Master's Thesis

Verification of Goroutines using Why3

Marc Schoolderman

July 2016

supervisor:
dr. F. Wiedijk
second reader:
dr. F. Verbeek

Radboud University Nijmegen
Institute for Computing and Information Sciences

Abstract

Concurrent software has become increasingly important in today's world; but reasoning formally about such programs is hard. The programming language Go offers a paradigm for writing programs that do not rely on memory sharing as a primary means of communication between processes. While this should make it easier to reason about concurrency, methods for formal verification of programs written in Go are absent.

In this thesis, we will investigate the capabilities of the Why3 verification platform to fill this gap. We first apply it to verify the complexity of heapsort, check an early program proof by Alan Turing, and solve other logical problems. We then use it as the foundation of a verification method for concurrent Go programs. With this method, we can verify partial correctness and liveness properties for concurrent programs written in a small subset of Go, using a high degree of automation. This technique is designed to be implementable as a tool, which could — if extended further — allow the partial verification of a larger range of Go programs.

Contents

1. Introduction

Reasoning about the correctness of programs is as old as the field of computer science itself. In 1949 Turing described a method for reasoning about the correctness of a process that computes $n!$ using only addition [1]. Turing imagines a situation where someone sits down to check whether this computational procedure actually does what it purports to do, and states:

> *In order to assist the checker, the programmer should make assertions about the various states that the machine can reach.*

Twenty years later, Hoare would propose formally defining the meaning of the programming languages we use, and a method of stating and checking such assertions with mathematical rigour [2]. It is still implied in this proposal that verification of a program is a manual, tedious, time-consuming and therefore a costly and error-prone activity, because someone has to perform the formal proofs.

Hoare's assumption that the cost of errors would eventually increase to a point where correctness proofs would become economically profitable has proven to be incorrect. Machine time is virtually free, and removing defects from software that has already been shipped happens all the time — even if that software has been shipped to Mars [3]. Software testing is still the main method of preventing bugs.

Painful mistakes in critical software such as Apple's 'goto fail' [4] show that that approach leaves much to be desired. Patching a security vulnerability does not help users whose systems have already been exploited. Embedded software is all around us, including in systems where its failure can mean loss of life. In such cases, the arguments in favour of formal verification are still sound: the challenge is to make it practical by reducing its cost.

1.1. Computer-aided reasoning

A major factor limiting the practical use of software verification as proposed by Hoare was that of doing formal proofs by hand. In present times, systems have been developed which remedy this situation by automation, removing the possibility of errors introduced by tedium, and significantly speeding up the process of finding a proof; or which have the ability to check a user-supplied proof mechanically and rigorously, thus ensuring its soundness. Prominent categories of these are:

Interactive Theorem Provers Prime examples of these are Coq [5], Isabelle/HOL [6], and PVS [7]. All of these provide a formal system which is rich in expressive power. Proofs in these systems must still be provided by the user, but are verified by a computer. Interactive theorem provers, while large programs in themselves, are usually built around a small trusted core so that they instil a high degree of confidence in the proofs.

Automated Theorem Provers The E theorem prover [8] and SPASS [9] fall into this category. These operate on the basis of a less expressive first-order logic, but find proofs without human intervention. This is usually based on *refutation*: to prove a proposition these systems try to derive a contradiction from its negation.

Satisfiability Solvers Examples of these are Alt-Ergo [10], CVC3 [11], CVC4 [12] and Z3 [13]. These operate on the basis of pure propositional logic (*SAT* solvers) or a subset of first-order logic with extensions for performing such things as basic arithmetic (*Satisfiability Modulo Theories*) These typically use a variant of the DPLL algorithm [14] to find a logical model satisfying a set of clauses, or show that none exists. As above, a proposition is proved by showing that its negation is not satisfiable by any model.

Model Checkers These allow the user to write a specification as a system of non-deterministic finite automata. Statements about a system are expressed in temporal logic, which are mechanically checked for all reachable states. SPIN [15] and NuSMV [16] are good examples.

When it comes to theorem provers, this classification is rather muddled, however. Most interactive theorem provers have support for a large degree of automation, and automated theorem provers can be instructed to look for proofs in a certain way. Also, all interactive theorem provers mentioned have a means to call external automated provers. In fact, the distinction between automated theorem provers and SMT solvers is of no real significance for our discussion, and we will generally use the term *automated theorem prover* to refer to both kinds of systems. On the other hand, model checking on the one hand and theorem proving on the other are very different, due to the differences in the way knowledge is represented in these systems.

1.2. Software analysis toolchains

Being able to perform large proofs is only one aspect of software verification. Perhaps even more important is the semantics of the programming language used. Modern programming languages have large specifications, and are also increasingly complex. If a programming language is officially standardized, it is usually in the form of a document written in *standardese* — language which is legalistic and often vague, and functions more like a contract upon implementers of the language than as a useful reference for the user.

Furthermore, when given a program fragment, it is hard to manually identify what exactly should be checked to verify it; especially since verification will often involve excluding a multitude of corner cases that are easily missed by a user, such as risks of pointer aliasing, integer overflow, ensuring that data structures remain in a well-behaved state when an exception occurs, and *data races* in multi-threaded environments.

To ensure validity of program verification, it is therefore necessary to also have a mechanical means of obtaining the formal conditions that must be satisfied for the program in question to be correct. Again, systems have been developed which assist with this, of which we should mention the following:

- OpenJML [17], the successor to ESC/Java, which generates verification conditions for programs written in Java in the form of propositions to be checked by SMT solvers.
- VCC [18] by Microsoft Research; this translates concurrent C programs to an intermediate language called *Boogie*. This intermediate language is then used to obtain verification conditions to be verified by Z3. Alternatively, it is also possible to use *Boogie* in concert with the Isabelle/HOL theorem prover [19].
- VeriFast [20], developed at the University of Leuven, is a tool based on separation logic for verification of C and Java programming languages. Like VCC, it uses Z3 to check the generated verification conditions.
- Frama-C [21], a suite of static analysis tools for C programs. One of these is the *Jessie* [22] plugin for generating verification conditions using the Why3 [23] verification framework. In this case, C programs are also translated to an intermediate language — WhyML — similar to the approach taken by VCC. Why3 will be discussed in more depth in the next chapter.

All of these systems also come equipped with some form of specification language (not part of the targeted programming language itself) for stating the necessary pre-conditions and post-conditions to be used for verification.

On a cautionary note, these and similar tools produce results that are mostly only valid for the semantics of a single implementation (which are usually not explicitly stated), or don't fully support the more infelicitous constructions of the target language. For instance, consider the following C declaration:

```
int x = (x=3) + (x=4);
```

This explicitly has undefined behaviour, as `x` is written to twice before a point in the program has been reached where all side-effects of expressions are guaranteed to have occurred. Value analysis with Frama-C will show, however, that it considers `x` to contain 8 afterwards. Running VCC will produce a warning, but also allows the verification of `x==8`.[1] VeriFast complains on *syntactic* grounds: according to it, the variable `x` is not in scope during the initializing expression.

A more rigorous approach would be to base a verification toolchain on an explicit, machine-readable formal version of a programming language specification, such as the one produced in the Formalin project [24]. The downside of this approach is the (current) lack of support for proof automation.

Another promising development is the arrival of new programming languages that are as practical as C, but avoid situations such as the above; or which contain a richer type system. Examples of this are the programming languages Rust [25] — which integrates the concept of *data ownership* into its type system — and Go [26], which is essentially an attempt at re-design of the C language. The disadvantage of these languages is that academic interest in them is much less than for more well-established languages, and so tools for formal verification of these languages are largely absent.

1.3. Concurrent programming

Modern computer systems usually consists of multiple CPU's. Software running on these systems can be optimized to make use of this parallelism by breaking a problem in small chunks that can be solved independently. This presents a special challenge, but also a more urgent need for software verification:

- Existing programming language specifications have been slow to catch up and standardize the guarantees (or lack thereof) given to multi-threaded software.
- In a single-threaded environment, non-determinism arises sparingly and consistently — for example, the order of evaluation of function arguments may be unspecified, but we can reasonably expect it to remain the same from one invocation to the next. In a multi-threaded environment, this is no longer true. Without explicit synchronization, it becomes impossible to tell what will happen exactly when.
- Various means for synchronization between threads exist, ranging from low-level primitives such as *atomic variables*, *semaphores*, *mutexes* and *condition variables* to higher level features such as *promises*, *futures*, and *message passing*. Using these can be tricky, and reasoning about the behaviour of programs involving these primitives in interesting ways is hard.
- Tools which allow static analysis of multi-threaded programs have only recently arrived. Older systems such as OpenJML are (currently) not able to take into account effects that may arise from interaction between multiple threads.

Model checkers are well-positioned to deal with this inherent non-determinism. Experience has shown that they can be used without much training to verify properties of (or detect bugs in) simple concurrent programs that go unnoticed when reasoned about by an experienced programmer [27]. But they have their drawbacks as well; in particular, it may not always possible to construct an accurate model for a given program.

[1]Which *is* consistent with the behaviour exhibited by Microsoft's compiler

1.4. Research question

The observations of the previous sections leads us to the main research question of this thesis.

The Go programming language, which is heavily influenced by Hoare's communicating sequential processes (CSP) [28], is our main motivation. A major appeal of Go is that concurrency is deeply part of its implementation — it is easy and practical to write efficient programs consisting of many thousands of concurrently executing subroutines that do not need to rely on data sharing for communication. The language design team actively encourages programmers to think of concurrency as a means of structuring a program, not necessarily just as an optimization strategy. Another appeal of Go is that it is a rather simple language, both in terms of its syntax, its type system, and memory model — as it has been specifically designed to be easily implementable, in contrast with a language such as C++.

This design choice should already make the *non-concurrent* parts of Go an easy target for developing formal verification tools — although this would present a significant amount of work, such work has been done before. This thesis therefore investigates whether and how Go's *concurrency primitives* can be reasoned about using a readily available verification platform. Why3 is a natural choice for this, as it allows for a high degree of flexibility.

The outline of the rest of this thesis is as follows.

- In Chapter 2 we will discuss the Why3 verification framework.
- Chapter 3 contains a number of case studies of simple but non-trivial non-concurrent programs and show how they can be verified using automated theorem proving by using the Why3 platform.
- Chapter 4 will briefly introduce the Go programming language, and describe the features marking it as an interesting target for concurrent software verification.
- Chapter 5 will describe a method for modelling a class of concurrent programs, correspondent with a small fragment of Go, in Why3. We will argue that this method can be in principle be used to verify properties of concurrent programs using automated theorem provers. This approach (and its drawbacks) will be demonstrated by verifying a concurrent version of a prime number sieve.
- Chapter 6 will discuss the results and offer suggestions for future work.

1.4.1. Reproducibility of results

In the interest of reproducibility, and since development of the aforementioned tools progresses rapidly, this section will list the verification tools used in the process of writing this thesis.

Why3 0.86.3 compiled using OCaml version 4.01.0

Alt-Ergo 0.99.1 compiled using OCaml version 4.01.0

CVC3 2.4.1 as packaged in the Debian GNU/Linux 8 distribution.

CVC4 1.4 compiled using GCC 4.9.2.

Z3 4.4.1 compiled using GCC 4.9.2.

Eprover 1.8 compiled using GCC 4.9.2.

SPASS 3.7 as packaged in the Debian 8 GNU/Linux distribution.

Coq 8.4pl4 as packaged in the Debian 8 GNU/Linux distribution.

All tests and examples were performed on a low-budget AMD64 laptop running Debian 8 (*"Jessie"*).

It should be noted that we have deliberately avoided the use of proprietary software, and especially software which is only available in binary form; all programs mentioned above are publicly available under an open source software license in some form.

All material related to this thesis will be made available at `http://github.com/squell/why3-go`.

1.5. A verification primer

Turing's verification method mentioned at the start of this chapter can be observed in Figure 1.2. We do not know whether he is working on the basis of an actual program for the Manchester Mark 1; he only provides an illustration of a program by giving us a control flow graph. While his method is basically sound and easy to understand, his attempt is not flawless in its execution, and sadly his presentation does not appear to have been very influential [1].

However, the control flow he presents can be implemented faithfully in x86-64 assembly language, shown in Figure 1.1. This is still an instructive example to demonstrate the process of modern program verification.[2] In this program, it is assumed that the natural number n resides in register `r10` at the start, which we will write as $r_{10} = n$ We can now make the following assertions, which are based on the ones given by Turing:

```
 1  L0: mov r9, 1
 2      mov r11, 1
 3  L1: mov r12, r11
 4      cmp r9, r10
 5      jae L3
 6      mov r8, 1
 7  L2: add r11, r12
 8      inc r8
 9      cmp r8, r9
10      jbe L2
11      inc r9
12      jmp L1
13  L3: ret
```

Figure 1.1.: Computing $n!$

- $r_{10} = n$ throughout the program.
- Whenever we are ready to execute the instruction labelled `L1`, it is the case that $1 \le r_9$, and $r_{11} = r_9!$. Also, $r_9 > n$ implies that $r_9 = 1$.
- Whenever we are at label `L2`, $1 \le r_8 \le r_9 < n$, $r_{12} = r_9!$, and $r_{11} = r_8 \cdot r_9!$.
- At label `L3`, $r_{11} = n!$, as desired.

The first assertion can be easily verified, as the register `r10` is not used as a destination register anywhere. Checking the other assertions requires *deductive reasoning*: we have to establish that they hold whenever whenever we transfer control to their respective entry points; consequently they may be used to prove other properties beyond that point.

For example, the second assertion holds when we reach `L1` the first time (since $r_9 = r_{11} = 1$). During verification of the fourth assertion, we may assume that it holds when we reach `L3`, since the only way to reach it is by taking the jump in line 5. But then we also know that $r_9 \ge n$, which means that either $r_9 = n$ and so $r_{11} = n!$, or $r_9 > n$, which means that $r_9 = 1$ and therefore by necessity $n = 0$. In this case we have $r_{11} = 1! = 0!$, and so again $r_{11} = n!$. Identifying the steps necessary to complete the verification of the second and third properties is left as an exercise to the reader.

[2]Especially if the reader hesitates at the sight of machine code.

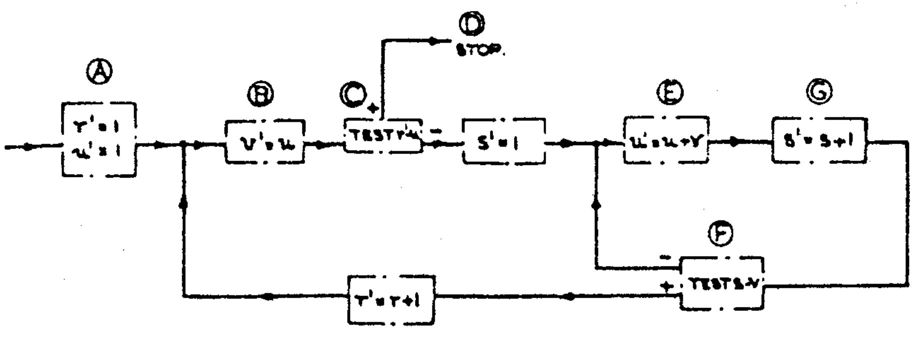
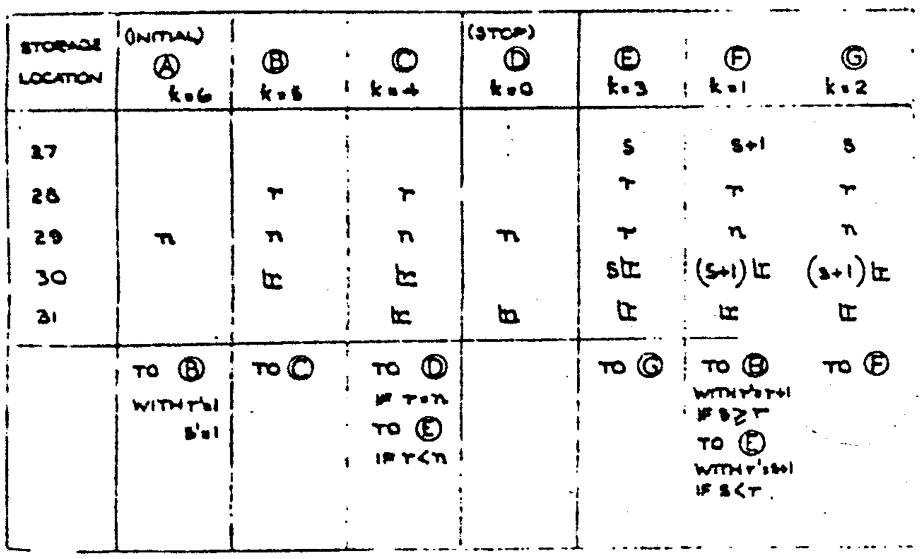


FIG. 1



STORAGE LOCATION	(INITIAL) A k = 6	B k = 5	C k = 4	(STOP) D k = 0	E k = 3	F k = 1	G k = 2
27					s	$s+1$	s
28		r	r		r	r	r
29	n	n	n	n	r	n	n
30		$\lfloor\underline{r}$	$\lfloor\underline{r}$		$s\lfloor\underline{r}$	$(s+1)\lfloor\underline{r}$	$(s+1)\lfloor\underline{r}$
31			$\lfloor\underline{r}$	$\lfloor\underline{n}$	$\lfloor\underline{r}$	$\lfloor\underline{r}$	$\lfloor\underline{r}$
	TO B WITH $r' = 1$ $s' = 1$	TO C	TO D IF $r = n$ TO E IF $r < n$		TO G	TO B WITH $r' = r + 1$ IF $s \geq r$ TO E WITH $r' = s + 1$ IF $s < r$.	TO F

FIG. 2

ISSUE 1 4.8.49.	T.R.E. M.O.S. DIAG. No. RTR 11/5790.

Figure 1.2.: Turing's original handwritten verification attempt, from the Turing Digital Archive. Note that Turing uses the peculiar notation $\lfloor\underline{n}$ to denote $n!$.

2. The Why3 verification platform

Why3 is a modular proof system developed for deductive program verification using a high degree of automation. In this chapter we will discuss its important features. The aim is to give a general feel for the strengths and drawbacks of the system.

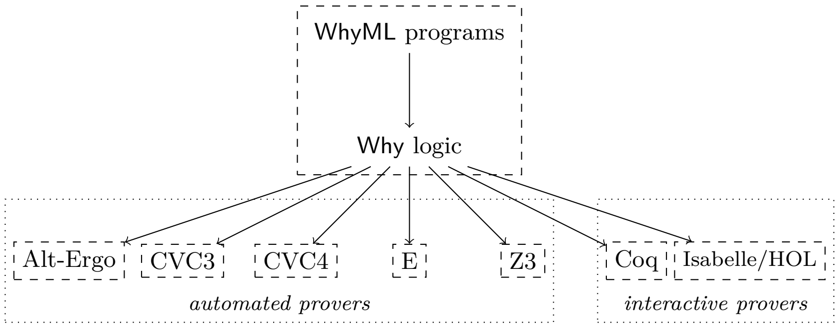
At the time of writing, Why3 [23] is under active development at INRIA by the Toccata project, and has its roots in research performed by Jean-Christophe Filliâtre in the area of verifying imperative programs using type theory [29]. This was followed in 2001 by the release of the Why tool, which could be used to produce verification conditions for programs written in a simple programming language for the proof assistants Coq [5] and PVS [7]. Other verification tools were created on top of this, such as *Krakatoa* for the verification of Java programs, and *Caduceus* for C programs [30] — which was then superseded by the *Jessie* plugin for the Frama-C static analysis platform [22, 21].

In 2010, Why3 was introduced to replace the earlier Why version, with a more powerful language, improved syntax, and an API to allow for easier user extensions. In addition to *Jessie* and *Krakatoa*, Why3 is being used as a back-end for the verification of Ada programs by the *SPARK 2014* tool [31], and the verification of cryptographic proofs in the *EasyCrypt* project [32].

2.1. Overview of Why3

Two related languages characterize the Why3 platform. The first of these (which we will simply call Why) is a purely logical language which is based on a typed first-order logic, with extensions to support polymorphism, algebraic data types, and (co)inductive predicates. The second language, called WhyML, is a simple yet sufficiently powerful programming language which is used for modelling programs for the purpose of generating verification conditions expressed in Why.

Why3 has only limited capability to reason about logic by itself, however; its main purpose is to translate its languages into input for various automated or interactive theorem provers. This design can be graphically represented as follows:



Both languages are modular in design. To use any definition, a user has to explicitly include the module that contains it. Also, not using a module means the theorems it introduces will not be visible to theorem provers. This is an important feature which serves three purposes:

- It improves the human readability of code written in Why/WhyML, similar to modern programming languages that use modularity and namespaces to decompose large programs into functionally self-contained parts.

- Reducing the amount of irrelevant information presented to external provers improves their performance, allowing for scalability of the system.
- By restricting the exposure of modules to each other, it means that Why3 has a coarse-grained mechanism for determining the interdependency of theorems, as this information cannot be determined by examining the output of theorem provers.

The last two facts also imply that it is beneficial in Why3 to structure formalizations in many insular components, as opposed to building a large 'kitchen sink' collection of theorems and definitions. This principle can be readily observed in Why3's standard library.

The WhyML programming language is syntactically quite similar to ML [33]. However, it is primarily an imperative programming language, supporting control-flow primitives as `while`/`for`-loops and exceptions, as well as familiar data types as mutable arrays. For verification purposes, WhyML has some features that are not commonly found in other programming languages. For instance, it has syntax for stating pre- and post-conditions, loop invariants, and allows writing so-called 'ghost' code, whose only function is to aid in verification.

Out of these conditions, Why3 extracts verification conditions from WhyML programs, using a weakest-precondition procedure [34]. Under certain conditions it is also possible to translate WhyML programs into OCaml, and run them directly.

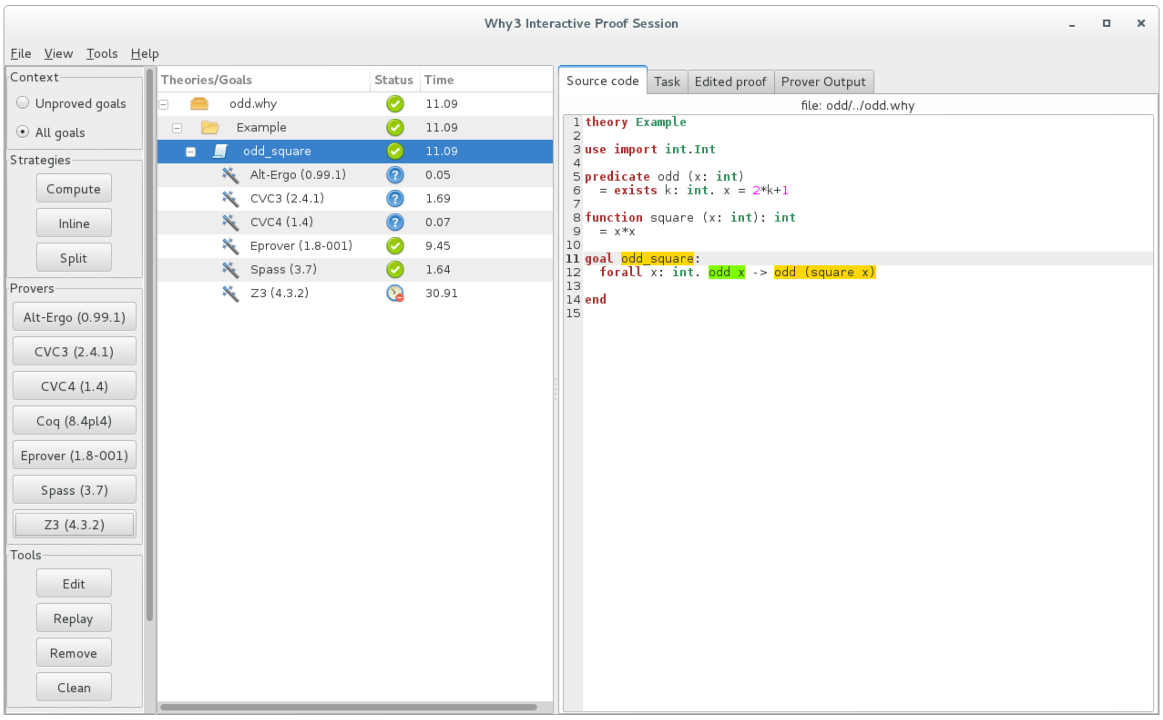
While Why3 is aimed at automated verification, this does not mean that it is a fully automatic system — in all but the most trivial WhyML programs, some form of logical transformation must still be applied to propositions in the logical language before they are simple enough to be solved by automated provers. This has to do with the that the logic of Why3 is a *logic of compromise*: it is chosen to be at a sufficiently expressive level to be easy to use, but it is not the native input language for any of the external provers [35]. Also, it is often the case that an automated prover struggles with a large conjunction, but can prove every term in it quickly if it is presented piece-by-piece, or that different provers are needed to prove different parts of the conjunction. This kind of proof transformation is done during a *interactive proof session* from within the Why3 IDE.

