise: it is a first-order logic with polymorphic types. There is no difficulty to embed it

1. Note that there is still space for unsoundness, such as a buggy translation which would not preserve the semantics of the imperative program.

into richer type systems [2]. On the other way round, however, a careful translation must be carried out, since a mere removal of types would be unsound. A solution was worked out by Jean-François Couchot and Stéphane Lescuyer in 2007 [29] and recently improved by Andrei Paskevich and François Bobot [17].

In 2004, Claude Marché and myself started a tool for the verification of C programs, Caduceus [46]. Following the methodology of Krakatoa, it translated C programs into Why programs, on top of a memory model *à la* Burstall-Bornat [23, 21]. Why was clearly appearing as a suitable *intermediate language*. It is worth mentioning that another such intermediate language, Boogie, was developed independently at Microsoft Research [4]. Later, Claude Marché identified another intermediate language to stick between C/Java and Why, to factor out common concepts related to memory models [54, 73]. Yet the role of Why was unchanged: computing weakest preconditions and feeding them to theorem provers. This is not incompatible with the direct use of Why to prove programs, which I kept doing all these years.

In 2007, a conference was organized in Bordeaux to celebrate Don Knuth who was receiving an Honoris Causa degree from University Bordeaux 1. In an attempt to honor the man and his work, I implemented a small tool to verify `MIX` programs using Why. `MIX` is the hypothetical machine introduced by Knuth in *The Art of Computer Programming* [60], as well as the name of its assembly language. So the point was to verify assembly programs and to use Why as an intermediate language one more time. The tool is a front-end to Why. It takes an annotated `MIX` program as input and turns it into a set of sequential Why programs. Since assembly programs have arbitrary control flow graphs, the notion of loop invariant is replaced by a more general notion of *invariant*. Invariants can be placed anywhere in programs. The only requirement is that any cycle in the control flow graph should contain at least one invariant. Then the control flow graph is traversed and any path from one invariant to another results in a separate Why program. The memory model of `MIX` is simple, with a bunch of references for the registers and status flags and a single array for the whole memory. More details are available in the conference paper [40].

In 2009, Bárbara Vieira (University do Minho, Portugal) visited ProVal and started the development of a tool to verify CAO programs. CAO is a domain-specific language for cryptography, developed within the CACE European Project. This tool translates CAO to Jessie, which is then translated to Why. More details are available from the related publication [2]. After Java, C, and `MIX`, this is yet another example of the use of Why as a verification condition generator.

After several years of development, the code of Why was becoming messy. In particular, the back-end of the code offering support for the various theorem provers

2. This is typically the case of proof assistants, with the notable exception of PVS, which does not provide declaration-level polymorphism. PVS supports type-parametric theories only, which turns the Why output into a nightmare of small theories.

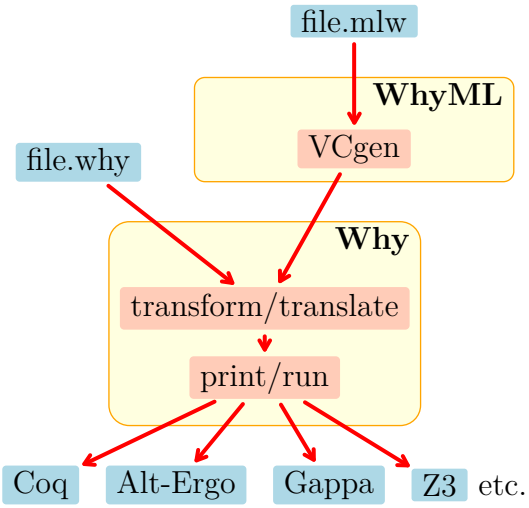



Figure 2.1: Overview of Why3's architecture.

was getting so complex [3] that it was almost impossible to add new features, such as support for algebraic data types for instance. Beside, we all know from *The Mythical Man-Month* [22] that "we have to throw one away". So on February 2010 we started a brand new implementation, from scratch, with Andrei Paskevich, Claude Marché, and François Bobot. It is called Why3 and was first released on December 2010.

## 2.2 Overview of Why3

Figure 2.1 gives an overview of Why3's architecture. It clearly distinguishes two parts: a purely logical back-end and a programs front-end.

Logical files include declaration or definition of types and symbols, axioms, and goals. Such declarations are organized in small components called *theories*, which are grouped in files with suffix `.why`. The purpose of Why3 is to extract goals from theories and to pass them to provers using suitable series of transformations and printers, as sketched in Fig. 2.1. For instance, the command line

```
why3 -P alt-ergo -t 10 -T EuclideanDivision int.why
```

extracts all goals from theory `EuclideanDivision` in file `int.why` and passes them to the SMT solver Alt-Ergo with a timeout of 10 seconds. This includes transforming

3. At the very beginning, the front-end for type checking programs and computing weakest preconditions was clearly the main part of the code; today, it is the opposite: the support for many provers is definitely dominating.

goals to remove logical features that are not known from Alt-Ergo, such as recursive and inductive definitions. Conversely, it makes use of Alt-Ergo's built-in theories, such as linear integer arithmetic or AC symbols.

Why3 provides a programming language, called WhyML. As depicted in Fig. 2.1 page 16, it is a mere front-end to what we have described so far. A WhyML input text is a file with suffix `.mlw`. It contains a list of theories and/or modules. Modules extend theories with *programs*. This front-end simply turns modules into theories, replacing programs by goals corresponding to the verification conditions — as computed by a weakest precondition calculus. WhyML files are handled by the tool `why3ml`, which has a command line similar to `why3`. For instance,

```
why3ml -P alt-ergo turing.mlw
```

will run the SMT solver Alt-Ergo on all verification conditions for the programs contained in file `turing.mlw`.

## 2.3 Logic

Why3 implements a polymorphic first-order logic. There are four kinds of declarations in this logic: types, function and predicate symbols, axioms, and goals.

Non Interpreted and Alias Types. Types can be either non interpreted, aliases for type expressions, or (possibly recursive) algebraic data types. Types can be mutually defined. A non interpreted type `t` is simply declared as

```
type t
```

A type alias introduces a shortcut for a type definition, *e.g.*,

```
type length = int
```

Technically, type aliases are expanded in a systematic manner. Built-in types include type `int` of integers, type `real` of real numbers, and tuple types.

Algebraic Data Types. An algebraic data type is defined with one or more constructors. For instance, the type of polymorphic lists is declared as

```
type list α = Nil | Cons α (list α)
```

Each constructor introduces a new function symbol. Values of algebraic data types can be examined using pattern matching, in both terms and formulas (see below). Records are particular cases of algebraic data types, with a specific syntax:

```
type polar = {| radius: real; angle: real |}
```

A record introduces projection functions for its fields. The constructor of a record type is not accessible; syntax {| . . . |} must be used instead, both for construction and destruction.

Type Expressions. A type expression is either a type variable or the application of a type symbol.

| | | | |
|---|---|---|---|
| $\tau$ | ::= | $\alpha$ | type variable |
| | \| | $s\ \tau \ldots \tau$ | type symbol application |

Functions and Predicates. Function and predicate symbols can be non interpreted or (mutually and/or recursively) defined. For instance, a non interpreted relation `rel` over type `t` can declared as

```
predicate rel t t
```

and a recursive function `length` over lists is defined as follows:

```
function length (l: list α): int =
  match l with
  | Nil → 0
  | Cons _ r → 1 + length r
  end
```

Why3 automatically verifies that recursive definitions are terminating. To do so, it looks for an appropriate lexicographic order of arguments that guarantees a structural descent.

Inductive Predicates. A predicate can also be defined inductively. For instance, the reflexive and transitive closure of `rel` is defined as follows:

```
inductive rstar t t =
  | Refl:  ∀ x: t. rstar x x
  | Trans: ∀ x y z: t. rstar x y → rel y z → rstar x z
```

Standard positivity restrictions apply to ensure the existence of a least fixed point.

Terms, Formulas, and Patterns. As usual in first-order logic, Why3 distinguishes terms and formulas. Terms are built as follows:

| | | | |
|---|---|---|---|
| $t$ | ::= | $x$ | variable |
| | \| | $c$ | constant |
| | \| | $s\ t \ldots t$ | function symbol application |
| | \| | if $f$ then $t$ else $t$ | conditional expression |
| | \| | match $t$ with $p \to t, \ldots, p \to t$ end | pattern matching |

Constants include integer and real literals. Note the intrusion of formulas into terms in conditional expressions. Formulas are built as follows:

| | | | |
|---|---|---|---|
| $f$ | ::= | $s\ t \ldots t$ | predicate symbol application |
| | \| | $\forall x : \tau . f$ | universal quantification |
| | \| | $\exists x : \tau . f$ | existential quantification |
| | \| | $f \wedge f$ | conjunction |
| | \| | $f \vee f$ | disjunction |
| | \| | $f \rightarrow f$ | implication |
| | \| | $f \leftrightarrow f$ | equivalence |
| | \| | not $f$ | negation |
| | \| | true | |
| | \| | false | |
| | \| | if $f$ then $f$ else $f$ | conditional expression |
| | \| | match $t$ with $p \rightarrow f, \ldots, p \rightarrow f$ end | pattern matching |

Formulas enrich the usual connectives of first-order logic with conditional and pattern matching expressions. Finally, patterns are built as follows:

| | | | |
|---|---|---|---|
| $p$ | ::= | $x$ | variable |
| | \| | $s\ t \ldots t$ | constructor application |
| | \| | _ | catch all |
| | \| | $p \vert p$ | or pattern |
| | \| | $p$ as $x$ | binding |

Axioms and Goals. Axioms can be introduced to enrich the logical context of subsequent declarations. The syntax is immediate:

```
axiom rel_sym: ∀ x y: t. rel x y → rel y x
```

A goal is declared with an equally obvious syntax:

```
goal test_length: length (Cons 2 (Cons 3 (Cons 5 Nil))) = 3
```

The context of a goal consists of all preceding declarations. Finally, a `lemma` declaration combines both notions of goal and axiom: it is a goal at the place where it appears and an axiom for subsequent declarations. A typical lemma is the following:

```
lemma length_nonnegative: ∀ l: list α. length l ≥ 0
```

In the case of this lemma, a proof by induction is needed and we may use Coq to prove it. As an axiom, it will be used in subsequent proofs by automated theorem provers that may not be powerful enough to prove it, such as SMT solvers.

Semantics. First-order logic is standard. For instance, a definition of many-sorted first-order logic can be found in Manzano's *Extensions of first-order logic* [72]. But *polymorphic* first-order logic is *not* standard. Informally speaking, polymorphism in Why3 is a declaration-level polymorphism, à la Hindley-Milner [77]. It means that each polymorphic declaration is a schema for the (possibly infinite) set of all its monomorphic instances. For instance, the following declaration of a polymorphic function symbol `length` over lists

```
function length: list α : int
```

can be understood as the declaration of an infinite set of function symbols:

```
function length_int:      list int        : int
function length_bool:     list bool       : int
function length_list_int: list (list int) : int
...
```

Similarly, a polymorphic axiom such as

```
axiom length_nonneg: ∀ l: list α. length l ≥ 0
```

is also to be understood as an infinite set of axioms

```
axiom length_nonneg1: ∀ l: list int.        length l ≥ 0
axiom length_nonneg2: ∀ l: list bool.       length l ≥ 0
axiom length_nonneg3: ∀ l: list (list int). length l ≥ 0
...
```

A polymorphic goal, on the contrary, can always be turned into a monomorphic goal, by instantiating all its type variables with fresh, uninterpreted types. A precise definition of polymorphic first-order logic is given by François Bobot and Andrei Paskevich [17].

In practice, polymorphic first-order logic cannot be reduced to many-sorted first-order logic using an infinite number of instantiations. Thus Why3 implements various methods for encoding polymorphism, that are proved to be sound and complete. This is described in the paper just mentioned.

### Theories

Declarations are organized in small components called *theories*, which are grouped in files with suffix `.why`. For instance, a file `list.why` may contain a first theory `List` defining the data type of polymorphic lists

```
theory List
  type list α = Nil | Cons α (list α)
end
```

and then another theory `Length` where the length of a list is defined:

```
theory Length
  use import int.Int
  use import List
  function length (l: list α) : int =
    match l with
    | Nil      → 0
    | Cons _ r → 1 + length r
    end
  lemma length_nonnegative: ∀ l: list α. length l ≥ 0
end
```

The declaration `use import` is used to import symbols from an existing theory. Here, integer arithmetic symbols are imported from the standard library `int.Int`, as well as the previously defined theory `List`. Imported symbols are shared: two `import`s of the same theory will result into a single addition of the corresponding symbols to the context. The purpose of theories is to control the logical context, which is an important matter when one uses automated theorem provers. If there is no need to import theory `Length`, we will avoid cluttering the logical context with unnecessary definitions and axioms.

The other way to use a theory is to *clone* it. Cloning a theory is making a copy of all its declarations, together with a possible instantiations of some of its non interpreted symbols. Here is an example. Let us introduce the following theory which defines the reflexive transitive closure of an abstract relation `rel` over an abstract type `t`.

```
theory RTClosure
  type t
  predicate rel t t
  inductive rstar t t =
    | Refl:  ∀ x:t. rstar x x
    | Trans: ∀ x y z:t. rstar x y → rel y z → rstar x z
end
```

Given a particular relation `next` over integers, we can clone this theory to get the reflexive transitive closure of `next`:

```
clone import RTClosure with type t = int, predicate rel = next
```

It is exactly as if we had typed in a fresh copy of the inductive definition above, with `int` and `next` respectively substituted for `t` and `rel`. It saves us the trouble — and the possibility of a mistake. Cloning theories is a powerful concept. For instance, we can clone the whole theory

```
clone import RTClosure
```

to get a fresh type symbol `t`, a fresh predicate symbol `rel`, and its reflexive transitive closure `star`. More subtly, we can clone it twice to get two relations and their reflexive transitive closures over a single type `elt`:

```
type elt
clone import RTClosure as R1 with type t = elt
clone import RTClosure as R2 with type t = elt
```

Equivalently, and more elegantly, we can reuse the type provided by the first cloning to instantiate the second one:

```
clone import RTClosure as R1
clone import RTClosure as R2 with type t = R1.t
```

The cloned theories are renamed using `as X` declarations to avoid name clashes.

