Let's Verify This with Why3

François Bobot, Jean-Christophe Filliâtre, Claude Marché, Andrei Paskevich

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Let's Verify This with Why3⋆

François Bobot[1], **Jean-Christophe Filliâtre**[2,3,4], **Claude Marché**[3,2,4], **Andrei Paskevich**[2,3,4]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
[2] Laboratoire de Recherche en Informatique (CNRS UMR 8623)
[3] Inria Saclay – Île-de-France
[4] Université Paris-Sud, France

Received: date / Revised version: date

Abstract. We present solutions to the three challenges of the VerifyThis competition held at the 18th FM symposium in August 2012. These solutions use the Why3 environment for deductive program verification.

1 Introduction

The Why3 environment for deductive program verification is built around a kernel that implements a formal specification language, based on typed first-order logic. Logical goals can be proved using a large set of automated and interactive external theorem provers, such as Alt-Ergo [4], CVC3 [2], CVC4 [1], Z3 [8], E [17], SPASS [19], Vampire [16], Coq [3], or PVS [15]. When a goal is sent to a prover that does not support some features of the language, Why3 applies a series of encoding transformations, for example, to eliminate pattern matching or polymorphic types [7].

On top of this kernel, Why3 features a programming language WhyML, where functions can be formally specified using contracts. A VC generator produces proof obligations that need to be discharged to prove that a program respects its specification [9].

In this paper we illustrate the use of Why3 by providing solutions to the three challenges that were given at the VerifyThis competition, held at the 18th FM symposium in August 2012. The description of the challenges can be found at `http://fm2012.verifythis.org/challenges/`. Reference Java implementations were given for the first two problems, and an algorithm in pseudocode was given for the third one.

We entered the competition with two teams, each with two members, both using Why3. By the end of the competition, our teams had partial solutions for the proposed challenges. After the competition, we merged our teams. For each challenge, we took the better approach of two as a basis upon which we built a complete solution. We estimate that we spent approximately 30 person-hours in that process. Our solutions use Why3 version 0.82 [6].

⋆ Work partly supported by the Bware project of the French national research organization (ANR-12-INSE-0010, `http://bware.lri.fr/`)

2 Why3 in a Nutshell

In this section we briefly describe Why3: the pure logic language of specifications, the programming language WhyML, and some peculiarities of program verification in Why3.

2.1 Specification Language

Why3 is based on first-order logic with ML-style polymorphic types and several extensions: recursive definitions, algebraic data types, and inductive predicates. The specification language of Why3 does not depend on any features of its programming language, and can serve as a rich common format of theorem proving problems, readily suitable (via Why3) for multiple external provers.

Types. Built-in types in Why3 include integers (`int`), real numbers (`real`), and tuples. A user-defined type can be non-interpreted or be a synonym for a type expression:

```
type map α β
type i_map γ = map int γ
```

Otherwise, it is an algebraic data type or a record. For instance, polymorphic lists and binary trees are defined in the standard library of Why3 as follows:

```
type list α = Nil | Cons α (list α)
type tree α = Empty | Node (tree α) α (tree α)
```

Record types are a special case of algebraic types with a single unnamed constructor and named fields. Here is a definition of a banker's queue:

```
type queue α = { front: list α; rear: list α }
```

All types must be inhabited and Why3 checks that every algebraic type declaration admits at least one value. For example, the above definition of the list type would be rejected without constructor `Nil`.

Function and Predicate Symbols. Every function or predicate symbol in Why3 has a polymorphic type signature. For example, an abstract function that applies a mapping to an element of the domain can be declared as follows:

```
function get (map α β) α : β
```

Both functions and predicates can be given definitions, possibly mutually recursive. As examples, we can specify that an `int`-to-`int` mapping is strictly increasing:

```
predicate increasing (m: map int int) =
  forall i j: int. i < j → get m i < get m j
```

or calculate the height of a tree:

```
function height (t: tree α): int = match t with
  | Node l _ r → 1 + max (height l) (height r)
  | Leaf       → 0
end
```

Why3 automatically verifies that recursive definitions are terminating. To do so, it looks for an appropriate lexicographic order of arguments that guarantees a structural descent. Currently, we only support recursion over algebraic types. Other kinds of recursive functions have to be axiomatized or defined as programs, where termination is proved using *variants* (see Sec. 2.2).

Another extension to the first-order language adopted in Why3 is inductive predicates. Such a predicate is the least relation satisfying a set of clauses. For instance, the subsequence relation over lists is inductively defined as follows:

```
inductive sub (list α) (list α) =
  | null : sub (Nil: list α) (Nil: list α)
  | cons : forall x: α, s1 s2: list α.
      sub s1 s2 → sub (Cons x s1) (Cons x s2)
  | dive : forall x: α, s1 s2: list α.
      sub s1 s2 → sub s1 (Cons x s2)
```

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

Terms and Formulas. First-order language is extended, both in terms and formulas, with pattern matching, **`let`**-expressions, and conditional expressions. We stay faithful to the usual distinction between terms and formulas that is made in first-order logic. Thus we make a difference between a predicate symbol and a function symbol which returns a `bool`-typed value, `bool` being defined with **`type`** `bool = True | False`. However, to facilitate writing, conditional expressions are allowed in terms, as in the following definition of absolute value:

```
function abs (x: int) : int =
  if x ≥ 0 then x else -x
```

Such a construct is directly accepted by provers not making a distinction between terms and formulas. In order to translate **`if-then-else`** constructs to traditional first-order language, Why3 lifts them to the level of formulas and rewrites them as conjunctions of two implications.

Theories. Pure logical definitions, axioms and lemmas are organized in collections, called *theories*. The standard library of Why3 contains numerous theories describing integer and real arithmetic, lists, binary trees, mappings, abstract algebraic notions, etc. To provide a fine-grained control of the premise set, we favor small theories which build on each other and introduce one or two concepts, such as Euclidean division, list membership, or injective maps. Instruction **`use`** imports a theory into the current context:

```
use import list.List
```

2.2 Programming Language

WhyML can be seen as a dialect of ML with extensive support for specification annotations. Program functions are provided with pre- and postconditions for normal and exceptional termination, and loop statements are annotated with invariants. To ensure termination, recursive functions and **`while`**-loops can be given variants, *i.e.*, terms that decrease at each recursive call or iteration with respect to some well-founded ordering. Statically checked assertions can be inserted at arbitrary points in a program.

Verification conditions are generated using a standard weakest-precondition procedure. In order to produce first-order proof obligations, WhyML is restricted to the first order, too: Nested function definitions are allowed but higher-order functions are not. Furthermore, in order to keep proof obligations more tractable for provers and more readable (hence debuggable) for users, the type system of WhyML requires all aliases to be known statically, at the time of verification condition generation. This allows us to apply the Hoare-style rule for assignment, without resorting to a heap memory model. One consequence of this discipline is that recursive data types cannot have mutable components. As we demonstrate below, these restrictions do not preclude us from writing and verifying complex algorithms and data structures.

Types. WhyML extends the pure types of the specification language in several ways. First and foremost, the mutable state of a computation is exclusively embodied in mutable fields of record data types:

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

A program type can be provided with an invariant, *i.e.*, a logical property imposed on any value of the type:

```
type array α
  model { length: int; mutable elts: map int α }
  invariant { 0 ≤ self.length }
```

Type invariants in WhyML are verified at the function call boundaries. Since WhyML type system tracks aliases and side effects statically, it is easy to detect whenever a type invariant must be re-established. Keyword **model** in place of the equal sign means that, inside WhyML programs, type `array` is not a record, but an abstract data type. Thus, an attempt to access the `elts` field in a program would be rejected. However, inside specifications, `array` is a record and its fields may be accessed.