As an example, consider the following proposition in Why, which states that the square of an odd number is also an odd number:

```
theory Example

use import int.Int
predicate odd (x: int) = exists k: int. x = 2*k+1

function square (x: int): int = x*x

goal odd_square:
  forall x: int. odd x -> odd (square x)

end
```

Loading this into a proof session, and trying various theorem provers, we end up in the situation depicted in Figure 2.1. We find that the automated theorem provers E and SPASS can prove the goal straight away (although E requires 9 seconds), Z3 doesn't reach a conclusion within the set time limit (of 30 seconds), and the other theorem provers quickly conclude that they cannot decide either way. Other less common but possible outcomes would be that a theorem prover finds a counterexample, runs out of available memory, or encounters an internal error. At the right hand side of the window, the source code is listed, with highlights to indicate the current proof state.

By pressing the Compute button, Why3 applies the `compute_in_goal` transformation: it applies all computations that it reasonably can — expanding the definitions of the predicate `odd` and function `square` — resulting in a transformed proof task. Running the same theorem provers again results in the situation depicted in Figure 2.2. This new state of affairs turns out to be detrimental to the

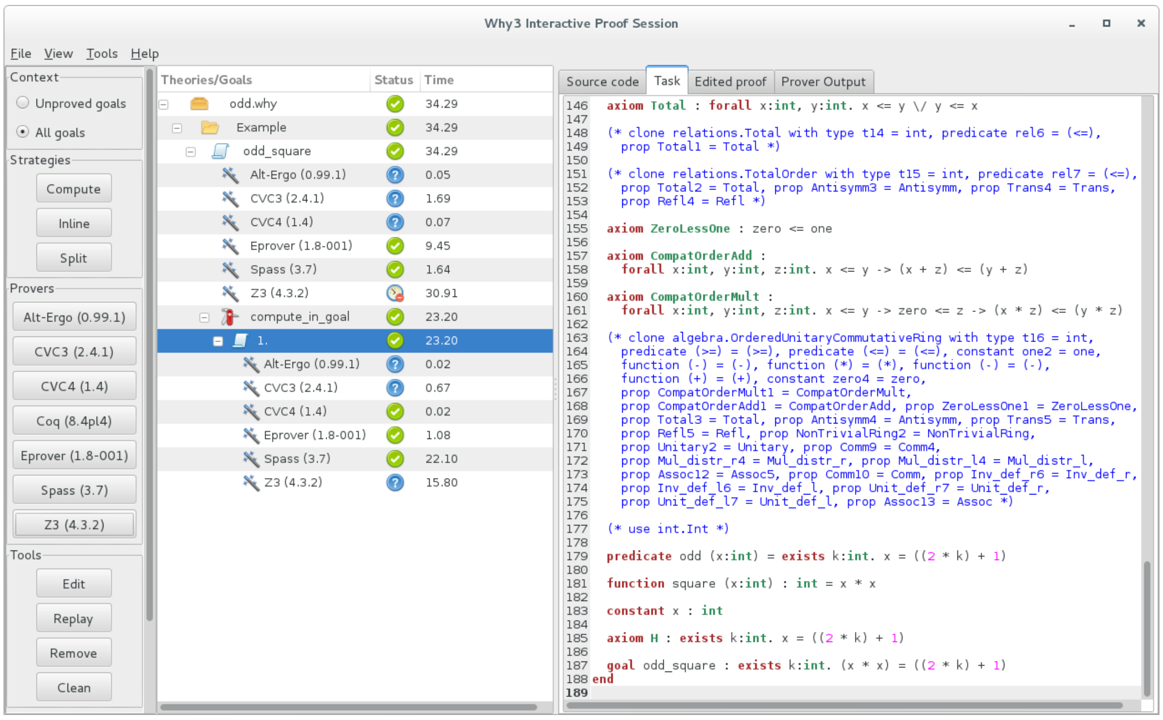


Theories/Goals	Status	Time
odd.why	✓	11.09
Example	✓	11.09
odd_square	✓	11.09
Alt-Ergo (0.99.1)	?	0.05
CVC3 (2.4.1)	?	1.69
CVC4 (1.4)	?	0.07
Eprover (1.8-001)	✓	9.45
Spass (3.7)	✓	1.64
Z3 (4.3.2)	⏱	30.91

file: odd/../odd.why

```
theory Example

use import int.Int

predicate odd (x: int)
  = exists k: int. x = 2*k+1

function square (x: int): int
  = x*x

goal odd_square:
  forall x: int. odd x -> odd (square x)

end

```

Figure 2.1.: Example Why3 proof session



Theories/Goals	Status	Time
odd.why	✓	34.29
Example	✓	34.29
odd_square	✓	34.29
Alt-Ergo (0.99.1)	?	0.05
CVC3 (2.4.1)	?	1.69
CVC4 (1.4)	?	0.07
Eprover (1.8-001)	✓	9.45
Spass (3.7)	✓	1.64
Z3 (4.3.2)	⏱	30.91
compute_in_goal	✓	23.20
1.	✓	23.20
Alt-Ergo (0.99.1)	?	0.02
CVC3 (2.4.1)	?	0.67
CVC4 (1.4)	✓	0.02
Eprover (1.8-001)	✓	1.08
Spass (3.7)	✓	22.10
Z3 (4.3.2)	?	15.80

```
  axiom Total : forall x:int, y:int. x <= y \/ y <= x

  (* clone relations.Total with type t14 = int, predicate rel6 = (<=),
    prop Total1 = Total *)

  (* clone relations.TotalOrder with type t15 = int, predicate rel7 = (<=),
    prop Total2 = Total, prop Antisymm3 = Antisymm, prop Trans4 = Trans,
    prop Refl4 = Refl *)

  axiom ZeroLessOne : zero <= one

  axiom CompatOrderAdd :
    forall x:int, y:int, z:int. x <= y -> (x + z) <= (y + z)

  axiom CompatOrderMult :
    forall x:int, y:int, z:int. x <= y -> zero <= z -> (x * z) <= (y * z)

  (* clone algebra.OrderedUnitaryCommutativeRing with type t16 = int,
    predicate (>=) = (>=), predicate (<=) = (<=), constant one2 = one,
    function (-) = (-), function (*) = (*), function (-) = (-),
    function (+) = (+), constant zero4 = zero,
    prop CompatOrderMult1 = CompatOrderMult,
    prop CompatOrderAdd1 = CompatOrderAdd, prop ZeroLessOne1 = ZeroLessOne,
    prop Total3 = Total, prop Antisymm4 = Antisymm, prop Trans5 = Trans,
    prop Refl5 = Refl, prop NonTrivialRing2 = NonTrivialRing,
    prop Unitary2 = Unitary, prop Comm9 = Comm4,
    prop Mul_distr_r4 = Mul_distr_r, prop Mul_distr_l4 = Mul_distr_l,
    prop Assoc12 = Assoc5, prop Comm10 = Comm, prop Inv_def_r6 = Inv_def_r,
    prop Inv_def_l6 = Inv_def_l, prop Unit_def_r7 = Unit_def_r,
    prop Unit_def_l7 = Unit_def_l, prop Assoc13 = Assoc *)

  (* use int.Int *)

  predicate odd (x:int) = exists k:int. x = ((2 * k) + 1)

  function square (x:int) : int = x * x

  constant x : int

  axiom H : exists k:int. x = ((2 * k) + 1)

  goal odd_square : exists k:int. (x * x) = ((2 * k) + 1)
end

```

Figure 2.2.: Why3 proof session after applying a transformation

time needed by SPASS, but benefits E. Also, CVC4 can now prove this fact instantly. We can also examine what exactly the transformation has changed by changing to the Task pane, which lists the current proof goal as Why code.

A proof session like this will be saved when we exit the IDE, and can be resumed or replayed at a later moment. If the theory under consideration has changed, Why3 detects which proofs are no longer up-to-date and marks them as obsolete. In addition, it marks all proofs which may depend on a proposition whose proof is obsolete as obsolete as well [36].

2.2. The Why language

The logical language Why3 allows the declaration of types, function symbols and predicates. These can be left abstract, or given a definition. New types can be created as record types or (potentially recursive) algebraic data types. For example, given the types:

```
type boxed 'a = { item: 'a; wraps: int }
type wrap 'a = Only 'a | Wrap (wrap 'a)
```

we can define the recursive function:

```
function shrinkwrap (obj: wrap 'a): boxed 'a
  = match obj with
    | Only x -> { item = x; wraps = 0 }
    | Wrap x -> let y = shrinkwrap x in { item = y.item; wraps = y.wraps+1 }
    end
```

Functions defined in this direct way must always be totally defined; it would be an error to omit the `Wrap x` in the pattern match. This also means that all terms in the logic language must be terminating; Why3 checks this automatically, and in many simple cases this suffices. In other cases, this does not work — for example, an inverse function of `shrinkwrap` would need to be recursive in its integer argument. In the present version of Why3 this means another method of defining such a function has to be used; we will examine such a way in Section 2.3.4. In future versions this limitation may be lifted [23].

A related property of Why3 is that all types must have at least one inhabitant; the above definition is rejected if we had omitted the case `Only 'a`, since in that case there does not exist a (finite) object for the type.

It is possible to write partially defined functions and predicates by using axiomatic definitions:

```
predicate five int
axiom five_def:
  five 5
```

This predicate is demonstrably true only for the number five; but it is also never demonstrably false. This kind of incomplete information is pervasive in Why3. For example, the value `div 1 0` is not defined; but the type system demands that it is a valid expression of type `int` — we just don't know what it is. This does not prohibit its use in subsequent calculations, and so, perhaps surprisingly, the following goal is provable:

```
use import int.Int
use import int.EuclideanDivision
goal div_identity:
  forall k: int. k * div k k = k
```

In a significant departure from pure first-order logic, Why3 also supports the definition of inductive predicates; these introduce the predicate symbol as the *least* possible relation satisfying the definition.

For instance, we can define the following equivalence relation on objects of type `wrap 'a`:

```
inductive similar (wrap 'a) (wrap 'a) =
| similar_0: forall x y: 'a. x = y -> similar (Only x) (Only y)
| similar_l: forall x y: wrap 'a. similar x y -> similar (Wrap x) y
| similar_r: forall x y: wrap 'a. similar x y -> similar x (Wrap y)
```

Using this relation, we can prove that `not similar (Only 1) (Only 2)` by applying the `inversion_pr` transformation described below, which would not the case had we defined the predicate axiomatically.

Another extension is the addition of higher-order functions and predicates, by using the built-in `HighOrd` theory, these become available so we can write a function such as:

```
use HighOrd
function mapbox (f: 'a->'b): boxed 'a -> boxed 'b
  = \x. { item = f x.item; wraps = x.wraps }
```

When using the `HighOrd` theory, currying and lambda abstractions also become useful tools, and so the language supports these as well. Function types can also be used for bound variables in logical quantifiers:

```
goal identity: exists f: int->int. f 2 = 3
```

However, since all automated theorem provers only support first-order logic, this is of limited use.

Logical facts in the Why logical language are introduced by three types of declarations:

Axioms A logical formula declared as an axiom is provable by definition.

If an axiom deals with newly introduced symbols, it is possible to establish criteria under which it is harmless. For instance, if P is a axiomatic statement concerning a user-defined symbol, and we can establish that for any statement Q in which this symbol does not occur, $P \rightarrow Q$ is provable only if Q itself is provable, then the axiom is admissible [37]. However, for obvious reasons, checking this is something that has to be done manually by the user.

Goals A logical formula declared as a goal must be proved; we already have seen these in the above.

Lemmas A logical formula declared as a lemma must be proved as well, but in any declaration following it, it is added to the set of assumed logical facts.

In conclusion, the Why language can best be viewed as a mostly first-order logic, with a total functional programming language for describing entities in the universe of discourse.

2.2.1. Modularization in Why3

Declarations in Why must be put in a *theory*; these can then be used in other theories. We have already seen several examples of this; for example, we had to import `int.Int` to be able to multiply two integers using the `*` operator. When using a theory, the user has control over how its definitions are introduced. When using `use import` directive, all definitions will be visible within in the current namespace; when using `use export`, they will become part of its namespace, and so will become visible to other theories that use the one being defined. If not importing a theory, a qualified identifier can be chosen by the user. As an example:

```
use import int.EuclideanDivision as E
use int.ComputerDivision as C
```

This will make the `div` function from the `EuclideanDivision` theory — which has nicer mathematical properties — visible as `div` and `E.div`, and at the same time allow us to use reason about the way division is defined by ordinary programming languages by referencing `C.div`.

Another possibility is to `clone` a theory. When we do this, the theory in question is not simply made accessible, but all of its definitions are *copied* into a new theory, which is then used instead. When doing this, we have the ability to provide an instantiation for types, functions and predicates that were left abstract in the original theory. Axioms governing these abstract symbols the original theory are cloned as well; to avoid risks of inconsistency, these can (and should) be turned into *goal*s and *lemma*s. For example, consider the following attempt to apply the principle of induction to show that all integers are the same:

```
predicate all_are_equal (n: int)
  = forall m: int. n = m

clone int.SimpleInduction
  with predicate p = all_are_equal, goal base, goal induction_step
```

This will fail to verify because the base case is false (since clearly not all integers are equal to 0). However, if we forgot to add `goal base`, Why3 will accept this (as the induction step is provable) without warning.

The WhyML language, explained further on, has a similar system of modularization, except that its declarations are bundled together in so-called *modules* instead of *theories*. The only distinction is that a `module` is allowed to contain WhyML code, whereas a `theory` cannot.

2.2.2. Logical transformations

As we have already discussed briefly, it is often the case that some manual intervention from within the Why3 IDE is required for automated theorem provers to work. Some of these transformations are rather specialized, but it is possible to distinguish three broad categories:

Computational transformations We have already used an instance of this in the example of Figure 2.2. Two of are so commonly applicable that they have a dedicated button in the IDE: the `inline_goal` transformation, which replaces function and predicate symbols by their definitions in the current goal, and `compute_in_goal` which applies other rewrite rules as well.

Eliminating transformations These are simplification strategies which remove features of the Why3 language which may not be handled well by theorem provers, and replace them by abstract symbols that are axiomatically defined. For example, the following goal turns out to be surprisingly hard for automated provers:

```
constant everything: set int = comprehension (\x: int. x = 42)
constant universe: set int = everything
goal question: universe = everything
```

But by applying the `eliminate_definition` transformation, the proof task is transformed into something entirely trivial:

```
constant everything
axiom everything_def: everything = comprehension (\x: int. x = 42)
constant universe: set int
axiom universe_def: universe = everything
goal question: universe = everything
```

Note that the `compute_in_goal` transformation also solves the original goal, without needing to resort to an external theorem prover at all.

Splitting transformations These represent strategies to break up large or complex propositions into smaller ones. The most common example of this is the `split_goal` transformation, which transforms a conjunction into a set of goals for each of its constituent terms. Also in this category are induction strategies for inductive predicates (`induction_pr`, `inversion_pr`) and algebraic data types (`induction_ty_lex`).

It is generally hard to to decide beforehand which transformation to apply to goals; when we are verifying many goals, this usually is a trial-and-error process during a proof session in the IDE. Some idea of what makes a proof difficult for automation can also be obtained by trying to 'debug' a goal by using an interactive theorem prover. For example, if we try to prove the `question` goal shown above, loading it into Coq gives us the proof obligation:

```
Parameter fc2: (Z -> bool).
Parameter fc3: (Z -> bool).
Axiom fc_def2 : forall (x:Z), ((fc2 x) = true) <-> (x = 42%Z).
Axiom fc_def3 : forall (x:Z), ((fc3 x) = true) <-> (x = 42%Z).
Theorem life : ((comprehension fc2) = (comprehension fc3)
```

Which, of course, involves arriving at a proof state where the functions `fc2` and `fc3` can be shown to have the same effect. One way to do this is by applying of set extensionality axiom provided by Why3:

```
apply set.Set.extensionality.
repeat intro.
repeat rewrite comprehension_def.
rewrite fc_def2.
rewrite fc_def3.
intuition.
```

But clearly this involves some creative thinking on the part of the user. The problem here seems to be that Why3 replaces the two constants with their definition as it generates input for provers; by applying the `eliminate_definition` transformation this is bypassed.

Like the choice of which external prover to run for which goal, transformations applied by the user are remembered between proof sessions, and so verifying (or in Why3 terminology, *replaying*) a completed Why3 session by a different user does not require this kind of interaction, as long as they are provided with the files in which Why3 stores the proof session in addition to the Why/WhyML sources.

2.2.3. Example: Tarski's fixed point theorem

To demonstrate a non-trivial use of Why's logic language, we will use it to develop a theory to formalize a well-known set-theoretic theorem, and show how we can use that theory elsewhere.

Theorem (Tarski's fixed point theorem)**.** *Given a set A and a function $f : \mathcal{P}(A) \to \mathcal{P}(A)$, if for any $X, Y \subseteq A$ it holds that $f(X) \subseteq f(Y)$, than there exists a μ such that $\mu = F(\mu)$.*

An idiomatic way to formalize this in Why3 is to build a theory around abstract symbols for A and f, which can be instantiated by the user. The skeleton for this is as follows:

```
theory Tarski

use import set.Set

type t

function f (set t): set t

axiom f_is_monotonic:
  forall x y: set t. subset x y -> subset (f x) (f y)

goal f_has_fixpoint:
```

```
13     exists x: set t. f x = x
14 
15   end
```

Lines 5 and 7 introduce the abstract symbols for the function f operating on sets of type t. The type t fulfils the role of the set A — its only role is to denote the elements that the powerset domain is built on top of. This subtle difference is, of course, a result of the fact that the foundation of Why3's logic language is a typed logic. In fact, to reason about sets we need to import the definitions and axioms of set theory in line 3.

One might optimistically load this theory in Why3 and check whether it can prove the stated goal automatically. Unsurprisingly, this will not succeed.

As is often the case with computer-assisted theorem proving, our first task is then to figure out why the theorem should hold at all, by finding an informal proof.

Informal Proof. Let $P := \{X \subseteq A : f(X) \subseteq X\}$ be the set of *pre-fixpoints* of f, and then consider $\mu := \bigcap P$. We will show that μ is the desired fixpoint by showing the inclusions $f(\mu) \subseteq \mu$ and $\mu \subseteq f(\mu)$.

By its construction, $\mu \subseteq X$ for all $X \in P$, and therefore $f(\mu) \subseteq f(X)$ by monotonicity of f, and since $f(X) \subseteq X$, we also have $f(\mu) \subseteq X$. Since this is the case for all $X \in P$, $f(\mu)$ is also a subset of $\bigcap P$, and thus we have shown the first inclusion. But then, again by monotonicity of f, we also have $f(f(\mu)) \subseteq f(\mu)$; i.e. $f(\mu) \in P$, and so it follows that $\mu \subseteq f(\mu)$, as desired. □

The above argument roughly can be divided into two parts:

- Construct the intersection of the set of pre-fixpoints of f.
- Verify that it is a fixed point by showing both inclusions.

Why3's support for set theory does not contain a function equal to the $\bigcap$-operator out of the box, but it is easy to construct using the `set.SetComprehension` theory:

```
use import set.SetComprehension

function intersect (fam: set (set t)): set t
  = comprehension (\x: t. forall y: set t. mem y fam -> mem x y)
```

which we can then use to define μ:

```
constant prefixpoints: set (set t) =
  comprehension (\x: set t. subset (f x) x)

constant mu: set t = intersect prefixpoints
```

We then introduce three lemmas.

```
lemma fmu_subset_of_mu:
  subset (f mu) mu

lemma mu_subset_fmu:
  subset mu (f mu)

lemma mu_is_fixpoint:
  f mu = mu
```

If we retry to prove this theory in Why3, we find out that the lemmas `fmu_subset_of_mu` and `mu_is_fixpoint` are still intractable for our suite of theorem provers.

In the case of `fmu_subset_of_mu`, after experimenting with and refining various lemmas that help prove this goal, it becomes clear that automated provers benefit by explicitly stating a fact about intersections that the observant reader may have noticed was conspicuously taken for granted in the informal proof above.

Proposition. *For any collection of sets S, the intersection $\bigcap S$ is the greatest common subset of all sets in the collection. That is:*

- *$\bigcap S \subseteq X$, for all $X \in S$.*
- *Whenever $Y \subseteq X$ for every $X \in S$, we also have $Y \subseteq \bigcap X$.*

Which we can easily formulate and get proven automatically in Why3:

```
lemma intersect_common_subset:
  forall fam: set (set t).
    forall x: set t. mem x fam -> subset (intersect fam) x

lemma intersect_greatest_common_subset:
  forall fam: set (set t),  s: set t.
    (forall x: set t. mem x fam -> subset s x) -> subset s (intersect fam)
```

In the case of the goal `mu_is_fixpoint`, what is needed is of course the fact that if $X \subseteq Y$ and $Y \subseteq X$, that it follows that $X = Y$.

By inspecting the `set.SetGen` theory to find out what makes this fact difficult, we discover that Why3 defines the predicate == to mean that two sets have the same members, and then relates this to ordinary equality by adding the axiom of extensionality for sets. So we can try to replace the final lemma with:

```
lemma fmu_equiv_of_mu:
  f mu == mu
```

The resulting theory can be proven (after judicious use of the `eliminate_definition` or `inline_goal` transformation on selected goals) by all theorem provers tested (Alt-Ergo, CVC3, CVC4, Z3, E); each goal can also be proven easily using a handful tactics in Coq. Using just Alt-Ergo, the entire theory can be verified in less than half a second.

The `Tarski` theory can now be used to get the fixed point for any monotonic function on sets by *cloning* it, supplying concrete instances for the abstract type and function symbols, and stating that the given function satisfies the axiom:

```
theory Example

use import int.Int
use import set.SetComprehension

function shift (s: set int): set int
  = add 0 (map \x: int. x+1) s

clone Tarski with type elt = int, function f = shift, goal f_is_monotonic

goal shift_has_fixpoint:
  exists s: set int. shift s = s

end
```

As a by-product of developing this theory, we have introduced a new function symbol (`intersect`) and by iterative refinement developed two useful lemmas about it, which both are more generally applicable than the theorem we set out to prove. It would now be good programming practice to spin this off into a separate compact theory that can be re-used elsewhere.

2.2.4. Example: Solving the Dining Philosopher's problem

A common use of automated provers is to find solutions to problems by encoding the problem as a logical puzzle. Even though Why3 was not developed with this goal in mind, we will show that it can be used for such a purpose.

For this, we will examine the well-known problem of the Dining Philosophers [38]. In this concurrency problem, five philosophers are situated at a circular table behind five plates of food, with a fork placed between each of them. All philosophers will want to eat, but to do that, they must pick up the forks on either side of their plates in some order, and they will have to wait if their neighbour is holding their fork. Once they have picked up a fork, however, they will not put it down until they have taken a bite.

It is not difficult to show that given this set of requirements, the philosophers have a problem: if at the start all philosophers pick up the fork to their right, no forks remain on the table, but no one will be able to eat, since nobody is in possession of both forks. Being unable to eat also means they are equally unable to put down the one fork already in their right hand. Thus we have arrived at a deadlock, which (in this sense quite literally) leads to starvation.

A solution to break the deadlock is to impose a strict order in which the forks will be taken from the table. If we impose the rule that four of the philosophers always pick up their right fork first, and instruct the fifth to do this in the reverse order, no deadlock can arise.

One possible way to model this problem is to model it as a non-deterministic finite automaton, and then show that no deadlock can occur if the above solution is used. Naturally, this approach lends itself very well to model checkers for exactly this reason. When using a theorem prover, it is less simple, as the machinery for a state transition system then has to be modelled explicitly.

An approach more suitable to automated theorem proving is to create a set of constraints that must be satisfied by the entire state space of the problem. In anthropomorphic terms, this is approaching the problem as if we have a photograph of five philosophers eating, and deducing on that basis alone whether they are following the rules given above, and if they are, whether or not they are in a deadlocked situation.

A benefit of this is that we can work abstractly; all that is required is that any state in our original problem is a model of the set of constraints — we do not need to capture every detail of the problem, or explicitly exclude all situations which are not part of the state space. With this in mind, we develop a logical specification for the Dining Philosophers. We begin by defining the structure of the logical domain:

```
type philosopher
type fork

function left  philosopher: fork
function right philosopher: fork

predicate grasped philosopher fork
```

That is, we assign left-hand and right-hand side forks to philosophers by a function, and define a predicate which should be interpreted as stating that a philosopher holds a fork in her hand.

We can now write down the constraints of the puzzle as a set of axioms. First of all, there are enough right-hand side and left-hand side forks for each philosopher. This can be ensured by requiring that both functions are injective:

```
axiom fork_availability:
  forall p q. (right p = right q \/ left p = left q) -> p = q
```

Also, philosophers are not allowed to use any other fork but those:

```
axiom bounded_resource:
  forall p, f. grasped p f -> (f = left p \/ f = right p)
```

A fork can only be held by a single philosopher at any time:

```
axiom mutual_exclusion:
  forall p q, f. grasped p f -> grasped q f -> p = q
```

Every fork is shared between (at least) two philosophers; that is, if a fork is a left-hand fork for some philosopher, it is the right-hand fork for someone else, and vice versa:

```
axiom shared_resource:
  forall f. exists p q. f = left p /\ f = right q
```

Note that in all cases, since the types of the quantified variables can be inferred from their use, they can be safely omitted.