Generally speaking, a theory with $n$ uninterpreted symbols can be cloned in up to $2^n$ different ways (not mentioning the substitution itself). As illustrated above, different cloning of a single theory can be meaningful. Why3's standard library is making a great use of theory cloning to avoid duplicated axioms; it is worth having a look at it to understand how useful this simple idea can be.

Though they were developed independently, the concept of theory cloning is similar to that of *theory interpretation* in PVS [84].

## 2.4 Proofs

Generally speaking, all information related to a given prover is gathered into a text file called a *driver*. Such a file contains the list of transformations to apply, the pretty-printer to use, the list of built-in symbols and corresponding axioms to ignore, etc. The purpose of drivers is to ease the addition or modification of a prover support, without changing Why3's code. It also eases experiments such as removing the use of a prover's built-in theory.

Beside its command line tool, Why3 comes with an IDE [4]. It displays goals and allows the user to perform proofs by repeated applications of provers and simple transformations such as splitting. A screenshot of the IDE is given in Fig. 2.2. The whole proof session can be saved into a file, to be reloaded from the IDE or to be replayed in a batch mode. The IDE is not limited to automated theorem provers. Interactive proofs using Coq can be edited, and then checked. Coq proofs are saved as part of the proof session.

4. Why3's IDE is mostly contributed by Claude Marché.

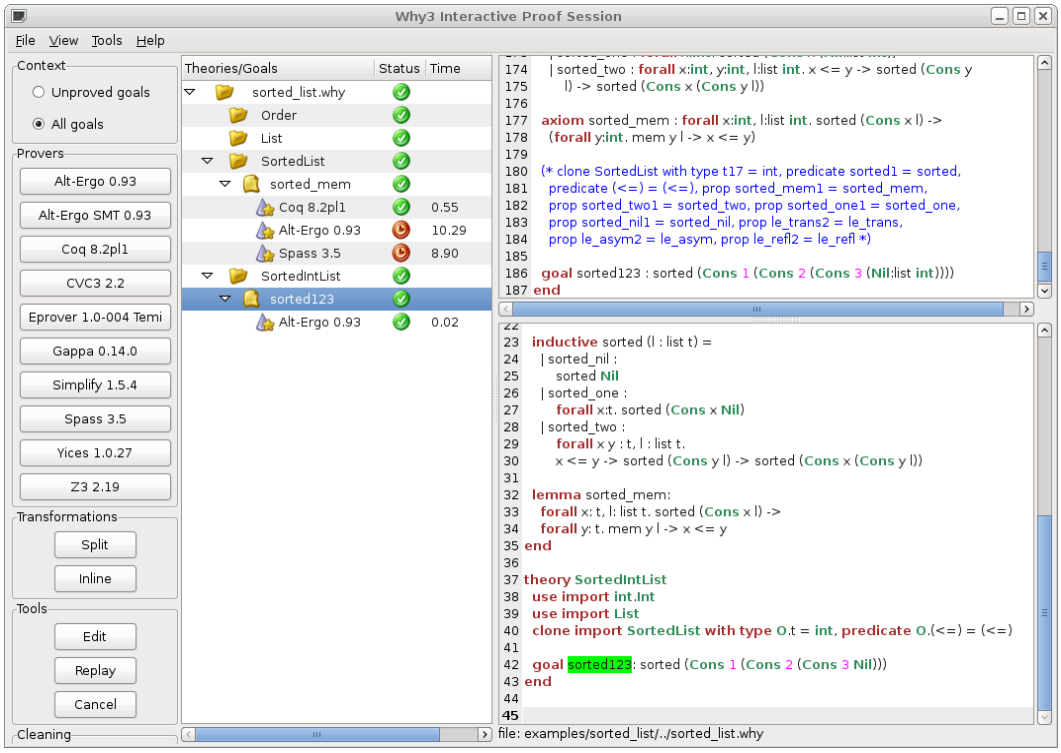



Figure 2.2: Screenshot of Why3's IDE.

## 2.5 Programs

As depicted in Fig. 2.1, Why3 is a verification condition generator for a programming language called WhyML. The main features of this language can be summarized as follows:

- it borrows many features from ML (syntax, polymorphism, algebraic data types, immutable variables, local and anonymous functions);
- yet it is mostly a first-order language *i.e.* there is currently no support for higher-order programs;
- it keeps to the main ideas of Hoare logic: (1) any term from the logic can be used in programs; and (2) aliasing is not allowed.

In the remaining of this section, we detail the most distinguishable aspects of WhyML.

### Mutable State

There is only one kind of mutable data structures in WhyML: records with mutable fields. The concept of mutable variable, or *reference* in ML-like terminology, is defined as a particular case of record type [5]:

```
type ref α = {| mutable contents: α |}
```

Then functions to create, access, and update references are easily defined, using respectively record creation, field access, and field update. For instance, function `:=` to update a reference is defined as follows:

```
let (:=) (r:ref α) (v:α) = {} r.contents ← v { !r = v }
```

Note the empty precondition and the postcondition stating that, in the final state, the value of `!r` is equal to `v`. Here is for instance Turing's early proof of program [101] written in WhyML using `for` loops and a reference `u`:

```
let routine (n: int) =
  { n ≥ 0 }
  let u = ref 1 in
  for r = 0 to n-1 do invariant { !u = fact r }
    let v = !u in
    for s = 1 to r do invariant { !u = s * fact r }
      u := !u + v
    done
  done;
  !u
  { result = fact n }
```

(Annotations make use of a function symbol `fact` imported from Why3's standard library.) Note that `for` loop indexes (`r` and `s` here) are not mutable, as in OCaml.

Mutable states mean that we may need to refer to previous values of states in annotations. Two constructs are provided for that purpose. First, a postcondition may refer to the value of term $t$ in the pre-state with `old` $t$. Second, labels can be introduced within code and the value of a term $t$ at a program point labeled with $L$ is denoted `at` $t$ $L$. From the parsing and type checking point of view, `old` and `at` are nothing else than polymorphic identity functions. They are interpreted later, in the weakest precondition calculus.

### Model Types

Let us consider modeling arrays in WhyML. As usual in Hoare logic, an array is modeled as a map from integers to values, which is mutated as a whole. Assuming

5. This is exactly how references are defined in OCaml.

we have maps available from Why3's standard library, an array can be defined as a reference containing a map or, equivalently, as a new record type with a mutable field:

```
use import map.Map
type array α = {| mutable elts: map int α |}
```

Although it is fine for many purposes, this model of arrays has a drawback: it does not prevent us from copying the entire contents of one array into another. This can indeed be achieved as simply as

```
a1.elts ← a2.elts
```

We would like to provide operations such as array access or array assignment, but certainly not the ability to make such an atomic copy. Actually, we intend the record type above to be a *model* of an array, not the array data type itself. For this purpose, WhyML introduces the notion of *model type*. It consists in replacing, in the type definition above, the equality sign by the `model` keyword:

```
type array α model {| mutable elts: map int α |}
```

This way, the type `array` $\alpha$ is no more a record type in programs, but rather an abstract type. In the logic, however, that is in annotations, it is still defined as a record type. It means that we can refer to the field `elts` when specifying operations over arrays and when annotating programs using arrays. The other way round, we cannot refer to field `elts` in programs anymore, which means that we cannot define operations such as array access or array assignment. We can only declare them. Here is for instance how array access is introduced:

```
val ([]) (a: array α) (i: int) :
  { 0 ≤ i < length a } α reads a { result = a[i] }
```

As a consequence, what we end up with is not an implementation of the array data structure, but rather a signature or *model* of this data type. This is perfectly fine; after all, an array in *not* implemented as a purely applicative map. The same technique can be used for other mutable data structures that cannot be implemented in WhyML. For instance, a mutable queue implemented as a singly linked list and two pointers to its extremities cannot be implemented in WhyML. But such a data structure can easily be modeled using purely applicative lists. It would start as follows:

```
type t α model {| mutable elts: list α |}
val push: x: α → q: t α →
  {} unit writes q { q.elts = old q.elts ++ Cons x Nil }
...
```

### Regions

We are not done with our model of arrays: the notion of array length is still missing (it is used in the precondition for function `[]` above). A simple solution is to have a notion of length associated to purely applicative maps, as a function from `map int` $\alpha$ to `int`. Then the length of array `a` is simply the length of `a.elts`. Equivalently, we could replace type `map int` $\alpha$ above with a product type `(int, map int` $\alpha$`)`.

This is not a good solution, however. Having the length tied to the map means that each modification to the array contents possibly modifies the length as well. Of course, axioms over maps or suitable postconditions for array operations could ensure that this is not the case. Unfortunately, it would also require the user to state the invariance of an array length in annotations. For instance, a loop over an array of size 10 would look like

```
for i = 0 to 9 do invariant { length a = 10 ∧ ... }
  a[i] ← 0
done
```

to account for the array length not being modified. Without this annotation, it is no more possible to prove that `a[i]` is a legal array access. Similarly, postconditions for function mutating arrays would have to ensure length invariance as well. Adding such annotations automatically would be an ugly hack, with no way to figure out the proper generalization. Finally, all these annotations will result in more complex verification conditions.

Instead, we turn to a much more elegant solution. We store the array length into another, immutable, field:

```
type array α model {| length: int; mutable elts: map int α |}
```

This way, any array operation which mutates the array contents results in a modification of field `elts`, while field `length` is left unchanged, with no need to say anything about it.

Clearly, record types with possibly mutable fields allow us to model arrays in a way that traditional Hoare logic cannot [6]. To do so, WhyML makes a slight shift from traditional Hoare logic. The primitive notion is not that of mutable variable, but rather that of a *region*. Regions are introduced by mutable record fields.

From a technical point of view, effects are now sets of regions and no more sets of variables as they used to be in Why2. Accordingly, weakest preconditions must now quantify over fresh values for regions. They must also reconstruct the values of all variables involving the corresponding regions. For instance, the weakest

6. Of course, we could model an array with *two* variables, one for its length and another one for its contents. But this would already be a translation from a language with arrays to another language without. What we seek here is a proper modeling of arrays directly in WhyML.

precondition for the loop above scanning a 10-element array will look like

$$\forall a_0, \ldots \forall m_1, \mathsf{let}\ a_1 = \{\texttt{length} = a_0.\texttt{length}; \texttt{elts} = m_1\}\ \mathsf{in}\ \ldots$$

The modification of the array within the loop body is interpreted as a new map $m_1$ and a new value $a_1$ for array `a` is built from its old value $a_0$ and $m_1$. As a consequence, any assumption over the length of $a_0$ will hold for $a_1$ as well. This new way of computing weakest preconditions is explained later in this section, on page 31.

### Abstract Syntax

The abstract syntax of WhyML expressions is given in Fig. 2.3. On top of that, several constructs are defined as syntactic sugar. We briefly describe these constructs here for the reader to be able to go through all the exemples from the next chapter. The sequence is the simplest one:

$$e_1;\ e_2 \stackrel{\text{def}}{=} \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$$

Access to a record field $f$ can be reduced to a logical term using a let binding:

$$e_1.f \stackrel{\text{def}}{=} \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ x_1.f$$

Indeed, any logical term can be used in programs. Additionally, predicate symbols can be used in programs as well, according to the following sugar:

$$\begin{aligned} s\ e_1\ \ldots\ e_n \stackrel{\text{def}}{=}\ & \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \ldots\ \mathsf{let}\ x_1 = e_1\ \mathsf{in} \\ & \mathsf{if}\ s\ x_1\ \ldots\ x_n\ \mathsf{then}\ \mathit{True}\ \mathsf{else}\ \mathit{False} \end{aligned}$$

This includes equality and integer comparisons for instance.

A function call $e_1\ e_2$, with $e_1$ of type $x_2 : \tau_2 \to \kappa_1$, is translated using the non-deterministic construct any:

$$e_1\ e_2 \stackrel{\text{def}}{=} \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ \mathsf{any}\ \kappa_1$$

The expression $\mathsf{any}\ \kappa_1$ stands for any computation with type, effect, and specification as specified by $\kappa_1$; in a context of modular verification, this is exactly what the function call means. Similarly, assignment of record field $f$ can be interpreted using the any construct, as if it would be a function call:

$$e_1.f \leftarrow e_2 \stackrel{\text{def}}{=} \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ \mathsf{any}\ \{\}\mathit{unit}\ \mathsf{writes}\ x_1.f\{x_1.f = x_2\}$$

| | | | |
|---|---|---|---|
| $e$ | ::= | $x$ | variable |
| | \| | $t$ | term |
| | \| | let $x = e$ in $e$ | local binding |
| | \| | let rec $d, \ldots, d$ in $e$ | local binding |
| | \| | fun $d$ | anonymous function |
| | \| | if $e$ then $e$ else $e$ | conditional |
| | \| | loop $e$ invariant $f$ | infinite loop |
| | \| | for $x = x$ to $x$ do $e$ invariant $f$ | for loop |
| | \| | match $x$ with $p \to e, \ldots, p \to e$ end | pattern matching |
| | \| | raise $(E\ e)$ | exception throwing |
| | \| | try $e$ with $E\ x \to e, \ldots, E\ x \to e$ | exception handling |
| | \| | $L : e$ | label |
| | \| | assert $f$ | assertion (cut) |
| | \| | check $f$ | assertion |
| | \| | assume $f$ | assumption |
| | \| | absurd | absurdity |
| | \| | any $\kappa$ | non-determinism |
| $d$ | ::= | $x : \tau, \ldots, x : \tau = \{f\}\ e\ \{q\}$ | function body |
| $\tau$ | ::= | $\alpha$ | type variable |
| | \| | $s\ r, \ldots, r\ \tau, \ldots, \tau$ | type application |
| | \| | $x : \tau \to \kappa$ | function type |
| $\kappa$ | ::= | $\{f\}\ \tau\ \epsilon\ \{q\}$ | specification |
| $\epsilon$ | ::= | reads $r, \ldots, r$ writes $r, \ldots, r$ raises $E, \ldots, E$ | effect |
| $q$ | ::= | $f, E \to f, \ldots, E \to f$ | postcondition |

Figure 2.3: Abstract syntax of WhyML.