Finally, a record field (or, more generally, an argument of a constructor in an algebraic type) can be declared *ghost*. Ghost data and ghost computations serve strictly for verification purposes. In particular, a typical use case for ghost fields is to equip a data structure with a pure logical "view". For example, some intricate implementation of sparse matrices may carry a ghost field[1]:

```
type sparse_matrix α =
  { ghost view : map (int,int) α; ... }
```

Formal arguments of program functions, as well as locally defined variables, can also be declared ghost.

In order to guarantee that ghost data and computations do not interfere with the program and cannot affect its final result, WhyML type system imposes a number of restrictions on their use. Ghost data cannot be used in a non-ghost computation. Ghost computations cannot modify a non-ghost mutable value or raise exceptions that escape into non-ghost code. However, ghost computations can read non-ghost values and ghost data can be used in program specifications.

Function Prototypes. Unlike other ML variants, WhyML does not separate interface and implementation. One can freely mix in the same WhyML module fully implemented program functions with abstract function prototypes carrying only a type and a specification. For "model" types like `array`, whose structure is inaccessible from programs, function prototypes is the only way to provide a usable interface. Here is how lookup and update are specified for arrays:

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

val ([]←) (a: array α) (i: int) (v: α) : unit
  requires { 0 ≤ i < a.length }
  writes   { a }
  ensures  { a.elts = set (old a.elts) i v }
```

The names `([])` and `([]←)` define mixfix operators `a[i]` and `a[i] ← v`. Clauses **reads** and **writes** specify the side effects in function prototypes; for example, `a[i] ← v` modifies the mutable field of `a`, namely `a.elts`. The term `(old a.elts)` in the postcondition of the second prototype refers to the pre-call value of the field `a.elts`, before it is modified by `([]←)`.

[1] Notice that the non-ghost fields of `sparse_matrix` can still be used in regular, non-ghost code, which would not be the case were `sparse_matrix` declared as a **model** type.

Programs. The syntax of WhyML programs should give no surprise to anyone familiar with ML. As examples, let us show several functions to handle mutable references:

```
function (!) (x: ref α) : α = x.contents

let (!) (r:ref α) : α
  ensures { result = !r }
= r.contents

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

let incr (r: ref int) : unit
  ensures { !r = old !r + 1 }
= r := !r + 1
```

Contrary to function prototypes, we do not indicate **reads** and **writes** effects, since Why3 can extract this information from the code. Notice that the same prefix symbol `(!)` is used as the name for both a pure access function and a program function. Since program symbols cannot appear in specifications, `!r` in pre- and postconditions can only refer to the pure function. In the program code, `!r` will refer to the WhyML function.

The last definition shows that pure types, functions, and predicates are accepted in WhyML programs. For instance, the type of integers and basic arithmetic operations are shared between specifications and programs. The only exception is made for logic functions and predicates specified directly on program types: such symbols can only be used in specifications. One reason for this restriction is that these functions and predicates have uncontrolled access to ghost components of program types. Had we not reused the `(!)` operator, WhyML would reject the last definition reporting that pure function `(!)` cannot appear in a program.

Modules. Akin to pure logical theories, WhyML declarations and definitions are grouped into *modules*. A module may import logical theories or contain pure declarations. The standard library modules introduce mutable references, arrays, hash tables, stacks, and queues.

2.3 Verifying Programs with Why3

Simple Why3 formalizations can be verified directly from the command line: The `why3` tool can run a designated automated prover on each proof obligation generated from a WhyML file and report eventual proof failures. For more complex developments, Why3 provides an interactive graphical environment whose main window is shown in Fig. 1. The big tree on the left shows the current state of the *session*. The root of the tree is the WhyML file in works, the first-level nodes are theories and modules, and the second-level nodes are primary proof obligations: logical lemmas and verification conditions for top-level WhyML functions.

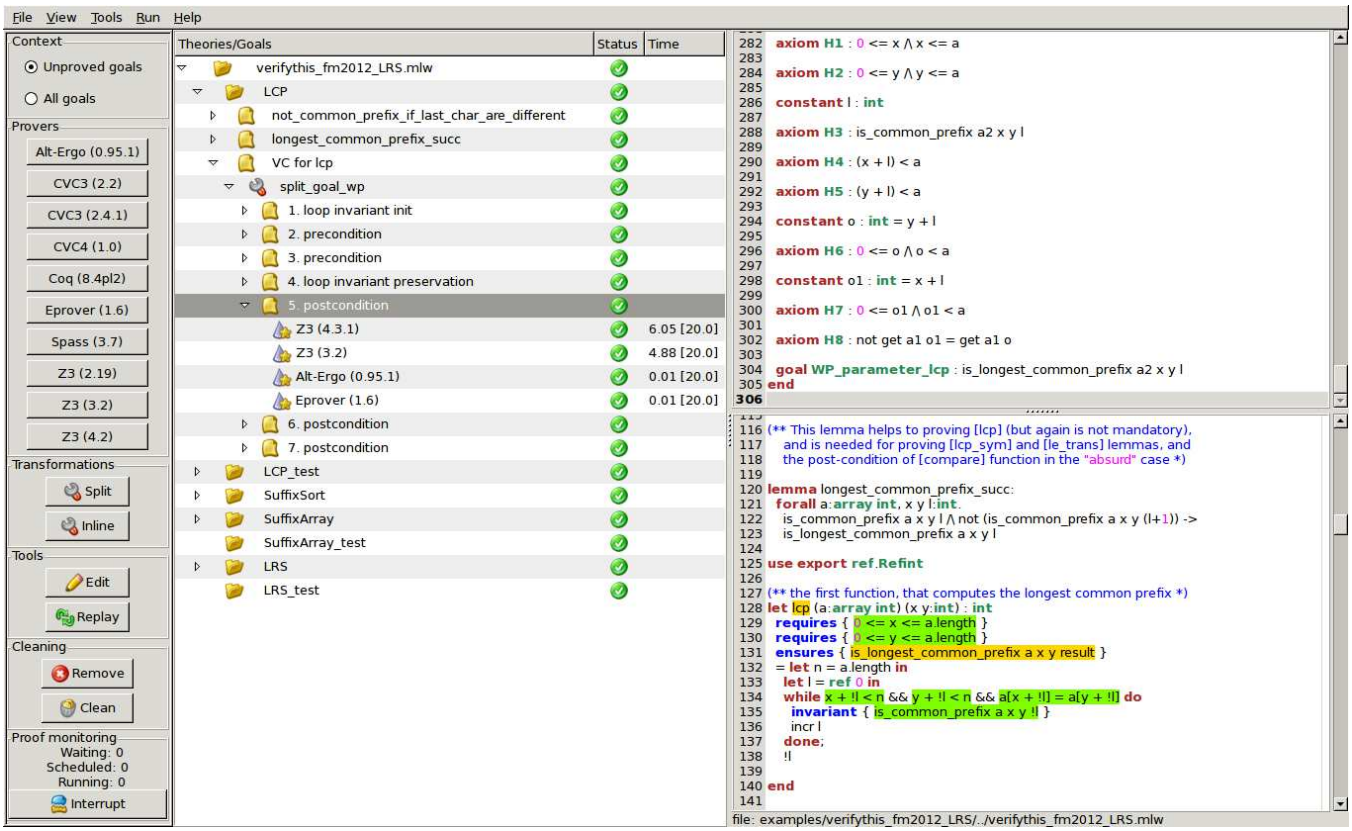


Fig. 1. Why3 graphical user interface.

Proof obligations, also called *proof tasks*, can be sent to automated provers or handled using interactive proof assistants (see the stack of buttons to the left of the session view). Why3 also puts at user's disposal a number of so-called *transformations* which can be used to simplify a proof obligation under consideration or to split it into a number of sub-tasks[2]. These sub-tasks appear at the lower levels of the session tree.

In the session shown on the figure, the file being checked is `verifythis_fm2012_LRS.mlw`. The first module in this file contains a WhyML function named `lcp` (whose source code is shown in the bottom-right frame), and the verification condition for this function is decomposed into seven sub-tasks: two preconditions ensuring safety of array access in the loop condition, loop invariant initialization and preservation, and three postconditions covering each branch of the negated loop condition. The cursor is positioned on the first postcondition, where the first two parts of the loop condition are true and the third one is false. The proof obligation itself is shown in the top-right frame. Four automated provers were able to discharge this proof task successfully.

Due to differences in prover technology (which are especially deep between SMT solvers and resolution-based provers), there is no single best prover for the purposes of program verification. Quite often, a proof obligation is only discharged by one or two provers out of half a dozen we use regularly. Being able to target diverse back-end provers is an important feature of Why3 which allows us to prove automatically more goals than we would be able to using just one dedicated prover.

Proof sessions are saved between runs of Why3, which facilitates development and debugging of verified programs. Special algorithms were devised for reconstruction of session trees after modification of the program code or specification [5]. Along with the session, Why3 stores proof scripts for interactive proof assistants (at this moment, Coq and PVS are supported) that handle proof obligations falling beyond the reach of automated provers. One characteristic case where one has to resort to interactive proof is proof tasks requiring reasoning by induction.

Another important functionality of Why3 is code extraction: a verified WhyML program can be translated to a compilable correct-by-construction OCaml program (a mechanism similar to that of Coq). Recently, work has started on a native WhyML evaluator which would allow for quick testing of programs and assertions, speeding up the development process.

[2] Another purpose of transformations is to eliminate and encode various high-level features of Why3 language, such as pattern matching or type polymorphism, in order to make proof obligations acceptable for a range of external provers. These transformations are invoked by a driver for a particular prover and do not need to be applied explicitly by the user.

3 Challenge 1: Longest Repeated Substring

This challenge aims at computing the *longest repeated substring* in a given string. The two occurrences of that substring are allowed to overlap. For example, the longest repeated substring in `ABCDBBCDA` and `ABCDBCDB` are respectively `BCD` and `BCDB`, with an overlap in the second case.

A naive algorithm that checks all possible occurrences of each substring would be of cubic complexity. This challenge proposes a more efficient algorithm, that amounts to computing the *array of suffixes* of the given string. It is an array of integers that lists the positions of the suffixes of the input string, in increasing lexicographic order. For example the sorted array of suffixes of `ABCDBCDB` is [0;1;4;7;2;5;3;6], corresponding to suffixes `ABCDBCDB`, `BCDBCDB`, `BCDB`, `B`, `CDBCDB`, `CDB`, `DBCDB`, `DB` in that order. The longest repeated substring is then the *longest common prefix* of two consecutive strings in this array.

A preliminary step of this challenge is thus to compute the longest common prefix of two given substrings of the input; this part was expected to be solved during the competition.

We provide a complete solution to the challenge. The Why3 code follows the Java reference implementation as closely as possible. A noticeable difference is that the function for sorting does not apply to a `SuffixArray` object: because the Java reference calls the `sort` method inside a constructor[3] which is not possible in Why3. Our solution is made of four Why3 modules:

- `LCP`: computation of longest common prefix of substrings.
- `SuffixSort`: helper functions for sorting suffixes, roughly corresponding to the private part of the `SuffixArray` class of the Java code.
- `SuffixArray`: the suffixArray data type, roughly corresponding to the public part of the `SuffixArray` class of the Java code.
- `LRS`: computation of longest repeated substring, corresponding to the `LRS` class of the Java code.

In the following we detail each module in turn, presenting first its specifications, and then the implementation and proofs. The full solution, including the complete Why3 input and the proof results, is available on the Toccata gallery of verified programs at `http://toccata.lri.fr/gallery/verifythis_fm2012_LRS.en.html`.

3.1 Longest Common Prefix

Our first module `LCP` implements the first part of the challenge: a function that computes the longest common prefix at two given indexes.

Specification. The specifications are given in Fig. 2. We import arrays from the Why3 standard library. The predicate `(is_common_prefix a x y l)` is true when the prefixes of length `l` at respective positions `x` and `y` in array `a` are identical, *i.e.*, when `a[x..x+l-1]` and `a[y..y+l-1]` are equal. The predicate `(is_longest_common_prefix a x y l)` is true when `l` is the maximal length of common prefixes at positions `x` and `y` in array `a`. The postcondition of the `lcp` function is the natural way of specifying that this function computes the longest common prefix.