An implication of the proposed solution is that there is always a philosopher who cannot be empty handed if she is holding a fork in her right hand; and that for all other philosophers the converse is true:

```
axiom lefties_and_righties:
  exists lefty. forall p. if p = lefty then grasped p (right p) -> grasped p (left p)
                                        else grasped p (left p) -> grasped p (right p)
```

Note that the name `lefty` in this proposition is somewhat misleading; for example, if all philosophers are holding either two forks or none, then any of them fits the requirement for `lefty`. However, the existence of exactly one *lefty* ensures that this axiom is always satisfied, which is all that matters.

Finally, we are left with formulating the property we wish to check. Suppose we are in a reachable state where there is a philosopher whose forks are not being held by a neighbour. Then we are clearly not in a deadlock, as this philosopher is either eating, or able to pick up a fork. So we can state:

```
goal dining:
  exists p. (forall q. grasped q (right p) -> p = q) /\
            (forall q. grasped q (left  p) -> p = q)
```

The resulting theory can be loaded into the Why3 IDE; the theorem provers SPASS is able to prove it within 15 seconds.

The verification time can be further improved by adding this lemma just before the previous goal:

```
lemma left_forks_are_all_forks:
  (forall p. exists q. grasped q (left p)) -> forall f. exists q. grasped q f
```

When we do this, CVC3 can verify the entire theory nearly instantaneously.

Note that the specification is very general. A restaurant in Hilbert's Grand Hotel with an infinite amount of philosophers sitting at a table for one with just one fork on it is also a model, a situation with a table where five right-handed philosophers are in a deadlock while a left-handed philosopher is eating happily by himself on a separate table is another. This also shows that the absence of deadlock does not mean an absence of starvation.

2.3. The WhyML programming language

We now turn our attention to the programming language used by Why3 for the actual purpose of software verification. The reader may at this point get confused — for we have already seen that the Why logic language can be used for a limited form of functional programming.

However, these limitations make it too far removed from commonly used imperative languages; in particular, the notion of a state is entirely lacking. Therefore a second language – WhyML – is needed. It is stateful in nature and well-suited to imperative programming. Another major difference is that whereas a function in the Why language essentially defines a rewrite rule — where any occurrence of the function call can be substituted by its body (as performed by the `inline_goal` transformation) — subroutines defined in WhyML define a *contract* expressed in the form of user-supplied preconditions and postconditions. For example:

```
let square (x: int): int
  requires { odd x }
  ensures { odd result }
= x*x
```

When calling this function from some other point, Why3 will generate a verification condition to check that the argument given is odd, and assume that the result is odd as well; but it will have no knowledge at that point that the result is equal to the square of the input; for that, we should have explicitly specified:

```
  ensures { result = x*x }
```

WhyML extends the type system of the Why language in two aspects:

- Record fields can be designated as *mutable*; these can be updated using the `<-` assignment syntax.
- It is possible to specify a *type invariant* for user-defined data types, which is then checked at (and only at) function entry/exit.

For example:

```
type oddint = { mutable value: int }
  invariant { exists k: int. self.value = 2*k+1 }

let square_by_ref (x: oddint)
  ensures { x.value = old (x.value*x.value) }
= x.value <- x.value*x.value
```

In this case, calling the procedure `square_by_ref` modifies the provided argument in-place; necessitating the use of the `old` keyword to reference the value of `x` before calling the function. The type invariant for `oddint` ensures that a verification condition will be added to ensure that the result is an odd value.

The WhyML standard library comes with several familiar data types that have mutable fields, most notably the types `ref 'a` and `array 'a`. which make programming with mutable data easier than in the above example. One of the restrictions not found in ordinary programming language is that aliasing of function arguments containing mutable data is forbidden, and that recursive data type definitions are not allowed to contain mutable data types — which also means that it is not possible to implement e.g. a destructively updated linked list in WhyML.

WhyML allows the creation of recursive functions using the `let rec` syntax. In this case, termination of the function can be proven by adding a `variant` declaration. This should denote an expression that diminishes during each recursive call; this can be an algebraic data type or an integer expression (that is guaranteed to remain non-negative), or a lexicographic ordering of multiple expressions separated by commas; for instance:

```
let rec ackermann (m n: int)
  requires { m >= 0 /\ n >= 0 }
  ensures { result >= 0 }
  variant { m, n }
= if m = 0 then
    n+1
  else if n = 0 then
```

```
    ackermann (m-1) 1
  else
    ackermann (m-1) (ackermann m (n-1))
```

If a (recursive or non-recursive) function is not supposed to terminate, this can be indicated by marking it as such by using the keyword `diverges`.

To allow for the verification of functions containing loops, loop invariants must be added by the user pertaining to all mutable data that is modified during the loop body. Loop termination is guaranteed when using `for`-loops; when using `while`-loops (or the `loop` construct), the user again has to specify a `variant` for each loop, as with recursive functions.

WhyML does not have a `break` or `return` statement, but programs are capable of non-local transfer of control flow by raising exceptions. If an exception is not caught in any `try`/`with` block, this means a function has an alternative means of terminating, and so in this case the user is required to specify an *exceptional post-condition*.

To assist complicated proofs, or to explicitly verify properties of a program, it is also possible to insert assertions at arbitrary locations in a WhyML program. These are analogous to the three types of logical declarations of Why. A statement introduced via an `assume` keyword is simply assumed to be true at the point in the program it occurs, a `check` can be used to verify a desired property but cannot be used to prove other facts, and an `assert` functions as a lemma — it generates a proof obligation but can also be used to prove subsequent facts. The use of `assert` statements is also beneficial when debugging proofs.

Note that all declarations that are valid in the Why language are also valid in WhyML. So, programs can be freely mixed with `lemma`'s, and function/predicate declarations expressed in the Why language. Such declarations reside in a separate namespace which is accessible from WhyML, which means that WhyML programs can use them — but not the other way around — or re-define them (without altering their meaning at the Why level).

2.3.1. Example: Sieve of Eratosthenes

As an demonstration of a small WhyML program, we will verify the ancient algorithm for finding all prime numbers up to a given limit m. In prose, this algorithm can be described as follows [39]:

1. Enumerate all integers between 2 and m.
2. Let p be the smallest number of this set; it is a prime number, add it to the set of discovered primes.
3. Remove all multiples of p from the list, including p itself.
4. Repeat steps 2 and 3 until the input set has been depleted.

A WhyML model of this algorithm, expressed using Why3's theory for handling finite sets of integers, is listed in Figure 2.3, complete with all annotations to allow it to be automatically verified. Its post-condition on line 12 expresses the fact that the result of this function is a set containing exactly all the prime numbers less than m.

A crucial step in proving the correctness of the above algorithm is, of course, determining why after removing all multiples of p it follows that the new smallest number on the list is automatically a prime number. To show this, we establish (in the first loop invariant) the property that if a number is in the input set, then all its positive divisors (except 1) are also in this set. This is clearly true initially, and also preserved in each iteration by the third step. From this loop invariant it can be deduced that the smallest number has no other divisors except itself and 1, and so is a prime number.

Showing that this algorithm doesn't accidentally 'skip' prime numbers might be intuitively obvious, but still needs to be proven rigorously as well. To do this, we use the second loop invariant, which states that all prime numbers less than m are present in the combination of the input set and the

```
1  module PrimeNumberSieve
2
3  use import ref.Ref
4  use import set.Fsetint
5  use import set.FsetComprehension
6  use import number.Divisibility
7  use import number.Prime
8
9  function sieve (n: int) (s: set int): set int = filter (\x. not (divides n x)) s
10
11 let primes (m: int): set int
12   ensures { forall n: int. prime n /\ n < m <-> mem n result }
13 = let nums = ref (interval 2 m) in
14   let primes = ref empty in
15   while not (is_empty !nums) do
16     invariant { forall n k: int. mem n !nums /\ 2 <= k /\ divides k n -> mem k !nums }
17     invariant { forall n: int. prime n /\ n < m -> mem n (union !nums !primes) }
18     invariant { forall n: int. mem n !primes -> prime n /\ n < m }
19     invariant { forall n: int. mem n !nums -> 2 <= n < m }
20     variant { cardinal !nums }
21     let p = min_elt !nums in
22     assert { subset (add p (sieve p !nums)) !nums };
23     primes := add p !primes;
24     nums := sieve p !nums;
25   done;
26   !primes
27
28 end
```

Figure 2.3.: Prime number sieve in WhyML

set of discovered primes. Initially, this is clearly true (as the input set contains all numbers); at termination (when the input set has become empty), it implies that we have discovered all the prime numbers.

The third and fourth invariant will seem rather pedantic; these simply state that nothing *else* is added to set of discovered primes, or the input set, during execution. Proving termination can be done in various ways; for example by taking the cardinality of the input set as the variant. By adding the assertion on line 22, it is possible to verify this variant automatically.

Loading this module into Why3, it can be verified in less than half a minute by both CVC3 and CVC4, after applying the `split_in_goal` transformation.

2.3.2. Abstract reasoning in WhyML

Like the Why language, declarations in WhyML can be left abstract. By introducing functions using the `val` keyword instead of `let`, the function body can be omitted, and any post-conditions of the function is simply accepted without question. If abstract functions have side effects (i.e., they alter mutable data), these must be explicitly stated, unlike ordinary functions, where Why3 is able to deduce them from the function definition. As an example:

```
val post_increment (x: ref int): int
  writes { x }
  ensures { result = old !x /\ !x = old !x+1 }
```

Without the `writes` annotation, calling this function would introduce an inconsistency by allowing `!x = !x+1` to become provable. In the above case, a function body satisfying the specification can easily be provided, and so the above declaration should morally be avoided.

WhyML also allows individual expressions to be abstract, using the expression "`any` *type*". An optional post-condition can be provided to partially specify the result. For example, to model a dice roll, we could use the non-determinism implied by the following abstract expression:

```
let dice = any int ensures { 1 <= result <= 6 } in (* .... *)
```

Instead of leaving out information, another method of achieving abstraction is that of hiding it. As stated earlier, WhyML programs have — in principle — full access to declarations expressed at the Why logical level. This may expose too much internal details of the logical model to the WhyML execution environment. For instance, an `array` in WhyML is modelled internally as a record holding a *length* field and a mutable `map`; but programs should treat it as a primitive type. WhyML allows to hide such information by declaring *model* types:

```
type wrap 'a model Only 'a | Wrap (wrap 'a)
```

In this case, the identifiers `Only` and `Wrap` are contained within the namespace of the Why sub-language, and no longer accessible from WhyML programs. The same holds for functions. To use the `shrinkwrap` function as given on page 15 in WhyML programs, we have to explicitly create a separate abstract WhyML-function, whose result is identical to that of the Why-function:

```
val shrinkwrap (obj: wrap 'a): boxed 'a
  ensures { result = shrinkwrap obj }
```

WhyML also allows preventing certain parts of a program from interfering with the proof of other parts, by wrapping it in an `abstract`-block:

```
let x = ref 0 in
abstract ensures { !x > old !x }
  x := !x+1
end
```

Code that appears after the `abstract` block will only know that `x` has been incremented, but not by how much. This can be helpful when developing proofs for larger WhyML programs, as altering the body of an abstract block will preserve earlier proofs outside it. It can also help in reducing the amount of redundant information presented to automated theorem provers, allowing them to work more efficiently.

2.3.3. Using ghost code to aid verification

In certain cases, additional computation may be necessary only for properly specifying verification conditions. For example, consider the following WhyML function for computing the inverse of `b` modulo `a`:

```
let mult_inverse (a b: int)
  requires { a > 1 /\ b > 0 /\ gcd a b = 1 }
  ensures { mod (result * b) a = 1}
= let (a, b)  = (ref a, ref b) in
  let (t0,t1) = (ref 0, ref 1) in
  let (s0,s1) = ghost (ref 1, ref 0) in
'Begin:
  while !b <> 1 do
    invariant { !s0*(at !a 'Begin) + !t0*(at !b 'Begin) = !a > 0 }
```

```
      invariant { !s1*(at !a 'Begin) + !t1*(at !b 'Begin) = !b > 0 }
      invariant { gcd !a !b = 1 }
      variant { !b }
      let q = div !a !b in
      (t0,t1) := (!t1, !t0 - q * !t1);
      (s0,s1) := (!s1, !s0 - q * !s1);
      (a,b) := (!b, mod !a !b)
    done;
    !t1
```

In this case, the extended Euclidean algorithm is being followed, using the table-driven method [40]. It is clear that in this code, *actually* computing `s0` and `s1` serves no purpose and can be erased from the function without altering its behaviour. However, actually doing this would hinder verification by making it harder to state an easily verifiable loop invariant.

Why3 supports *ghost code* precisely for situations such as these [41]. Expressions and declarations (including record fields and function parameters) can be marked as `ghost`, which then becomes part of their type. Why3's type system ensures *non-interference*: ghost code is not allowed to influence non-ghost code, or have non-ghost side-effects. This also means that ghost code, unlike non-ghost code, is required to be terminating.

The 'ghostness' of an expression propagates through all WhyML code that is exposed to it, making it ghost as well, and subject to the same restrictions. For example, an ordinary function being passed a ghost value as an argument is considered a ghost function for the duration of that call. The benefit of ghost code is two-fold:

1. If using WhyML as an intermediate language, ghost code can be safely added anywhere to ease verification, since Why3's type system ensures it will have no effect on the non-ghost part of the program.

2. If Why3 is being used to extract OCaml programs from WhyML code, ghost-code is erased from the resulting program.

Note that, as mentioned in [41], it can be useful to think of ghostness as a property pertaining to information flow. Information that is contained in ghost code is 'classified' — it is unable to find its way or influence towards non-ghost or 'unclassified' code.

2.3.4. Writing proofs using `let lemmas`

Given a ghost function with precondition P and postcondition Q that returns nothing, it is clear — due to the non-interference property — that it can be called explicitly at any point in a WhyML program when P holds to obtain the knowledge that Q also holds. Logically, this is equivalent to having the implication $P \rightarrow Q$ everywhere in a program.

To be able to use such an implication directly, and implicitly, WhyML supports the concept of `let lemma`'s. When a WhyML procedure is declared as such, it is bound to the same restrictions as other ghost procedures, but in addition the implication captured by its pre- and post-conditions is added to the set of logical facts, which can then be used to prove other facts anywhere in a WhyML module.

Another way of viewing this is that the function body of a `let lemma` serves as the *proof* for the logical statement implied by its pre- and post-conditions — instead of using logic to verify properties of WhyML programs, we can use WhyML programs to prove logical facts. While complicated proofs can also be discharged using interactive theorem provers, proofs using `let lemmas` have the advantage of being part of the Why3 environment, and having a more declarative feel to them, both of which makes them easy to use.

For example, suppose we want to use the fact that if two sorted sub-ranges of an array are a permutation of each other, that they are in fact identical. As often with such 'obvious' truths, no automatic prover seems able to prove it. However, the recursive `let lemma`:

```
let rec lemma sorted_permut_eq (a b: array int) (l u: int)
  requires { permut a b l u /\ sorted_sub a l u /\ sorted_sub b l u }
  ensures { array_eq_sub a b l u }
  variant { u }
= assert { permut b a l u };
  if l < u then begin
    assert { a[u-1] = b[u-1] };
    sorted_permut_eq a b l (u-1);
  end
```

can be quickly proven by Alt-Ergo. As a corollary, it can now be quickly proven by CVC3, CVC4 and Z3 that:[1]

```
lemma sorted_permut_all:
  forall a b: array int. permut_all a b /\ sorted a /\ sorted b -> array_eq a b
```

Since in this way, WhyML programs can be used to influence logical facts known at the level of its Why sub-language, it also provides a means for WhyML programs to define functions in Why in an indirect way — by defining a Why function as returning an arbitrary element of a set of outcomes, and then proving that this set is not empty by writing a `let lemma`, thereby proving that the function is well-defined.

For example, suppose we want to introduce a log function for integers. Directly writing this as a Why function is impossible, as it would necessitate a recursive definition that Why3 cannot prove to be terminating:

```
function log (base x:int): int
  = if x < base then 0 else 1 + log base (div x base)
```

But we *can* define a predicate defining the *relation* between the input and result of the integer log function:

```
predicate is_log_of (base x result: int)
  = result >= 0 /\ power base result <= x < power base (result+1)
```

We can then use this as described to create a function:

```
function log (base x: int): int
  = choose (comprehension (\e. is_log_of base x e))
```

But to be able to use this function meaningfully, we need to show that it is well-defined, i.e. that there are circumstances in which the set used in its definition is not empty. We do this using a constructive proof in WhyML. It is easy to define a suitable log function function imperatively:

```
let log (base x: int): int
  requires { base > 1 /\ x > 0 }
  ensures { is_log_of base x result }
= let (n, y) = (ref 0, ref 1) in
  while !y <= x do
    invariant { 0 <= !n /\ 0 < !y = power base !n <= x*base }
    variant { x - !y }
    n += 1; y *= base;
  done;
  (!n-1)
```

[1]Note that there is no extensionality principle for arrays — it cannot be deduced that `a = b`.

So, given the stated preconditions, this function produces a witness for the fact that the set of outcomes contains at least one element, which is precisely what we need. And although this is a perfectly ordinary WhyML function, it can be used from within a `let lemma` to produce this witness as well:

```
18  let lemma log_is_well_defined (base x: int)
19    requires { base > 1 /\ x > 0 }
20    ensures { is_log_of base x (log base x) }
21  = log base x; ()
```

Note again that the `log` identifier on line 21 calls the WhyML function, whereas the identifier on line 20 denotes the Why function.

Of course, we could also have defined the `log` function by stating the proposition proven by the `let lemma` using an axiom; but the above approach has the advantage that it does not allow us to introduce inconsistencies in the system.

3. WhyML verification case studies

In this chapter, we will apply Why3 to three verification tasks. In the first task, we will verify heapsort implemented in WhyML, and then show how Why3 can also be used to prove its complexity. Secondly, we will use WhyML as a modelling language to finish the proof for the small assembly program encountered in Section 1.5. Thirdly, we will examine how WhyML can be used to perform reachability analysis, which we will need later for verifying liveness in Go programs. We set ourselves the target of using only automated proofs to discharge the resulting proof obligations.

3.1. Verifying heapsort

Heapsort is a well-known comparison sorting algorithm with a worst-case (and average-case) complexity of $O(n \log n)$ comparisons and swaps. Together with quicksort, it is widely used in practical software, as it forms one part of the introsort algorithm [42] that is used in the GNU C++ standard library.

Heapsort has been verified before in Why3 [43], but in a indirect manner — by first creating a model for binary heaps in Why3 and then building a heapsort using that. This method also requires manual proofs in the Coq assistant to prove certain lemmas.

We set ourselves a more direct target: verify the implementation of the heapsort algorithm as it is commonly presented, by directly manipulating an array representing the heap.

To be a correct sorting algorithm, three conditions must be met by a procedure:

1. Elements in the final array appear in order.

2. The final array is a permutation of the array given as input.

3. The procedure must terminate.

All conditions are necessary; if we drop one, it is not hard to come up with an algorithm which can be proven (in Why3) to satisfy the remaining two but might fail to sort the input. The last requirement, it could be argued, is something of an artefact of Hoare logic: a postcondition only specifies what happens *if* a procedure terminates, and so termination is needed to ensure they will actually hold. On the other hand, since a sorting algorithm should terminate when it has achieved its purpose, and every array can be sorted (given a suitable partial order), it also follows for that reason that termination is required.

An abstract WhyML function for sorting an array of integers which satisfies these conditions can be specified as follows:

```
val sort (a: array int): unit
  writes { a }
  ensures { forall n m: int. 0 <= n <= m < length a -> a[n] <= a[m] }
  ensures { permut_all (old a) a }
```

Here, the array is passed by reference and modified in-place; the first post-condition states that the resultant array is sorted. The second post-condition uses a predicate from the `array.ArrayPermut` module to express that the output is a permutation of the input. Another, stronger predicate we will use later on is `permut_sub`, which not only expresses that two arrays are permutations of each other, but also that they are an exact match outside a given range.

Heapsort operates on the principle of a binary heap, which is a fully balanced binary tree where the value stored in every node is guaranteed to be not greater than the value stored in its parent node — thus, the root of the heap contains its largest element. Since the heap is fully balanced, it can be represented using a simple array, where the root of the heap is at index 0, and the children of the node with index i are located at index $2i+1$ and $2i+2$. A careful implementation can use the same array that is being sorted for (temporarily) storing the heap.

The heapsort implementation we will consider consists of three routines:

- A function `make_heap` which permutes the input array so that it initially satisfies the heap property.
- The function `sort`, which extracts the top element from a heap, and places it at the end of the array, thereby shrinking the heap. The previous element found at that location is re-inserted into the heap.
- An auxiliary routine, `push_down` which moves an element downward in the heap to the correct location.

This last function is the most complicated routine of the heapsort algorithm, and so we will consider it first; it is listed in Figure 3.1. When calling this function, the input array should satisfy the heap property for the range $[pos, limit)$, except that the element at location *pos* may be in an incorrect location, to be moved further down. When it terminates, it has re-established the heap property. Three auxiliary definitions are used in its annotations:

```
function parent (pos: int): int = div (pos-1) 2

predicate heap (a: array int) (l u: int) =
  forall i. l <= parent i /\ i < u -> a[i] <= a[parent i]

predicate broken_heap (a: array int) (l u: int) (h: int) =
  (forall i. l <= parent i /\ i < u /\ parent i <> h -> a[i] <= a[parent i]) /\
  (forall i. l <= parent h < h = parent i /\ i < u -> a[i] <= a[parent h])
```

The `heap` predicate expresses that the array `a` satisfies the heap property (as discussed informally above) for indices in the range $[l, u)$. The `broken_heap` predicate relaxes this somewhat, to allow the heap to have a *hole* node at index h. This value of its contents should still not be greater than any of its parents, but other than that it is not considered part of the heap. The other listed conditions and loop invariants concern themselves with ensuring that all array indices remain within bounds, and that the function only permutes its input, and should be obvious.

By successive calls to the `push_down` function, we can incrementally construct a heap out of any array. The `make_heap` function listed in Figure 3.2 demonstrates this. In this case, the postconditions and loop invariants are simply a succinct explanation of what the function is doing.

The `sort` function, finally, combines the two other functions — it calls `make_heap` to transform the input into a heap, and at each step, after extracting an element from the heap, re-instates the heap property by a call to `push_down`. Two first two loop invariants deserve special attention. The first states that the the output array is slowly becoming sorted, and is necessary for achieving the postcondition. The second captures the consequence of extracting the largest element of the heap at each step — namely, that all elements remaining in the heap will be less than the elements already extracted.

To verify this second loop invariant, we need the fact that, indeed, the top element of the heap is a largest element of the heap. To prove this requires some extra manipulation in Why. First, we define an inductive predicate which corresponds with the transitive closure of the parent-child relation in the heap:

```
inductive descendant int int =
| Refl: forall n: int. descendant n n
| Step: forall p q: int. p <= q /\ descendant p (parent q) -> descendant p q
```

This can be used to prove the following consequence of the heap property:

```
lemma descendant_order:
  forall a: array int, limit p q: int.
    0 <= p <= q < limit <= length a /\ heap a p limit /\ descendant p q -> a[p] >= a[q]
```

Which can be easily proven by automatic provers after applying the `induction_pr` transformation. We also need that all indices are a descendant of the root index 0; which is, of course, a simple case of applying the principle of strong induction:[1]

```
predicate descendant_of_0 (n: int) = descendant 0 n
clone int.Induction with predicate p = descendant_of_0, constant bound = zero
```

Both facts combined imply the desired property about heaps we need to help finish the proof of the `sort` function:

```
lemma heap_top:
  forall a: array int, limit m: int.
    0 <= m < limit <= length a /\ heap a 0 limit -> a[0] >= a[m]
```

Trying to verify all three routines, we discover that the hardest remaining challenge for automated provers turns out to be proving the `permut_all` and `permut_sub` loop invariants —- which may seem rather odd, as a simple static analysis could already determine that we only ever modify the array by swapping two elements. However, adding the following last lemma solves this problem:

```
lemma permut_sub_transitive:
  forall a b c: array int, l u: int.
    permut_sub a b l u -> permut_sub b c l u -> permut_sub a c l u
```

This completes the effort needed for verification — the resultant module can be completely verified by automated provers (after suitably using the `inline_goal` and `split_goal` in some choke points), using a combination of Alt-Ergo, CVC4 and Z3, in less than a minute of CPU time.

This case study also demonstrates another advantage of using Why3 for verification of software — after finding a succinct set of annotations, the resultant program is not just verified, but also self-documenting. In principle, anyone familiar with first-order logic and programming should be able to understand (roughly) why a WhyML program is correct, even without running Why3.

3.1.1. Verifying the complexity of heapsort

In addition to verifying the *functional correctness*, we may also be interested in a non-functional property of a program. In the case of heapsort, such a property is the claim that it sorts an array using $O(n \log n)$ comparison.