A while loop is syntactic sugar for an infinite loop where the predefined exception *Exit* is used to exit:

$$\begin{array}{l}\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{invariant}\ I\ \mathsf{variant}\ t \stackrel{\text{def}}{=}\\ \quad \mathsf{try\ loop\ if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else\ raise}\ \mathit{Exit}\ \mathsf{invariant}\ I\ \mathsf{variant}\ t\\ \quad \mathsf{with}\ \mathit{Exit}\_ \to ()\end{array}$$

Within an infinite loop, the variant is some syntactic sugar for an assertion placed at the end of the loop body:

$$\mathsf{loop}\ e\ \mathsf{invariant}\ f\ \mathsf{variant}\ t \stackrel{\text{def}}{=} \mathsf{loop}\ L : e;\ \mathsf{assert}\ t \prec \mathsf{at}\ t\ L\ \mathsf{invariant}\ f$$

A fresh label $L$ is used to denote the starting point of the loop body, in order to state that the variant is decreasing ($\prec$ is the order relation associated to the variant $t$).

Variants for recursive functions is also syntactic sugar. We use the function preconditions to insert the corresponding assertions. We proceed in two steps. First, we save the value of the variants at the function entry points, in local variables $v_i$:

$$\begin{array}{l}\mathsf{let\ rec}\ f_1\ \vec{x_1}\ \mathsf{variant}\ t_1 = e_1\\ \mathsf{and}\ f_2\ \vec{x_2}\ \mathsf{variant}\ t_2 = e_2\\ \ldots\\ \mathsf{and}\ f_n\ \vec{x_n}\ \mathsf{variant}\ t_n = e_n\\ \mathsf{in}\ e\end{array} \stackrel{\text{def}}{=} \begin{array}{l}\mathsf{let\ rec}\ f_1\ \vec{x_1} = \mathsf{let}\ v_1 = t_1\ \mathsf{in}\ e_1\\ \mathsf{and}\ f_2\ \vec{x_2} = \mathsf{let}\ v_2 = t_2\ \mathsf{in}\ e_2\\ \ldots\\ \mathsf{and}\ f_n\ \vec{x_n} = \mathsf{let}\ v_n = t_n\ \mathsf{in}\ e_n\\ \mathsf{in}\ e\end{array}$$

Then we type each function body $e_i$ within an environment where function $f_j$ has type

$$f_j : \vec{x_j} \to \{t_j(\vec{x_j}) \prec v_i \wedge p_j\}\tau_j\epsilon_j\{q_j\}$$

where $\prec$ is the order relation associated to all variants $t_i$ (it must be unique).

Finally, lazy operators && and || are syntactic sugar for conditional constructs.

$$e_1\ \texttt{\&\&}\ e_2 \stackrel{\text{def}}{=} \begin{cases}\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathit{andb}\ x_1\ e_2 & \text{if } e_2 \text{ is pure,}\\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ \mathit{false} & \text{otherwise.}\end{cases}$$

We make a particular case when $e_2$ is pure (that is, without any effect other than read effects) to limit the number of conditional constructs. This helps in getting smaller and more intuitive verification conditions. Operator || is interpreted in a similar way.

### Type Checking

The purpose of Why3 type checking is three-fold:

1. to perform traditional ML-like type checking;

2. to infer effects;
3. to exclude aliases.

Regarding point 1, we use a standard W algorithm [77]. We avoid the issue of polymorphic references as follows: types are not generalized at local binders and global binders only bind values, hence the value restriction applies [102]. Point 2 is also standard: it amounts to compute the sets of regions possibly accessed and possibly modified for each program sub-expression, as well as the set of possibly raised exceptions. It follows Talpin and Jouvelot's *type and effect discipline* [96, 97] in a straightforward way. The resulting effects are used to compute weakest preconditions.

Point 3 is the detection, and then rejection, of possible region aliases. This is a requirement for the weakest precondition calculus to be sound, as it assumes distinct regions to denote distinct memory cells. There are two different ideas. First, functions are polymorphic w.r.t. regions, exactly as they are polymorphic w.r.t. types. For instance, a function that swaps the contents of two references has a type such as

$$\texttt{swap} : \forall r_1, r_2.\ \forall \alpha.\ \texttt{ref}\ r_1\ \alpha \to \texttt{ref}\ r_2\ \alpha \to\ \texttt{unit}$$

(where we omit annotations for clarity). When we implement and prove correct a function with such a type, we assume regions $r_1$ and $r_2$ to be distinct. Therefore, when we apply this function to two references, we check that the corresponding references are indeed distinct. As a consequence, an expression such as

$$\mathsf{let}\ r = \texttt{ref}\ 0\ \mathsf{in}\ \texttt{swap}\ r\ r$$

is ill-typed. A function taking twice the same reference as argument should rather have a type such as

$$\texttt{swap'} : \forall r.\ \forall \alpha.\ \texttt{ref}\ r\ \alpha \to \texttt{ref}\ r\ \alpha \to\ \texttt{unit}.$$

However, to keep the system as simple as possible (but no simpler), regions are hidden from the user. To make this possible, a compromise is made. First, regions provided by function arguments must all be distinct. This rules out functions such as `swap'` above, but they are of poor interest anyway. Second, regions returned as part of the function result must be freshly allocated, *i.e.*, allocated during the function call. This rules out functions such as the identity over references:

$$\texttt{id} : \forall r.\ \forall \alpha.\ \texttt{ref}\ r\ \alpha \to \texttt{ref}\ r\ \alpha.$$

Again, such functions are seldom useful. These two conditions are ensured by the Why3 type checker.

### Weakest Preconditions

Given a program expression $e$, a postcondition $f$, and $\vec{q}$ a set of exceptional postconditions $E_1 \to f_1, \ldots, E_n \to f_n$, we define the weakest precondition $\mathsf{wp}(e, f, \vec{q})$ recursively over the structure of $e$.

Purely Applicative Constructs. The case of a program variable is immediate:

$$\mathsf{wp}(x, f, \_) \stackrel{\text{def}}{=} f[\mathit{result} \leftarrow \lceil x \rceil],$$

and so is the case of a logical term:

$$\mathsf{wp}(t, f, \_) \stackrel{\text{def}}{=} f[\mathit{result} \leftarrow t].$$

The weakest precondition of a pattern matching is a mere swapping of the two constructs:

$$\begin{array}{l}\mathsf{wp}(\mathsf{match}\ x\ \mathsf{with}\ p_1 \to e_1, \ldots, p_n \to e_n\ \mathsf{end}, f, \vec{q}) \stackrel{\text{def}}{=} \\ \quad \mathsf{match}\ x\ \mathsf{with}\ p_1 \to \mathsf{wp}(e_1, f, \vec{q}), \ldots, p_n \to \mathsf{wp}(e_n, f, \vec{q})\ \mathsf{end}.\end{array}$$

Of course, this is that simple since we have pattern matching in the logic as well. Getting rid of it is left to Why3's back end (see Section 2.4).

Sequence and Conditional. Sequence and conditional follow the standard rules. The sequence is here generalized into a let binding:

$$\mathsf{wp}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e_1, \mathsf{wp}(e_2, f, \vec{q})[\lceil x \rceil \leftarrow \mathit{result}], \vec{q}).$$

In the particular case of a true sequence $e_1; e_2$, that is when $x$ does not appear in $e_2$, it simplifies as expected:

$$\mathsf{wp}(e_1;\ e_2, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e_1, \mathsf{wp}(e_2, f, \vec{q}), \vec{q}).$$

When $e_1$ and $e_2$ cannot raise an exception, it simplifies even further, to the traditional rule

$$\mathsf{wp}(e_1;\ e_2, f) \stackrel{\text{def}}{=} \mathsf{wp}(e_1, \mathsf{wp}(e_2, f)).$$

The conditional rule is also standard:

$$\mathsf{wp}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e_1, \mathsf{if}\ \mathit{result}\ \mathsf{then}\ \mathsf{wp}(e_2, f, \vec{q})\ \mathsf{else}\ \mathsf{wp}(e_3, f, \vec{q}), \vec{q}).$$

However, in the particular case where $e_2$ and $e_3$ are both logical terms or variables, we avoid the duplication of $f$ using a conditional term:

$$\mathsf{wp}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e_1, f[\mathit{result} \leftarrow \mathsf{if}\ \mathit{result}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3], \vec{q}).$$

Assertions. Rules for assertions are straightforward to derive. An assertion introduced with assert accumulates to the postcondition:

$$\mathsf{wp}(\mathsf{assert}\ f_1, f_2, \_) \stackrel{\text{def}}{=} f_1 \wedge f_2.$$

In practice, such a conjunction is tagged so that a later split of this formula will produce $f_1$ and $f_1 \Rightarrow f_2$, instead of merely $f_1$ and $f_2$. This way, $f_1$ acts as a logical cut, which is the purpose of assert. A variant of assert, called check, results into a symmetric conjunction. An assertion introduced with assume is turned into an hypothesis:

$$\mathsf{wp}(\mathsf{assume}\ f_1, f_2, \_) \stackrel{\text{def}}{=} f_1 \Rightarrow f_2.$$

Finally, the absurd statement turns the whole weakest precondition into falsity:

$$\mathsf{wp}(\mathsf{absurd}, \_, \_) \stackrel{\text{def}}{=} \mathsf{false}.$$

Labels. When a label is crossed, we simply erase the corresponding label within annotations. Thus we set

$$\mathsf{wp}(L : e, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e, f, \vec{q})[\mathsf{at}\ t\ L \leftarrow t]$$

where the substitution ranges over all occurrences of terms at $t$ $L$ in the weakest precondition.

Exceptions. Let $\vec{q} = E_1 \to f_1, \ldots, E_n \to f_n$ be the set of exceptional postconditions. When considering the statement raise $(E_i\ e)$, we can always assume that $E_i$ belongs to $\vec{q}$ — otherwise we extend $\vec{q}$ with a true postcondition for $E_i$. Then we have

$$\mathsf{wp}(\mathsf{raise}\ (E_i\ e), \_, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e, f_i, \vec{q}).$$

For a try with construct, we can assume that handled exceptions are $E_1, \ldots, E_m$, with $m \leq n$, without loss of generality. Then we have

$$\begin{array}{l}\mathsf{wp}(\mathsf{try}\ e\ \mathsf{with}\ E_1\ x_1 \to e_1, \ldots, E_m\ x_m \to e_m, f, \vec{q}) \stackrel{\text{def}}{=} \\ \quad \mathsf{wp}(e, f, (E_1 \to \mathsf{wp}(e_1, f_1, \vec{q}), \ldots, E_m \to \mathsf{wp}(e_m, f_m, \vec{q}), \vec{q}_{m+1..n})).\end{array}$$

A New Quantification. Before we go any further, we need to introduce a new operation. In traditional Hoare logic, anytime we need to generalize the weakest precondition w.r.t. the state, we simply quantify over all the corresponding program variables. In our case, however, it is a slightly more complex operation. The reason is that we allow *sub-parts* of data structures to be mutated. Therefore we need to introduce new values for these sub-parts and then to reconstruct all the values they belong to, as we informally explained earlier on page 26.