```
use import array.Array

predicate is_common_prefix (a: array int) (x y l: int) =
  0 ≤ l ∧ x+l ≤ a.length ∧ y+l ≤ a.length ∧
  (forall i:int. 0 ≤ i < l → a[x+i] = a[y+i])

predicate is_longest_common_prefix (a: array int) (x y l:int) =
  is_common_prefix a x y l ∧
  forall m:int. l < m → ¬ (is_common_prefix a x y m)

val lcp (a: array int) (x y: int) : int
  requires { 0 ≤ x ≤ a.length }
  requires { 0 ≤ y ≤ a.length }
  ensures { is_longest_common_prefix a x y result }
```

Fig. 2. Challenge 1: Specification of longest common prefixes.

```
lemma not_common_prefix_if_last_different:
  forall a:array int, x y:int, l:int.
    0 ≤ l ∧ x+l < a.length ∧ y+l < a.length ∧
    a[x+l] ≠ a[y+l] → ¬ is_common_prefix a x y (l+1)

lemma longest_common_prefix_succ:
 forall a:array int, x y l:int.
  is_common_prefix a x y l ∧ ¬ (is_common_prefix a x y (l+1))
  → is_longest_common_prefix a x y l

let lcp (a: array int) (x y: int) : int =
  let n = a.length in
  let l = ref 0 in
  while x + !l < n && y + !l < n && a[x + !l] = a[y + !l] do
    invariant { is_common_prefix a x y !l }
    variant { n - !l }
    incr l
  done;
  !l
```

Fig. 3. Challenge 1: Implementation of the `lcp` function.

Implementation and Proofs. The Why3 code for the function that computes the longest common prefix is given in Fig. 3. It is a direct translation of the Java reference implementation. Since we need a mutable variable `l` for this code, we use a reference (`incr` is a function of the standard library that increments an integer reference, see Sec. 2.2). The loop invariant is natural, since the algorithm amounts to incrementing `l` until the characters at `x+l` and `y+l` differ.

To prove this function, we first state and prove two lemmas (Fig. 3). These lemmas are not strictly necessary for the proof, though they help some automated provers, but we need them later anyway. The first lemma states that if characters at positions `x+l` and `y+l` are different, then there is no common prefix of length `l+1`. The second lemma states that, for the

[3] Calling a method on an object before it is fully constructed is indeed not recommended [18].