To verify this with WhyML, we can introduce a ghost counter, and a function which acts like an the identity function, but as a side effect updates the ghost counter:

```
val ghost count: ref int

let event (expr: 'a): 'a
  ensures { !count = old !count+1 /\ result = expr }
= count += 1; expr
```

[1]Instead of cloning `int.Induction`, we could also have used a simple `let lemma`

```
let push_down (a: array int) (pos: int) (limit: int)
  requires { 0 <= pos < limit <= length a }
  requires { broken_heap a pos limit pos }
  ensures  { heap a pos limit }
  ensures  { permut_sub (old a) a pos limit }
= 'Start:
  let cur = ref pos in
  while !cur*2+1 < limit do
    invariant { pos <= !cur }
    invariant { broken_heap a pos limit !cur }
    invariant { permut_sub (at a 'Start) a pos limit }
    variant { limit - !cur }
    let child = ref (!cur*2+1) in
    if !child+1 < limit && a[!child] < a[!child+1] then
      child := !child+1;
    if a[!cur] < a[!child] then begin
      swap a !cur !child;
      cur := !child;
    end else
      cur := limit;
  done
```

Figure 3.1.: Heap insertion function

```
let make_heap (a: array int)
  ensures { heap a 0 (length a) }
  ensures { permut_all (old a) a }
= 'Start:
  for i = div (length a) 2-1 downto 0 do
    invariant { heap a (i+1) (length a) }
    invariant { permut_all (at a 'Start) a }
    push_down a i (length a);
  done
```

Figure 3.2.: Heap initialization function

```
let sort (a: array int)
  ensures { forall n m: int. 0 <= n <= m < length a -> a[n] <= a[m] }
  ensures { permut_all (old a) a }
= 'Start:
  make_heap a;
  'Loop:
  for i = length a-1 downto 1 do
    invariant { forall n m: int. i < n <= m < length a -> a[n] <= a[m] }
    invariant { forall n m: int. 0 <= n <= i < m < length a -> a[n] <= a[m] }
    invariant { heap a 0 (i+1) }
    invariant { permut_all (at a 'Start) a }
    swap a 0 i;
    push_down a 0 i;
  done
```

Figure 3.3.: The selection sort loop

To verify the complexity of heapsort then only requires wrapping the array element comparisons in the function `push_down` (on lines 14 and 16) in a call to the `event` function, and adding appropriate post-conditions and loop invariants to each function.

For instance, utilizing the `log` function as we have defined in section 2.3.4, we can add the following loop invariant to the function `push_down`:

```
invariant { !count - at !count 'Start <= 2*log 2 (if !cur >= limit
                                                   then limit else !cur+1) }
```

which is sufficient to prove a new postcondition:

```
ensures { !count - old !count <= 2*log 2 limit }
```

By applying the same process to `make_heap` and `sort`, we can then obtain as final post-condition for the function `sort`, proving its bound of $O(n \log n)$ comparisons:

```
ensures { let n = length a in !count - old !count <= 4*n*log 2 n }
```

Adding these extra invariants and post-conditions comes at a price, however. While the new post-conditions and loop invariants dealing are discharged easily enough by automated theorem provers, the extra informational 'baggage' means that several of the original verification conditions can no longer be verified automatically (within a reasonable amount of time), even though we added nothing to the non-ghost semantics of the program.

To re-establish full automatic verification the proof, we need to add several assertions to the code, as well as allowing automated provers to run longer. Verifying the heapsort algorithm in this manner requires just over three minutes of CPU time on our test machine. The resulting module is listed in Appendix A.

3.2. Verifying simple assembly programs

In Section 1.5, we briefly mentioned Turing's proof attempt for a function which computes $n!$, gave a modern assembly language version in Figure 1.1, and stated several assertions that were claimed to be sufficient to prove its correctness, but left the full proof as an exercise to the reader.

We will now show how we can complete the verification by translating the given x86-64 code into WhyML. To do this, we need to create an ad-hoc model for the instructions used, and find some way to model the unstructured control flow into WhyML. The approach that we take here consists of the following parts:

- Partition the assembly code in *basic blocks*, each of which starts at a labelled instruction and ends right before the next labelled instruction.
- Translating the instructions inside each *basic block* in a one-to-one correspondence into WhyML statements.
- Wrap the translated blocks in a simple state machine that handles the transition from one *basic block* to the next one.

Verification of assembly language using Why3 has been previously examined in more depth in [44], in the context of MIX programs. There is a definite similarity here; the prime difference is that we use WhyML primitives instead of a more explicit *sequentialization* process, and that we will also prove termination of the program given.

We begin with building a crude, but sufficient model for the arithmetic operations used in the code, which includes the *carry* and *zero flags*:

```
val cf: ref bool
val zf: ref bool

val cmp (a b: int): unit
  writes { cf, zf }
  ensures { a = b <-> !zf }
  ensures { a < b <-> !zf }

val add (a b: int): int
  writes { cf, zf }
  ensures { result = a+b }

val inc (a: int): int
  writes { cf, zf }
  ensures { result = a+1 }
```

Note that, like Turing, we ignore the representational issues of numbers on computers, i.e. possible issues caused by two's complement and wrap around.

Next, we define an algebraic data type corresponding with the labels present in assembly code (for use in our state machine), and two exceptions which are used to model `jmp` and `ret` instructions.

```
type label = L0 | L1 | L2 | L3

exception Jump label
exception Return
```

The idea here is most instructions will be executed sequentially; a jump should interrupt this flow, and exceptions are a natural means of expressing this in WhyML. After catching the `Jump` exception, it is easy to transfer execution in the WhyML model to the corresponding basic block.

We can also reach a label by simply falling into it after executing the instruction preceding it; in this case we need to transfer control (in our state machine) to the next basic block, and so we need to explicitly specify the order of the program labels:

```
constant undefined: label

function next (l: label): label =
  match l with L0 -> L1 | L1 -> L2 | L2 -> L3 | L3 -> undefined end
```

And now all that is left is translating the assembly code of Section 1.5 and the associated assertions into WhyML. For notational convenience, the CPU registers are modelled as an array of size 16. Since we will be using the `Return` exception to signify the conclusion of the program, we specify the result in the form of an exceptional post-condition.

```
let x86fac (n: int) (r: array int)
  requires { length r = 16 }
  requires { r[10] = n >= 0 }
  raises { Return -> r[11] = fact n }
= let ip = ref L0 in
  loop
    invariant { n = r[10] }
    invariant { !ip=L1 -> 1 <= r[9] /\ r[11]=fact r[9] /\ (r[9] > n -> r[9]=1) }
    invariant { !ip=L2 -> 1 <= r[8] <= r[9] < n /\ r[12]=fact r[9] /\ r[11]=r[8]*fact r[9] }
    invariant { !ip=L3 -> r[11]=fact n }
    try
      match !ip with
      | L0 -> r[9] <- 1;
              r[11] <- 1;
      | L1 -> r[12] <- r[11];
```

```
                cmp r[9] r[10];
                if not !cf then raise (Jump L3);
                r[8] <- 1;
        | L2 -> r[11] <- add r[11] r[12];
                r[8] <- inc r[8];
                cmp r[8] r[9];
                if !cf || !zf then raise (Jump L2);
                r[9] <- inc r[9];
                raise (Jump L1);
        | L3 -> raise Return
        end;
        ip := next !ip
      with
        Jump pos -> ip := pos
      end
    end
```

This code has been written to closely resemble the assembly version given in Figure 1.1: every instruction maps to precisely one WhyML statement. The invariants are the same as the assertions given in Section 1.5.

Starting a Why3 proof session with the above definitions, we find that the verification condition for `x86fac` can be verified by Alt-Ergo in a couple of seconds, without any proof transformations.

3.2.1. Proving termination

So far, we have ignored termination of this program. But, as Turing also pointed out:

> *Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.*

He then proposes the quantity $(r_{10} - r_9)\omega^2 + (r_9 - r_8)\omega + k$, where k is a quantity which is decreasing for successive labels [1]. This is equivalent to using a lexicographic ordering on these three separate expression, which is supported by Why3.

Sadly, plugging this expression into the above model, we will discover that it is incorrect: if we execute the basic block corresponding with the label `L1`, the register `r8` gets set to 1. In many cases this will cause the quantity $r_9 - r_8$ to increase. This seems to be an error on Turing's part, as the same problem occurs in his original diagram if we move from state C to state E (see Figure 1.2 on page 11). A smaller, more pedantic problem is the initialization that occurs at the beginning of the program: we are not told the initial value of r_9, and so $r_{10} - r_9$ cannot be proven to decrease when executing the basic block corresponding to label `L0`.

Luckily, it is possible to identify a different quantity: the algorithm increments r_{11} until it reaches $n!$, and so we can use this as the basis for proving termination. To be precise, we can add the following variant to the `loop` above:

```
      variant { fact n - if !ip=L0 then 0 else r[11], 3 - (to_int !ip) }
```

Which defines a lexicographic ordering on the pair $n! - r_{11}$ (avoiding the unspecified value of r_{11} at the start of the program), and a value which decreases with successive labels. In this definition we use the following trivial translation from program labels to integers:

```
function to_int (l: label): int =
  match l with L0 -> 0 | L1 -> 1 | L2 -> 2 | L3 -> 3 end
```

Proving that this variant is strictly decreasing in the above code is relatively easy even by hand. However, automated theorem provers find it difficult to show that the quantity $n! - r_{11}$ remains non-negative, and so we need to add the following lemmas stating trivial facts about the factorial function:

```
let rec lemma fact_monotone (n m: int)
  requires { n >= m >= 0 }
  ensures { fact n >= fact m >= 0 }
  variant { n }
= if n > m then fact_monotone (n-1) m else
  if n > 0 then fact_monotone (n-1) (n-1)

lemma fact_order:
  forall n: int. 0 <= n -> n*fact n < fact (n+1)

lemma fact_order_corollary:
  forall n m k: int. 0 <= k <= m < n -> k*fact m < fact n
```

Alt-Ergo can verify most of these; E is needed to prove the post-condition of the `let lemma`.

Using these lemmas, Alt-Ergo is able to prove correctness *and* termination of the factorial function written in assembly. The total CPU time needed for verification does not exceed one minute.

3.3. Reachability analysis using WhyML

WhyML provides the `absurd` keyword to allow us to prove that code may be unreachable; it is equivalent to an `assert { false }` statement. For example, in the following loop, it is clear that `x` cannot ever be higher than 1000:

```
for x = 0 to 1000 do
  if x > 1000 then absurd;
done
```

However, we would like to also reason about the *reachability* of code. In simple cases we can solve this problem by introducing a `ghost bool` variable, which is set to true at the location whose reachability we want to prove, and subsequently check at the end of the function whether the `ghost bool` variable has become true. However, this technique will not work if a program may contain non-terminating code:

```
let reached = ref false in
let x = ref 0 in
while true do
  if prime !x then begin
    (* this code is reachable *)
    reached := true
  end;
  x := !x+1;
done;
check { !reached }
```

Here, the final `check` is useless, since the post-condition of the `while` loop will allow us to prove anything. This indicates the need to perform the check at an earlier point. This involves a kind a 'backward' reasoning that is not natively supported by WhyML. In this section we propose a logical *hack* which does allow us to do this.

The general idea is that we add a hidden state variable, called `obligation`, which keeps track of logical facts that must provably hold at the end of a function:

```
type obligation model { ghost mutable value: bool }

val ghost obligation: obligation

function (?) (x: obligation): bool
  = x.value

val ghost init (): unit
  writes { obligation }
  ensures { ?obligation }

val ghost oblige (x: bool): unit
  writes { obligation }
  reads { obligation }
  ensures { ?obligation <-> x /\ old ?obligation }
```

We declared this variable as ghost to prevent it from having an effect on ordinary WhyML code. The above code is not special at all: an `obligation` is essentially a `ref bool` with a restricted set of operations. The trick lies in applying it to propositions that are undecidable at the point of call to `oblige`, but which become provable at the locations we are interested in. By recording these propositions in the global variable `obligation`, they are tracked throughout the program until we eventually reach those points.

For example, we can define an abstract type and predicate to reason about locations and whether they have been reached, and a WhyML abstract function allowing the use of this predicate at the WhyML level as a ghost value:

```
type location
predicate reached location

val ghost reached (id: location): bool
  ensures { result = reached id }
```

We can then use this to show that the following `for` loop will eventually hit a prime number:

```
  init();
  let x_prime = any location in
  oblige (reached x_prime);
  for x = 0 to 1000 do
    invariant { x > 37 -> ?obligation }
    if prime x then begin
      assume { reached x_prime } (* reachable code *)
    end
  done;
```

The loop invariant ensures that, after we have hit the number 37, all obligations have been fulfilled. However, the same technique can now also be applied to non-terminating code: replacing the `for` loop with a `while` will not alter the validity of the loop invariant.

We can also reason about a statement being reached *multiple* times, instead of just once. In this case, we need to alter the `location` type to make it mutable:

```
type dummy
type location = { ghost mutable dummy: dummy }
```

and introduce an abstract function to replace the `assume` statement in the previous example, which also produces a fresh proposition whose truth value is again undecidable:

```
 val ghost satisfy (id: location): unit
   writes { id }
   ensures { old (reached id) }
```

This can then be used to show that, in the previous `for` loop, whenever we find a prime number less than 37, we will find another prime:

```
   let x_prime = any location in
   oblige (reached x_prime);
   for x = 0 to 1000 do
     invariant { ?obligation <-> (x > 37 \/ reached x_prime) }
     if prime x then begin
       satisfy x_prime;
       if x < 37 then
         oblige (reached x_prime);
     end
   done;
   check { ?obligation }
```

In this case, the `check` at the end of the `for` loop will succeed, ensuring that all obligations that were entered into during execution are fulfilled.

It can also be useful to show that in non-terminating programs, certain statements can be reached *infinitely many* times. In the case of the prime number loop we have been using, this obviously requires a Why3 proof that there are an infinite number of primes, which can be done in a remarkably declarative fashion, as shown in Figure 3.4. This module can be proven automatically using a combination of Alt-Ergo, CVC3 and E.

In a non-terminating program, the final `check` can obviously not be relied upon. In order to show that obligations will be fulfilled, we instead introduce the notion of a *conditional variant*. This is a quantity which must be shown to decrease during each iteration, unless some initial condition became provable. We would like to denote this in WhyML using the syntax:

```
   variant { not condition -> quantity }
```

But since this is not available, we can emulate its behaviour by inserting a `check` at the end of the `while` loop which generates exactly the same verification condition as WhyML's `variant`, but with the implication added. When a `while` loop has a conditional variant, we know that one of two things must be the case: either the condition specified in the variant must eventually become true in a finite number of iterations, or the loop will terminate before it reaches that point.

In the case of the prime number loop we have been demonstrating, a suitable *conditional variant* is the distance between `x` and some arbitrary prime number greater than `x`, which we can maintain as a ghost variable, to conceptually use the *conditional variant*:

```
     variant { not ?obligation -> !next_prime - !x }
```

In actual WhyML, this approach results in the following program:

```
   init();
   let x_prime = any location in
   let ghost next_prime = ref 2 in
   let x = ref 0 in
   oblige (reached x_prime);
   while true do
     invariant { ?obligation <-> reached x_prime }
     invariant { prime !next_prime /\ (!x < !next_prime \/ prime !x) }
 'Begin:
     if prime !x then begin
```

```
        next_prime := some_larger_prime !x;
        satisfy x_prime;
        oblige (reached x_prime);
      end;
      x := !x+1;
      check { "expl:conditional variant decrease"
              not (at ?obligation) 'Begin ->
              0 <= at (!next_prime - !x) 'Begin /\
              (!next_prime - !x) < at (!next_prime - !x) 'Begin
      };
    done;
```

In general, the notion of a *conditional variant* is applicable to more than the examples we have given. For example, in a lot of software, simple termination is not an interesting property, or even an undesirable one — it is much more important that a program does not *hang*, i.e. enter a state from which point on it will never be able to produce any side-effects. By specifying a conditional variant, such properties can be proven.

```
module InfinitePrimes

use import int.Int
use import number.Prime
use import number.Divisibility
use import ref.Ref

let rec fac (n: int): int
  ensures { 0 < result /\ forall p. 1 <= p <= n -> divides p result }
  variant { n }
= if n > 1 then n*fac(n-1) else 1

let lemma finite_prime_contradiction (n: int)
  requires { forall p: int. prime p -> p <= n }
  ensures { false }
= let prod = fac n in
  assert { exists p: int. prime p /\ divides p prod /\ divides p (prod+1) }

goal Euclid:
   forall n: int. exists p: int. p > n /\ prime p

use import set.SetComprehension

function some_larger_prime (n: int): int
  = choose (comprehension (\p: int. p > n /\ prime p))

lemma some_larger_prime_def:
  forall n: int. some_larger_prime n > n /\ prime (some_larger_prime n)

end
```

Figure 3.4.: WhyML code to prove the existence of an infinite number of primes

4. The Go programming language

The Go programming language [26] was developed at Google by Rob Pike, Ken Thompson and Robert Griesemer, and had its first major release in 2012. It is firmly the Algol tradition of languages, with a static type system and imperative constructs. Its syntax owes much to C, and is in some respects even more spartan in nature by leaving out most symbols not necessary for disambiguating a program; it can be viewed as a modern re-imagining of a higher-level version of C.

The most interesting aspect of Go is perhaps that, unlike most programming languages which *add* features, Go's actually *removes* features deemed to be unnecessary and tries to be simpler. Its designers stress that it is not an experimental language, and that Go in fact does not contain any fundamentally new ideas about programming; its language definition has, as a result, changed very little after its initial release.

The rationale given for the development of Go is that the complexity of modern programming languages hinders productivity. For example, complex type hierarchies in object-oriented programming languages must usually be thought out before all information that should influence their design is known; and changing such a design later is costly. The complexity of programming languages also influences the toolchain which must be used to implement them, which as a result becomes less efficient. For example, in C++, compilation may actually involve running *meta-programs* at compile-time.

In this chapter, we will briefly demonstrate Go syntax, and list some of its interesting features, before taking a more in-depth view into its concurrency model. It is assumed the reader is familiar with common higher level programming languages.

4.1. A Go primer

The syntax of Go is essentially a simplified and reduced version of that of C. For example, in an `if` or `for` statement, the parentheses around the condition are no longer necessary. Also, while the grammar formally requires a semicolon, in many cases the lexer automatically inserts these while processing a source file; allowing us to write:

```go
package main

import "fmt"
import "os"

func main() {
    if len(os.Args) <= 1 {
        fmt.Println("Hello world!")
    } else {
        for i := 1; i < len(os.Args); i++ {
            var str string = os.Args[0]
            fmt.Println("Hello", str)
        }
    }
}
```

Since the lexer for Go inserts semicolons at the end of lines, a K&R coding style — with the opening brace at the end of the line the starts a block — is enforced in Go. Also, the braces in the if statement are not optional, unlike in C.

Another interesting difference between Go and C is its style of declarations. On line 11, the keyword var introduces the variable str of type string, and initializes it. Go can deduce the type of the variable from its initializing expression, and so we might as well have written:

```
11                  var str = os.Args[i]
```

In fact, Go offers an even more terse syntax, called the *shorthand declaration*, achieving the same:

```
11                  str := os.Args[i]
```

Go's type system supports many of the same types that C does. One difference is that, unlike C, but much like Pascal, arrays are a primitive type, and their (fixed) size is part of their static type; e.g. an array of four integers ([4]int) is a different type than [5]int. Since this would make arrays almost unusable [45], Go adds a second array-like type in the form of a *slice*, which present a *view* on some underlying array. A slice type is declared by simply leaving out the length from an array declaration. Slices can be used to more safely handle many situations which would require pointer arithmetic in C:

```
package main

import "fmt"
import "errors"

func binsearch(s string, a []string) (string, error) {
    for len(a) > 0 {
        mid := len(a) / 2
        switch {
        default:
            return a[mid], nil
        case s > a[mid]:
            a = a[mid+1:]
        case s <= a[mid]:
            a = a[:mid]
        }
    }
    return "", errors.New(s + " not found!")
}

var names = []string{"Arthur", "Ford", "Trillian", "Zaphod"}

func main() {
    s, err := binsearch("Marvin", names)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(s)
}
```

Other things to note in this example:

- Go does not have a distinct while loop; a for loop is used instead in which the first and third expressions are eliminated.

- The switch statement is generalized, and does not have the automatic fall-through behaviour as in C.
- Functions can return multiple values; the illustration of using an error value to indicate whether an item has been found is in fact idiomatic in Go, since the language does not support exceptions.

Go's syntax for types is entirely different from C, and perhaps closest in notation to Algol68 [46]; they can be read simply from left to right. To take an extreme example:

```
type continuation func ([]*int) (*[64]func(int), continuation)
```

This declares continuation to be the type of a function which takes a slice of pointers to int as an argument and returns a pointer to an array of 64 functions (which take an int and return nothing), and another object of the type continuation.

4.1.1. Memory allocation

Go is a garbage collected language, and there is no need to explicitly mention how objects get allocated — static analysis is used instead to determine which objects may leak out of a function. Those that do not leak are simply allocated on the stack. For instance, to create a simple linked list, it suffices to use:

```
type list struct {
    name string
    next *list
}

func mklist(args ...string) (result *list) {
    tail := &result
    for _, s := range args {
        obj := list{s, nil}
        *tail, tail = &obj, &obj.next
    }
    return
}
```

Note that in this case, we have also used Go's *named result* feature, as well as a *range-based* for loop to iterate over the variable number of arguments given. The compiler will find out that obj must be allocated dynamically, and will do so.

Go's memory management also allows function objects to do more than is possible using function pointers in C. For example, the following function constructs a function which could be used for appending elements to a list efficiently:

```
func builder(tail **list) func(string) {
    return func(s string) {
        obj := list{s, nil}
        *tail, tail = &obj, &obj.next
    }
}
```

So, after calling:

```
    var phil *list = nil
    f := builder(&phil)
    f("Hobbes")
    f("d'Holbach")
```

The effect would be the same as if we had simply called

```
    phil := mklist("Hobbes", "d'Holbach")
```

4.1.2. Interfaces

Go does not support object orientation in the usual sense of the word; there are no class hierarchies. However, it is possible to create *methods* for structures:

```go
func (data *list) greet(prefix string) {
    for ; data != nil; data = data.next {
        fmt.Println(prefix, data.name)
    }
}
```

Which allows us to call `mklist("Freek","Freek").greet("Hello")`. In case we implement similar methods for different object types, an interface can be created to get a form of dynamic dispatch:

```go
type Greetable interface {
    greet(string)
}
```

Now, automatically, any type which has methods conforming to this interface can be used as an instance of it. So, if we add the following type:

```go
type empty struct {}

func (e empty) greet(prefix string) {
    fmt.Println(prefix,"world")
}
```

We can write code such as:

```go
func main() {
    var x Greetable
    x = empty{}
    x.greet("Hello")
    x = mklist("Freek", "Freek")
    x.greet("Hello")
}
```

Which will output:

```
Hello world
Hello Freek
Hello Freek
```

This is similar to the use of virtual methods in object-oriented languages; except that no complicated type hierarchy needs to be created.

4.2. Concurrency in Go

The designers cite Hoare's idea of *Communicating Sequential Processes (CSP)* [28] as their primary influence for the model of concurrency implemented in Go. Instead of using threads and locks, Go's concurrency model is built around the idea of *goroutines* and channels.

A goroutine is essentially a coroutine, except that like a thread, it is also allowed to run *simultaneously* with other goroutines. Two goroutines can communicate by sending a message over a channel.

```go
package main

import "fmt"

func sieve(in chan int, out chan int) {
    myPrime := <-in
    out <- myPrime
    filt := make(chan int)
    go sieve(filt, out)
    for n := range in {
        if n%myPrime != 0 {
            filt <- n
        }
    }
}

func main() {
    outp := make(chan int)
    inp := make(chan int)
    go sieve(inp, outp)
    go func() {
        for x := 2; ; x++ {
            inp <- x
        }
    }()
    for n := range outp {
        fmt.Println(n)
    }
}
```

Figure 4.1.: Concurrent prime number sieve in Go

This communication happens synchronously; i.e., whenever a goroutine A want to send a message to B, it must wait until B has read this message before the execution of A can proceed. Channels in Go are statically typed, but may otherwise communicate any allowed data type — which includes other channels.

A key idea in Go is that *concurrency* is not just an means of making software run faster, but can also function as an organizing model for software. The idea being that large software can be understood as being composed of various independent processes which communicate with each other. Therefore, Go allows and encourages a programmer to create as many separate goroutines as needed.

Whether these processes are in fact run simultaneously is an implementation detail — the Go runtime may in fact run these processes by using simple cooperative multitasking, or use as many threads as there are goroutines, or (most commonly) a combination of both. The user has some control over the amount of threads used by setting an environment variable (`GOMAXPROCS`).

As an example of goroutines, consider the program in Figure 4.1. This program implement the Sieve of Eratosthenes (encountered in Section 2.3.1) as a concurrent program, by starting (recursively) a new goroutine for every prime number encountered; and so we will quickly be in a situation where thousands of goroutines are operating concurrently.

This is far from efficient, since the overhead involved in sending messages between so many instances of `sieve` is larger compared to the actual work performed by each goroutine. However, it does illustrates the flexibility of Go's concurrency model. A similar program in C using threads is much harder to write correctly, and starting thousands of threads will quickly overpower the operating system.