Let $f$ be a formula (typically the weakest precondition computed so far) and $\vec{r} = r_1, \ldots, r_n$ be a set of regions (typically the *writes* effect of some program subexpression). We define the formula $\nabla\vec{r}.\, f$ as follows. Let $v_1, \ldots, v_m$ be the set of free variables of $f$ whose types contain at least one region from $\vec{r}$. Then we set

$$\begin{array}{rcl} \nabla\vec{r}.\, f & \stackrel{\text{def}}{=} & \forall\vec{r}.\ \mathsf{let}\ v_1 = \mathsf{update}\ v_1\ \vec{r}\ \mathsf{in} \\ & & \ldots \\ & & \mathsf{let}\ v_n = \mathsf{update}\ v_n\ \vec{r}\ \mathsf{in} \\ & & f \end{array}$$

The value for each variable $v_i$ is rebuilt using function update. This function decomposes the old value and rebuilds its fields one by one:

$$\begin{array}{rcl} \mathsf{update}\ v\ \vec{r} & \stackrel{\text{def}}{=} & \mathsf{match}\ v\ \mathsf{with} \\ & & C\ y_1 \ldots\ y_m \to C\ (\mathsf{update}_1\ y_1\ \vec{r}) \ldots\ (\mathsf{update}_m\ y_m\ \vec{r}). \end{array}$$

$C$ is the constructor for the type of $v$, which exists since $v$ contains some regions in its type. Function $\mathsf{update}_i$ rebuilds the value $y_i$ for field $i$, as follows:

$$\mathsf{update}_i\ y_i\ \vec{r} \stackrel{\text{def}}{=} \begin{cases} r_j & \text{if } r_j \text{ is the region for field } i, \\ y_i & \text{if } \vec{r} = \emptyset, \\ \mathsf{update}\ y_i\ (\vec{r} \cap \mathit{reg}(y_i)) & \text{otherwise.} \end{cases}$$

Here $\mathit{reg}(y_i)$ stands for the set of regions occurring in the type of $y_i$. Here is an example. Let us assume the declaration of two record types *array* and *state* as follows:

$$\begin{array}{l} \mathsf{type}\ \mathit{array}\ r\ \alpha = \{\mathsf{mutable}\ \mathit{elts} : \mathit{map}\ \mathit{int}\ \alpha\} \\ \mathsf{type}\ \mathit{state}\ r_1\ r_2\ = \{a : \mathit{array}\ r_1\ \mathit{bool};\ b : \mathit{array}\ r_2\ \mathit{real}\} \end{array}$$

Let $s$ be a variable of type $\mathit{state}\ r_1\ r_2$ and $f$ a formula. We assume that $s$ is the only variable from $f$ whose type contains $r_1$. Then

$$\begin{array}{rl} \nabla r_1.\, f = & \forall r_1 : \mathit{map}\ \mathit{int}\ \mathit{bool}. \\ & \mathsf{let}\ s = \mathsf{match}\ s\ \mathsf{with} \\ & \quad C\ y_1\ y_2 \to C\ (\mathsf{match}\ y_1\ \mathsf{with}\ C'\_ \to C'\ r_1)\ b\ \mathsf{in} \\ & f. \end{array}$$

Loops. The rule for the infinite loop is defined as follows:

$$\mathsf{wp}(\mathsf{loop}\ e\ \mathsf{invariant}\ I, \_, \vec{q}) \stackrel{\text{def}}{=} I \wedge \nabla \mathit{writes}(e).\, I \Rightarrow \mathsf{wp}(e, I, \vec{q}).$$

Note that the normal postcondition — second argument of wp — is of no use here since the loop cannot terminate normally. On the contrary, one can exit the loop

using an exception, hence the use of $\vec{q}$. The case of a for loop is rather tricky. That is the main reason why this construct is not defined as syntactic sugar.

$$\begin{array}{lll} \mathsf{wp}(\mathsf{for}\ x = x_1\ \mathsf{to}\ x_2\ \mathsf{do}\ e\ \mathsf{invariant}\ I(x), f, \vec{q}) \stackrel{\text{def}}{=} & & \\ \quad x_1 > x_2 \Rightarrow f & & \\ \wedge \quad x_1 \leq x_2 \Rightarrow & I(x_1) & \\ & \wedge \quad \nabla \mathit{writes}(e). & \forall i. x_1 \leq i \leq x_2 \Rightarrow I(i) \Rightarrow \mathsf{wp}(e, I(i+1), \vec{q}) \\ & & \wedge \quad I(x_2+1) \Rightarrow f \end{array}$$

First, it distinguishes the case where the for loop is not even entered, that is when $x_1 > x_2$. Otherwise, $I$ must hold initially (that is, $I(x_1)$), must be preserved by $e$ (that is, $I(i)$ must imply $I(i+1)$, informally speaking), and must imply $f$ on loop exit. The last two conditions must hold within a fresh state, hence the $\nabla$ operator. Note that the latter condition mentions $I(x_2+1)$, not $I(x_2)$. Indeed, there are $x_2 - x_1 + 1$ different states in a for loop and we chose the convention that $I(x)$ refers to the beginning of the loop body, as in other loops.

Function Call. A function call is interpreted using the non-determinism construct any. The expression any $\{p\}\ \tau\ \epsilon\ \{q\}$ stands for any computation with precondition $p$, result of type $\tau$, effect $\epsilon$, and postcondition $q$. When computing the weakest precondition, we may assume that both postconditions are referring to the same set of exceptions, since we may always extend them using true postconditions. Then we have

$$\begin{array}{l} \mathsf{wp}(\mathsf{any}\ \{f_0\}\ \tau\ \epsilon\ \{f_1, \vec{q_1}\}, f_2, \vec{q_2}) \stackrel{\text{def}}{=} \\ \quad f_0 \wedge \nabla \mathit{writes}(\epsilon).\, (\forall r : \tau.\, f_1 \Rightarrow f_2) \wedge \bigwedge_i (\forall r : \tau_i.\, q_{1,i} \Rightarrow q_{2,i}) \end{array}$$

where $\tau_i$ is the type of the value carried by exception $E_i$. There is only one quantification $\nabla \mathit{writes}(\epsilon)$, for both normal and exceptional postconditions. We do not make a distinction between effects for normal output and effects for exceptional output.

Function Definition. The weakest precondition for an anonymous function only amounts to verifying that the definition is correct:

$$\mathsf{wp}(\mathsf{fun}\ d, \_, \_) \stackrel{\text{def}}{=} \mathsf{correct}(d)$$

Indeed, the result of this expression being a function, it cannot appear in the postcondition. Predicate correct is defined below. On the contrary, recursive functions are necessarily bound into an expression. Thus the weakest precondition contains two parts:

$$\mathsf{wp}(\mathsf{letrec}\ d_1, \ldots, d_n\ \mathsf{in}\ e, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e, f, \vec{q}) \wedge \bigwedge_i \mathsf{correct}(d_i)$$

The predicate correct expresses than a function definition satisfies all its annotations. It is defined using predicate wp:

$$\begin{array}{l}\mathsf{correct}(x_1 : \tau_1, \ldots x_n : \tau_n \ = \{p\}\ e\ \{f, \vec{q}\}) \stackrel{\text{def}}{=} \\ \quad \forall x_1, \ldots, x_n.\, \nabla\, \mathit{writes}(e) \cup \mathit{reads}(e).\, p \Rightarrow \mathsf{wp}(e, f, \vec{q})\end{array}$$

There is a subtle point here: contrary to what we have done so far for loop, for, and any, we also quantify over $\mathit{reads}(e)$. It expresses that such a function may be used in states that are possibly different from the one where it is declared.

Top-level Declarations. In Why3, top-level declarations are limited to declarations and function definitions. A declaration induces no verification condition. A function definition is either let $d$ or letrec $d_1, \ldots, d_n$ and is translated into a verification condition using predicate correct, as we did above for local definitions. Finally, it is turned into a closed formula using universal quantification over all program variables.

# 3

# Case Studies in Program Verification

Over the past years, I have used the Why tool to verify several dozens of programs. All these proofs can be obtained from Why3's on-line "gallery of verified program", at `why3.lri.fr`. There are also part of Why3's source distribution. This includes, though not exhaustively:

- historical examples, such as proofs from Turing's, Floyd's, and Hoare's papers [101, 49, 53] and McCarthy's 91 function [71];
- examples from books: binary search from Bentley's *Programming Pearls* [10], Knuth's prime numbers algorithm from *The Art of Computer Programming* [60], all program proofs from *Software Foundations* [86];
- challenges in program verification: all problems from VSTTE'10 competition [91], some problems from VACID-0 [66];
- several algorithms: Bresenham line drawing, Knuth-Morris-Pratt string searching, Floyd's "tortoise and hare", Levenshtein distance, Dijkstra's Dutch national flag, Dijkstra's shortest path;
- several sorting algorithms: mergesort, selection sort, insertion sort, and quicksort;
- mathematically oriented programs: Fibonacci numbers (in linear and logarithmic time), fast exponentiation (recursive and iterative), greatest common divisor (with Bezout coefficients), $N$-queens puzzle (counting the number of solutions [42]);
- data structures: sparse arrays, same fringe problem;

This chapter presents a selection of these proofs. It intends to strengthen the discussion of the previous chapters. Additionally, it can serve as a companion to Why3's manual, which already contains a detailed description of the solutions for the five VSTTE'10 verification problems, and to some of my publications [39, 45].

## 3.1 Termination

We focus here on some program verification which involve tricky termination proofs.

### McCarthy's 91 Function

This famous function was precisely introduced as a challenge for termination proofs. It is defined over integers by

$$f(n) = \begin{cases} n - 10 & \text{if } n > 100, \\ f(f(n + 11)) & \text{otherwise.} \end{cases}$$

We can show that it returns 91 as soon as $n \leq 101$, hence its name. In Why3, we define it recursively, as above. To prove it terminates, we have to provide a variant. Here, it is as simple as

$$101 - n$$

but proving that it decreases for each function call is tricky. Indeed, it actually requires us to also prove a postcondition which states how the return value relates to $n$. Thus we prove both termination and behavior at the same time.

```
let rec f91 (n:int) : int variant { 101-n } =
  { }
  if n ≤ 100 then
    f91 (f91 (n + 11))
  else
    n - 10
  { (n ≤ 100 and result = 91) or (n ≥ 101 and result = n - 10) }
```

The resulting weakest precondition is discharged in no time by any SMT solver — which means, in our case, either Alt-Ergo, CVC3, or Z3.

Let us now consider a non-recursive implementation of McCarthy's 91 function [1]. We do not need to make a full stack explicit. We can simply remember how many calls to $f$ are still to be performed. Let $e$ be that number, which is initially set to 1. If $n > 100$ we simply subtract 10 from $n$ and decrement $e$ by 1, since we have just performed one call to $f$. Otherwise, we add 11 to $n$ and increment $e$ by 1, which precisely mean two more calls to $f$ since we just did one. The pseudocode reads as follows:

1. The following code was suggested by Judicaël Courant.

```
e ← 1
while e > 0 do
  if n > 100 then
    n ← n − 10
    e ← e − 1
  else
    n ← n + 11
    e ← e + 1
return n
```

Proving the termination of this loop is at the same time simpler and more difficult than proving the termination of the recursive function. It is simpler because we do not need to say anything about the behavior of the code. But it is also more difficult since the variant is now a pair, namely

$$(101 - n + 10e,\ e).$$

It is ordered lexicographically, where the first and second components are both compared using the usual order relation over natural integers. This order relation over pairs of integers is defined in Why3's standard library and we import it:

```
use import int.Lex2
```

We could stop here. But exactly as the recursive implementation above was a challenge for termination, this non-recursive implementation looks appealing for a proof of correctness. It is not completely trivial why it indeed computes $f(n)$. The loop invariant is

$$f^e(n) = f(n_0)$$

where $n_0$ stands for the initial value of $n$. Therefore, when we exit with $e = 0$, we have $n = f(n_0)$ and we return it. To express this invariant, we first introduce the logic function $f$.

```
function f (x: int) : int = if x ≥ 101 then x-10 else 91
```

It is not recursive but instead captures the extensional meaning of McCarthy's 91 function. Then we define $f^k(x)$. Such a logical function cannot be defined recursively over $k$, since it is not structural (see page 18). So we axiomatize it.

```
function iter int int : int
axiom iter_0: ∀ x: int. iter 0 x = x
axiom iter_s: ∀ k x: int. 0 < k → iter k x = iter (k-1) (f x)
```

More generally, the standard library contains a theory `int.Iter` that axiomatizes the iteration of any function. So we can replace the three declarations above with a theory cloning:

```
use import module ref.Ref
use import int.Lex2

function f (x: int) : int = if x ≥ 101 then x-10 else 91

clone import int.Iter with type t = int, function f = f

let f91_nonrec (n0: int) =
  { }
  let e = ref 1 in
  let n = ref n0 in
  while !e > 0 do
    invariant { e ≥ 0 and iter e n = f n0 }
    variant   { (101 - n + 10 * e, e) } with lex
    if !n > 100 then begin
      n := !n - 10;
      e := !e - 1
    end else begin
      n := !n + 11;
      e := !e + 1
    end
  done;
  !n
  { result = f n0 }
```

Figure 3.1: Non-recursive implementation of McCarthy's 91 function.

```
clone import int.Iter with type t = int, function f = f
```

The code and its specification are given in Fig. 3.1. As for the recursive version, the weakest precondition is easily discharged by all SMT solvers.

### Tortoise and Hare Algorithm

The tortoise and hare algorithm is a cycle detection algorithm. It appears in volume 2 of *The Art of Computer Programming* [61, ex. 6 p. 7] and is credited to Robert Floyd. Let $f$ be a function over a type $T$ and $x_0$ a value of type $T$. We can define an infinite sequence with $x_{n+1} = f(x_n)$. Clearly, the sequence $(x_n)$ has finitely many distinct values if and only if it becomes cyclic at some point. The purpose of Floyd's algorithm is precisely to detect such a cycle. It is extraordinarily

```
type t
function f t : t

function x0: t
clone import int.Iter with type t = t, function f = f
function x (i: int): t = iter i x0

function mu     : int
function lambda : int

axiom mu_range: 0 ≤ mu
axiom lambda_range: 1 ≤ lambda
axiom distinct:
  ∀ i j: int. 0 ≤ i < mu+lambda → 0 ≤ j < mu+lambda →
  i ≠ j → x i ≠ x j
axiom cycle: ∀ n: int. mu ≤ n → x (n + lambda) = x n
```

Figure 3.2: Tortoise and Hare Algorithm: Assumptions.

simple:

$$
\begin{array}{l}
T, H \leftarrow f(x_0), f(f(x_0)) \\
\mathsf{while}\ T \neq H\ \mathsf{do} \\
\quad T, H \leftarrow f(T), f(f(H))
\end{array}
$$

That is, tortoise $T$ advances at speed 1, while hare $H$ advances at speed 2, and we stop whenever they are equal. If so, we have detected a cycle. Otherwise, the algorithm loops forever and there is no cycle. The beauty of this algorithm is that it runs in constant space. Besides, it has several meaningful applications. For instance, it can be used to check whether a singly linked list is cyclic — is that case, it will even stop in both cases, since either it detects a cycle or reaches the `null` pointer.