```
predicate lt (a: array int) (x y: int) =
  let n = a.length in 0 ≤ x ≤ n ∧ 0 ≤ y ≤ n ∧
  exists l:int. is_common_prefix a x y l ∧
                (y+l < n ∧ (x+l = n  ∨  a[x+l] < a[y+l]))

val compare (a: array int) (x y: int) : int
  requires { 0 ≤ x ≤ a.length }
  requires { 0 ≤ y ≤ a.length }
  ensures { result = 0 → x = y }
  ensures { result < 0 → lt a x y }
  ensures { result > 0 → lt a y x }

predicate le (a: array int) (x y: int) = x = y  ∨  lt a x y

predicate sorted_sub
       (a: array int) (data: Map.map int int) (l u: int) =
  forall i1 i2: int. l ≤ i1 ≤ i2 < u →
    le a (Map.get data i1) (Map.get data i2)

predicate sorted (a: array int) (data: array int) =
  sorted_sub a data.elts 0 data.length

use array.ArrayPermut  (* provides the [permut] predicate *)

use map.MapInjection
predicate range (a: array int) =
  MapInjection.range a.elts a.length

val sort (a: array int) (data: array int) : unit
  requires { data.length = a.length }
  requires { range data }
  ensures { sorted a data }
  ensures { ArrayPermut.permut (old data) data }
```

Fig. 4. Challenge 1: Specification of lexicographic comparison and sorting.

longest common prefix to be of length `l`, it suffices to have a common prefix of length `l` but not `l+1`. The two lemmas and the verification conditions for the `lcp` function are proved automatically, using SMT solvers. (Detailed statistics on proofs are summarized in Section 3.5.)

3.2 Sorting Suffixes

Our next module deals with lexicographic order on suffixes and sorting.

Specification. In the specifications of this module (Fig. 4) we define the lexicographic order as a predicate `lt`: the suffix at position `x` is smaller than the suffix at position `y` if there is some length `l` such that prefixes of length `l` are the same, and the next character in `x`, if any, is smaller than the next character in `y`. The comparison function is then specified with postconditions that tell, depending on the sign of the result, whether suffixes at positions `x` and `y` are equal, smaller or greater respectively.

To specify the sorting algorithm, we need several ingredients (Fig. 4). First we define the "less or equal" predicate `le`, which is the ordering relation used for sorting. As usual with sorting algorithms, the postcondition is two-fold: first, it says that the resulting array is sorted; second, that it contains a permutation of its initial contents. This sorting function is peculiar since it does not take only one array as usual, but takes two arrays, `a` and `data` as arguments, and sorts `data` using an order relation that depends on `a`. Because of this peculiarity, we cannot reuse the `sorted` predicate from the Why3 standard library to specify the `sort` function: the definition of this predicate is parametrized by only one array (see Appendix A.1 for details). We thus had to write our own version (Fig. 4) by making a copy of the relevant part of the Why3 standard library and adding an extra argument `a`.

On the other hand, we can reuse the `permut` predicate from the standard library (see Fig. 24 in the Appendix A.1 for details). Finally, additional requirements are that the input arrays have the same length l, and that the values in array `data` range in $0 \ldots l-1$, for which we can reuse the corresponding `range` predicate from the standard library, where `(range m n)` means that `m` maps the domain $[0, \ldots, n-1]$ into $[0, \ldots, n-1]$ (See Fig. 25 in the Appendix).

Implementation and Proofs. The implementation is given in Fig. 5. The code of the `compare` function is close to the Java reference implementation. Notice the use of the **`absurd`** keyword to say that the last case is unreachable. All VCs for these functions are proved automatically.

To prove the sorting algorithm, we first have to show that `le` is transitive (lemma `le_trans`). To prove this lemma automatically, we pose an auxiliary lemma `lcp_same_index` to handle the equality case. The loop invariants needed are no different from the ones needed for a generic insertion sort algorithm (see for instance `http://toccata.lri.fr/gallery/sorting.en.html`). We just add the fact that values in array `data` remain within $0 \ldots l-1$. The two assertions in the code are intended to help automated provers.

3.3 The `suffixArray` Data Structure

This module corresponds to the public part of the SuffixArray class in the Java reference implementation.

Specification. We first declare a type `suffixArray` (Fig. 6) corresponding to the `SuffixArray` class in the Java code. We equip this type with an invariant that specifies that the two arrays have the same length l, that the suffix array is a permutation of $0, \ldots, l-1$, and that it is sorted in increasing order. The three functions correspond to the methods of the same name in the Java code, and their postconditions are natural.

Implementation and Proofs. The implementation (Fig. 7) of `select` and `lcp` are simple and easy to prove. The `create` function acts as a constructor, and the difficulty is to establish the invariant. For this we pose a lemma `permut_permutation`: when one is permuting elements of an array that represents a permutation, the result is still a permutation. Such a lemma cannot be proved automatically, because it requires induction (the `permut` predicate is defined inductively). We perform this proof in Coq, which requires a few lines of tactics. The notion of permutation of an interval

```
let compare (a: array int) (x y: int) : int =
  if x = y then 0 else
  let n = a.length in
  let l = LCP.lcp a x y in
  if x + l = n then -1 else
  if y + l = n then 1 else
  if a[x + l] < a[y + l] then -1 else
  if a[x + l] > a[y + l] then 1 else
  absurd

lemma lcp_same_index : forall a:array int, x:int.
  0 ≤ x ≤ a.length →
  LCP.is_longest_common_prefix a x x (a.length - x)

lemma le_trans : forall a:array int, x y z:int.
  le a x y ∧ le a y z → le a x z

let sort (a: array int) (data: array int) : unit =
  'Init:
  for i = 0 to data.length - 1 do
    invariant { permut (at data 'Init) data   }
    invariant { sorted_sub a data.elts 0 i  }
    invariant { range data }
    let j = ref i in
    while !j > 0 && compare a data[!j-1] data[!j] > 0 do
      invariant { 0 ≤ !j ≤ i }
      invariant { range data }
      invariant { permut (at data 'Init) data   }
      invariant { sorted_sub a data.elts 0 !j }
      invariant { sorted_sub a data.elts !j (i+1) }
      invariant { forall k1 k2:int. 0 ≤ k1 < !j ∧
                    !j+1 ≤ k2 ≤ i → le a data[k1] data[k2] }
      variant { !j }
      'L:
      let b = !j - 1 in
      let t = data[!j] in
      data[!j] ← data[b];
      data[b] ← t;
      (* to prove invariant [permut ...] *)
      assert { exchange (at data 'L) data (!j-1) !j };
      decr j
    done
    (* to prove invariant [sorted_sub a data.elts 0 i] *)
    assert { !j > 0 → le a data[!j-1] data[!j] }
  done
```

Fig. 5. Challenge 1: Implementation of comparison and sorting.

$0 \ldots l-1$ and this lemma are good candidates for addition in the Why3 standard library.

3.4 Longest Repeated Substring

Specification. This module contains only the main function for computing the longest repeated substring, specified in Fig. 8. As in the Java code, there are two global variables `solStart` and `solLength` that hold the results of the computation. To specify this function, we add an extra global *ghost* variable `solStart2` that holds the index of the second occurrence of the repeated substring. The postconditions thus say that the two indexes `solStart` and `solStart2` are distinct, that `solLength` is the length of their longest common prefix, and that for any other pair of two indexes, the longest common prefix is not longer than `solLength`.

```
predicate permutation (m: Map.map int int) (u: int) =
  MapInjection.range m u ∧ MapInjection.injective m u

type suffixArray = {
    values : array int;
    suffixes : array int;
}
invariant { self.values.length = self.suffixes.length ∧
  permutation self.suffixes.elts self.suffixes.length ∧
  SuffixSort.sorted self.values self.suffixes }

val select (s: suffixArray) (i: int) : int
  requires { 0 ≤ i < s.values.length }
  ensures { result = s.suffixes[i] }

(** constructor of suffixArray structure *)
val create (a: array int) : suffixArray
  ensures { result.values = a }

val lcp (s: suffixArray) (i: int) : int
  requires { 0 < i < s.values.length }
  ensures { LCP.is_longest_common_prefix
              s.values s.suffixes[i-1] s.suffixes[i] result }
```

Fig. 6. Challenge 1: Specification of SuffixArray.