If a channel is only used by a function as an input- or output-only channel (such as in the prime number sieve), this can be indicated in its type; for example, in the prime number sieve, we would have replaced the signature of the function `sieve` with:

```go
func sieve(in <-chan int, out chan<- int) {
```

To indicate that `in` is used for input and `out` for output.[1]

Channels may be shared between any number of goroutines; if multiple goroutines are trying to send or receive on a channel, two will non-deterministically be selected for synchronization. It is also possible to wait for communication on more than one channel by using the `select` statement. For example:

```go
    select {
    case x := <-a:
        fmt.Println("received", x, "from a")
    case x := <-b:
        fmt.Println("received", x, "from b")
    }
```

This will wait for reception of a message on either channel `a` or channel `b`.

4.3. Dining Philosophers

As an illustration of Go's concurrency features, we will revisit the Dining Philosopher problem from Section 2.2.4, and show how it can be reproduced in Go. Note that in this problem, the difficulty is the presence of mutual exclusion: The forks represent a shared resource which can only be owned by (at most) a single process at a time. This notion is best captured by the use of *mutexes*. Implementing it using channels may therefore seem somewhat odd, but it also illustrates the concept of *synchronization* between goroutines nicely.

Using a **`chan int`** to communicate the availability of each fork, we can capture the behaviour of a philosopher by a goroutine that, after waiting for the reception of a message indicating the availability of the right and left fork, then sends a message to indicate that it is eating. Subsequently, to signal the return of both forks to table, a message is sent to the corresponding channels.

```go
func Philosopher(id int, eating chan int, left chan int, right chan int) {
    for {
        x := <-right
        y := <-left
        eating <- id
        right <- x
        left <- y
    }
}
```

The actual values of `x` and `y` don't matter, as we will show in a moment. Also note that the order in which the forks are put down is fixed.

The above routine seems superficially similar to a solution based on mutexes, but is different in the sense that we also require a process communicating at the other end of the provided communication channels. In the case of the `eating` channel, this can simply be a process reporting that the philosopher is eating. For the forks, it is tempting to let each `Philosopher` processes communicate

[1]Note the direction of the `<-` as either pointing to or away from the keyword `chan`

directly with its neighbours — but this would imply that philosophers don't actually put down the forks on the table, but hand each other the fork directly, because two processes must *synchronize* when communication over an unbuffered channel takes place. To get an appropriate model, we need an extra process type which represents the location on the table for each fork:

```
11  func Table(id int, fork chan int) {
12      for {
13          fork <- id
14          <-fork
15      }
16  }
```

When started as a goroutine, this routine is ready to hand one fork out to the first process that wants it; and after that, is ready to receive it back again. Again, the actual value that is sent over the channel is not important. Using these two building blocks, all that is left is to initialize the communication channels and processes:

```
18  func main() {
19      var forks [5]chan int
20      for i := range forks {
21          forks[i] = make(chan int)
22          go Table(i, forks[i])
23      }
24
25      var eating = make(chan int)
26      for i := range forks {
27          go Philosopher(i, eating, forks[i], forks[(i+1)%5])
28      }
29
30      for {
31          fmt.Printf("philosopher %d eating\n", <-eating)
32      }
33  }
```

The function `main`, which defines the start of the Go program, consists of three parts. In the first part, the variable `forks` is declared as an array of 5 channels used for sending/receiving integers. Each element of this array still needs to be initialized, which is handled in the subsequent `for`-loop. Also, for each fork, the function `Table` is started as a goroutine. The second part similarly creates the `eating` channel, and starts the philosophers. Note that in line 25, the type of the variable declaration is missing — it can be deduced from the initializing expression `make(chan int)`. We could also have used the shorthand declaration `eating := make(chan int)` at this point. The final `for`-loop should be self-explanatory.

Analysis

To see why the previous section models the Dining Philosopher problem, we can provide an informal argument consisting of three relevant observations about the given code fragments:

1. We assign a notion of *ownership* to each fork. Initially each fork is owned by a `Table` goroutine.
2. Ownership is transferred by sending a message on the channel corresponding to the fork. By inspecting the code we may convince ourselves that only a single process can be the owner of a fork at any given time.
3. Lastly, obviously a `Philosopher` can only eat when it is in possession of both forks, and it relinquishes ownership of both after having eaten.

This doesn't mean that every fork is always transferred to its `Table` process when a `Philosopher` puts it down: if multiple goroutines are trying to receive (or send) a message, the scheduler will arbitrarily select one which succeeds, making the other goroutines wait. Since both a `Table` and `Philosopher` process may be listening on a channel corresponding to a fork, it may happen that a fork is transferred directly between philosophers, as described earlier. In this sense, the `Table` processes are just there to allow a philosopher to put down a fork regardless of the appetite of its neighbour.

When this program is actually executed on a system with only a single thread (i.e., `GOMAXPROCS = 1`), the output of the program will show that each philosopher gets to eat in turn, with the deterministic behaviour of the scheduler unintentionally avoiding the deadlock scenario. However, if we increase the number of threads available (`GOMAXPROCS ≥ 2`), it will eventually happen that the `main` function is waiting for a philosopher to start eating, whilst all philosophers are waiting for the availability of a fork. When this happens, the program will exit with a fatal error, complaining about a deadlock.

The observant reader may wonder why we have not used a `fmt.Printf` call in the body of a `Philosopher` directly, instead of sending a message to the `eating` channel, only for it to get translated into a print statement body of `main` anyway. The answer has to do with the execution model of Go:

1. By making `func main` wait on a channel serviced by the `Philosophers`, we get a runtime error when no goroutine can proceed.

2. If we replace the final loop in `main` with an empty loop (`"for {}`"), the `Philosopher` goroutines are not guaranteed to get CPU time, unless we yield it explicitly by using the runtime function `runtime.Gosched`.

This is the result of a goroutine being a hybrid between a coroutine than a thread: Go will only execute them if there is a thread available, and one sure way to make one available is by making the current process dependent on its execution. This feature of goroutines is important to keep in mind, as it implies that not all goroutines are created equally: the `main` function comprises the only thread whose progress of execution we are to actually care about. Whenever it communicates via a channel, this responsibility for progress is passed on to the goroutine(s) connected to that channel, which may in turn pass it on to other goroutines, and so on. This aspect of communication being directly linked to the notion of progress heavily informs the way we model this aspect of Go programs in Why3 in chapter 5.

5. Modelling Go with WhyML

In this chapter, we will first restrict the range of allowed Go programs to a small subset; we will then show how we can verify *weak correctness* — that is, correctness under the assumption that liveness is assured — for programs written in this subset. After that we will extend this primitive model to allow us to prove the *strong correctness* of a concurrent program.

The fragment of the Go language that we will consider is illustrated by Figure 5.1. This fragment, which we will refer to as Go$^-$, is similar to a simple WHILE-type language extended with channels. In particular, the only data types we admit are `int`, `bool` and `chan int`; furthermore, the only permissible actions on a `chan int` are sending/receiving data, and passing it to goroutines. All non-local control flow (such as `break` and `return`) are eliminated. These are not hard to implement in Why3 using exceptions, but we have not found them necessary in what follows. To keep our verification examples simple, we will pretend that the data type `int` can hold unbounded integers; modelling `int` as an integer which is at least 32 bits in size is not hard in WhyML, but would complicate proofs, and for our present discussion would be an unnecessary distraction.

```
stmt          ::= var-decl | chan-decl
                | ident = expr ;
                | expr
                | for expr block
                | if expr block else block
                | go expr ;
                | send-or-recv ;

block         ::= { stmt* }

var-decl      ::= ident := expr ;

chan-decl     ::= ident := make(chan int) ;

send-or-recv  ::= ident <- expr | ident := <- expr

fun-decl      ::= func ident ( [ param-list ] ) [type] { stmt* return [expr]}

param-list    ::= ident type [, param-list]

type          ::= int | bool | <-chan int | chan<- int

expr          ::= ( expr ) | unary-op expr | expr bin-op expr | fun-call
                | ident | number | true | false

unary-op      ::= - | !

binary-op     ::= + | - | / | * | % | < | <= | > | >= | == | != | && | ||

fun-call      ::= ident ( [ arg-list ] )

arg-list      ::= expr [, arg-list]

program       ::= package main ; fun-decl*
```

Figure 5.1.: Grammar of the Go$^-$ language used for verification

5.1. Static analysis of Go$^-$ programs

We place restrictions on the types of concurrency allowed in Go$^-$. These should be verified using a static analysis before verification using Why3 can commence, and are intended to bring Go more closer to a form where it is amenable to proof techniques developed for CSP:

Unidirectionality A goroutine must use each channel it is given (or that it creates) solely as an input or an output channel; so, for example, we disallow the following construction:

```
func bounce_1(c chan int) {
    x := <-c
    c <- x
}
```

No sharing of channels At any point in a program, every channel has (at most) one process responsible for reading from it, and (at most) one process responsible for writing to it, prohibiting code such as:

```
func sink(in <-chan int} { <-in }

func main() {
    c := make(chan int)
    go sink(c)
    go sink(c)  // c may not be read by two processes
    c <- 5
}
```

As well as:

```
func main() {
    c := make(chan int)
    go sink(c)
    <-c         // c may not be read by both main and sink
```

Static knowledge of data flow It must be possible to statically determine which processes are responsible for channels which are created inside a function. As a consequence, goroutines may only be started at the outermost lexical level; i.e. not inside an `if` or `for` statement. For the same reason, channels may not be copied, or passed around to or returned from ordinary functions, prohibiting constructions such as:

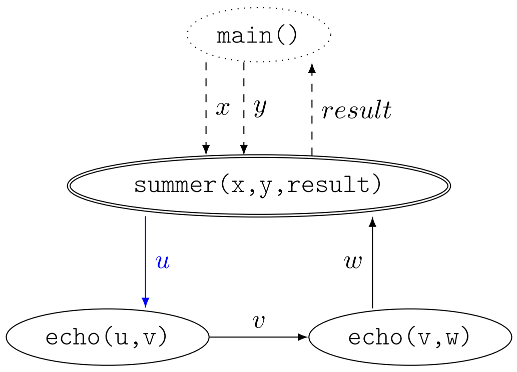
```
func foo(par bool) <-chan int {
    c := make(chan int)
    if par {
        go sink(c) // may not start a goroutine here
    } else {
        sink(c)    // cannot pass c to ordinary function
    }
}
```

These restrictions are designed to ensure that at every point during execution, we can meaningfully talk about two processes being *connected* via a channel, where one acts as a *receiver* and the other as a *sender*. In other words, at every send or receive statement we can visualize the data flow of a program as a directed graph where every channel corresponds to exactly one edge in the graph.

For example, consider the following Go code:

```
1  func echo(in <-chan int, out chan<- int) {
2      for {
3          x := <-in
4          out <- x
5      }
6  }
7
8  func summer(x <-chan int, y <-chan int, result chan<- int) {
9      u,v,w := make(chan int), make(chan int), make(chan int)
10     go echo(u, v)
11     go echo(v, w)
12     for {
13         tmp := <-x
14         u <- tmp
15         a := <-w
16         b := <-y
17         result <- a + b
18     }
19 }
20
21 func main {
22     x, y, result := make(chan int), make(chan int), make(chan int)
23     go summer(x, y, result)
24     x <- 37
25     y <- 5
26     n := <-result
27 }
```

The function main launches summer as a goroutine; As soon as its execution reaches the point at line 14, we can statically deduce the following data flow graph:



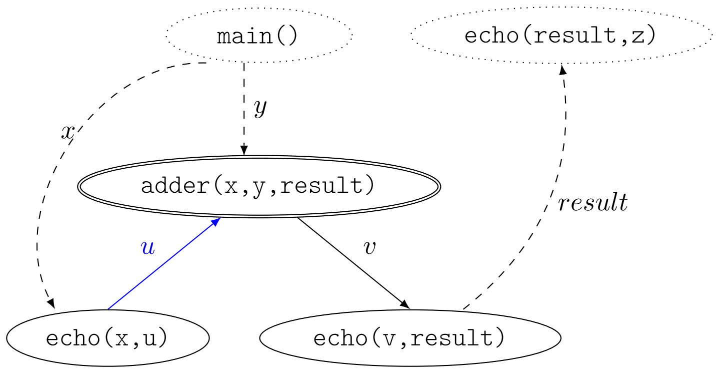
Since summer is the *current process*, i.e. whose execution we are considering, we have drawn it using double lines. The edge corresponding to the channel u has been coloured blue to indicate that it is about to be activated. After executing the send statement, the summer function will then wait for reception of a message on the channel w. This will only be able to proceed if, in between, the two instances of the echo function have communicated with each other over the channel v. We will call a channel such as v that is not directly connected to a process, but that is part of the data flow graph a *forwarding channel*, as its function is to forward the communication initiated by the current process to other processes.

The channels that were passed as arguments — x, y and z — were drawn using dashed lines to indicate that, from the point of view of the summer function, they connect to an unknown process, which we indicate by drawing the node for `main` using dots. Since communication over these channels are what drives the execution of a goroutine, we will we will call the processes that are assumed to be connected at the other end of these channels a *requesting process*. Processes that were spawned by `summer` we will denote as *child processes*, which in this case are the two instances of `echo`.

A *forwarding channel* can be connected directly to a *requesting process*. Also, a requesting process does not necessarily have to be equal to a parent process. For instance:

```
1  func adder(x <-chan int, y <-chan int, result chan<- int) {
2      u, v := make(chan int), make(chan int)
3      go echo(x, u)
4      go echo(v, result)
5      for {
6          b := <-y
7          a := <-u
8          v <- a + b
9      }
10 }
11
12 func main() {
13     x, y := make(chan int), make(chan int)
14     z, result := make(chan int), make(chan int)
15     go adder(x, y, result)
16     go echo(result, z)
17     x <- 37
18     y <- 5
19     n := <-z
20 }
```

In this case, the data flow graph of the situation at line 7 has become more complicated: A



The function `adder` now has two *requesting processes*: `main`, and an instance of `echo`. The channels `x` and `result` are now *forwarding channels*, as they do not connect to `adder`. At the chosen line, `adder` is trying to receive a message over the channel `x`, which will require the `echo(x,u)` process to in turn receive one from `main`. Note that the existence of the channel `z` is not known to the function `adder` as it can only be reached by going through a node for a *requesting process*, about which `adder` has no knowledge, and so we have left it out of the graph.

Before a channel is connected to a process, we call the channel *inoperative*; we will disallow any send or receive operation on such a channel, and so do not need to consider this case further.

5.2. Translating non-concurrent Go to WhyML

The non-concurrent parts of Go$^-$ map onto WhyML in a straightforward fashion. Let $\llbracket - \rrbracket$ denote a translation on the abstract syntax tree of a Go$^-$ program. We define this translation using Table 5.1. We use the ellipsis notation $(a_i \ldots)$ to denote repetition of a syntax element; its exact meaning on the left-hand side should be clear from the context and the grammar listed in Figure 5.1. On the right-hand side, if zero syntax elements have matched the ellipsis, its translation should result in an empty expression. Some name mangling will be needed to prevent Go identifiers from clashing with names in the WhyML environment; we indicate this by using the notation $\aleph n$ for the mangled version of the identifier n. Note that in the translation of a function declaration, we need to explicitly *copy* the function arguments, since they can be used as local variables in Go code, but not in WhyML.

$\llbracket$ **func** f (a_i τ_i, ...) $[\tau]$ { $S_j \ldots$ **return** $[r]$ } $\rrbracket$	$\overset{\text{fun-decl}}{\Longmapsto}$	**let** $\llbracket f \rrbracket$ ($\llbracket a_i \rrbracket$: $\llbracket \tau_i \rrbracket$) ... = $\llbracket a_i$:= a_i;$\rrbracket$;... $\llbracket S_j \rrbracket$;... $\llbracket r \rrbracket$
$\llbracket$ **int** $\rrbracket$	$\overset{\text{type}}{\Longmapsto}$	**int**
$\llbracket$ **bool** $\rrbracket$	$\overset{\text{type}}{\Longmapsto}$	**bool**
$\llbracket$ { $S_i \ldots$ } $\rrbracket$	$\overset{\text{stmt}}{\Longmapsto}$	**begin** $\llbracket S_i \rrbracket$;... **end**
$\llbracket$ n := x ; $\rrbracket$	$\overset{\text{stmt}}{\Longmapsto}$	**let** $\aleph n$ = **ref** $\llbracket x \rrbracket$ **in** ()
$\llbracket$ x = y ; $\rrbracket$	$\overset{\text{stmt}}{\Longmapsto}$	$\llbracket x \rrbracket$:= $\llbracket y \rrbracket$
$\llbracket$ **for** x S $\rrbracket$	$\overset{\text{stmt}}{\Longmapsto}$	**while** $\llbracket x \rrbracket$ **do** $\llbracket S \rrbracket$ **done**
$\llbracket$ **if** x S **else** T $\rrbracket$	$\overset{\text{stmt}}{\Longmapsto}$	**if** $\llbracket x \rrbracket$ **then** $\llbracket S \rrbracket$ **else** $\llbracket T \rrbracket$
$\llbracket$ *expr* ; $\rrbracket$	$\overset{\text{stmt}}{\Longmapsto}$	$\llbracket expr \rrbracket$
$\llbracket$ (e) $\rrbracket$	$\overset{\text{expr}}{\Longmapsto}$	($\llbracket e \rrbracket$)
$\llbracket$ *op* x $\rrbracket$	$\overset{\text{expr}}{\Longmapsto}$	$\llbracket op \rrbracket$ $\llbracket e \rrbracket$
$\llbracket$ x *op* y $\rrbracket$	$\overset{\text{expr}}{\Longmapsto}$	($\llbracket op \rrbracket$) $\llbracket x \rrbracket$ $\llbracket y \rrbracket$
$\llbracket$ f (a_i,...) $\rrbracket$	$\overset{\text{expr}}{\Longmapsto}$	$\aleph f$ ($\llbracket a_i \rrbracket$)...
$\llbracket$ *ident* $\rrbracket$	$\overset{\text{expr}}{\Longmapsto}$	!$\aleph$*ident*
$\llbracket$ *constant* $\rrbracket$	$\overset{\text{expr}}{\Longmapsto}$	*constant*
$\llbracket$ *op* $\rrbracket$	$\overset{\text{op}}{\Longmapsto}$	$\begin{cases} \texttt{not} & \text{if } op = \text{‘}\texttt{!}\text{’} \\ \texttt{div} & \text{if } op = \text{‘}\texttt{/}\text{’} \\ \texttt{mod} & \text{if } op = \text{‘}\texttt{\%}\text{’} \\ \texttt{=} & \text{if } op = \text{‘}\texttt{==}\text{’} \\ \texttt{<>} & \text{if } op = \text{‘}\texttt{!=}\text{’} \\ op & \text{otherwise} \end{cases}$

Table 5.1.: Translation schema for sequential Go$^-$ to WhyML

For adding logical annotations to Go$^-$ code, we apply the convention that these must be given in comments, using Why syntax. As an example, consider the following function written in Go$^-$:

```
//why3: use import number.Divisibility
//ensures: forall m. 1 <= m <= n -> divides m result
//variant: n
func fac (n int) int {
    r := 1;
    if n > 1 { r = n*fac(n-1); }
    else { }
    return r
}
```

Which we can, by strictly applying the transformation scheme above (handling expressions in an obvious manner), be translated into:

```
use import number.Divisibility

let rec fac (n: int)
  ensures { forall m. 1 <= m <= n -> divides m result }
  variant { n }
= let n = ref n in ();
  let r = ref 1 in ();
  if !n > 1 then begin r := !n * fac (!n - 1); end else begin end;
  !r
```

In the examples that follow, we will not rigidly adhere to the structure of the grammar in Figure 5.1, but continue to write idiomatic Go in such a way that a semantically equivalent transformation from Go → Go$^-$ is obvious. Similarly, since we are performing the translation to WhyML by hand, we will also allow ourselves to perform transformations on the generated WhyML code. In the `fac` example above, the `if`-statement can be made much more readable by simply writing it as:

```
  if !n > 1 then r := !n * fac (!n - 1);
```

However, the translation from Go to WhyML is designed with a program for automatic translation in mind, similar to the Jessie plugin for Frama-C [22]; due to time constraints we have not developed such a tool yet.

5.3. Verifying weak correctness

Many methods for extending Hoare logic to prove the correctness of concurrent programs have been proposed. In a 1976 paper, Owicki and Gries [47] already describe a very general method for proving the correctness of concurrent programs, by introducing an `await` B `then` S primitive which executes the statement S atomically as soon as the condition B becomes true. While this primitive is too powerful to be implemented directly in a programming language, they show how it can be used to model other, more easily implemented, mechanisms such as semaphores. This mechanism could in principle be used to model the concept of synchronous communication channels as well.

Proof techniques directly targeting CSP programs are presented in [48] and [49] — the approach we will take bears a lot of resemblance to these, exploiting the similarity of Go to CSP programs.

Both treatments start with the concept of *weak correctness*, verifying conditions under the assumption that liveness (or at least, the absence of deadlock) is guaranteed. This is conceptually similar to using classical Hoare logic to prove properties without an accompanying proof of termination.

The technique for proving weak correctness of concurrent programs can be broken up into three separate parts [48]:

1. A sequential proof of each process involved, by adding rules to a system for non-concurrent proofs (such as provided by vanilla WhyML) for sending an receiving over a channel, which will involve adding some assumptions that are not provable from within the process.

2. A satisfaction proof to show that the assumptions made in the sequential proofs are valid when multiple processes are combined into a concurrent program.

3. A non-interference proof to show that concurrently running processes do not interfere with the assertions of the sequential proof. As shown in [50], many of these proofs can be avoided.

These three parts are a necessary ingredient of *any* concurrent proof technique. For instance, the proof technique by [49] dispenses with non-interference proofs by not relying on shared variables, and has an mechanism which performs a similar role to that of a satisfaction proof as part of the rule of parallel composition of programs.

Our WhyML model can also be explained in the context of these three ingredients. We will also dispense with non-interference proofs, by disallowing the sharing of state between processes.[1]

The sequential proof is, of course, performed by extending the translation of the previous section with operations on channels. We also insert assertions which take care of the satisfaction proof obligations for channels directly connected to a process.

Finally, a procedure is used to generate WhyML assertions for providing the satisfaction proof obligations needed to ensure the weak correctness of communication that occurs over a *forwarding channel*.

5.3.1. Modelling Go channels

We model Go channels using the WhyML module listed in Figure 5.2. It uses an interface similar to the WhyML array module — except that where arrays have a fixed size and mutable contents, a channel consists of fixed contents (representing the *communication history*) and a mutable size: once an element has been sent over a channel, it becomes part of the known communication history over that channel.

The use of a full communication history, instead of just reasoning about a single element that is sent over a channel, derives from the proof technique used by [49]; it allows us to reason about communication without resorting to explicitly shared ghost variables between processes.

The current end of the *known* communication history can be obtained using the `cursor` function. Outside the known history, the contents are unspecified; in particular, nothing meaningful can be said about the *future* of a communication channel. The function `last` can be used if we are only interested in the most recent element sent. A `chan` is declared as a *model* type (as discussed in Section 2.3.2), so that it cannot be manipulated by WhyML programs except through the four functions that make up its interface.

Note that we could also have opted to make the communication history *mutable*. In that case, only the post-condition of the `forward` function — whose purpose will be explain in Section 5.3.3 — needs to be strengthened by adding:

```
ensures { forall i: int. i < cursor (old q) -> (old q)[i] = q[i] }
```

The downside of mutable contents is that it becomes harder for provers to see that elements that were previously sent remain part of the history. This is a similar problem to the one encountered in Section 3.1, where automated provers had difficulties checking that an array is a permutation of its original value, even though the only operation performed on it are repeated swaps of two elements. It may also necessitate the use of extra loop invariants.

However, since `chan` is a model type, and the interface does not allow WhyML code access to the `contents` field separately from the `pointer` field, it is not possible for an inconsistency to arise. We are further helped by the fact that WhyML itself also disallows creating aliases of mutable types:

[1]In line with the Go advice: *"Do not communicate by sharing memory; instead share memory by communicating."*

```
module Chan

use import array.Array
use import map.Map as M
use import int.Int

type chan 'a model { contents: map int 'a; mutable pointer: int }
  invariant { 0 <= self.pointer }

function enqueued (x: 'a) (q: chan 'a): chan 'a
  = { contents=q.contents[q.pointer<-x]; pointer=q.pointer+1 }

function get (q: chan ~'a) (i: int): 'a = M.get q.contents i
function ([]) (q: chan ~'a) (i: int): 'a = get q i

function cursor (q: chan ~'a): int = q.pointer
function last (q: chan ~'a): 'a = get q (cursor q - 1)

val make unit: chan 'a
  ensures { result.pointer = 0 }

val send (x: ~'a) (q: chan ~'a): unit
  writes { q }
  ensures { q = enqueued x (old q) }

let recv (q: chan ~'a): 'a
  writes { q }
  ensures { q = enqueued result (old q) }
= let x = any 'a in send x q; x

val forward (q: chan ~'a): unit
  writes { q }
  ensures { q.pointer >= old q.pointer }

end
```

Figure 5.2.: WhyML model for weakly correct Go channels

```
1  let send_two (x y: chan int) =
2    send 1 x;
3    send 2 y
4
5  let main() =
6    let x = make(): chan int in
7    send_two x x
```

In this case, WhyML will complain about line 7 that:

```
This application creates an illegal alias
```

Since this is the case, it is vastly preferable to use the guarantees offered by the type system of Why3 to ensure the soundness of our construction, instead of explicitly generating proof obligations.