Let us assume that there is a cycle and let us prove that Floyd's algorithm terminates. We start with a bunch of assumptions, gathered in Fig. 3.2. We introduce function $f$ and the infinite sequence $(x_n)$, reusing library `int.Iter` for that purpose. The existence of a cycle means that there exist smallest integers $\mu \geq 0$ and $\lambda \geq 1$ such that $x_\mu = x_{\mu+\lambda}$. It is convenient to introduce $\mu$ and $\lambda$ as constants — this is nothing more than skolemization. The properties of $\mu$ and $\lambda$ are decomposed into four axioms. Instead of the characterization above, we adopt Knuth's one, which is equivalent: values $x_0, x_1, \ldots, x_{\mu+\lambda-1}$ are all distinct, and $x_{n+\lambda} = x_n$ for $n \geq \mu$.

The challenge is obviously to prove the termination, *i.e.*, to figure out a variant for the `while` loop. Actually, we are even going to prove that it runs in time $O(\mu+\lambda)$. The code is given in Fig. 3.3. The obvious part of the loop invariant states that

the tortoise is some $x_t$ while the hare is $x_{2t}$. Moreover, we have checked so far that $x_i \neq x_{2i}$ for $1 \leq i < t$. Finally, we add $t \leq \mu + \lambda$ to the invariant, which bounds the running time.

The termination argument is the following: once it is inside the cycle, the tortoise cannot be overtaken by the hare. Thus the distance between the hare and the tortoise decreases. We have to turn that into a variant, which requires some care. First, we introduce `dist` $i$ $j$, the distance between $x_i$ and $x_j$. Since this it is a partial function, we only define it for $\mu \leq i, j$. Second, we define a custom order relation, `rel`. It compares two "tortoises" and requires them to be consecutive values $x_i$ and $x_{i+1}$ with $i \leq \mu + \lambda$. Additionally, it requires the distance between $x_{2i+2}$ and $x_{i+1}$ to be smaller than the distance between $x_{2i}$ and $x_i$, whenever the two tortoises are within the cycle, that is $i \geq \mu$. Then the variant is simply the tortoise, tortoises being compared with `rel`.

The proof decomposes into four goals: one lemma to prove that $x_{n+k\lambda} = x_n$ whenever $n \geq \mu$, and three verification conditions (the loop invariant is initialized, the loop invariant is preserved, and the variant decreases). Apart from the loop invariant initialization, which is discharged automatically, the remaining goals are discharged interactively using Coq. Slightly more than 80 lines of tactics are needed.

It is worth pointing out that, beside termination, we have also proved that there is no more than $\mu + \lambda + 1$ loop steps.

## 3.2 Mathematics

### Fibonacci Numbers

The sequence of Fibonacci numbers is the well-known integer sequence defined as follows:

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \quad \text{for } n \geq 2. \end{cases}$$

In Why3, we cannot define a logical function recursively over integers. Thus it is axiomatized.

```
theory Fibonacci "Fibonacci numbers"
  use export int.Int
  function fib int : int
  axiom fib0: fib 0 = 0
  axiom fib1: fib 1 = 1
  axiom fibn: ∀ n:int. n ≥ 2 → fib n = fib (n-1) + fib (n-2)
end
```

```
(* the minimal distance between x i and x j *)
function dist int int : int

(* it is defined as soon as i,j >= mu *)
axiom dist_def: ∀ i j: int. mu ≤ i → mu ≤ j →
  dist i j ≥ 0 ∧
  x (i + dist i j) = x j ∧
  ∀ k: int. 0 ≤ k → x (i + k) = x j → dist i j ≤ k

predicate rel (t2 t1: t) =
  ∃ i: int. t1 = x i ∧ t2 = x (i+1) ∧ 1 ≤ i ≤ mu + lambda ∧
                (i ≥ mu → dist (2*i+2) (i+1) < dist (2*i) i)

let tortoise_hare () =
  let tortoise = ref (f x0) in
  let hare = ref (f (f x0)) in
  while !tortoise ≠ !hare do
    invariant {
      ∃ t: int. 1 ≤ t ≤ mu+lambda ∧
      !tortoise = x t ∧ !hare = x (2*t) ∧
      ∀ i: int. 1 ≤ i < t → x i ≠ x (2*i) }
    variant { !tortoise } with rel
    tortoise := f !tortoise;
    hare     := f (f !hare)
  done
```

Figure 3.3: Tortoise and Hare Algorithm: Code.

Let us consider now the computation of $F_n$. A naive computation of $F_n$ following the recursive scheme above would have time complexity $O(\phi^n)$ where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. A better solution is obviously to compute it iteratively, storing two consecutive values $F_k$ and $F_{k+1}$ in two variables $a$ and $b$ and repeating the assignment

$$a, b \leftarrow b, a + b$$

which leads to a $O(n)$ algorithm. There is actually an even better way to compute $F_n$. It is based on the identity

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \tag{3.1}$$

which holds for all $n \geq 0$ if we set $F_{-1} = 1$. It is easily proved by induction over $n$. Using a fast exponentiation algorithm, it is thus possible to compute $F_n$ in time $O(\log n)$. Let us prove the correctness of a program computing $F_n$ along this idea.

We do not need to compute the four coefficients of the matrix. Computing only $F_n$ and $F_{n-1}$ is enough since we have $F_{n+1} = F_n + F_{n-1}$. Thus we define a recursive function `logfib` which takes $n$ as argument and returns the pair $(F_{n-1}, F_n)$. The pseudocode reads as follows:

```
logfib n =
  if n = 0 then
    (1, 0)
  else
    let (a, b) = logfib (n/2) in
    if n mod 2 = 0 then
      (a^2 + b^2, b(2a + b))
    else
      (b(2a + b), (a + b)^2 + b^2)
```

We could specify this code with a postcondition which simply states that it returns $(F_{n-1}, F_n)$. But proving it correct would require proving identities relating $F_{2k}$ and $F_{2k-1}$ to $F_{k-1}$ and $F_k$. Instead we keep to equation (3.1), which only requires a trivial induction, and thus we choose the specification

$$\mathsf{let}\ (a,b) = \texttt{logfib}\ n\ \mathsf{in}\ \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} a+b & b \\ b & a \end{pmatrix}.$$

This is a natural cut, which makes the proof easier. We introduce a custom theory for $2 \times 2$ integer matrices in Fig. 3.4. It uses a record type for matrices. It only contains the identity matrix and two operations: multiplication and exponentiation. The latter is obtained from a generic theory `Exponentiation`. This theory includes the lemma $x^{n+m} = x^n \cdot x^m$, from which we can deduce the patterns for fast exponentiation, namely $x^{2n} = x^n \cdot x^n$ and $x^{2n+1} = x^n \cdot x^n \cdot x$.

Fig. 3.5 gives the annotated code for `logfib` and the subsequent `fibo` function which returns $F_n$ as the second component of `logfib` $n$. In the middle, we insert the lemma corresponding to equation (3.1). It is limited to the first column of the matrix. It is indeed enough for the induction to be carried out, since

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{n+1} & ? \\ F_n & ? \end{pmatrix} = \begin{pmatrix} F_{n+2} & ? \\ F_{n+1} & ? \end{pmatrix}.$$

Therefore we avoid specifying $F_{-1} = 1$.

Coq is needed to prove the postcondition of `logfib` (in the non-trivial case), as well as the lemma `fib_m`. The remaining goals are discharged automatically.

```
theory Mat22 "2x2 integer matrices"
  use import int.Int
  type t = {| a11: int; a12: int; a21: int; a22: int |}

  function id : t = {| a11 = 1; a12 = 0; a21 = 0; a22 = 1 |}

  function mult (x: t) (y: t) : t =
    {| a11 = x.a11 * y.a11 + x.a12 * y.a21;
       a12 = x.a11 * y.a12 + x.a12 * y.a22;
       a21 = x.a21 * y.a11 + x.a22 * y.a21;
       a22 = x.a21 * y.a12 + x.a22 * y.a22; |}

  clone export int.Exponentiation
    with type t = t, function one = id, function (*) = mult
end
```

Figure 3.4: A theory for $2 \times 2$ integer matrices.

### Knuth's Program for Prime Numbers

Knuth's program for prime numbers [60, p. 147] was discussed in the introduction (see page 5) as an example of a program whose safety proof is difficult. Knuth's code is given in left of Fig. 3.6. It fills the array $p$ with the first 500 primes. Note that $p$ is indexed from 1, not 0. Variables are used as follows: $j$ counts the number of primes found so far; $n$ stands for the candidate for next prime; $k$ scans $p$ to use previously computer primes.

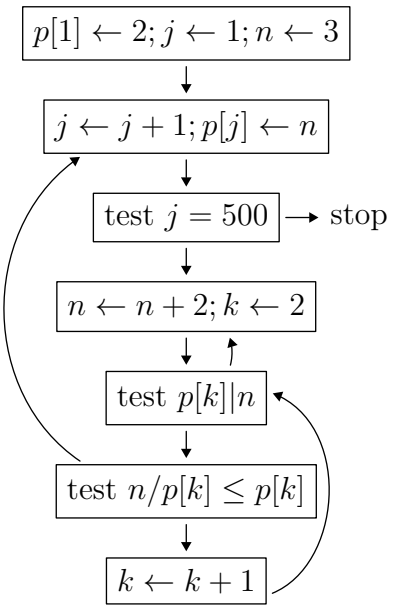
We make a few changes with respect to Knuth's code. First, the array is indexed from 0. Second, we compute the first $m$ prime numbers, for some arbitrary $m \geq 2$. Last, we make a slight change in the code flowchart. Knuth's flowchart is well suited for assembly language, but not for a structured language. Thus we turn the code into a `for` loop on $j$ and a local recursive function `test`. The resulting pseudocode is given in right of Fig. 3.6.

The notions of divisibility (predicate `divides`) and of prime number (predicate `prime`) are imported from Why3's standard library. Fig. 3.7 contains additional material for the specification. Predicate `first_primes` $p\ u$ states that $p[0], \ldots, p[u-1]$ contains the first $u$ prime numbers. It makes use of predicate `no_primes_in` $l\ u$, which states that there is no prime number between $l$ and $u$; it will be conveniently reused in the code's annotation. Bertrand's postulate is introduced as an axiom — we do not intend to prove it.

The annotated code is given in Fig. 3.8. Most of the specification is obvious. The termination of the recursive function `test` is ensured using a pair of integers

```
module FibonacciLogarithmic
  use import Fibonacci
  use import int.EuclideanDivision
  use import Mat22

  function m1110 : t = {| a11 = 1; a12 = 1;
                          a21 = 1; a22 = 0 |}

  let rec logfib n variant { n } =
    { n ≥ 0 }
    if n = 0 then
      (1, 0)
    else begin
      let a, b = logfib (div n 2) in
      let c = a + b in
      if mod n 2 = 0 then
        (a*a + b*b, b*(a + c))
      else
        (b*(a + c), c*c + b*b)
    end
    { let a, b = result in
      power m1110 n = {|a11=a+b; a12=b; a21=b; a22=a|} }

  lemma fib_m :
    ∀ n: int. n ≥ 0 →
    let p = power m1110 n in fib (n+1) = p.a11 and fib n = p.a21

  let fibo n =
    { n ≥ 0 }
    let _, b = logfib n in b
    { result = fib n }
end
```

Figure 3.5: A program computing $F_n$ in time $O(\log n)$.




$p[0] \leftarrow 2; p[1] \leftarrow 3; n \leftarrow 5$
for $j = 2$ to $m - 1$ do
  let rec `test` $k =$
    if $n \mod p[k] = 0$ then
      $n \leftarrow n + 2$; `test` 1
    else if $n/p[k] > p[k]$ then
      `test` $(k + 1)$
  in
  `test` 1; $p[j] \leftarrow n; n \leftarrow n + 2$

Figure 3.6: Knuth's program for prime numbers: original assembly code (left) and equivalent structured code (right).

```
predicate no_prime_in (l u: int) =
  ∀ x: int. l < x < u → not (prime x)

(* p[0]...p[u-1] are the first u prime numbers *)
predicate first_primes (p: array int) (u: int) =
  p[0] = 2 ∧
  (* sorted *)
  (∀ i j: int. 0 ≤ i < j < u → p[i] < p[j]) ∧
  (* only primes *)
  (∀ i: int. 0 ≤ i < u → prime p[i]) ∧
  (* all primes *)
  (∀ i: int. 0 ≤ i < u-1 → no_prime_in p[i] p[i+1])

axiom Bertrand_postulate:
  ∀ p: int. prime p → not (no_prime_in p (2*p))
```

Figure 3.7: Knuth's program for prime numbers: specification.

```
let prime_numbers (m: int) =
  { m ≥ 2 }
  let p = make m 0 in
  p[0] ← 2;
  p[1] ← 3;
  let n = ref 5 in (* candidate for next prime *)
  for j = 2 to m - 1 do
    invariant { first_primes p j ∧
      p[j-1] < !n < 2*p[j-1] ∧ odd !n ∧ no_prime_in p[j-1] !n }
    let rec test (k: int) variant { (2*p[j-1] - !n, j - k) } with lex =
      { 1 ≤ k < j ∧ first_primes p j ∧
        p[j-1] < !n < 2*p[j-1] ∧ odd !n ∧ no_prime_in p[j-1] !n ∧
        ∀ i: int. 0 ≤ i < k → not (divides p[i] !n) }
      if mod !n p[k] = 0 then begin
        assert { not (prime !n) }; n += 2; test 1
      end else if div !n p[k] > p[k] then
        test (k + 1)
      else
        assert { prime !n }
      { p[j-1] < !n ∧ prime !n ∧ no_prime_in p[j-1] !n }
    in
    test 1;
    p[j] ← !n;
    n += 2
  done;
  p
  { first_primes result m }
```

Figure 3.8: Knuth's program for prime numbers: code.

ordered lexicographically. Indeed, either $n$ is increased and $k$ is reset to 1, or $k$ is increased and $n$ is not modified. A key annotation to establish the validity of this variant is

$$\texttt{p[j-1]} < \texttt{!n} < \texttt{2*p[j-1]}.$$

That is where Bertrand's postulate is needed: when $n$ is increased, the right inequality still holds, since it would otherwise contradict Bertrand's postulate. Thus we can use `2*p[j-1] - !n` as first component in the variant.