```
let select (s: suffixArray) (i: int) : int = s.suffixes[i]

(** needed to establish type invariant in function [create] *)
lemma permut_permutation : forall a1 a2:array int.
  ArrayPermut.permut a1 a2 ∧ permutation a1.elts a1.length
  → permutation a2.elts a2.length

let create (a: array int) : suffixArray =
  let n = a.length in
  let suf = Array.make n 0 in
  for i = 0 to n-1 do
    invariant { forall j:int. 0 ≤ j < i → suf[j] = j }
    suf[i] ← i
  done;
  SuffixSort.sort a suf;
  { values = a; suffixes = suf }

let lcp (s: suffixArray) (i: int) : int =
  LCP.lcp s.values s.suffixes[i] s.suffixes[i-1]
```

Fig. 7. Challenge 1: Implementation of SuffixArray.

```
val solStart : ref int
val solLength : ref int
val ghost solStart2 : ref int

val lrs (a: array int) : unit
  requires { a.length > 0 }
  ensures { 0 ≤ !solLength ≤ a.length }
  ensures { 0 ≤ !solStart ≤ a.length }
  ensures { 0 ≤ !solStart2 ≤ a.length ∧
            !solStart ≠ !solStart2 ∧
            LCP.is_longest_common_prefix
              a !solStart !solStart2 !solLength }
  ensures { forall x y l:int. 0 ≤ x < y < a.length ∧
    LCP.is_longest_common_prefix a x y l → !solLength ≥ l }
```

Fig. 8. Challenge 1: Specification of longest repeated substring.

```
let lrs (a: array int) : unit =
  let sa = SuffixArray.create a in
  solStart := 0;
  solLength := 0;
  solStart2 := a.length;
  for i=1 to a.length - 1 do
    invariant { 0 ≤ !solLength ≤ a.length }
    invariant { 0 ≤ !solStart ≤ a.length }
    invariant { 0 ≤ !solStart2 ≤ a.length ∧
                !solStart ≠ !solStart2 ∧
                LCP.is_longest_common_prefix
                  a !solStart !solStart2 !solLength }
    invariant { forall j k l:int. 0 ≤ j < k < i ∧
      LCP.is_longest_common_prefix a
        sa.suffixes[j] sa.suffixes[k] l → !solLength ≥ l }
    let l = SuffixArray.lcp sa i in
    if l > !solLength then begin
       solStart := SuffixArray.select sa i;
       solStart2 := SuffixArray.select sa (i-1);
       solLength := l
    end
  done;
  (** the following assert needs [lcp_sym] lemma *)
  assert { forall j k l:int.
    0 ≤ j < a.length ∧ 0 ≤ k < a.length ∧ j ≠ k ∧
    LCP.is_longest_common_prefix a
      sa.suffixes[j] sa.suffixes[k] l → !solLength ≥ l};
  (* we state explicitly that sa.suffixes is surjective *)
  assert { forall x y:int. 0 ≤ x < y < a.length →
    exists j k : int.
      0 ≤ j < a.length ∧ 0 ≤ k < a.length ∧ j ≠ k ∧
      x = sa.suffixes[j] ∧ y = sa.suffixes[k] };
  ()
```

Fig. 9. Challenge 1: Implementation of longest repeated substring.

```
(* needed for [lrs] function, first assert *)
lemma lcp_sym : forall a:array int, x y l:int.
  0 ≤ x ≤ a.length ∧ 0 ≤ y ≤ a.length →
  LCP.is_longest_common_prefix a x y l →
    LCP.is_longest_common_prefix a y x l

(* allows CVC to prove the next lemma *)
lemma le_le_common_prefix:
  forall a:array int, x y z l:int.
    SuffixSort.le a x y ∧ SuffixSort.le a y z →
    LCP.is_common_prefix a x z l →
      LCP.is_common_prefix a y z l

(* proved by Alt-Ergo and CVC. But only by Alt-Ergo
   if previous lemma is removed *)
lemma le_le_longest_common_prefix:
  forall a:array int, x y z l m:int.
    SuffixSort.le a x y ∧ SuffixSort.le a y z →
    LCP.is_longest_common_prefix a x z l ∧
    LCP.is_longest_common_prefix a y z m → l ≤ m
```

Fig. 10. Challenge 1: Lemmas for longest repeated substring.

Implementation and Proofs. The implementation of `lrs` follows the Java code, and is given in Fig. 9. The first three loop invariants are naturally derived from the first three postconditions. The most difficult part is to establish the last postcondition, for which we add a fourth loop invariant and two assertions after the loop.

The fourth loop invariant states that we already have computed the longest repeated substring for all suffixes in `sa.suffixes[0]`,...,`sa.suffixes[i-1]`. To show that it is preserved by a loop iteration, where only suffixes `sa.suffixes[i-1]` and `sa.suffixes[i]` are considered, it is necessary to show that for any j between 0 and $i-1$, the longest common prefix of `sa.suffixes[j]` and `sa.suffixes[i]` is smaller that those of `sa.suffixes[i-1]` and `sa.suffixes[i]`. This property is true only because suffixes are sorted lexicographically. To complete the proof with automated provers, we pose the lemma `le_le_longest_common_prefix` given on Fig. 10. To prove it, we also pose the auxiliary lemma `le_le_common_prefix`. Both lemmas are proved automatically.

The last part of the proof is to show the fourth postcondition, knowing that the fourth loop invariant is true at loop exit. To achieve this, we need to add two assertions. The first assertion is just to make the final loop invariant symmetric in j and k, and needs the lemma `lcp_sym` of Fig. 10. The second assertion states that the array of suffixes is *surjective*, that is we are sure that all indexes have been considered when the loop is finished. That second assertion could not be proved automatically, we needed to perform the proof in Coq. The main reason is that it uses existential quantification, which is typically hard to handle automatically. Another difficulty is that this surjectivity property follows from the classical but non-trivial mathematical result that an injective function that maps a finite set (namely, $0, \ldots, l-1$) to itself is also surjective. Fortunately, such a result is already proved in the Why3 standard library, so that we can reuse it in our Coq proof, which, in the end, is just a dozen lines long.

Finally, proving the postcondition from the second assertion must be done in Coq, it seems that the automated provers are not able to exploit the existential quantifications successfully (3 lines of tactics only).

3.5 Proof Statistics

The detailed proof results are available at URL `http://toccata.lri.fr/gallery/verifythis_fm2012_LRS.en.html`. The table below summarizes these results. The time limit given to automated provers is 10 seconds.

Module	number of VCs	automatically proved
LCP	10	10 (100%)
SuffixSort	62	62 (100%)
SuffixArray	22	21 (95%)
LRS	30	28 (93%)
Total	124	121 (97%)

The table below shows the results per prover, among the 121 VCs proved automatically.

Prover	number of VCs proved
Alt-Ergo 0.95.2	116
CVC3 2.4.1	106
CVC4 1.3	105
Eprover 1.6	77
Vampire 0.6	75
Z3 3.2	85
Z3 4.3.1	82

3 VCs are proved only by Alt-Ergo. The three interactive proofs done using Coq amount to a total of 49 lines of script added manually, most of them being for proving the lemma `permut_permutation`.

3.6 Lessons Learned

The first part of the challenge, namely our `LCP` module, was easy and was indeed solved almost in the same way within the 45 minutes slot. On the other hand, achieving the other parts required significantly more work, that we estimate to a dozen of hours. It should be noticed that although the final specifications are, to our opinion, relatively short and natural, it was not easy to obtain them at first: a significant amount of time is needed to find adequate specifications that make the proofs almost fully automated.

Our proofs were not done fully automatically, as we had to invoke Coq and write manually a few lines of proof script. In the case of the `permut_permutation` lemma, this was necessary because an inductive reasoning was needed, on the definition of the permutation predicate. For the other VCs, this was needed to manually give witnesses of some existential quantifications, which could not be found by automated provers at our disposal. Notice that when we go to Coq to discharge a goal, we typically use a few lines of Coq tactics, e.g. to perform an induction. For the resulting subgoals, we can still use automated provers via the `why3` Coq tactic, which amounts to send back a Coq goal to Why3 and call a given automated prover. It is an interesting question whether an interactive proof assistant is mandatory for such purposes, we will come back to this in the concluding section.

Another point is that we were able to reuse a significant amount of theories from the standard library. This is good news since good library support is certainly an important point for the efficient use of a verification environment. However we also realized that some parts were not sufficiently generic, namely the `sorted` predicate for which the ordering relation cannot depend on another extra argument. A simple solution would be to add an extra parameter in the `le` predicate in the library. Another direction would be to allow some form of partial application in the logic of Why3, that would be a first step towards higher-order logic.

4 Challenge 2: Sums of Prefixes

This challenge proposes to compute the sums of the prefixes of an array. More precisely, given an array `a`, it amounts to computing each partial sum $\sum_{0 \le k < i} \mathtt{a}[k]$ and storing it back into $\mathtt{a}[i]$. The array is thus modified in-place. The proposed algorithm requires the length of `a` to be a power of 2, and does not use any extra storage.