Ideally, the same would hold for the type invariant: since no operation allows us to construct a channel with a negative size, it should trivially hold with no explicit proof needed. However, since Why3 only assumes type invariants at function entry, we are required to add loop invariants for every channel modified in a `while` loop to allow Why3 to know that the type invariant is preserved.

Translating communication statements is simple; we extend the translation operator $\llbracket - \rrbracket$ defined in Table 5.1 with the following rules:

$\llbracket$ **`chan<-`** τ $\rrbracket$	$\overset{\text{type}}{\Longmapsto}$	**`chan`** $\llbracket \tau \rrbracket$
$\llbracket$ **`<-chan`** τ $\rrbracket$	$\overset{\text{type}}{\Longmapsto}$	**`chan`** $\llbracket \tau \rrbracket$
$\llbracket$ n **`:= make(chan int);`** $\rrbracket$	$\overset{\text{chan-decl}}{\Longmapsto}$	**`let`** $\aleph n$ **`= make(): chan int in ()`**
$\llbracket$ **`go`** $fun - call$ $\rrbracket$	$\overset{\text{go-stmt}}{\Longmapsto}$	*(* empty *)*
$\llbracket$ ch **`<-`** e**`;`** $\rrbracket$	$\overset{\text{stmt}}{\Longmapsto}$	**`send`** $\llbracket e \rrbracket$ $\aleph ch$
$\llbracket$ n **`:= <-`**ch**`;`** $\rrbracket$	$\overset{\text{stmt}}{\Longmapsto}$	**`let`** $\aleph n$ **`= ref(recv`** $\aleph ch$**`)`**

Table 5.2.: Translation schema for Go channel operations to WhyML

To easily state properties about the communication history, we also introduce an auxiliary module, listed in Figure 5.3, which introduces several predicates over channels to use with supporting lemmas:

- `mem` x ch, which states that the element x is a member of the known communication history of the channel ch.
- `mem_sub` x ch i j, which states that the element x is in the communication history with an index in the range $[i, j)$.
- `mem_add` x ch ch', which states the element x has become part of the communication history of ch after its cursor position was at the one indicated by ch'. This is useful to use in WhyML in combination with the `old` keyword, e.g: `mem_add 5 x (old x)` can be used to express the notion that the number 5 has been sent over the channel x after a process has acquired it.

```
module Mem

use import Chan
use import int.Int

predicate mem_sub (e: 'a) (q: chan 'a) (a b: int)
  = exists i: int. a <= i < b <= cursor q /\ q[i] = e

predicate mem (e: 'a) (q: chan 'a)
  = mem_sub e q 0 (cursor q)

lemma mem_0: forall x: 'a, q: chan 'a.
  0 <= cursor q -> mem x (enqueued x q)

lemma mem_S: forall x y: 'a, q: chan 'a.
  mem x q -> mem x (enqueued y q)

lemma mem_sub_0: forall x: 'a, q: chan 'a, i: int.
  0 <= i <= cursor q -> mem_sub x (enqueued x q) i (cursor q + 1)

lemma mem_sub_S: forall x y: 'a, q: chan 'a, i j: int.
  mem_sub x q i j -> mem_sub x (enqueued y q) i j

lemma mem_sub_R: forall x: 'a, q: chan 'a, i' j' i j: int.
  0 <= i <= i' <= j' <= j <= cursor q -> mem_sub x q i' j' -> mem_sub x q i j

predicate mem_add (e: 'a) (q: chan 'a) (old_q: chan 'a)
  = mem_sub e q (cursor old_q) (cursor q)

lemma mem_add_R: forall x: 'a, q u v: chan 'a.
  cursor v <= cursor u -> mem_add x q u -> mem_add x q v

end
```

Figure 5.3.: WhyML support module for reasoning about channel communication histories

5.3.2. Sequential verification of concurrent programs

An important part is still missing: when a WhyML function calls the `recv` function, it is only guaranteed to obtain the next item from the channel, about which we know nothing. To remedy this, right after calling `recv`, we have to insert an assumption about the communication history of the channel. This is highly similar to the seemingly dubious Hoare rule for communication given in [48], which in Go-syntax would look like:

$$\{P\}\ x\ \texttt{:=}\ \texttt{<-}ch\ \{Q\}$$

which is then validated during the satisfaction proof, by verifying that for all possible matching send statements of the form $\{P'\}\ ch\texttt{<-}y\ \{Q'\}$ the implication $(P \wedge P') \rightarrow (Q \wedge Q')$ is valid.

In our approach, each channel that is passed to a goroutine P as an argument should have accompanying annotations which specify the condition that the function *requires* to hold *after* receiving a message from the *requesting process*: these are then simply assumed in the function at that point.

Similarly, annotations should be given to express the *guarantees* that a function provides to a *requesting process* when it sends a message over a channel; these are then checked at that point.

Conversely, in the function that launched P as a goroutine, assertions are inserted *after* sending a message to validate the assumptions made above, and upon reception the guarantees provided by P may be assumed. Since this is analogous to ordinary function calls in WhyML, we can express these conditions naturally using the notion of *channel-related pre- and post-conditions.* For example:

```
//requires: cursor input = cursor output = 0
//requires[input]: odd (last input)
//ensures[output]: odd (last output)
//ensures[output]: cursor input = cursor output
func squarer(input <-chan int) (output chan<- int) {
    //invariant: cursor output = cursor input
    for {
        x := <-input
        output <- x
    }
}
```

The first precondition is a regular precondition, which is checked when this function is launched as a goroutine: in this case the provided channels may not have been used before.

The second annotation is the *channel precondition* for `input`. It just expresses the fact that the last integer sent must be odd. This is also guaranteed for the integer being sent. The final *channel postcondition* for `output` simply states that this function will send exactly as much as it receives.

Channel pre- and postconditions must, like regular pre- and post-conditions, be expressed in terms of function parameters, but are otherwise unrestricted. In particular, we can refer to the entire communication history; not just the last element, and allow the use of the `old` keyword to refer to the state channels were in when the goroutine was started.

If we translate the above Go function to WhyML, also adding the assertions in the manner just described, we get a function which looks like this:

```
1   let squarer (input: chan int) (output: chan int)
2     requires { cursor input = cursor output = 0 }
3   = while true do
4       invariant { 0 <= cursor input /\ 0 <= cursor output }
5       invariant { cursor output = cursor input }
6       let x = ref (recv input) in
7       assume { odd (last input) };
8       send (!x * !x) output;
9       assert { cursor output = cursor input };
10      assert { odd (last output) };
11    done
```

The loop invariant on line 4 has to be added to ensure the type invariant of the channels `input` and `output`, as described in Section 5.3.1; otherwise, it is a straightforward translation of the Go code.

If a program launches the function `squarer` as a goroutine, then after every send to the `input` channel, an `assert` is inserted to verify the channel pre-condition, validating the assumption made above, and similarly, assumptions will be inserted after every receive. For example:

```
    x := make(chan int)
    y := make(chan int)
    go squarer(x, y)
    x <- 42
    result := <-y
    //check: odd result
```

Will translate to WhyML as:

```
1       let x := make(): chan int in
2       let y := make(): chan int in
3       (* go squarer(x, y) *)
4       assert { cursor x = cursor y = 0 };
5       send 42 x;
6       assert { odd x[cursor x - 1] };
7       let result = ref(recv y) in
8       assume { odd y[cursor y - 1] };
9       check { odd !result };
```

Launching a goroutine has little immediate effect: its only influence is that its precondition is checked (see line 4). Note that the channel pre-condition on line 6 is violated, since 42 is not an odd integer.

5.3.3. Generating satisfaction proof obligations

To finish our verification technique for weak correctness, we must now generate the appropriate satisfaction proof obligations for communicating processes; that is, show that all the assumptions we made in a function that is used as a goroutine are validated by assertions made in other functions.

If we perform the sequential verification process described in the previous section for a function P, we see that we have already inserted all the assertions needed to guarantee the satisfaction of channels that are directly connected to P. To be more precise, when we send a message over the channel u:

- If the message is to a *requesting process*, we have asserted P's *channel post-condition* for u, this validating the assumptions made about u in the *requesting process*.
- If the recipient is a *child process*, we have asserted the child's *channel pre-condition* for u, validating the assumptions made inside the child process.

As we have seen in the data flow graphs for the `summer` and `adder` functions on pages 51 and 51, there may also exist *forwarding channels*, which are required to operate successfully for a program to be correct. For example, in the case of the data-flow graph for the `summer`, it must also be shown that the channel pre-condition of `echo(v,w)` for v is satisfied by the channel post-condition for v of the `echo(u,v)`.

In the case of the `adder` function, there are two forwarding channels connected to a *requesting process*. In this case, it must be shown that `adder`'s channel pre-condition for x satisfies the channel pre-condition for x `echo(x,u)`, and similarly, that the channel post-condition for *result* of the process `echo(v,result)` implies `adder`'s post-condition for the same channel.

Since we have no control or knowledge about the amount of messages sent over a *forwarding channel*, we have defined the `forward` function in Figure 5.2 in order to move its cursor by an unspecified amount. After `forward` has been called, we can then insert an appropriate assertion. For example, in the case of `summer`, after the translation of the send operating on line 14 in the program on page 51, we can emit the following WhyML code to emit a satisfaction obligation:

```
    (* [[ u <- tmp ]] *)
    send !tmp u;
    assert { precondition of echo(u,v) };
    forward v;
    assert { postcondition of echo(u,v)
          -> precondition of echo(v,w) };
```

In the case of `adder` on page 52, reading from channel u requires generating a satisfaction obligation for the forwarding channel x, but not for the forwarding channel *result*, since it is not part of the data-flow path leading towards the channel u:

```
    (* [[ x := <-u ]] *)
    let x = ref(recv u) in
    assume { postcondition of echo(x,u) };
    forward x;
    assert { precondition of adder(x,y,result)
          -> postcondition of echo(x,u) };
```

In Section 5.4.4, verification of forwarding channels is an important part of verifying the *liveness* of Go programs. To see why generating these assertions is necessary even if we are only interested in weak correctness, consider the following program:

```
//ensures[out]: forall i. 0 <= i < cursor out -> out[i] = x
func source(x int, out chan<- int) {
    out <- x
}

//requires[in]:  forall i. 0 <= i < cursor in -> in[i] >= 0
//ensures [out]: forall i. 0 <= i < cursor in -> out[i]*out[i] <= in[i]
//ensures [out]: cursor out = cursor in
func root(in <-chan int, out chan<- int) {
    x := <-in
    out <- int(math.Sqrt(float64(x)))
}

func main() {
    x, y := make(chan int), make(chan int)
    go source(-1, x)
    go root(x, y)
    sqrt := <-y
    //assert: sqrt*sqrt <= -1
}
```

Clearly, this program should not pass verification, as the assertion indicates an inconsistency. If we had sent the value −1 to the `root` process directly, the explicit check of its precondition would cause verification to fail. In this case, a child process is communicating with `root` *on behalf of* the current process; generating an assertion for the *forwarding channel* `x` as just described causes this precondition to be tested as well.

The general procedure for generating verification conditions for forwarding channels is as follows. After sending or receiving a message from within a process P over the channel u, proceed along the following steps:

1. Construct the static data flow graph for the channel operation (see pages 51 and 52)
2. Find every *forwarding channel* that is relevant to the current operation. These are all the edges in the data flow graph that are reachable via a path that leads from u and does not contain P itself. Furthermore, all edges on the path must be oriented in the same direction as u.
3. For every channel v found, emit the `forward` instruction followed by an assertion in the form of an implication which captures its satisfaction proof obligations.

This same procedure is also to be followed as soon as a goroutine is launched which turns a channel into a *forwarding channel.*

The fact that `forward` grows the communication history of a channel by an unspecified number of elements is important to ensure the *non-interference* of logical assertions that involve *forwarding channels* — while it is allowed to reason in a general sense about the communication history (as guaranteed by the post-conditions of the involved goroutines), a proof of correctness is not allowed to depend on the *exact* current size of this history — as the processes can communicate with each other at any time.

5.3.4. Example: Verifying the concurrent prime number sieve

As a more complete example of the verification method we have developed in the previous sections, we will verify the weak partial correctness of the concurrent prime number sieve introduced in Section 4.1. We first need to rewrite this algorithm to comply with the restrictions of Go$^-$. In particular, we need to eliminate the lambda function and for-each loop. Subsequently, the program needs to be annotated in the style of WhyML. The result is listed in Figure 5.4. It consists of three functions, which we will examine in order.

Verification of the function `generate` The channel post-condition of the function `generate` states that if the communication history is of size n, then it will contain all numbers in the range $[init, n + init)$. Note that this is an underspecification: it does not necessarily specify the order in which they will be generated.

Translating `generate` to WhyML is a straightforward application of Section 5.3.2, since there are no *forwarding channels* present at any moment.

```
let generate (init: int) (out: chan int)
  requires { cursor out = 0 }
= 'Start:
  let x = ref init in
  while true do
    invariant { "expl:type invariant preservation" 0 <= cursor out }
    invariant { forall x: int. init <= x < cursor out + init <-> mem x out }
    invariant { cursor out+init = !x }
    send !x out;
    assert { "expl:channel postcondition"
             forall x: int. init <= x < cursor out + init <-> mem x out };
    x := !x+1;
  done
```

This WhyML version can be verified quickly using just CVC4.

Verification of the function `sieve` The pre-condition of the function `sieve` should be compared to the first loop invariant in the algorithm presented in Section 2.3.1; as it fulfils essentially the same function. The post-condition is again a underspecification: the function will output prime numbers, and it is forbidden to generate them spontaneously — but in principle the post-condition does allow for it to skip any prime number, or to repeatedly output the same prime number.

The loop invariants govern the communication histories of the channels `filt` and `out`; in particular we need to remember that we have already sent the prime number p.

The channel out becomes a forwarding channel after it is handed to the (recursively) launched goroutine. We mark the starting point of that separate goroutine by asserting its pre-condition, emitting the label `Entry_sieve` (to be able to refer to the initial state of its channel arguments), and a call to the `forward` function to make its position indeterminable.

Since out is a *forwarding channel*, we need to generate a forwarding condition on line 30..

Note that an assertion (on line 31 in Figure 5.4, and line 27 in the WhyML code) is needed to help the verification process along.

```
1  let sieve (src: chan int) (out: chan int)
2    requires { cursor src = 0 }
```

```go
package main

import "fmt"

//requires: cursor out = 0
//ensures[out]: forall x: int. init <= x < cursor out + init <-> mem x out
func generate(init int, out chan<- int) {
    x := init
    //invariant: forall x: int. init <= x < cursor out + init <-> mem x out
    //invariant: cursor out + init = !x
    for {
        out <- x
        x++
    }
}

//requires: cursor src = 0
//requires[src]: forall x: int. mem x src -> 2 <= x /\ forall d: int. 2 <= d ->
//               divides d x -> mem d src
//ensures[out]: forall x: int. mem_add x out (old out) -> mem x src /\ prime x
func sieve(src <-chan int, out chan<- int) {
    p := <-src
    out <- p
    filt := make(chan int)
    go sieve(filt, out)
    //invariant: forall x: int. mem x src /\ not (divides !p x) <-> mem x filt
    //invariant: mem_add !p out (at out 'Start)
    for {
        n := <-src
        if n%p != 0 {
            //assert: forall x: int. mem x src /\ not (divides !p x) /\ x <> !n ->
            //        mem x filt
            filt <- n
        }
    }
}

func main() {
    outp := make(chan int)
    inp := make(chan int)
    go sieve(inp, outp)
    go generate(2, inp)
    for {
        n := <-outp
        //check: prime !n
        fmt.Println(n)
    }
}
```

Figure 5.4.: Concurrent prime number sieve in Go, annotated

```
= 'Start:
  let p = ref (recv src) in
  assume { "expl:channel precondition" sieve_pre src };

  send !p out;
  assert { "expl:channel postcondition" sieve_post src out (at out 'Start) };

  let filt = make(): chan int in
  assert { "expl:goroutine precondition" cursor src = 0 };
'Entry_sieve:
  forward out;
  assert { "expl:channel forward" sieve_post filt out (at out 'Start)
                               -> sieve_post src out (at out 'Start) };

  while true do
    invariant { "expl:type invariant preservation"
                0 <= cursor src /\ 0 <= cursor out /\ 0 <= cursor filt }
    invariant { forall x: int. mem x src /\ not (divides !p x) <-> mem x filt }
    invariant { mem_add !p out (at out 'Start) }

    let n = ref (recv src) in
    assume { "expl:channel precondition" sieve_pre src };

    if (mod !n !p <> 0) then begin
      assert { forall x. mem x src /\ not (divides !p x) /\ x <> !n -> mem x filt };
      send !n filt;
      assert { "expl:channel precondition" sieve_pre filt };
      forward out;
      assert { "expl:channel forward" sieve_post filt out (at out 'Start)
                                   -> sieve_post src out (at out 'Start) };
      end;
    end
  done
```

To allow for readability of the WhyML model, we have defined predicates for the pre- and post-conditions that were found in the Go annotated program:

```
predicate sieve_pre (src: chan int) =
  forall x: int. mem x src -> 2 <= x /\ forall d. 2 <= d -> divides d x -> mem d src
predicate sieve_post (src out old_out: chan int) =
  forall x: int. mem_add x out old_out -> mem x src /\ prime x
```

During verification, this extra layer of definition actually hinders automated provers, requiring extra computational transformations to remove them. For example, it is necessary to add

```
meta "rewrite_def" predicate sieve_pre
meta "rewrite_def" predicate sieve_post
```

To allow the use of the `compute_specified` transformation, which allows us to selectively unfold these definitions. Still, using a combination of Alt-Ergo, CVC3 and CVC4, this function can be automatically verified in less than a minute of CPU time.

Verification of `main` The function `main` contains no annotations or loop invariants; but since it spawns two goroutines, the WhyML model is not as succinct. Again note that on line 18 a channel forwarding assertion has to be generated, since `inp` is a forwarding channel.

```
let main ()
= let outp = make(): chan int in
  let inp = make(): chan int in
  assert { "expl:goroutine precondition" cursor inp = 0 };
'Entry_sieve:
  assert { "expl:goroutine precondition" cursor inp = 0 };
'Entry_generate:
  forward inp;
  assert { "expl:channel forward"
           forall x: int. 2 <= x < cursor inp + 2 <-> mem x inp
        -> sieve_pre inp };
  while true do
    invariant { "expl:type invariant preservation"
                cursor inp >= 0 /\ cursor outp >= 0 }
    let n = ref (recv outp) in
    assume { "expl:channel postcondition" sieve_post inp outp (at outp 'Entry_sieve) };
    forward inp;
    assert { "expl:channel forward"
             forall x: int. 2 <= x < cursor inp + 2 <-> mem x inp
          -> sieve_pre inp };
    end;
    check { prime !n };
    (* fmt.Println(!n) *)
  done
```

The verification conditions generated by this function can all be discharged by Alt-Ergo in less than 5 seconds.

5.4. Verifying strong correctness

Up to this point, we have no bothered with the possibility of deadlocks. However, due to the ease with which goroutines can be strung together, this is an important topic. Consider again the functions summer and adder, but now in simplified form:

```
func summer(x <-chan int, y <-chan int, result chan<- int) {
    a := <-x
    b := <-y
    result <- a + b
}

func adder(x <-chan int, y <-chan int, result chan<- int) {
    b := <-y
    a := <-x
    result <- a + b
}
```

These functions are not interchangeable in all circumstances; but they are hard to distinguish using just weak correctness. Another shortcoming of weak correctness is that programs that do trigger deadlocks may end up in a logically inconsistent state. For example, consider the following function:

```
//ensures[less]: x < 0
//ensures[more]: x > 0
func cmp(x int, less chan<- int, more chan<- int) {
    if x < 0 { less <- x }
    if x > 0 { more <- x }
}
```

Clearly, the two post-conditions are inconsistent with each other, but they both are perfectly valid, as the function `cmp` will never be able to activate both channels. However, if a program tries to read from both channels, it will cause a deadlock. In our model of weak correctness, it results in an inconsistent state:

```
        x, y := make(chan int), make(chan int)
        go cmp(1,x,y)
        a := <-x
        b := <-y
        //check: 1 = 2
```

Which in the WhyML translation can be proven, since it translates to:

```
        let x = make(): chan int
        let y = make(): chan int
        let a = ref (recv x) in
        assume { 1 < 0 };
        let b = ref (recv y) in
        assume { 1 > 0 };
        check { 1 = 2 }
```

In some sense, this is in the spirit of Hoare logic: *if* the operation `<-y` terminates, it will indeed be true that $1 = 2$, just as would be the case had we replaced `<-y` it with an infinite loop. When using Hoare logic, this problem is solved by providing an associated termination proof. For concurrent programs, the equivalent is a proof that deadlocks cannot occur.

Both [48] and [49] use the same method for proving the absence of deadlocks, which basically requires performing an exhaustive search of all possible combinations of synchronization between send/receive statements, and then showing the conjunction of all their respective preconditions either leads to a contradiction — i.e., the state cannot happen — or that progress can be made in that state. This is somewhat similar to the technique we used in Section 2.2.4 to prove the absence of deadlocks in the Dining Philosophers problem.

A downside of that approach is that it is hard to implement for Go; for example, in the prime number sieve of Figure 5.4, the number of matching communication pairs is essentially infinite, since the function `sieve` recursively launches itself as a goroutine. Secondly, in many cases a stronger notion of *liveness* than the mere absence of deadlock is actually a more interesting property. For example, if we connect the function `sieve` to a slightly different goroutine which sends it all numbers starting with 1, it will only produce output once:

```
    func main() {
        x := make(chan int)
        y := make(chan int)
        go generate(1, x)
        go sieve(x, y)
        <-y
        <-y // blocks forever
    }
```

However, this blocking situation is not a deadlock; the two goroutines are in fact very busily searching for a number that is not divisible by 1!

Therefore, we want to provide the stronger guarantee of *liveness*, which we in this case define as the ability of the primary goroutine — that is, the one that `main` starts in — to make progress. In

Section 3.3, we have already discussed a technique that can be used to ensure progress in a program. It is this idea that we combine with the notion of weak correctness to arrive at a verification technique for *strong correctness* of concurrent Go programs.

5.4.1. Liveness annotations

To reason about liveness, we add a new kind of proposition: a *liveness annotation*, in the form of the expression *p* => *ch*. A liveness annotation simply expresses the fact that process *p* will guarantee that a send or receive operation on the channel *ch* will not block.

As an example, consider again the echo process as defined on page 51. When echo is launched as a goroutine, it will guarantee to synchronize first on the channel input, and after that alternate between the two channels output and input. We could therefore annotate it as follows:

```go
//startup: this => input
//ensures[input]: this => output
//ensures[output]: this => input
func echo(input <-chan int, output chan<- int) {
    //obligation: this => input, variant: 0
    for {
        x := <-input
        output <- x
    }
}
```

Here the identifier this denotes the goroutine echo itself. A new annotation (startup) is necessary to specify the immediate effects of launching echo as a goroutine; as the only effect can be that some channels become safe to synchronize with, this must invariably be a liveness annotation. In this case echo guarantees to synchronize with input.

The following two annotations are channel-related post-conditions, and express the synchronizing behaviour of echo described informally. Note that the model for strong correctness also allows input channels to have post-conditions, unlike the model for weak synchronization where they would have served no logical purpose.

A new type of loop invariant is introduced to keep track of the current synchronization obligation of a function. Since echo promises to read from the channel input at startup, and after writing to output, this should be this => in. The obligation annotation also requires a *conditional variant*, which is a quantity stated to decrease until the obligation is met. In this case, the obligation is fulfilled by the first statement of the loop body, and so this variant is trivial.

A process communicating with an instance of an echo goroutine needs some way to denote this instance. We do this by adding a *process annotation* to statements launching a goroutine. For example:

```go
func main() {
    i, o := make(chan int), make(chan int)
    //process: echo1
    go echo(i,o)
    u <- 5
    //invariant: echo1 => o
    for {
        a := <-o
        i <- a
    }
}
```

After the statement `a := <-o`, the post-condition for `echo1` will ensure that the channel i can be written to, after which the loop invariant is re-established.

The annotation `echo1 => i` expresses something markedly different than `this => i`: in the first case, some other process is giving a guarantee that can be relied upon; in the latter, a guarantee is given that formally binds the current process to perform a certain action. There is a deliberate asymmetry in this model: whenever two processes are involved with each other, one is responsible for providing guarantees, whilst the other is relying on them. Guarantees are given to *requesting processes* (as defined in Section 5.1) by the current process; and relied upon by the current process when dealing with *child processes*. Most importantly, a goroutine is not required to ensure that its child processes can always make progress (as indeed was not the case in most examples we used throughout this chapter).