Five goals require interactive proof using Coq (one lemma and four verification conditions). The remaining goals (30 verification conditions) are discharged automatically.

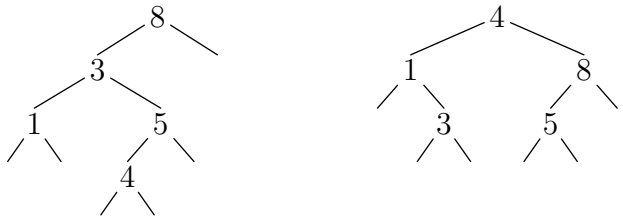
## 3.3 Data Structures

### Same Fringe

"Same fringe" is a famous problem among functional programmers. There is no reference for this problem; it's simply part of folklore. The problem can be stated as follows:

> Given two binary trees,
> do they contain the same elements when traversed in order?

Here we assume binary trees with elements stored at inner nodes. (The problem would be equally interesting with elements stored at leaves.) Here is an example of two trees with different shapes but the same list of elements, namely 1,3,4,5,8.




The nature of the elements is of no importance here. We make no assumption about it [2], using an uninterpreted type `elt` for elements:

```
type elt
type tree =
  | Empty
  | Node tree elt tree
```

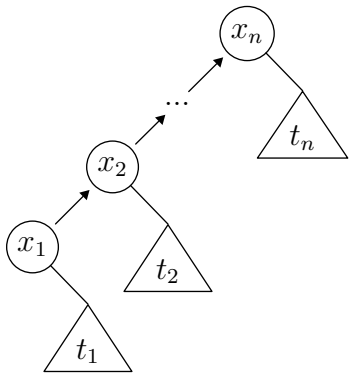
On way to *specify* the problem is to turn the two trees into lists and then compare them. Turning a tree into a list is defined recursively over the structure of the tree, with the help of list concatenation:

```
function elements (t : tree) : list elt = match t with
  | Empty → Nil
  | Node l x r → elements l ++ Cons x (elements r)
end
```

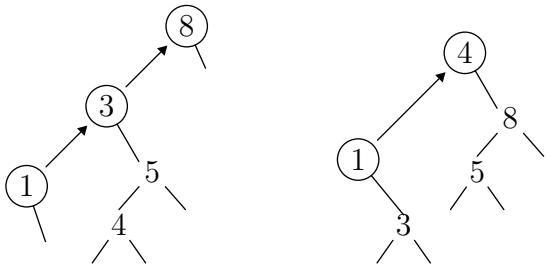
*Solving* the same fringe problem is another matter. If we naively build the two lists and then compare them, it is likely to be inefficient, from both space and time point of view. Ideally, we would like to come up with a $O(\min(n, m))$ solution, where $n$ and $m$ are the sizes of the two trees. Though it can be done, we are slightly less

2. Though, it can be pointed out that when elements are equipped with a total order and sorted from left to right, such trees are binary search trees. Thus the same fringe problem is that of deciding whether two binary search trees contain the same elements, a problem of genuine interest.

ambitious here and we propose a $O(\max(n, m))$ solution. The idea is to consider the leftmost branch of each tree, as a list ordered from bottom up (its head is the leftmost element). Each element on this branch comes with its associated right subtree, which may be empty. This can be depicted as follows:




We call such a list an enumerator. Building it is easy: we simply descend the left branch until we reach the empty tree. On the example above, the two enumerators are the following:




The purpose of enumerators is to propose an easy and lazy comparison of the trees: if their heads differ, we are done; otherwise, we remove it and prepend the enumerator for its right subtree. Enumerators correspond to the following data type:

```
type enum =
  | Done
  | Next elt tree enum
```

Such a data structure is a degenerated form of Huet's zipper [55]. (There is no way to descend to the right.)

Fig. 3.9 gives the full specification and annotated code for the same fringe problem. This includes a function `enum_elements` to turn an enumerator into a list, for the purpose of specification, and three functions to solve the problem: `enum` to turn a tree into an enumerator; `eq_enum` to compare enumerators; and finally `same_fringe`

```
type elt
type tree =
  | Empty
  | Node tree elt tree
function elements (t : tree) : list elt = match t with
  | Empty → Nil
  | Node l x r → elements l ++ Cons x (elements r)
end
type enum =
  | Done
  | Next elt tree enum
function enum_elements (e : enum) : list elt = match e with
  | Done → Nil
  | Next x r e → Cons x (elements r ++ enum_elements e)
end
let rec enum t e variant { length (elements t) } =
  { }
  match t with
  | Empty → e
  | Node l x r → enum l (Next x r e)
  end
  { enum_elements result = elements t ++ enum_elements e }
let rec eq_enum e1 e2 variant { length (enum_elements e1) } =
  { }
  match e1, e2 with
  | Done, Done →
      True
  | Next x1 r1 e1, Next x2 r2 e2 →
      x1 = x2 && eq_enum (enum r1 e1) (enum r2 e2)
  | _ →
      False
  end
  { result=True ↔ enum_elements e1 = enum_elements e2 }
let same_fringe t1 t2 =
  { }
  eq_enum (enum t1 Done) (enum t2 Done)
  { result=True ↔ elements t1 = elements t2 }
```

Figure 3.9: Solution to the same fringe problem.

```
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1;
  while (l <= u) {
    int m = (l + u) / 2;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m - 1;
    else
      return m;
  }
  return -1;
}
```

Figure 3.10: Binary search within a sorted array (C code).

to solve the problem. Thanks to functions `elements` and `enum_elements`, the specification is obvious. Note the use of list lengths to ensure termination.

The very nice thing about this proof is that it is entirely automated: Alt-Ergo succeeds in discharging all verification conditions, in no time. Though it looks like such a proof requires induction, it is here performed implicitly, hidden within the verification of both recursive functions `enum` and `eq_enum`. This is induction without induction, somehow.

## 3.4 Using Why3 as an Intermediate Language

As explained in Section 2.1, Why has been used for years at ProVal as an intermediate language to verify C and Java programs. This section sketches the main ideas of this process.

Following Jon Bentley, we use the *challenge of binary search* as an example [10]. Let us consider a C function, `binary_search`, which looks for a given integer value within a sorted array. Its code is given in Fig. 3.10. The three arguments are `t`, the array in which the search is performed, `n`, the size of this array, and `v`, the value which is searched for.

This C program contains two elements which are not part of the Why3 language:

1. the pointer type `int*` and the associated operation `t[m]`;
2. the presence of a `return` inside a `while` loop.

Since the code is actually not performing any pointer arithmetic, one could model the type `int*` directly as an array. However, we choose to keep to a low-level model

```
type pointer
type memory
function get memory pointer int : int
val mem : memory ref
```

Figure 3.11: A minimal memory model.

of pointers, with more complex programs in mind. Thus we introduce a Why3 type for C pointers of type `int*`:

```
type pointer
```

We introduce another type, `memory`, for the state of the memory:

```
type memory
```

Then we can introduce a `get` operation to access memory with a given pointer:

```
function get memory pointer int : int
```

If `m` is some given memory state, `p` a pointer and `i` an integer, then `get m p i` accesses `m` at address `p+i`. We assume here that the memory is only word-addressed, *i.e.* we only have pointers of type `int*`. A global reference `mem` contains the current state of memory:

```
val mem : ref memory
```

It completes our first *memory model*, which is summarized in Fig. 3.11.

Translating function `binary_search` in this memory model is straightforward: `t` becomes an argument of type `pointer` and expression `t[m]` is translated into `get !mem t m`. Of course, the memory model could be simplified even further. One could simply declare a function `function t int : int`, suppress argument `t` and translate `t[m]` into `t m`. Though, as explained above, we want to keep to a realistic memory model. If the memory was to be mutated (which is not the case in this program), we would introduce another operation

```
function set memory pointer int int : memory
```

and we would translate a C assignment into a mutation of reference `mem`.

The second obstacle is the presence of a `return` statement inside the `while` loop. To translate it, we introduce a Why3 exception carrying an integer value:

```
exception Return int
```

Then we translate `return m` into `raise (Return_int m)`. The whole while loop is surrounded with a `try with`, as follows:

```
let binary_search (t:pointer) (n:int) (v:int) =
  try
     ...
  with Return r →
    r
  end
```

The remaining of the translation is straightforward. Local variables `l`, `u` and `m` are translated into Why3 local references, since the C operator `&` is not used to obtain their addresses. For variable `m`, we can even use a non-mutable variable since it is not mutated in the C code.

We still need to give the function its specification. The precondition requires the array to be sorted.

```
{ ∀ k1 k2:int.
     0 ≤ k1 ≤ k2 ≤ n-1 → get mem t k1 ≤ get mem t k2 }
```

The loop invariant maintains inequalities over `l` and `u` and the crucial property that `v`, if it occurs in `t[0..n-1]`, then necessarily occurs in `t[l..u]`.

```
{ invariant 0 ≤ l and u ≤ n-1 and
    ∀ k:int. 0 ≤ k < n → get(mem,t,k) = v → l ≤ k ≤ u }
```

To prove termination, we simply add the variant `u-l`. Finally, the postcondition states soundness and completeness properties, that is, either the result is non-negative and `v` occurs in `t` at this position, or the result is `-1` and `v` does not occur in `t`.

```
{ (result ≥ 0 and get mem t result = v) or
  (result = -1 and ∀ k:int. 0≤k<n → get mem t k ≠ v) }
```

Function `binary_search` is given Fig. 3.12. It is proved correct automatically.

Let us make this proof slightly more realistic by adding *array bound checking*. To do this, we first introduce the notion of block size in our memory model:

```
function block_size memory pointer : int
```

In this new model, each pointer points to the beginning of a block, whose size is given by function `block_size`. Now, instead of accessing memory with function `get` inside the program, we will use instead a function with a precondition:

```
val get_ : p:pointer → ofs:int →
  { 0 ≤ ofs < block_size mem p }
  int reads mem
  { result = get mem p ofs }
```

In the Why3 code for function `binary_search`, we replace the two occurrences of `get !mem t m` with `get_ t m`, getting a a new verification condition for each. Both

```
exception Return int

let binary_search (t : pointer) (n : int) (v : int) =
  { ∀ k1 k2:int.
      0 ≤ k1 ≤ k2 ≤ n-1 → get mem t k1 ≤ get mem t k2 }
  try
    let l = ref 0 in
    let u = ref (n-1) in
    while !l ≤ !u do
      invariant { 0 ≤ l and u ≤ n-1 and
        ∀ k:int. 0 ≤ k < n → get mem t k = v → l ≤ k ≤ u }
      variant { u-l }
      let m = div (!l + !u) 2 in
      if get !mem t m < v then l := m + 1
      else if get !mem t m > v then u := m - 1
      else raise (Return m)
    done;
    raise (Return (-1))
  with Return r →
    r
  end
  { (result ≥ 0 and get mem t result = v) or
    (result = -1 and ∀ k:int. 0≤k<n → get mem t k ≠ v) }
```

Figure 3.12: Function `binary_search` translated to Why3.

amount to showing

$$0 \leq \texttt{(l + u) / 2} < \texttt{block\_size mem t}.$$

To be able to discharge them, we need to add an extra precondition to function `binary_search`, which states that `n` is no larger than the block size:

```
let binary_search (t : pointer) (n : int) (v : int) =
  { n ≤ block_size mem t and ... }
  ...
```

All VCs are discharged automatically.

Finally, let us make this proof even more realistic by showing the *absence of arithmetic overflow* in this C code. For simplicity, we consider here that C type `int` stands for 32 bit integers. We introduce a new Why3 type to model this C type.

```
type int32
```

The key idea for reasoning about fixed size integers is the following:

> Specifications, and thus verification conditions, should only use mathematical integers, that is Why3's type `int`.

There are two reasons for this. First, we do not want to loose the ATPs capabilities regarding arithmetic, and this is the only arithmetic they are aware of. Second, we cannot introduce partial function symbols in the logic, such as an addition over 32 bit integers. If an addition is to appear within a specification, it should be the usual addition over mathematical integers. Thus we introduce a function which maps values of types `int32` to the corresponding value in type `int`.

```
function to_int int32 : int
```

If the value of a C variable `x` is referred to within an annotation, we will write `to_int x` instead. More generally, we do so for any expression of type `int32` occurring within an annotation. For instance, the loop invariant used to be

```
0 ≤ l and u ≤ n-1 and
∀ k:int. 0 ≤ k < n → get mem t k = v → l ≤ k ≤ u
```

and now it is cluttered with `to_int`s:

```
0 ≤ to_int l and to_int u ≤ to_int n - 1 and
∀ k:int. 0 ≤ k < to_int n →
to_int (get mem t k) = to_int v → to_int l ≤ k ≤ to_int u
```

Note that the universal quantification ∀ `k` is still over type `int`. To axiomatize `int32` and `to_int`, we first introduce an axiom which defines the range of 32 bit integers.

```
axiom int32_domain :
  ∀ x:int32. -2147483648 ≤ to_int x ≤ 2147483647
```

Then we introduce an operation to build a value in type `int32` from a value in type `int`, provided it lies in the appropriate range.

```
val of_int : x:int →
  { -2147483648 ≤ x ≤ 2147483647 }
  int32
  { to_int result = x }
```

To translate a C operation over 32 bit integers, say an addition such as

$$e_1 +_{32} e_2$$

we convert both operands to type `int` with `to_int`, apply the corresponding operation over type `int`, and then convert back to type `int32` with `of_int`:

$$\texttt{of\_int}\ (\texttt{to\_int}\ e_1 + \texttt{to\_int}\ e_2)$$

Of course, we perform this recursively in both $e_1$ and $e_2$, up to variables and constants. For instance, the initialization of `u` used to be

```
let u = ref (n - 1) in
```

and now reads as follows:

```
let u = ref (of_int (to_int n - to_int (of_int 1))) in
```

It is worth pointing out that writing `to_int (of_int 1)` instead of `1` enforces us to prove that constant 1 fits in the range of 32 bit integers. Even if obvious in this case, absence of arithmetic overflows in constants are to be checked as well [3]. It is clear from the two excerpts above that translating both specifications and programs to this new model is quickly intractable in a manual process. Fortunately, this can be easily automated during a translation from C to Why3, since the compiler has access to abstract syntax trees fully decorated with types.