```
use import int.Int
use import int.Power
use import array.Array
use import array.ArraySum

predicate is_power_of_2 (x: int) =
  exists k:int. (k ≥ 0 ∧ x = power 2 k)

val compute_sums (a: array int) : unit
  requires { a.length ≥ 2 && is_power_of_2 a.length }
  writes   { a }
  ensures  { forall i:int. 0 ≤ i < a.length →
               a[i] = sum (old a) 0 i }
```

Fig. 11. Challenge 2: Specification of sums of prefixes.

```
inductive sumtree (left: int) (right: int)
                  (a0: array int) (a: array int) =
| Leaf : forall left right:int, a0 a : array int.
   right = left + 1 → a[left] = a0[left] →
   sumtree left right a0 a
| Node : forall left right:int, a0 a : array int.
   right > left + 1 →
   sumtree (go_left left right) left a0 a →
   sumtree (go_right left right) right a0 a →
   a[left] = sum a0 (left-(right-left)+1) (left+1) →
   sumtree left right a0 a
```

Fig. 12. Challenge 2: Modeling the binary tree.

The Why3 specification of the algorithm is given in Fig. 11. We import the function `sum` from the theory `array.ArraySum` of the Why3 standard library, where `(sum a i j)` denotes the sum $\sum_{\mathtt{i} \le k < \mathtt{j}} \mathtt{a}[k]$.

The challenge is to verify a sequential version of a parallelizable algorithm. The idea is to identify the array with a binary tree. The algorithm traverses this tree by gathering information firstly from the leaves to the root (the *upsweep* phase) and secondly from the root to the leaves (the *downsweep* phase). First, we specify the state of array `a` during the *upsweep* and *downsweep* phases. Then we explain each phase separately.

The full solution is available on the Toccata gallery of verified programs at `http://toccata.lri.fr/gallery/verifythis_PrefixSumRec.en.html`

4.1 Modeling the Binary Tree

The main difficulty in this challenge is to capture the operations performed by the upsweep phase. The array is identified with a complete binary tree and this operation updates nodes with sums of subtrees in an intricate way. The simplest idea[4] is to use an inductive predicate `sumtree` (see Fig. 12) that

[4] that came to our mind during the competition

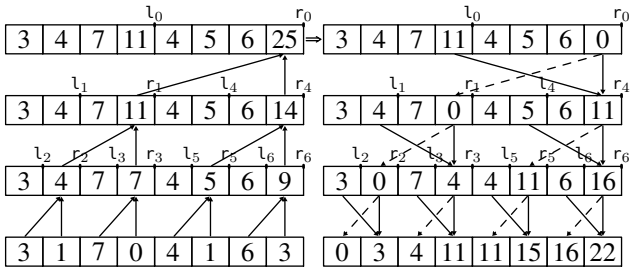


Fig. 13. Challenge 2: Schematic view of the algorithm. The left column shows the *upsweep* phase, the right column the *downsweep* phase. The indices (l_i,r_i) correspond to the subtrees and to the parameters (`left`,`right`) of the recursive calls. The initial array is at the bottom left, the array after the *upsweep* phase is at the top left, the array before the *downsweep* phase is at the top right, the resulting array is at the bottom right. A plain arrow stands for a sum and an assignment performed in the first and second phase. A dashed arrow stands for an assignment `a[left]`←`tmp` performed in the second phase.

mimics the recursive structure of the algorithm. It takes as arguments two indices `left` and `right` to identify a subtree, and two arrays `a0` and `a`. Array `a0` stands for the initial contents of the array, and array `a` for its current state with partial sums.

A subtree is represented by a pair of indices (`left`, `right`), with `space = right - left` being a power of two. The elements of this subtree span from (`left-space+1`) to `right`, included. We introduce two functions `go_left` and `go_right` to descend into the left and right subtrees:

```
function go_left (left right:int) : int =
  let space = right - left in left - div space 2
function go_right (left right:int) : int =
  let space = right - left in right - div space 2
```

The left subtree is (`go_left left right`, `left`) and the right subtree is (`go_right left right`, `right`). A schematic representation of the algorithm and of the tree view of the array is in Fig. 13.

The main idea in predicate `sumtree` is to describe the value in `a[left]` but not the value in `a[right]`. The main reason is that the root of a left subtree is not modified anymore once the left subtree has been processed. Since `sumtree` is inductively defined, it describes the values at indices from `left` to `right-1`, since they are all roots of left subtrees. For instance, (`sumtree 3 7 a0 a`) defines the values stored at indices from 0 to 6. On the contrary, the value `a[right]` is not specified in `sumtree`. Indeed, functions `upsweep` and `downsweep` have different specifications for `a[right]`.

4.2 The "upsweep" Phase

The implementation and the complete specification of `upsweep` is presented in Fig. 14. This function on (`left`, `right`) sets `a[right]` to the sum of the elements of the subtree. So when the left subtree has been processed, `a[left]` contains the sum of the elements of the left subtree. It is indeed the property which is described in `sumtree`. Moreover

```
let rec upsweep (left right: int) (a: array int) : unit
  requires { 0 ≤ left < right < a.length ∧
    -1 ≤ left - (right - left) ∧ is_power_of_2 (right - left)}
  variant { right-left }
  ensures { let space = right - left in
    sumtree left right (old a) a ∧
    a[right] = sum (old a) (left-(right-left)+1) (right+1) ∧
    (forall i: int. i ≤ left-space → a[i] = (old a)[i]) ∧
    (forall i: int. i > right → a[i] = (old a)[i]) }
  = 'Init:
    let space = right - left in
    if right > left+1 then begin
      upsweep (left - div space 2) left a;
      upsweep (right - div space 2) right a;
      assert {
        sumtree (left - div space 2) left (at a 'Init) a };
      assert {
        sumtree (right - div space 2) right (at a 'Init) a };
      assert { a[left] =
        sum (at a 'Init) (left-(right-left)+1) (left+1) };
      assert { a[right] = sum (at a 'Init) (left+1) (right+1) }
    end;
    a[right] ← a[left] + a[right];
    assert { right > left+1 →
       sumtree (left - div space 2) left (at a 'Init) a };
    assert { right > left+1 →
       sumtree (right - div space 2) right (at a 'Init) a }
```

Fig. 14. Challenge 2: Upsweep phase.