This asymmetry has two important consequences:

1. It allows local reasoning; a current process only has to reason about its child processes in order to prove that communicating with it will succeed — and a goroutine is only required to guarantee its liveness to its instantiating process.

2. It also allows us to reason compositionally: the liveness of a goroutine follows by combining the liveness properties of any child processes it communicates with. At some point, this will result in correctness proof for the primary goroutine of a Go program (i.e., the one that starts `main`).

The annotations we have described in this section suffice to reason about communication that happens across channels that are directly connected to the current process. When forwarding channels are relevant to the correctness of a function, extra annotations are required — we will handle this case in Section 5.4.4 below.

5.4.2. Strengthening the model for Go channels

To implement the annotations described in the previous section, we adapt the Go model for channels given in Section 5.3.1, and combine it with the `Obligation` module developed in Section 3.3; it is listed in Figure 5.5. The primary modifications are:

- A predicate for the liveness annotation `=>` has been added; as well as a type to model processes, and a global object `this` to denote the current process.

- The `send` and `recv` operations now also take an extra argument indicating the process that is responsible for ensuring the liveness of the channel.

  These functions now also require (as a precondition) that they will not block; i.e., that the mentioned process has guaranteed that it possible to send or receive on the given channel. This means that, similar to out-of-bounds array accesses, liveness is now a *sine qua non* for the verification of a concurrent Go program.

- A predicate `handled` is used for exactly the same purpose as the `location` type in Section 3.3 — it is true for all channel objects that for which the synchronization obligations has been fulfilled, because a send or receive can be proven to have been performed on them.

- A function ch`[`Q`<-`P`]` replaces the `forward` function; it captures the transfer of one element between two processes, instead of indeterminably many. It again is only used for generating verification conditions of *forwarding* channels between processes P and Q, in Section 5.4.4.

```
module Chan

use import array.Array
use import map.Map as M
use import int.Int
use export obligation.Obligation

type id

type chan 'a model { contents: map int 'a; mutable pointer: int; id: id }
  invariant { 0 <= self.pointer }

function enqueued (x: 'a) (q: chan 'a): chan 'a
  = { contents=q.contents[q.pointer<-x]; pointer=q.pointer+1; id=q.id }

type process model { mutable state: id }
val this: process

predicate handled (q: chan ~'a)
predicate (=>) (p: process) (q: chan ~'a)

function get (q: chan ~'a) (i: int): 'a = M.get q.contents i
function elem (q: chan ~'a): map int 'a = q.contents
function ([]) (q: chan ~'a) (i: int): 'a = get q i

function cursor (q: chan ~'a): int = q.pointer
function last (q: chan ~'a): 'a = get q (cursor q - 1)

val make unit: chan 'a
  ensures { result.pointer = 0 }

val send (x: ~'a) (q: chan ~'a) (p: process): unit
  writes { q, p.state }
  requires { p => q }
  ensures { q = enqueued x (old q) }
  ensures { handled (old q) }

let recv (q: chan ~'a) (p: process): 'a
  writes { q, p.state }
  requires { p => q }
  ensures { q = enqueued result (old q) }
  ensures { handled (old q) }
= let x = any 'a in send x q p; x

val ([<-]) (q: chan ~'a) (b a: process): unit
  writes { q, a.state, b.state }
  requires { a => q /\ b => q }
  ensures { q.pointer = old q.pointer+1 }
  ensures { handled (old q) }

val responsible (q: chan 'a): unit
  ensures { (?obligation -> handled q) -> this => q }

val delegated (that: process) (q: chan 'a): unit
  ensures { that => q -> handled q /\ this => q }

end
```

Figure 5.5.: WhyML model for Go channels for strong correctness

5.4.3. Sequential verification of liveness annotations

For the most part, the process of providing the sequential proof of goroutine's, described in Section 5.3.2, is used unaltered: liveness annotations are a predicate like any other, and can be dealt with as any other proposition when asserting or assuming channel pre- and post-conditions.

The required extensions to this process are as follows:

- At the start of a function, the `startup` conditions needs to be asserted. Conversely, whenever a `go` statement launching a child process occurs, the startup condition of the child goroutine needs to be assumed.
- Since an input channel can now also have a post-condition, these must be asserted after executing a receive operation, and assumed after executing a send operation, as is the case for post-conditions for output channels.

To ensure the guarantees given by a function, the technique from Section 3.3 is applied in the following manner:

- If a function needs to assert a post-condition where `this =>` *ch* is a consequence, i.e. it guarantees to operate on *ch*, an obligation needs to entered into which states that the predicate `handled` will become true for the channel *ch*. To be more precise, whenever an assertion of the form *cond* → `this =>` *ch* is made, a call has to be inserted of the form `oblige (`*cond* `->` `handled` *ch*`)`.
- The `obligation` annotation is translated to a loop invariant which specifies the state of the `?obligation` object.

  In addition, all `while` loops in the generated WhyML code need to have a *conditional variant* inserted ensuring that all obligations entered into will eventually be fulfilled. At the end of a function, and at every point where it communicates with the *requesting process* — i.e. at every point where control is transferred — a check also needs to be inserted to ensure that the obligations are all met at that point.

In addition, some WhyML 'logical glue' is needed to allow `this =>` *ch* to be provable if we have that `?obligation -> handled` *ch*. We would this to be a defining axiom or `let lemma` in our system, but since both channels and process states are mutable values, this is not available to us. The function `responsible` listed in Figure 5.5 takes care of this.

The end result is a translation process which is quite arduous to perform by hand, but is well suited to mechanization. For example, translating the definition of the `echo` function as given in the previous section is done as follows:

```
let echo (input: chan int) (output: chan int)
= init();
  (* startup *)
  oblige (handled input);
  responsible input;
  assert { this => input };
  while true do
    invariant { cursor input >= 0 /\ cursor output >= 0 }
    (* obligation: this => input, variant: 0 *)
    invariant { ?obligation <-> handled input }
    (* [[ x <- input ]] *)
    responsible input;
    let x = ref(recv input this) in
    check { ?obligation };
    (* ensures[input]: this => output *)
```

```
      oblige (handled output);
      responsible output;
      assert { this => output };
      (* [[ output <- x ]] *)
      send !x output this;
      check { ?obligation };
      (* ensures[output]: this => input *)
      oblige (handled input);
      responsible input;
      assert { this => input };
    done;
    check { ?obligation }
```

The translation of the `main` process is easier, since it does not need to make any guarantees, and is mostly along the lines of Section 5.3.2:

```
let main()
= let i = make(): chan int in
  let o = make(): chan int in
  (* [[ go echo(i,o) ]] *)
  let echo1 = any process in
  assume { echo1 => i };
  (* [[ i <- 5 ]] *)
  send 5 i echo1;
  assume { echo1 => o };
  while true do
    invariant { cursor i >= 0 /\ cursor o >= 0 }
    invariant { echo1 => o }
    [[ a := <- o ]]
    let a = ref(recv o echo1) in
    assume { echo1 => i };
    [[ i <- a ]]
    send !a i echo1;
    assume { echo1 => o };
  done
```

5.4.4. Satisfaction proofs in the strongly correct model

By applying the process of sequential verification, satisfaction obligations for *directly connected* processes will already have been generated, as was the case for weak correctness.

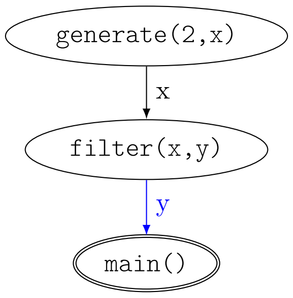
This leaves the case of *forwarding channels*, which (for weak correctness) we dealt with in Section 5.3.3 by inserting extra assertions, combined with a call to the `forward` function. In this case we would simply assume that the synchronization between the involved processes would be successful. In the case of strong correctness, we have to verify this as well.

To be more precise, if we are trying to write to (or read from) a channel u, and the success of that operation may depend on other processes communicating with each other, it must be shown that the communication between those two processes will eventually result in channel u becoming available. For example, in the case of the of the prime number sieve of Figure 5.4, the output channel of `sieve` will not become available if all numbers presented in its input channel are divisible by the first number sent to it. In a simpler example:

```
//startup: this => input
//ensures[input]: if last input < x then this => input else this => output
```

```
//ensures[output]: this => input
func filter(x int, input <-chan int, output chan<- int) {
    for {
        y := <-input
        if y >= x { output <- y }
    }
}

func main() {
    x, y := make(chan int), make(chan int)
    //process: gen
    go generate(2, x)
    //process: filter
    go filter(7, x, y)
    <-y
}
```

In this case, the channel y in the last line only becomes available as soon as the `generate` process has sent the number 7 or higher to the `filter` process. Here, we take `generate` to be the same as in Figure 5.4, i.e., it simply emits successive integers, starting at 2. The associated data flow graph of this situation is as follows:



In WhyML we ensure this by simply emulating the communication between `gen` and `filter` element-by-element, in a generated `while`-loop, right before the statement `<-y` is to be executed. In this case, the form of the loop would look as follows:

```
while not (filter => y) do
  if (gen => x /\ filter => x) then begin
    x[filter <- gen];
    assume { postcondition of gen };
    assert { precondition of filter };
    assume { postcondition of filter };
  end;
done
```

We will call this construction the *forwarding loop*. Clearly, as soon as the loop terminates, we will have `filter => y` as a condition, indicated that it is safe to read from channel y, as desired. The important thing to show about this loop is that it is terminating; which should be done by giving a variant. In this case, `100 - cursor x` would suffice.

Of course, the *forwarding loop* also requires loop invariants; in this case, at least `gen => x` ∧ `filter => x` would be a requirement, in addition to a loop invariant from which it can be deduced that the `gen` process will eventually reach the number 7.

The major difference between this process and the one described in Section 5.3.3 is that whereas earlier, we would insert a check for the forwarding channel once, and simply move the cursor of the forwarding channel by an unspecified amount, we now check the communication between the involved processes one tiny step at a time.

More precisely, we replace the procedure listed on page 61 by the following. Before sending or receiving a message from within process P over the channel u, which connects to a child process R, proceed along the following steps:

1. Construct the static data flow graph for the channel operation, and find every *forwarding channel* that is relevant to the current operation, exactly as described on page 61.
2. If no forwarding channels are found, stop. Otherwise:
3. Emit a *forwarding loop* (`while not (`R` => `u`) do ...`).
4. Emit the loop variant and necessary loop invariants. With the exception of the type invariant for channels, these must be supplied by the user.
5. For every forwarding channel v found, of which process A is responsible for guaranteeing the availability of the channel for reading, and B is responsible for guaranteeing the same for writing, emit a conditional statement which:
   a) Checks whether the condition A`=>`v and B`=>`v holds.
   b) If that is the case, executes v`[`B`<-`A`]`, followed by the appropriate assumptions and assertions for the pre- and post-conditions of A and B.

Note that the processes A or B mentioned above may be equal to the current process P, if the *forwarding channel* found connects to a *requesting process*. For example, in the data flow graph seen on page 52, when generating the *forwarding loop* for the channel x, A would equal the process identifier corresponding to `echo(x,u)`, whereas B would be equal to `this`, since `adder` is responsible for guaranteeing that the channel x will be read from.

One large problem of this approach is that the user is required to provide loop invariants and variant for something which is not present in the original Go program; e.g. in the `adder`, e.g.:

```
//startup: this => y
//ensures[y]: this => x
//ensures[x]: this => result
//ensures[result]: this => x
func adder(x <-chan int, y <-chan int, result chan<- int) {
    u, v := make(chan int), make(chan int)
    go echo(x, u)
    go echo(v, result)
    //obligation: this => y
    for {
        b := <-y
        //forward[u] variant: cursor u + 1 - cursor x
        //forward[u] invariant: cursor x <= cursor u
        //forward[u] obligation: if cursor x < cursor u then this => x else this => y
        a := <-u
        v <- a + b
    }
}
```

In this case, it is not intuitive that anything would even need to be specified, or what to specify. In many cases, it is probable that much of this can be deduced from the post-conditions of the channels involved. It would be helpful for any program implementing this translation scheme to offer some support to the user, or to find a more elegant way of expressing the invariants needed in the *forwarding loop* at the Go level.

5.4.5. Recursion and goroutines

In the prime number example of Figure 5.4, a goroutine launches instances of itself. We have ignored this until now, but to prevent such a pattern from leading to a runaway creation of processes, an extra kind of check will be necessary. Luckily, addressing this is very easy, as there are two possibilities:

1. The recursion can be proven to be finite; in this case the user can supply a `variant`.

2. The recursion may not be finite (as in the case of the prime number sieve). In this case, to prevent the runaway effect it is sufficient to require that a function interacts with its *requesting process* — i.e., sends or receives a message from a channel given to the function as an argument — before it starts a copy of itself.

This second condition can be verified fully automatically, without requiring an annotation, since we can simply check whether the cursor of a channel has been moved. This will prevent degenerate code such as the following:

```
func chain(x chan<- int) {
    //process: go_chain
    go chain(x)
}
```

Since it will translate to WhyML as:

```
let chain (x: chan int)
= 'Start: init();
  (* go chain(x) *)
  let go_chain = any process in
'Entry_go_chain:
  check { cursor x > at (cursor x) 'Start }
```

And this final check will obviously fail.

5.4.6. Evaluation: Liveness in the prime number sieve

In Appendix B, we show how liveness annotations can be applied to the concurrent prime number sieve given earlier. Translating this to WhyML results in around 200 lines of code, and takes about 4 minutes of CPU time to be automatically verified using a combination of CVC3, CVC4 and Alt-Ergo.

In the function `main`, the *forwarding loop* for the channel `outp` was the hardest part to verify. Obviously we need the fact that there are an infinite number of primes to show that this loop is terminating, since the `sieve` function will not produce any output if it no longer receives any unseen prime numbers. But even then invariants were needed to state 'obvious' facts that turned out not to be so obvious — such as that if an element has been sent by the `generate` process to the channel `inp`, it is not already part of the known history of `outp` (expressed by the loop invariant on line 68).

Various other invariants have also turned out to be necessary to allow the verification of the liveness annotations themselves, which were not necessary for verifying weak correctness — this is similar to the situation where supporting invariants are needed to ensure that an index into an array is not out of bounds, which may in turn require the proof of other correctness properties.

Such supporting proofs are not optional, as the preconditions of the `send` and `recv` functions *require* channels to be non-blocking. This may at times hinder developing a proof of a program in an incremental method. A solution can also be found by taking inspiration from Why3's `array.Array` module: we could introduce a `defensive_send` and `defensive_recv` method in the model for Go channels of Figure 5.5, which do not check the pre-condition, but are stated to throw an exception in case they may block:

```
exception Blocked

val defensive_send (x: ~'a) (q: chan ~'a) (p: process): unit
  writes { q, p.state }
  ensures { q = enqueued x (old q) }
  ensures { handled (old q) }
  raises { Blocked -> not old (p => q) }
```

A proof of liveness then simply entails showing that this exception can never occur, for example by adding a

```
raises { Blocked -> false }
```

annotation to a function — but we would now also have the option of simply ignoring the exception.

The code in Appendix C was created by adding the liveness annotations directly to the WhyML version shown in Section 5.3.4. The amount of effort necessary to produce this verification was about double that of proving the weak correctness of the prime number sieve; a major influencing factor in this is that as proofs become more complex, automated provers become less efficient, and the development cycle becomes slower.

In conclusion, this example on the one hands teaches us that verifying software is hard, and verifying concurrent software is even harder. But it also shows that the amount of extra work needed is not necessarily insurmountable: a verification of heapsort also takes a considerable amount of time. Considering the amount of added non-determinism that can occur in a concurrent program, the increased amount of annotations needed is actually modest (with the exception of the `forward` annotations), and they manage to convey the *meaning* of the program remarkably well. For example, the important function `sieve` has just four pre- and post-conditions; for its input channel, they are:

- If a number has been sent to `sieve`, then so must all of its divisors greater than or equal to 2:

  ```
  forall x. mem x src -> 2 <= x /\ forall d. 2 <= d -> divides d x -> mem d src
  ```

- If a prime number has been sent to `sieve` that has not yet been read from its output channel, then its output channel is safe to read from; otherwise it is ready to accept new inputs.

  ```
  if exists x. mem x src /\ prime x /\ not mem_add x out (old out)
    then this => out else this => src
  ```

For its output channel, they are even simpler:

- Numbers sent to the output channel are all prime, and originate from the input channel:

  ```
  forall x. mem_add x out (old out) -> mem x src /\ prime x
  ```

- After reading a single output, writing to the input channel is allowed:

  ```
  this => src
  ```

Which should be easily understandable at a high level, even if someone is not intricately familiar with all the machinery for verifying these conditions.

Note that the second condition and third conditions in concert imply a stronger result than we had for the weakly verified prime number sieve, as it is no longer possible for it to 'skip' prime numbers that are sent to it; although this specification still permits for it to repeatedly output the same prime number. Fixing this last gap would require verification of another post-condition for the channel out; we leave the exact form that it should take as an exercise to the reader.

6. Conclusion

We have reviewed how the Why3 verification platform can be used for various software verification tasks. In particular, in Chapter 3 we applied it in a manner which it is already well-suited for — verifying algorithms in a classical, single-thread environment. We verified heapsort, and shown how WhyML can also be easily used as an intermediate language for the verification of assembly language programs.

In Chapter 5, we applied Why3 outside its normal application area, by using it to describe a method for verifying concurrent programs written in Go that use synchronous message passing as a sole means of communication between processes. If we ignore the possibility of synchronization issues, such as deadlock or livelock, our method can best be seen as an application of Levin's [48] proof technique for CSP programs.

We have investigated how the principle of underspecification (which is natural in Hoare logic) can also be used to reason about *liveness* in a concurrent Go program. This method should suffice for simple programs (which are expected to follow simple concurrency patterns), but can become laborious when applied to difficult scenarios. Another downside of our method is that the user needs to be aware of the way the translation operates; particularly in the case where large chains of communication are involved. As a proof of concept, we have verified a concurrent version of the Sieve of Eratosthenes, written in Go, and shown which steps were necessary to achieve this result.

6.1. Reliability of automated provers

The Electric Monk was a labour-saving device, like a dishwasher or a video recorder. Dishwashers washed tedious dishes for you, thus saving you the bother of washing them yourself, video recorders watched tedious television for you, thus saving you the bother of looking at it yourself; Electric Monks believed things for you, thus saving you what was becoming an increasingly onerous task, that of believing all the things the world expected you to believe. — Douglas Adams, Dirk Gently's Holistic Detective Agency

An important question in verification of software is the one raised by Pollack [51]: how can we actually believe that an algorithm verified by a computer is correct? This is more than a purely philosophical question, especially when using automated provers. Finding assertions and lemmas to allow something to be proven using Why3 can be done using a somewhat uninformed trial and error method until one of the theorem provers says that it has found a proof — without stating what it entails. Often automated provers make short work of theorems that are not at all trivial to perform by hand, or have difficulty with seemingly obvious facts; perhaps finding complicated proofs.

Making matters even harder are the following observations, encountered during the main research of this thesis:

Differences between prover versions Newer versions of the same automated theorem prover can act differently than their predecessor. Why3 even allows us to use different versions in parallel. This raises the question: what if an older version finds a proof of something, but a newer version does not? Does that mean we are relying in a soundness bug in the older version?

Differences between runs of the same prover Theorem provers usually require a random seed, which guides their search for a proof. This means that the time taken to find a proof may vary between runs, sometimes wildly. This can be a problem for the reproducibility of results obtained earlier.

Generation of prover input Why3 has to transform and generate input for the various theorem provers. This demonstrably contains some bugs. For example, if we define a predicate with the name `equiv`, and run the theorem prover Z3, we can see some error messages indicating a name clash with a built-in symbol. Also, generated Coq files have on occasion contained name clashes.

Bugs in the WhyML front-end The following program will be verified by Why3 v0.86.3, even though it appears to reach an unreachable statement (“`absurd`”):

```
let oops ()
= if false then 'A: begin end;
  absurd
```

The label `'A` in front of the empty statement appears to add an axiom for `false` to the logical system, suppressing the generation of verification conditions after the point it occurs. This bug is unlikely to occur ‘in the wild’ as the use of a label in this case does not make sense; and its effects are immediately obvious — but still, it is a soundness bug.

The possibility of inconsistencies in a Why3 theory It is easy to introduce new axioms at various places, using the `axiom`, `val` or `clone` declarations; each of which may introduce an inconsistency. First-order logic is hard, and typo’s are easily made [52]. Ironically, the one sure sign that a theory we constructed is inconsistent is, of course, that all theorem provers have no trouble finishing all proof obligations in record time.

Soundness bugs in provers We have encountered one soundness bug in the commonly used Z3 theorem prover, version 4.3.2. Undoubtedly, many more exist. While we have not needed Z3 much to verify the results in this thesis, it is likely that Alt-Ergo, CVC3 and CVC4 are equally susceptible to problems.

The reader with an interest in software verification from solid foundations will at this point feel somewhat uneasy, and not unjustly so. However, Why3 is a tool which is highly pragmatic in nature; it can be highly useful in detecting and preventing bugs in software with a much larger accuracy than software testing. Also, there are several precautions that we can take to bolster our believe in the correctness proofs that it delivers.

Foremost among these is viewing Why3 as a system in which we can formally specify the conditions for a correctness proof that should also appeal to human understanding. One of the major appeals of a WhyML annotated programs, as opposed to, say, a Coq proof is that the conceptual gap between annotated WhyML code and ‘real’ code is much smaller. Ideally, using Why3 can result in better programs (and better programmers) without getting bogged down by many ‘boring’ individual proofs. On the other hand, if one is only interested in the verdict of a system, it is beyond a doubt that Coq’s *“Proof completed”* has more authority to it than the list of green check-marks seen in the Why3 IDE.

Second, bugs in the generation of prover input (or in the provers themselves) can be avoided by applying as many as possible to every verification condition. If two different automated provers manage to prove a statement, the chances that this is due to a soundness bug are obviously much lower than if we had just used one.

Finally, a Why3 user can avoid introducing inconsistent proofs by being diligent. For example, we have often used Why3’s support for set theory to construct logical objects, instead of defining them by introducing new axioms. In similar vein, by applying the following set of guidelines, we can increase our trust in verification using Why3:

- If a theory contains an axiom, this theory should be cloned at least once, where all abstract symbols get instantiated and all axioms are verified as goals — thus ensuring the satisfiability and thereby the internal consistency of the theory. Similarly, every time an abstract symbol gets instantiated during cloning, all axioms mentioning it should be turned into lemmas in the `clone` directive.

- No `assume` statements should be used, and no `any`-expressions with an `ensures` post-condition, since these allow us to introduce axioms in a WhyML program.
- If an abstract function is defined using `val`, it should only be used to 'forward' an already existing pure Why function to the WhyML world.
- If Coq is used to discharge proof obligations, the assumptions made in the proof should be inspected. It is easy to prove false statements using Coq: the `admit` tactic is part of its feature set, and Why3 generates many axioms in Coq input files (see page 18).

These checks can in principle be made automatically; we suggest the addition of a `why3 lint` command that performs these. Also note that some WhyML constructions given in this thesis violate some of these criteria; in particular the reachability hack of Section 3.3 would be challenging to rewrite in a way to make it pass such a `why3 lint` check.

6.2. Future work

The method in Chapter 5 is necessarily incomplete, due to the limited scope of this research. We can identify several areas in which further work would be needed to make the contribution of this thesis practically useful:

- The transformation of Go programs to WhyML code is tedious to do by hand; especially when also proving liveness of programs. However, the method was designed to be mechanizable. A program should be developed which actually implements this mechanization. This program should ideally also try to deduce the liveness annotations necessary.
- Go's `select` statement, and programs where a channel can have more than one reader or writer, are not yet supported. Adding these is unlikely to present insurmountable difficulties.
- Closing channels (using the `close` built-in) is important for the runtime of Go, as it allows goroutines that can no longer be communicated with to be garbage collected. Support for `close` should be added, in concert with an method to prove that channels that have not been closed will not leak.
- Support for the non-concurrent fragment of Go is very limited; in particular, we cannot handle any data type besides `int` and `bool`. Structured data types (arrays, slices, structs) and pointers need to be added. When adding pointers, we should also add a notion of ownership that can be transferred between processes when a pointer is passed over a channel, and start proving non-interference properties explicitly using Why3.
- Channels are ordinary data types in Go, and can be passed around freely, which our method cannot handle. To extend it in this direction could, perhaps, be achieved by 'linking' two channels logically, verifying that the pre- and post-conditions are logically compatible, and showing the absence of channel aliases.
- A dedicated assertion language for Go would need to be developed for a more seamless integration between Go and Why3; similar to the ACSL language used by Frama-C/Jessie [22].

It would also be interesting to look at a combination of theorem proving and model checking — where an automated theorem prover is used to prove the Go program equivalent to some abstract automaton, which is then proved to be free of deadlocks/livelocks using a model checker.

6.3. Why verified software?

A common response to proposals for formal verification is a sympathetic, but sceptical one: the generally held view is still that it is too hard and costs too much time. In the introduction, we have responded to this by stating that in certain application areas, the certainty provided by formal methods is still a worthwhile goal.