When we try to prove this third model of the C program, we are left with one unproved verification condition. It is related to the C statement

```
int m = (l + u) / 2;
```

and more precisely with the absence of arithmetic overflow in the computation of `l + u`. Proving that it is greater or equal to $-2^{31}$ is easy (we know that both `l` and `u` are non-negative at this point) but we cannot prove that it is smaller or equal to $2^{31} - 1$. And indeed this is not provable: for a large enough array, we could have for instance `l` $=$ `u` $= 2^{30}$ and the addition would overflow. We already discussed this famous bug and its possible fixes in the first chapter. For the fix

```
int m = l + (u - l) / 2;
```

all VCs are easily discharged automatically. Proving the other fix proposed by Joshua Bloch [12], namely `int m = (l + u) >>> 2` in Java, is another matter. Indeed, the addition may still overflow but the shift will restore a zero sign bit, ending up with the expected value. To prove such a fix correct, we can no longer use the model above, since it does not allow overflows. A more elaborated model would be needed, for instance one that is faithful to the machine's 32-bit signed arithmetic.

Realistic Memory Models. Tools such as Krakatoa or the Jessie plug-in of Frama-C are using even more complex memory models, to account for pointer arithmetic, heap-allocated data, dynamic memory allocation, and so on. The basic idea is to model memory according to structure (or objects) fields, following an old idea by Burstall [23]. At ProVal, this was first experimented in Krakatoa and Caduceus, and later refined using a static separation analysis of pointers [54], leading to the intermediate language Jessie [73].

3. Most compilers report arithmetic overflows in constants.

# 4
# Perspectives

My research perspectives are split into general perspectives for deductive program verification (Sec. 4.1) and perspectives specific to the Why3 tool (Sec. 4.2).

## 4.1 Deductive Program Verification

### Verification of Algorithms

Among many others, a legitimate goal for deductive program verification is the following:

> We should be able to pick up any algorithm from a standard textbook and prove it to be correct in time and space no larger than the textbook proof.

Let us clarify what me mean in the statement above. First, writing the code itself in a formal language suitable for program verification should be easy, at least no more difficult than writing pseudo-code or code in some mainstream programming language. It means that high-level programming constructs should be available, as well as suitable libraries. Second, performing the proof should be of a similar difficulty to a pen-and-paper proof. Of course, we assume here that the proof is done for the first time on both sides. Both kinds of proof require to exhibit invariants. Then the proof process itself differs, but the time spent in writing down a pen-and-paper proof should be comparable to the time spent in elaborating a formal proof, provided a reasonable amount of proof automation. The latter means suitable decision procedures, *e.g.* for lists, finite sets, or reflexive transitive closure. With such tools being available, teachers and scientists should be able to ensure the correctness of algorithms presented in lectures and research papers in a reasonable amount of time. As far as I am concerned, I have this perspective in mind. This document has explained what I have done so far in that direction and I definitely intend to make further progress in the future.

### Verification of Static Analysis Tools

Textbook algorithms are legitimate targets for deductive verification but they are not *real* programs. This is somehow frustrating. Among all real programs, relevant candidates for formal verification are static analysis tools: compilers, abstract interpretors, model checkers, slicers, etc. They are often used on critical code, and thus are critical programs themselves. We already mentioned Leroy's verified compiler CompCert, but most static analyzers are still waiting to be proved sound. I think Why3 can be seriously considered to tackle such challenges: indeed, it features algebraic data types (for abstract syntax trees), inductive predicates (to define semantics), relevant libraries (*e.g.* finite sets), and a nice combination of proof automation and interactive proof.

### Bootstrapping Verification Tools

Among static analysis tools are the verification tools themselves. They are obvious candidate for being verified, as we want them to be sound in the first place. There had been already several steps in that direction. For instance, Nipkow formalized the first 100 pages of Winskel's book using Isabelle/HOL [82]. This includes soundness and relative completeness of Hoare logic, the latter using a weakest precondition calculus. Later, Yves Bertot conducted a similar experience in Coq, though introducing new ingredients such as computational reflection and abstract interpretation [11]. Along these lines, the undergoing Ph.D. thesis of Paolo Herms [51] at ProVal aims at certifying the whole chain from C programs to verification conditions used in Frama-C.

However, formally verifying a tool such as Why3 is *a lot* of work and it is not that obvious that we should do it right now. We should probably focus first on improving the capacity of our tools before considering proving them. Who cares about a sound, but useless tool? I remember a discussion with Natarajan Shankar during my postdoc on a related topic: I was (very naively) comparing the relative soundness of Coq and PVS and Shankar was replying that providing new, useful features to PVS was more important than increasing the trust in PVS's own soundness. Today, I definitely agree with him.

## 4.2 Why3

We can already do a lot of program verification with Why3, as demonstrated in the previous chapter. Yet we have a lot of possible improvements in mind. We discuss them here.

Ghost Code. It is sometimes useful to insert additional code into a program for the sole purpose of writing its specification. Such a code is called *ghost code*. Ghost code is similar to regular code: it is parsed, type checked, and turned into verification conditions in the same way. The only constraint is that it does not *interfere* with regular code, in the sense that we can remove ghost code without affecting the behavior of the regular code. This does not prevent ghost code from performing side effects, but it limits those side effects to ghost data. We have already demonstrated the usefulness of ghost code in a paper describing the proof of a two line C program computing the number of solutions to the $N$-queens problem [42].

Somehow, ghost code is reminiscent of program extraction in Coq [78, 70]: sort `Set` is used for programs and sort `Prop` for annotations, and the latter can be safely removed thanks to non-interference ensured by type checking. However, separating regular and ghost code using types is not the only way to go. One can instead tag some variables and functions as ghost. This is the way followed in Dafny [65] for instance.

Program Execution. Currently, there is no way to execute Why3 code. We plan to provide a translator from Why3 to OCaml, so that Why3 programs can be executed interactively in the OCaml interpreter or compiled and linked into standalone OCaml programs. OCaml is the target of choice since it provides almost all WhyML features (exceptions, polymorphism, algebraic data types, records with mutable fields, etc.). Getting executable code from Why3 raises some interesting issues:

- First, there are several possible reasons for a Why3 code not to be executable: it may refer to a program function which is declared but not defined (`val`); equivalently, it may use the non-deterministic program construct (`any`); finally, it may involve some logical term which is not executable. Thus the user must be able to provide some OCaml implementation to fill the gaps. An example is type `int`, which should be mapped to some implementation of arbitrary-precision arithmetic. Such mapping could be done using a configuration file along the lines of prover drivers. But this could also be done by turning the whole code into an OCaml functor whose parameters correspond to the missing implementations.
- Second, since WhyML programs may involve arbitrary logic expressions, we need to be able to translate them to OCaml as well. This is not always possible. An obvious case is when a logic symbol is declared but not defined. Even when a logic symbol is defined, it may not be possible to translate it to OCaml code. For instance, its definition may involve quantifiers or non deterministic inductive predicates, which are not executable in most cases. It may also involve the equality predicate. If it is used on some algebraic data type, we can use OCaml's polymorphic and structural equality. If it is used

on some uninterpreted type, we cannot.
- Finally, one can imagine turning annotations into code as well, to perform *run-time assertion checking*. This implies turning logic expressions into code, which we just discussed. But this also implies saving state values in order to interpret constructs `old` and `at` within annotations. This is a well-known issue in the run-time assertion checking community (see for instance [62]). Even if it is likely to be simpler in the case of Why3, due to the absence of aliases, it remains a nontrivial problem.

Invariants. Currently, there is no easy way to associate an invariant to a given Why3 data type. For instance, we showed earlier how arrays are modeled in Why3 (see page 26), namely

```
type array α model {| length: int; mutable elts: map int α |}
```

but we have no way to state and prove that the length of any array is non-negative. We should be able to do that, however, since `array` is an abstract data type and since the only operations returning arrays (`make`, `append`, etc.) all ensure that the returned array has a non-negative length. Currently, the user has to insert suitable preconditions regarding array lengths, when necessary. Said otherwise, we already have a mechanism for data abstraction so we should be able to give it a suitable mechanism for data invariant.

Encapsulation. Our model of arrays is elegant and genuinely useful. However, arrays belong to the category of data structures that we *do not* implement in Why3; we only give them a signature. On the contrary, there are many data structures that we can implement in Why3, such as hash tables, binary search trees, priority queues, etc. For such data structures, we want to define a signature and an implementation, separately, and then to confront them. For instance, the signature for imperative sets of integers would look like

```
type t model {| mutable elts: set int |}
val create: unit → {} t { result.elts = empty }
val add: x: int → s: t →
         {} unit writes s { s.elts = add x (old s.elts) }
...
```

and a possible implementation using hash tables would look like

```
type t = array (list int)
let create () = Array.make 17 Nil
...
```

As usual in data abstraction, verifying the implementation against the signature will typically require invariants. In the example above, the invariant will relate the

abstract set to the set of all elements appearing in the hash table. Currently, there is no way to do that in Why3: we can define the signature, we can use it in some client code, and, independently, we can implement a data structure, but we cannot verify the implementation against the signature — unless we manually insert the specifications from the signature into the implementation.

Module Cloning. As we showed earlier, theory cloning is a powerful concept to reuse specifications. It is natural to consider doing the same for modules. *Module cloning* would be a way to get a new module by partially instantiating some uninterpreted symbols from another module (types, constants, functions), reusing code, specification, and proof in a single step. Somehow, module cloning is no different from ML functors [68]; it is only easier to use and more flexible. As an example, let us consider priority queues. A signature module can be designed as follows. An abstract type for elements is declared

```
module Q
  type elt
```

and is equipped with a total order

```
  clone export relations.TotalOrder with type t = elt
```

The priority queue itself can be modeled as a list of elements, though it is not of importance here:

```
  type pqueue model {| mutable elts: list elt |}
```

Then operations can be declared, such as `create`, `push`, `pop`, and so on. Once the signature module `Q` is complete, module cloning would allow us to instantiate type `elt`, and possible relation `rel` as well, to get priority queues specialized for this type. For instance, priority queues containing integers could be used as easily as

```
  clone import module Q with type elt = int, predicate rel = (≤)
```

A nice idea by Andrei Paskevich is that data encapsulation, as discussed in the previous paragraph, could be implemented using module cloning: cloning the signature module and instantiating its parameters with functions from the implementation module should generate the expected verification conditions.

Higher-Order Programs. WhyML is already so close to OCaml that it is natural to consider going one step further and tackle higher-order programs. Actually, there is already some limited support for higher-order functions in Why3. Indeed, the type checker already allows function arguments with arbitrary types. Since function types include specifications, one can perfectly write some code like

```
  let f (g: n:int → {} unit writes x { !x = old !x + n }) = ...
```

(assuming some global reference `x`). After all, this is no different from having a global function `g` in the context and then prove function `f`. However, it is already limited when it comes to *applying* function `f` to some particular function: the argument function must have exactly the type of `g`. Thus the following is legal

```
f (fun n:int → {} x := !x + n { !x = old !x + n })
```

but applying `f` to a function with a different specification or effect is not. Obviously, we could relax this a little bit to allow a function with a weaker precondition, a stronger postcondition, or a smaller effect. Yet we would be far from idiomatic higher-order programs. Indeed, higher-order is particularly powerful when combined with polymorphism. A typical example is an *iterator*. Iterators are the idiomatic way to traverse the elements of a collection in functional programming, either to perform some action on each (`iter`), to build a value from them (`fold`), or to rebuild a new collection (`map`). Here is for instance the type of the `iter` function over lists from OCaml standard library:

```
List.iter: (α → unit) → α list → unit
```

Specifying, proving, and using such a function in a verification context is quite a challenge. Indeed, the function argument of type $\alpha \to$ `unit` may have some arbitrary effect and specification, thus we need some kind of effect polymorphism and higher-order logic. A huge step in that direction was made in Johannes Kanig's PhD thesis [56], which extended previous work by Regis-Gianas on purely applicative programs [88] to programs with effects. We still need to find a way to integrate Kanig's work smoothly into Why3. (Incidentally, it is Johannes's work that inspired the use of regions in Why3.)

Obviously, specifying and proving higher-order programs quickly requires higher-order logic, as demonstrated in Kanig's thesis for instance. Since automated theorem provers are limited to first-order logic, higher-order logic must be encoded into first-order logic in some way or another. A simple, yet limited, solution is to spot higher-order *schemes* and to translate all their instantiations into first-order logic, as done by Leino and Monahan [67]. Otherwise a systematic translation can be considered, as done for instance by Meng and Paulon in Isabelle/HOL [75] or by Regis-Gianas and Kanig in the aforementioned work [88, 56].

Interactive Proof. When a verification condition is not discharged by any automated theorem prover, we turn to a proof assistant, typically Coq. There we apply tactics to discharge the goal interactively. This includes high-level reasoning steps, such as applying an induction principle or inverting some inductive predicate definition. Once these are performed, we are left with simpler goals, most of them could be discharged by automated theorem provers. Unfortunately, there is currently no way to use these provers from Coq. We consider implementing a Coq plug-in to do this.