```
lemma sumtree_frame:
  forall left right:int, a0 a a' : array int.
  (forall i:int.
    left-(right-left) < i < right → a[i] = a'[i]) →
  sumtree left right a0 a → sumtree left right a0 a'

lemma sumtree_frame2:
  forall left right:int, a0 a0' a : array int.
  (forall i:int.
    left-(right-left) < i < right → a0[i] = a0'[i]) →
  sumtree left right a0 a → sumtree left right a0' a
```

Fig. 15. Challenge 2: Frame properties for `sumtree`.

the array `a` is modified by the recursive calls, so we need to specify the extent of these modifications. This is the purpose of the last two formulas in the postcondition. The conditions of preservation of `sumtree` are described in the two frame lemmas given in Fig. 15. These two lemmas are proved by induction over `sumtree`, using the Coq proof assistant.

4.3 The "downsweep" Phase

The implementation of the `downsweep` function is presented in Fig. 16. We need the ghost argument `a0` that contains the initial array. Indeed (**`old`** `a`) cannot be used because it represents the intermediate state between the *upsweep* and *downsweep* phase. The traversal is the same as in the first phase so the inductive predicate `sumtree` is used directly for the proof. During the second phase the value `a[right]` is used again in a special way. Before the call to (`downsweep left right a0 a`), it contains the prefix sum

```
predicate partial_sum (left:int) (right:int) (a0 a:array int) =
  forall i : int. (left-(right-left)) < i ≤ right →
    a[i] = sum a0 0 i

let rec downsweep
  (left right: int) (ghost a0: array int) (a: array int) : unit
  requires {
    0 ≤ left < right < a.length ∧
    -1 ≤ left - (right - left) ∧
    is_power_of_2 (right - left) ∧
    a[right] = sum a0 0 (left-(right-left) + 1) ∧
    sumtree left right a0 a }
  variant { right-left }
  ensures { partial_sum left right a0 a ∧
    (forall i: int. i ≤ left-(right-left) →
                                a[i] = (old a)[i]) ∧
    (forall i: int. i > right → a[i] = (old a)[i]) }
  = 'Init:
    let tmp = a[right] in
    assert { a[right] = sum a0 0 (left-(right-left) + 1) };
    assert { a[left] = sum a0 (left-(right-left)+1) (left+1) };
    a[right] ← a[right] + a[left];
    a[left] ← tmp;
    assert { a[right] = sum a0 0 (left + 1) };
    if right > left+1 then
    let space = right - left in
    assert { sumtree (go_left left right) left a0 (at a 'Init)};
    assert {
      sumtree (go_right left right) right a0 (at a 'Init) };
    assert { sumtree (go_right left right) right a0 a };
    downsweep (left - div space 2) left a0 a;
    downsweep (right - div space 2) right a0 a;
    assert { partial_sum (left - div space 2) left a0 a };
    assert { partial_sum (right - div space 2) right a0 a }
```

Fig. 16. Challenge 2: Downsweep phase.

of the value of the initial array `a0` before the subtree (`left`, `right`):

```
a[right] = (sum a0 0 (left-(right-left) + 1)).
```

Finally, the function (`downsweep left right a0 a`) ensures that all the values of the subtree (`left`, `right`) are the prefix sums of the array `a0`. As for the function `upsweep`, we need to specify the extent of the modification of `downsweep` but we do not have to write and prove the frame lemmas for `partial_sum` because they can be derived without induction from the frame lemmas of `sum` defined in the Why3 standard library. The frame lemma `sumtree_frame` is also used for the frame of the first recursive call.

4.4 The Main Procedure

The main procedure `compute_sums`, in Fig. 17, calls the two phases sequentially and initializes `a[right]` to the prefix sum of the index `0`, which is `(sum a0 0 0) = 0`. The harness test proposed in the challenge is also proved (we do not give the code in this paper).

```
let compute_sums (a: array int) : unit
  requires { a.length ≥ 2 && is_power_of_2 a.length }
  ensures { forall i : int. 0 ≤ i < a.length →
            a[i] = sum (old a) 0 i }
 = let a0 = ghost (copy a) in
   let l = a.length in
   let left = div l 2 - 1 in
   let right = l - 1 in
   upsweep left right a;
   (* needed for the precondition of downsweep *)
   assert { sumtree left right a0 a };
   a[right] ← 0;
   downsweep left right a0 a;
   (* needed to prove the postcondition *)
   assert { forall i : int.
     left-(right-left) < i ≤ right →
     a[i] = sum a0 0 i }
```

Fig. 17. Challenge 2: Main procedure.

4.5 Proof Statistics

The detailed proof results are available at `http://toccata.lri.fr/gallery/verifythis_PrefixSumRec.en.html`. The table below summarizes these results. The time limit given to automated provers is 10 seconds.

Function	number of VCs	automatically proved
`lemmas`	4	2 (50%)
`upsweep`	24	24 (100%)
`downsweep`	29	29 (100%)
`compute_sums`	12	12 (100%)
`test_harness`	20	20 (100%)
Total	89	87 (98%)

The table below shows the results per prover, among the 87 VCs proved automatically.

Prover	number of VCs proved
Alt-Ergo 0.95.2	100
CVC3 2.4.1	85
CVC4 1.2	96
Z3 3.2	64
Z3 4.3.1	70

The two lemmas `sumtree_frame` and `sumtree_frame2` are proved using Coq by 4 lines of tactics: one induction on the inductive predicate `phase1` and then the subgoals are discharged using the `why3` tactic, as in challenge 1. All remaining proof obligations are discharged by at least two automated provers.

4.6 Lessons Learned

As for challenge 1, we had to use the interactive proof assistant Coq to discharge some VCs. See the general conclusion for a discussion on such a use of Coq.

The hardest part of this challenge is the specification of the state between the two phases. During the competition,

```
type loc
constant null: loc
type node = { left: loc; right: loc; data: int; }
type memory = map loc node
val mem: ref memory
```

Fig. 18. Challenge 3: Preliminaries.

when we came up with the idea of using an inductive predicate, we got the indexes wrong several times. When the automated provers failed to prove a proof obligation, we added assertions that should have helped to find particular facts that could not be proved. We also tried to prove them in Coq in order to find the hole in the proof. After the competition, when the indexes in the specification were corrected, we removed the useless assertions and Coq proofs. This raises the general question on how we can debug the specifications. For such a purpose, we are currently implementing a step-by-step evaluator of WhyML functions. The user will thus be able to easily compare the behavior of the program to its specification.

5 Challenge 3: Deletion in a Binary Search Tree

The third challenge is to verify a procedure that removes the node with the minimal key from a binary search tree. The pseudocode given at the competition descends along the leftmost branch of the tree using a while loop. When it reaches a node with no left child, it makes its right child the new left child of its parent. The tree is mutated in-place. Our solution respects the reference implementation. The full solution is available on the Toccata gallery of verified programs at `http://toccata.lri.fr/gallery/verifythis_fm2012_treedel.en.html`

5.1 Preliminaries

Why3 has no native support for mutable trees. Hence we build a minimal memory model, given in Fig. 18. It introduces some uninterpreted type `loc` to denote memory locations, with a particular value `null`. Then the heap is modeled as a global reference `mem` holding a purely applicative map from memory locations to nodes. A node is a record with two fields `left` and `right` of type `loc` and a third field `data` of type `int`. To account for possible `null`-dereference, we do not access these three fields directly, but we use instead functions `get_left`, `get_right`, and `get_data` with suitable preconditions:

```
val get_left (p: loc) : loc
  requires { p ≠ null }
  ensures  { result = !mem[p].left }
```

(and similarly for `get_right` and `get_data`). In this model, we assume that any pointer that is not `null` can be safely dereferenced, which is the case in languages that do not permit explicit deallocation, *e.g.* Java.