To see this, let us assume we have a specification for a piece of software. According to it, potential actions by software can be classified into three categories:

- Actions that the program *must not* do, for example: *"stepping on a brake pedal should not accelerate a vehicle"*.
- Actions a program *must* do: *"stepping on a brake pedal should slow down a vehicle"*.
- Actions that the specification does not mention, such as the car radio being tuned to a different station when the brake pedal is used.

The first is an example of *negative correctness*. Formal verification is not required to ensure this kind of behaviour — runtime checks can be used to prevent bad things from happening. For example, Go checks automatically that array indexes are not out of bounds.

The second case is an example of *positive correctness*. In these scenarios, runtime checking does not help. It is no good if the software component responding to the brake pedal keeps aborting because of a failed run-time check.

The third case illustrates *abstraction*, and this is in some sense the most important one to consider. Ultimately, the goal of a specification is to be a useful tool for deciding on the acceptability of a piece of software; things must be left out for it to be understandable.

Using Why3 allows us to focus on deriving such a specification, instead of on performing all the proofs in every detail. In the future, by moving towards increasingly higher degrees of automation, we may end up with with a practical method for formally deducing, given a set of requirements for a program as a whole, a minimal set of conditions that each component must satisfy — and verification will be not so much about proving correctness, but more about deriving proper specifications; so that we can eventually know what the critical software we are relying on is actually up to.

A. Complexity of heapsort verified using WhyML

In the following, the Counter module implements the ghost counter as described on page 32, and the log function is as described in Section 2.3.4.

```
module HeapSort

  use import int.Int
  use import int.ComputerDivision
  use import ref.Ref
  use import array.Array
  use import array.ArraySwap
  use import array.ArrayPermut
  use import Logarithm
  clone Counter as C

  function parent (pos: int): int = div (pos-1) 2

  predicate heap (a: array int) (start: int) (limit: int) =
    forall i: int. start <= parent i /\ i < limit -> a[i] <= a[parent i]

  predicate broken_heap (a: array int) (start: int) (limit:int) (hole: int) =
    (forall i: int. start <= parent i /\ i < limit /\ parent i <> hole -> a[i] <= a[parent i]) /\
    (forall i: int. start <= parent hole < hole = parent i /\ i < limit -> a[i] <= a[parent hole])

  lemma permut_sub_transitive:
    forall a b c: array int, l u: int.
      permut_sub a b l u -> permut_sub b c l u -> permut_sub a c l u

  let push_down (a: array int) (pos: int) (limit: int)
    requires { 0 <= pos < limit <= length a }
    requires { broken_heap a pos limit pos }
    ensures  { heap a pos limit }
    ensures  { permut_sub (old a) a pos limit }
    ensures  { !C.count - old !C.count <= 2*log 2 limit }
  = 'Start:
    let cur = ref pos in
    while !cur*2+1 < limit do
      invariant { pos <= !cur <= limit }
      invariant { broken_heap a pos limit !cur }
      invariant { permut_sub (at a 'Start) a pos limit }
      variant { limit - !cur }
      invariant { !C.count - at !C.count 'Start <= 2*log 2 (if !cur < limit then !cur+1 else limit) }
      let child = ref (!cur*2+1) in
      if !child+1 < limit && C.event (a[!child] < a[!child+1]) then
        child := !child+1;
      if C.event (a[!cur] < a[!child]) then begin
        swap a !cur !child;
        assert { heap a pos (!cur+1) };
        cur := !child;
      end else
        cur := limit;
    done

  let make_heap (a: array int)
    ensures { heap a 0 (length a) }
    ensures { permut_all (old a) a }
    ensures { let n = length a in !C.count - old !C.count <= n*2*log 2 n }
  = 'Start:
    for i = div (length a) 2-1 downto 0 do
      invariant { heap a (i+1) (length a) }
      invariant { permut_all (at a 'Start) a }
      invariant { let n = length a in !C.count - at !C.count 'Start <= (n - i - 1) * 2*log 2 n }
      push_down a i (length a);
    done

  inductive descendant int int =
  | Refl: forall n: int. descendant n n
  | Step: forall p q: int. p <= q /\ descendant p (parent q) -> descendant p q

  lemma descendant_root_order:
    forall a: array int, pos limit i: int.
      0 <= pos <= i < limit <= length a -> heap a pos limit -> descendant pos i -> a[pos] >= a[i]

  predicate descendant_of_0 (n: int) = descendant 0 n
  clone int.Induction with predicate p = descendant_of_0, constant bound = zero

  lemma heap_top:
    forall a: array int, n m: int. 0 <= m < n <= length a /\ heap a 0 n -> a[0] >= a[m]

  let sort (a: array int)
    ensures { forall n m: int. 0 <= n <= m < length a -> a[n] <= a[m] }
    ensures { permut_all (old a) a }
    ensures { let n = length a in !C.count - old !C.count <= 4*n*log 2 n }
  = 'Start:
    make_heap a;
    'Loop:
    for i = length a-1 downto 1 do
      invariant { forall n m: int. i < n <= m < length a -> a[n] <= a[m] }
      invariant { forall n m: int. 0 <= n <= i < m < length a -> a[n] <= a[m] }
      invariant { heap a 0 (i+1) }
      invariant { permut_all (at a 'Start) a }
      invariant { let n = length a in !C.count - at !C.count 'Loop <= (n - i - 1) * 2*log 2 n }
      swap a 0 i;
      assert { forall n: int. 0 <= n < i -> a[n] <= a[i] };
      push_down a 0 i;
      assert { forall n: int. 0 <= n < i -> a[n] <= a[i] };
    done

end
```

B. Concurrent prime number sieve in Go, with liveness annotations

```
package main

import "fmt"

//requires: cursor out = 0
//startup: this => out
//ensures[out]: forall x. init <= x < cursor out + init <-> mem x out
//ensures[out]: forall i. 0 <= i < cursor out -> out[i] = init+i
//ensures[out]: this => out
func generate(init int, out chan<- int) {
    x := init
    //invariant: forall x: int. init <= x < cursor out + init <-> mem x out
    //invariant: forall i. 0 <= i < cursor out -> out[i] = init+i
    //invariant: cursor out + init = !x
    //obligation: this => out
    for {
        out <- x
        x++
    }
}

//requires: cursor src = 0
//startup: this => src
//requires[src]: forall x. mem x src -> 2 <= x /\ forall d. 2 <= d -> divides d x -> mem d src
//ensures[src]: if exists x. mem x src /\ prime x /\ not mem_add x out (old out)
//                then this => out else this => src
//ensures[out]: forall x. mem_add x out (old out) -> mem x src /\ prime x
//ensures[out]: this => src
func sieve(src <-chan int, out chan<- int) {
    p := <-src
    out <- p
    filt := make(chan int)
    go sieve(filt, out) //process: go_sieve
    //invariant: forall x. mem x src /\ not (divides !p x) <-> mem x filt
    //invariant: mem_add !p out (old out)
    //invariant: if exists x. mem x filt /\ prime x /\ not mem_add x out (at out 'go_sieve)
    //             then go_sieve => out else go_sieve => filt
    //obligation: this => src
    //obligation: (exists x. mem x src /\ prime x /\ not mem_add x out (old out)) -> this => out
    for {
        n := <-src
        if n%p != 0 {
            //assert: forall x. mem x src /\ not (divides !p x) /\ x <> !n -> mem x filt
            //forward[filt] invariant: mem_add !p out (old out)
            //forward[filt] invariant: go_sieve => filt \/ go_sieve => out
            //forward[filt] invariant: go_sieve => filt \/ pre (cursor out)+1 >= cursor out
            //forward[filt] variant: pre (cursor out)+1 - cursor out
            //forward[filt] obligation: this => src
            //forward[filt] obligation: (exists x. mem x src /\ mem x prime /\
            //                           not mem_add x out (old out)) -> this => out
            filt <- n
        }
    }
}

func main() {
    outp := make(chan int)
    inp := make(chan int)
    go sieve(inp, outp) //process: go_sieve
    go generate(2, inp) //process: go_generate
    //invariant: go_generate => inp /\ go_sieve => inp
    //invariant: forall x. mem x outp -> mem x inp
    for {
        //forward[outp] invariant: go_generate => inp
        //forward[outp] invariant: go_sieve => inp \/ go_sieve => outp
        //forward[outp] invariant: forall x. mem x outp -> mem x inp
        //forward[outp] invariant: cursor inp >= pre (cursor inp)
        //forward[outp] invariant:
        //  (cursor inp > pre (cursor inp) -> not mem (last inp) outp) &&
        //  (go_sieve => outp \/ last inp < pre (some_larger_prime (last inp)))
        //forward[filt] variant: pre (some_larger_prime (last inp)) - cursor inp
        n := <-outp
        //check: prime !n
        fmt.Println(n)
    }
}
```

C. WhyML model of the concurrent prime number sieve with liveness annotations

```
module PrimeNumberSieve

use import ref.Ref
use import int.Int
use import int.ComputerDivision
use import chan.Chan as P
use import chan.Mem

use import number.Divisibility
use import number.Prime

predicate generate_go_pre (ready: process) (init: int) (out: chan int)
  = cursor out = 0
predicate generate_go_post (ready: process) (init: int) (out: chan int)
  = ready => out
predicate generate_post (ready: process) (init: int) (out: chan int)
  = (forall x: int. init <= x < cursor out+init <-> mem x out) /\
    (forall i. 0 <= i < cursor out -> out[i] = init+i) /\
    ready => out

let generate (init: int) (out: chan int): unit
  requires { generate_go_pre this init out }
  diverges
= P.init();
  P.oblige (any bool ensures { result <-> handled out });
  responsible out;
  assert { "expl:goroutine postcondition" generate_go_post this init out };
  let x = ref init in
  while true do
    invariant { "expl:type invariant preservation" 0 <= cursor out }
    invariant { forall x: int. init <= x < cursor out+init <-> mem x out }
    invariant { forall i. 0 <= i < cursor out -> out[i] = init+i }
    invariant { cursor out+init = !x }
    invariant { ?obligation <-> handled out }
'Begin:
    responsible out;
    send !x out this;
    check { "expl: obligations met" ?obligation };
    P.oblige (any bool ensures { result <-> handled out });
    responsible out;
    assert { "expl:channel postcondition" generate_post this init out };
    x := !x+1;
    check { "expl:conditional variant decrease"
            not (at ?obligation) 'Begin ->
            0 <= at (0) 'Begin /\ (0) < at (0) 'Begin
    };
  done;
  check { "expl: obligations met" ?obligation }

predicate sieve_go_pre (ready: process) (src: chan int) (out: chan int)
  = cursor src = 0
predicate sieve_go_post (ready: process) (src: chan int) (out: chan int)
  = ready => src
predicate sieve_pre_1 (ready: process) (src: chan int) (out: chan int) (old_out: chan int)
```

```
  = forall x: int. mem x src -> 2 <= x /\ forall d: int. 2 <= d -> divides d x -> mem d src
predicate sieve_post_1 (ready: process) (src: chan int) (out: chan int) (old_out: chan int)
  = (if exists x: int. mem x src /\ prime x /\ not mem_add x out old_out then ready => out else ready => src)
predicate sieve_post_2 (ready: process) (src: chan int) (out: chan int) (old_out: chan int)
  = (forall x: int. mem_add x out old_out -> mem x src /\ prime x) /\
    ready => src

meta "rewrite_def" predicate sieve_pre_1
meta "rewrite_def" predicate sieve_post_2

let sieve (src: chan int) (out: chan int): unit
  requires { sieve_go_pre this src out }
  diverges
= 'Start:
  P.init();
  P.oblige (any bool ensures { result <-> handled src });
  responsible src;
  responsible out;
  assert { "expl:goroutine postcondition" sieve_go_post this src out };

  responsible src;
  responsible out;
  let p = ref (recv src this) in
  check { "expl: obligations met" ?obligation };

  P.oblige (any bool ensures { result <-> handled out });
  responsible src;
  responsible out;
  assume { "expl:channel precondition" sieve_pre_1 this src out (at out 'Start) };
  assert { "expl:channel postcondition" sieve_post_1 this src out (at out 'Start) };

  responsible src;
  responsible out;
  send !p out this;
  check { "expl: obligations met" ?obligation };
  P.oblige (any bool ensures { result <-> handled src });
  responsible src;
  responsible out;
  assert { "expl:channel postcondition" sieve_post_2 this src out (at out 'Start) };

  let filt = make(): chan int in
  let go_sieve = any process in
  check { "expl:recursion safety" cursor src > at (cursor src) 'Start \/ cursor out > at (cursor out) 'Start };
  assert { "expl:goroutine precondition"  sieve_go_pre go_sieve filt out };
  assume { "expl:goroutine postcondition" sieve_go_post go_sieve filt out };
'Entry_sieve:
  while true do
    invariant { "expl:type invariant preservation" 0 <= cursor src /\ 0 <= cursor out /\ 0 <= cursor filt }
    invariant { forall x: int. mem x src /\ not (divides !p x) <-> mem x filt }
    invariant { mem_add !p out (at out 'Start) }
    invariant { if exists x: int. mem x filt /\ prime x /\ not mem_add x out (at out 'Entry_sieve)
                  then go_sieve => out else go_sieve => filt }
    invariant { ?obligation <-> handled src /\ ((exists x: int. mem x src /\ prime x /\
                                               not mem_add x out (at out 'Start)) -> handled out) }
'Begin:
    delegated go_sieve out;
    responsible src;
    let n = ref (recv src this) in
    check { "expl: obligations met" ?obligation };

    P.oblige (any bool ensures { result <-> handled src });
    P.oblige (any bool ensures { result <-> (exists x: int. mem x src /\ prime x /\
                                           not mem_add x out (at out 'Start)) -> handled out });
    delegated go_sieve out;
    responsible src;
```

```
      assume { "expl:channel precondition" sieve_pre_1 this src out (at out 'Start) };
      assert { "expl:channel postcondition" sieve_post_1 this src out (at out 'Start) };

      if (mod !n !p <> 0) then begin
        assert { forall x. mem x src /\ not (divides !p x) /\ x <> !n -> mem x filt };

'Fwd:
        while not any bool ensures { result <-> go_sieve => filt } do
          invariant { "expl:type invariant preservation" 0 <= cursor out /\ 0 <= cursor src /\ 0 <= cursor filt }
          invariant { mem_add !p out (at out 'Start) }
          invariant { ?obligation <-> handled src /\ ((exists x: int. mem x src /\ prime x /\
                                                  not mem_add x out (at out 'Start))-> handled out) }
          invariant { go_sieve => filt \/ go_sieve => out }
          invariant { go_sieve => filt \/ at (cursor out) 'Fwd+1 >= cursor out }
          variant { at (cursor out) 'Fwd+1 - cursor out }
          delegated go_sieve out;
          responsible src;
          out[this <- go_sieve];

          P.oblige (any bool ensures { result <-> (exists x: int. mem x src /\ prime x /\
                                                not mem_add x out (at out 'Start)) -> handled out });
          delegated go_sieve out;
          responsible src;
          assume { "expl:channel postcondition" sieve_post_2 go_sieve filt out (at out 'Entry_sieve) };
          assert { "expl:channel postcondition" sieve_post_2 this src out (at out 'Start) };
        done;

        send !n filt go_sieve;
        assert { "expl:channel precondition" sieve_pre_1 go_sieve filt out (at out 'Entry_sieve) };
        assume { "expl:channel postcondition" sieve_post_1 go_sieve filt out (at out 'Entry_sieve) };
      end;
      check { "expl:conditional variant decrease"
              not (at ?obligation) 'Begin ->
              0 <= at (0) 'Begin /\ (0) < at (0) 'Begin
      };
    done;
    check { "expl:obligations met" ?obligation }

use import infprime.InfinitePrimes

let main (): unit
  diverges
= let outp = make(): chan int in
  let inp = make(): chan int in
  let go_sieve = any process in
  assert { "expl:goroutine precondition"  sieve_go_pre go_sieve inp outp };
  assume { "expl:goroutine postcondition" sieve_go_post go_sieve inp outp };
'Entry_sieve:
  let go_generate = any process in
  assert { "expl:goroutine precondition"  generate_go_pre go_generate 2 inp };
  assume { "expl:goroutine postcondition" generate_go_post go_generate 2 inp };
  while true do
    invariant { "expl:type invariant preservation" 0 <= cursor inp /\ 0 <= cursor outp }
    invariant { go_generate => inp /\ go_sieve => inp }
    invariant { forall x. mem x outp -> mem x inp }

'Fwd:
    while not any bool ensures { result <-> go_sieve => outp } do
      invariant { "expl:type invariant preservation" 0 <= cursor inp /\ 0 <= cursor outp }
      invariant { go_generate => inp /\ (go_sieve => inp \/ go_sieve => outp) }
      invariant { forall x. mem x outp -> mem x inp }
      invariant { cursor inp >= at (cursor inp) 'Fwd }
      invariant { (cursor inp > at (cursor inp) 'Fwd -> not mem inp[cursor inp-1] outp) &&
                  (go_sieve => outp \/ inp[cursor inp-1] < at (some_larger_prime inp[cursor inp-1]) 'Fwd) }
      variant { at (some_larger_prime inp[cursor inp-1]) 'Fwd - cursor inp }
```

```
'Begin:
      inp[go_sieve <- go_generate];
      assume { "expl:channel postcondition" generate_post go_generate 2 inp };
      assert { "expl:channel precondition" sieve_pre_1 go_sieve inp outp (at outp 'Entry_sieve) };
      assume { "expl:channel postcondition" sieve_post_1 go_sieve inp outp (at outp 'Entry_sieve) };
    done;

    let n = ref (recv outp go_sieve) in
    assume { "expl:channel postcondition" sieve_post_2 go_sieve inp outp (at outp 'Entry_sieve)  };
    check { prime !n };
  done

end
```

Bibliography

[1] F. L. Morris and C. B. Jones, "An early program proof by Alan Turing," *IEEE Ann. Hist. Comput.*, vol. 6, pp. 139–143, Apr. 1984.

[2] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.

[3] M. W. Maimone, P. C. Leger, and J. J. Biesiadecki, "Overview of the mars exploration rovers autonomous mobility and vision capabilities," in *IEEE International Conference on Robotics and Automation (ICRA*, 2007.

[4] A. Langley, "Apple's SSL/TLS bugs." `https://www.imperialviolet.org/2014/02/22/applebug.html`, 2014.

[5] "The Coq proof assistant reference manual." Accessible at `https://coq.inria.fr/documentation`, 2015.

[6] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[7] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, (London, UK, UK), pp. 748–752, Springer-Verlag, 1992.

[8] S. Schulz, "System Description: E 1.8," in *Proc. of the 19th LPAR, Stellenbosch* (K. McMillan, A. Middeldorp, and A. Voronkov, eds.), vol. 8312 of *LNCS*, Springer, 2013.

[9] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski, "SPASS version 3.5," in *Proceedings of the 22Nd International Conference on Automated Deduction*, CADE-22, (Berlin, Heidelberg), pp. 140–145, Springer-Verlag, 2009.

[10] S. Conchon and E. Contejean, "The Alt-Ergo automatic theorem prover." `http://alt-ergo.lri.fr/`, 2008.

[11] C. Barrett and C. Tinelli, "CVC3," in *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)* (W. Damm and H. Hermanns, eds.), vol. 4590 of *Lecture Notes in Computer Science*, pp. 298–302, Springer-Verlag, July 2007. Berlin, Germany.

[12] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, (Berlin, Heidelberg), pp. 171–177, Springer-Verlag, 2011.

[13] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[14] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[15] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, pp. 279–295, May 1997.

[16] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, (London, UK, UK), pp. 359–364, Springer-Verlag, 2002.

[17] D. R. Cok, *OpenJML: JML for Java 7 by Extending OpenJDK*, pp. 472–479. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[18] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *Theorem Proving in Higher Order Logics*, pp. 23–42, Springer, 2009.

[19] S. Böhme, M. Moskal, W. Schulte, and B. Wolff, "HOL-Boogie - an interactive prover-backend for the Verifying C Compiler," *Journal of Automated Reasoning*, vol. 44, no. 1-2, pp. 111–144, 2010.

[20] B. Jacobs, J. Smans, and F. Piessens, "A quick tour of the VeriFast program verifier," in *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, APLAS'10, (Berlin, Heidelberg), pp. 304–311, Springer-Verlag, 2010.

[21] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C," in *Software Engineering and Formal Methods*, pp. 233–247, Springer, 2012.

[22] C. Marhé and Y. Moy, "The Jessie plugin for deductive verification in Frama-C," *INRIA Saclay Île-de-France and LRI, CNRS UMR*, 2012.

[23] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Why3: Shepherd your herd of provers," in *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pp. 53–64, 2011.

[24] R. Krebbers, *The C standard formalized in Coq*. PhD thesis, PhD thesis, Radboud University, 2015. Manuscript accepted by the committee, available online at http://robbertkrebbers. nl/research/thesis_draft. pdf, 2015.

[25] N. D. Matsakis and F. S. Klock, II, "The rust language," *Ada Lett.*, vol. 34, pp. 103–104, Oct. 2014.

[26] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. Addison-Wesley Professional, 1st ed., 2015.

[27] R. Hamberg and F. Vaandrager, "Using model checkers in an introductory course on operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 101–111, Oct. 2008.

[28] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, Aug. 1978.

[29] J.-C. Filliâtre, "Verification of non-functional programs using interpretations in type theory," *J. Funct. Program.*, vol. 13, pp. 709–745, July 2003.

[30] J. C. Filliâtre and C. Marché, "The Why/Krakatoa/Caduceus platform for deductive program verification," in *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, (Berlin, Heidelberg), pp. 173–177, Springer-Verlag, 2007.

[31] D. Hoang, Y. Moy, A. Wallenburg, and R. Chapman, "SPARK 2014 and GNATprove," *Int. J. Softw. Tools Technol. Transf.*, vol. 17, pp. 695–707, Nov. 2015.

[32] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, "Easycrypt: A tutorial," in *Foundations of Security Analysis and Design VII*, pp. 146–166, Springer, 2014.

[33] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.

[34] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Let's verify this with Why3," *Int. J. Softw. Tools Technol. Transf.*, vol. 17, pp. 709–727, Nov. 2015.

[35] J.-C. Filliâtre, "One logic to use them all," in *Proceedings of the 24th International Conference on Automated Deduction*, CADE'13, (Berlin, Heidelberg), pp. 1–20, Springer-Verlag, 2013.

[36] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich, *Preserving User Proofs across Specification Changes*, pp. 191–201. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

[37] P. Suppes, *Axiomatic set theory*. Courier Corporation, 1960.

[38] Wikipedia, "Dining philosophers problem — Wikipedia, the free encyclopedia," 2016. [Online; accessed 23-May-2016].

[39] Wikipedia, "Sieve of Eratosthenes — Wikipedia, the free encyclopedia," 2016. [Online; accessed 22-May-2016].

[40] C. Plumb, "Computing inverses modulo n," 1994. Accessible at `ftp://ftp.csc.fi/index/crypt/math/inverses-modulo-n.txt`.

[41] J.-C. Filliâtre, L. Gondelman, and A. Paskevich, "The spirit of ghost code," in *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, (New York, NY, USA), pp. 1–16, Springer-Verlag New York, Inc., 2014.

[42] D. R. Musser, “Introspective sorting and selection algorithms,” *Softw., Pract. Exper.*, vol. 27, no. 8, pp. 983–993, 1997.

[43] A. Tafat and C. Marché, “Binary heaps formally verified in Why3,” Research Report RR-7780, INRIA, Oct. 2011.

[44] J.-C. Filliâtre, “Formal verification of MIX programs,” *Journées en l’honneur de Donald E. Knuth, Bordeaux, France*, 2007.

[45] B. W. Kernighan, *Why Pascal is not my favorite programming language.* Bell Laboratories, 1981.

[46] A. Van Wijngaarden, B. Mailloux, J. Peck, C. Koster, M. Sintzoff, C. Lindsey, L. Meertens, and R. Fisker, “Revised Report on the Algorithmic language,” *Acta Informatica*, vol. 5, pp. 1–236, 1975.

[47] S. Owicki and D. Gries, “An axiomatic proof technique for parallel programs i,” *Acta Informatica*, vol. 6, no. 4, pp. 319–340, 1976.

[48] G. M. Levin and D. Gries, “A proof technique for communicating sequential processes,” *Acta Informatica*, vol. 15, no. 3, pp. 281–302, 1981.

[49] N. Soundararajan, “Axiomatic semantics of communicating sequential processes,” *ACM Trans. Program. Lang. Syst.*, vol. 6, pp. 647–662, Oct. 1984.

[50] T. P. Murtagh, “Redundant proofs of non-interference in Levin-Gries CSP program proofs,” *Acta Informatica*, vol. 24, no. 2, pp. 145–156, 1987.

[51] R. Pollack, “How to believe a machine-checked proof,” *Twenty Five Years of Constructive Type Theory*, vol. 36, p. 205, 1998.

[52] A. Paskevich, “[Why3-club] Equivalence an asymmetric relation?.” `http://lists.gforge.inria.fr/pipermail/why3-club/2014-June/001090.html`.