This is not a new idea, as such a technology already exists in Isabelle/HOL — this is the Sledgehammer tool [75] we already mentioned in the first chapter.

However, we consider doing it slightly differently. First, we do not really care about reconstructing a proof in Coq, since we are already assuming the soundness of automated theorem provers. (Of course, checking such proofs when they exist cannot harm, but the point here is that we are perfectly fine with calling provers that do not produce any proof.) Second, we will use Why3 and its OCaml API to implement this Coq plug-in, thus performing a round trip from Why3 to Coq and back. Hopefully, this round trip should not have any impact on the terms and formulas coming from the original Why3 goal. Of course, there is no reason why such a plug-in could not be used for other kinds of Coq proofs, at least when they use the first-order fragment of Coq's logic. Somehow, it would provide a first (modest) step towards a Sledgehammer-like tactic for Coq. This Coq plug-in is already work in progress.

# Bibliography


[1] Jean-Raymond Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.

[2] M. Barbosa, Jean-Christophe Filliâtre, J. Sousa Pinto, and B. Vieira. A Deductive Verification Platform for Cryptographic Software. In *4th International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010)*, volume 33, Pisa, Italy, September 2010. Electronic Communications of the EASST.

[3] Romain Bardou, Jean-Christophe Filliâtre, Johannes Kanig, and Stéphane Lescuyer. Faire bonne figure avec Mlpost. In *Vingtièmes Journées Francophones des Langages Applicatifs*, Saint-Quentin sur Isère, January 2009. INRIA.

[4] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.

[5] Mike Barnett, K. Rustan M. Leino, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. pages 49–69. Springer, 2004.

[6] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[7] Clark Barrett and Cesare Tinelli. CVC3. In Damm and Hermanns [31], pages 298–302.

[8] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.

[9] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[10] J.L. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[11] Yves Bertot. *From Semantics to Computer Science, essays in Honour of Gilles Kahn*, chapter Theorem proving support in programming language semantics, pages 337–361. Cambridge University Press, 2009.

[12] Joshua Bloch. Nearly all binary searches and mergesorts are broken, 2006. `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

[13] François Bobot. *Logique de séparation et vérification déductive*. Thèse de doctorat, Université Paris-Sud, 2011.

[14] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.

[15] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. *The Why3 platform*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, February 2011. `http://why3.lri.fr/`.

[16] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[17] François Bobot and Andrei Paskevich. Expressing Polymorphic Types in a Many-Sorted Language, 2011. Preliminary report. `http://hal.inria.fr/inria-00591414/`.

[18] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Formal Proof of a Wave Equation Resolution Scheme: the Method Error. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of the first Interactive Theorem Proving Conference*, volume 6172 of *LNCS*, pages 147–162, Edinburgh, Scotland, July 2010. Springer. (merge of TPHOLs and ACL2).

[19] Sylvie Boldo and Jean-Christophe Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.

[20] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for Certifying Floating-Point Programs. In *16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning*, volume 5625 of *Lecture Notes in Artificial Intelligence*, pages 59–74, Grand Bend, Canada, July 2009. Springer.

[21] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[22] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[23] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[24] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, September 2011. ACM.

[25] Sylvain Conchon and Jean-Christophe Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

[26] Sylvain Conchon and Jean-Christophe Filliâtre. A Persistent Union-Find Data Structure. In *ACM SIGPLAN Workshop on ML*, pages 37–45, Freiburg, Germany, October 2007. ACM.

[27] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-Persistent Data Structures. In *17th European Symposium on Programming (ESOP'08)*, Budapest, Hungary, April 2008.

[28] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a Generic Graph Library using ML Functors. In Marco T. Morazán, editor, *Trends in Functional Programming Volume 8: Selected Papers of the $8^{th}$ International Symposium on Trends in Functional Programming (TFP'07), New York, USA*, volume 8. Intellect, 2008.

[29] Jean-François Couchot and Stéphane Lescuyer. Handling polymorphism in automated deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.

[30] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May*

*16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009.

[31] Werner Damm and Holger Hermanns, editors. *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, Berlin, Germany, July 2007. Springer.

[32] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. `http://research.microsoft.com/projects/z3/`.

[33] Leonardo de Moura and Bruno Dutertre. Yices: An SMT Solver. `http://yices.csl.sri.com/`.

[34] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[35] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.

[36] Jean-Christophe Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris-Sud, July 1999.

[37] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[38] Jean-Christophe Filliâtre. Backtracking iterators. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

[39] Jean-Christophe Filliâtre. Formal Proof of a Program: Find. *Science of Computer Programming*, 64:332–240, 2006.

[40] Jean-Christophe Filliâtre. Formal Verification of MIX Programs. In *Journées en l'honneur de Donald E. Knuth*, Bordeaux, France, October 2007. `http://knuth07.labri.fr/exposes.php`.

[41] Jean-Christophe Filliâtre. A Functional Implementation of the Garsia–Wachs Algorithm. In *ACM SIGPLAN Workshop on ML*, Victoria, British Columbia, Canada, September 2008. ACM.

[42] Jean-Christophe Filliâtre. Verifying two lines of C with Why3: an exercise in program verification. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, Philadelphia, USA, January 2012.

[43] Jean-Christophe Filliâtre and K. Kalyanasundaram. Functory: A distributed computing library for Objective Caml. In *Trends in Functional Programming*, Madrid, Spain, May 2011.

[44] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *Proceedings of The European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 370–384, Barcelona, Spain, April 2004.

[45] Jean-Christophe Filliâtre and Nicolas Magaud. Certification of Sorting Algorithms in the System Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, Nice, France, 1999.

[46] Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods*, pages 15–29. Springer, 2004.

[47] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Damm and Hermanns [31], pages 173–177.

[48] Jean-Christophe Filliâtre and F. Pottier. Producing All Ideals of a Forest, Functionally. *Journal of Functional Programming*, 13(5):945–956, September 2003.

[49] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[50] The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`.

[51] Paolo Herms, Claude Marché, and Benjamin Monate. A certified multi-prover verification condition generator. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *VSTTE*, Lecture Notes in Computer Science. Springer, 2012.

[52] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[53] C. A. R. Hoare. Proof of a program: Find. *Commun. ACM*, 14:39–45, January 1971.

[54] Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, pages 81–93, Braga, Portugal, March 2007.

[55] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, Septembre 1997.

[56] Johannes Kanig. *Spécification et preuve de programmes d'ordre supérieur*. Thèse de doctorat, Université Paris-Sud, 2010.

[57] Johannes Kanig and Jean-Christophe Filliâtre. Who: A Verifier for Effectful Higher-order Programs. In *ACM SIGPLAN Workshop on ML*, Edinburgh, Scotland, UK, August 2009.

[58] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[59] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, June 2010.

[60] Donald E. Knuth. *The Art of Computer Programming, volume 1 (3rd ed.): Fundamental Algorithms.* Addison Wesley Longman Publishing Co., Inc., 1997.

[61] Donald E. Knuth. *The Art of Computer Programming, volume 2 (3rd ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., 1997.

[62] Piotr Kosiuczenko. An abstract machine for the old value retrieval. In Claude Bolduc, Jules Desharnais, and Béchir Ktari, editors, *MPC*, volume 6120 of *Lecture Notes in Computer Science*, pages 229–247. Springer, 2010.

[63] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.

[64] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, 2005.

[65] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Springer, editor, *LPAR-16*, volume 6355, pages 348–370, 2010.

[66] K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.

[67] Rustan Leino and Rosemary Monahan. Automatic verification of textbook programs that use comprehensions. In *Formal Techniques for Java-like Programs (FTfJP 2007)*, Berlin, Germany, July 2007.

[68] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[69] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[70] Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud, July 2004.

[71] Z. Manna and J. Mccarthy. Properties of programs and partial function logic. In *Machine Intelligence*, volume 5, 1970.

[72] María Manzano. *Extensions of first order logic*. Cambridge University Press, New York, NY, USA, 1996.

[73] Claude Marché. Jessie: an intermediate language for Java and C verification. In *Programming Languages meets Program Verification (PLPV)*, pages 1–2, Freiburg, Germany, 2007. ACM.

[74] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.

[75] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40:35–60, 2008.

[76] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, Hemel Hempstead, 1992.

[77] R. Milner. A theory of type polymorphismn in programming. *Journal of Computer and System Sciences*, 17, 1978.

[78] Christine Paulin Mohring and Benjamin Werner. Synthesis of ml programs in the system coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[79] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *IEEE Annals of the History of Computing*, 6(2):139–143, April 1984.

[80] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *Proceedings of ICFP'08*, 2008.

[81] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[82] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[83] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[84] Sam Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, SRI International, 2001.

[85] Lawrence C. Paulson. Introduction to isabelle. Technical report, University of Cambridge, 1993.

[86] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Distributed electronically, 2011.

[87] Silvio Ranise and David Deharbe. Light-weight theorem proving for debugging and verifying units of code. In *Proc. SEFM'03*, Canberra, Australia, September 2003. IEEE Computer Society Press. `http://www.loria.fr/equipes/cassis/softwares/haRVey/`.

[88] Yann Régis-Gianas and François Pottier. A Hoare logic for call-by-value functional programs. In *Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC'08)*, pages 305–335, 2008.

[89] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[90] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[91] Natarajan Shankar and Peter Mueller. Verified Software: Theories, Tools and Experiments (VSTTE'10). Software Verification Competition, August 2010. `http://www.macs.hw.ac.uk/vstte10/Competition.html`.

[92] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31:1–12, 1984.

[93] Julien Signoles. *Extension de ML avec raffinement : syntaxe, sémantiques et système de types*. Thèse de doctorat, Université Paris-Sud, July 2006.

[94] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[95] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[96] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[97] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(4):245–298, June 1994.

[98] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. `http://coq.inria.fr`.

[99] Laurent Théry. Proving pearl: Knuth's algorithm for prime numbers. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *LNCS*. Springer-Verlag, 2003.

[100] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.

[101] Alan Mathison Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculing Machines*, pages 67–69, Cambridge, 1949. Mathematical Laboratory.

[102] Andrew K. Wright. Simple imperative polymorphism. *LISP and symbolic computation*, 8:343–355, 1995.

# Index


$\nabla$, 32
42, 42
91 function, 4, 38

alias, 30
Alt-Ergo, 11, 14, 16, 17, 38, 52
arithmetic
    overflow, 3, 55
    Presburger, 13
array, 24

Barthe, Gilles, x
Bentley, Jon, 4, 52
Bertot, Yves, x, 60
Bertrand's postulate, 5
binary search, 3, 52
Bobot, François, ix, 15, 16, 20
Boldo, Sylvie, ix

Caduceus, ix, 15, 57
Castéran, Pierre, x
Charguéraud, Arthur, 9
cloning
    modules, 63
    theories, 21
CompCert, xi, 9, 10, 60
Conchon, Sylvain, x
consistance, *see* crème de marrons
Coq, ix, 5, 13, 14, 60, 64, 65
Couchot, Jean-François, 15
Courant, Judicaël, 38
Cousineau, Guy, x
crème de marrons, *see* consistance
CVC Lite, 14
CVC3, 10, 38
cycle detection, 4, 40

Dafny, 6, 9, 61
Danvy, Olivier, xi
di Cosmo, Roberto, xi
Doligez, Damien, x
driver, 22

exponentiation
    fast, 44

Fibonacci, 42
FIND (program), 3
Floyd, Robert, 4, 40
Frama-C, 57, 60
fringe, *see* same fringe

Gappa, x
ghost code, 61

haRVey, 14
Herms, Paolo, 60
higher-order, 63
Hoare, Tony, 3
HOL Light, 14
HOL4, 14
Huet, Gérard, 50
Huet, Gérard, xi

IDE

of Why3, 22
induction, 52
invariant, 15
Isabelle/HOL, 14

Jessie, 15, 57
Jouvelot, Pierre, 30

Kanig, Johannes, 64
Knuth, Donald E., 5, 15, 45
Krakatoa, 14, 15, 57

Leino, K. Rustan M., x, 64
Leroy, Xavier, x, xi, 9
Lescuyer, Stéphane, 15
Letouzey, Pierre, ix
lexicographic order, 18, 39, 48

Magaud, Nicolas, 13
Marché, Claude, ix, 14–16, 22
McCarthy, John, 4, 38
Melquiond, Guillaume, x
memory model, 53
`MIX`, 5, 15
Mizar, 14
model type, 24, 25

*N*-queens, 37

OCaml, x, 24, 61, 63–65
overflow, 3

Paskevich, Andrei, 15, 16, 20, 63
Paulin, Christine, xi, 14
prime number, 45
procrastination, 1–78
programs
of Why3, *see* WhyML
ProVal, ix, x, 13, 15, 52, 57, 60
PVS, 14, 15, 22, 60

reference, 24
region, 26
run-time assertion checking, 62

same fringe, 49
Shankar, Natarajan, x, 60
Signoles, Julien, x
Simplify, 14

Talpin, Jean-Pierre, 30
termination, 4, 38
Théry, Laurent, 5, 13
theory, 20
tortoise and hare (algorithm), 4, 40
Turing, Alan, 2

Urbain, Xavier, 14

variant, 4, 29, 38, 41
Vieira, Bárbara, 15

W, algorithm, 30
weakest precondition, 31
Why, x, 5, **13**, 14, 15, 37, 52
Why3, 13, **13**, 16–18, 20, 22–25, 29–31, 35, 37–39, 42, 45, 52–57, 59–65
WhyML, 17, 23, **23**, 24–28, 61, 63, 78

Yices, 10
Ynot, 9

Z3, 10, 38
Zenon, 14
zipper, 50