```
inductive istree (m: memory) (p: loc) (t: tree loc) =
  | leaf: forall m: memory.
      istree m null Empty
  | node: forall m: memory,p: loc,l r: tree loc.
      p ≠ null →
      istree m m[p].left l → istree m m[p].right r →
      istree m p (Node l p r)

val tree_delete_min (t: loc)
  (ghost it: tree loc) (ghost ot: ref (tree loc))
  : (loc, int)
  requires { t ≠ null }
  requires { istree !mem t it }
  requires { distinct (inorder it) }
  ensures  { let (t',m) = result in istree !mem t' !ot ∧
    match inorder it with
    | Nil → false
    | Cons p l → m = !mem[p].data ∧ inorder !ot = l
    end }
```

Fig. 19. Challenge 3: Specification.

5.2 Specification

We import polymorphic immutable lists and trees from the Why3 standard library (see Sec. 2.1 for definitions). We also import a function `inorder (tree` α`) : list` α that lists the elements of a tree according to inorder traversal and a predicate `distinct (list` α`)` that expresses that all elements of a given list are distinct. Types `list` and `tree` will only appear in specifications.

Our specification for problem 3 is given in Fig. 19. The central component is the inductively defined predicate `istree`. Such an inductive definition should be read as a set of inference rules:

$$\frac{}{\texttt{istree}\ m\ \texttt{null}\ \texttt{Empty}}$$

$$\frac{p \neq \texttt{null} \quad \texttt{istree}\ m\ m[p].\texttt{left}\ l \quad \texttt{istree}\ m\ m[p].\texttt{right}\ r}{\texttt{istree}\ m\ p\ (\texttt{Node}\ l\ p\ r)}$$

Given a memory state m, a loc p, and a tree of memory locations t (of type `(tree loc)`), the predicate (`istree` $m\ p\ t$) means that memory m contains a well-formed tree rooted at p, whose shape is exactly t. The inductive nature of predicate `istree` ensures that such a tree is both finite and acyclic. But nothing prevents the heap-allocated tree to be a DAG; we will take care of that later.

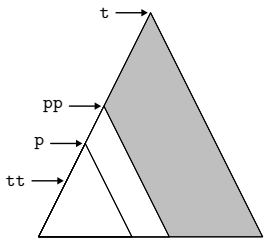
The tree deletion operation `tree_delete_min` takes a non-null loc `t` as argument. It is required to be the root of a tree, that is `(istree !mem t it)`, where `it` is a tree of locations passed as an additional ghost parameter. To account for the absence of sharing in `t`, we also require that all locations in `it` are distinct, which is conveniently written as `distinct (inorder it)`. Let `(t',m)` be the pair returned by the function. The postcondition makes use of a second ghost parameter `ot` to describe the shape of `t'`. It simply says that `(inorder it) = (Cons p (inorder ot))`, for

some location p, with m being the data field at p. We avoid the use of an existential quantifier over p by performing pattern-matching over (inorder it) instead.

It is worth pointing out that our postcondition simply says that we deleted the first element from (inorder it), that is the leftmost innermost element in the tree rooted at t. There is no need for the notion of binary search tree to show up. It would be an orthogonal (and easy) lemma to show that the minimal element in a binary search tree is located at the leftmost innermost node, and that after removal the remaining tree is still a binary search tree.

5.3 Proof

The code itself is straightforward. First, it handles the particular case where there is no left subtree. Otherwise, it descends along the leftmost branch of the tree, using three variables pp, p, and tt to hold three consecutive nodes on this branch. This can be depicted as follows:



We are done with the descent when tt becomes null. Then we simply remove node p by turning the right subtree of p into the left subtree of pp.

Proving the code, however, is not straightforward. The whole code annotated with assertions and equipped with ghost statements is given in Fig. 20. The loop performs some local computation, focusing on the subtree rooted at pp, but the postcondition we wish to establish is related to the whole tree rooted at t. Thus we have to account for the "tree with a hole" depicted in gray in the picture above. Fortunately, there is a convenient way to define such a notion: Huet's zipper [11]. The idea is to define a subtree placeholder as the path from the root of the subtree to the root of the whole tree. In the general case, this path should indicate whether we took the left or right subtree during the descent. In our case, however, we are always moving to the left subtree, so the zipper degenerates into a mere list, that is

```
type zipper α = Top | Left (zipper α) α (tree α)
```

The zipper (Left z x t) denotes a hole in place of the left subtree of a node with value x, with right subtree t, and with some upper context z. The zipper Top denotes a hole at the root of the tree. For instance, the "tree with hole" (Node (Node □ x_2 t_2) x_1 t_1) is denoted by the zipper (Left (Left Top x_1 t_1) x_2 t_2).

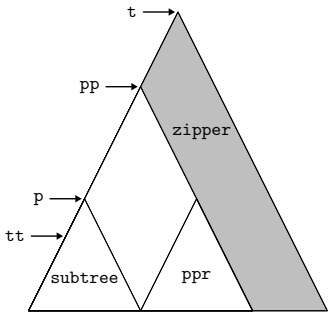
From a zipper z and a subtree t, we can recover the whole tree with the following recursive function that rebuilds the nodes along the path:

```
let tree_delete_min (t: loc)
  (ghost it: tree loc) (ghost ot: ref (tree loc))
  : (loc, int) =
  let p = ref (get_left t) in
  if !p = null then begin
    let m = get_data t in
    let tt = get_right t in
    ghost match it with
    | Empty → absurd
    | Node l _ r → assert { l = Empty }; ot := r
    end;
    (tt, m)
  end else begin
    let pp = ref t in
    let tt = ref (get_left !p) in
    let ghost zipper = ref Top in
    let ghost ppr = ref (right it) in
    let ghost subtree = ref (left it) in
    while !tt ≠ null do
      invariant { !pp ≠ null ∧ !mem[!pp].left = !p }
      invariant { !p ≠ null ∧ !mem[!p].left = !tt }
      invariant { let pt = Node !subtree !pp !ppr in
                  istree !mem !pp pt ∧ zip pt !zipper = it}
      variant { !subtree }
      assert { istree !mem !p !subtree };
      ghost zipper := Left !zipper !pp !ppr;
      ghost ppr := right !subtree;
      ghost subtree := left !subtree;
      pp := !p;
      p := !tt;
      tt := get_left !p
    done;
    assert { istree !mem !p !subtree };
    assert { !pp ≠ !p };
    let m = get_data !p in
    tt := get_right !p;
    mem := set !mem !pp { !mem[!pp] with left = !tt };
    let ghost pl = left !subtree in assert {pl = Empty};
    ghost ot := zip (right !subtree)
                    (Left !zipper !pp !ppr);
    (t, m)
  end
```

Fig. 20. Challenge 3: Implementation.

```
function zip (t: tree α) (z: zipper α): tree α =
  match z with
  | Top → t
  | Left z' x r → zip (Node t x r) z'
  end
```

The idea behind our proof is to maintain the zipper for the subtree rooted at pp in a ghost variable zipper, as well as its left and right subtrees in two additional ghost variables subtree and ppr. This can be depicted as follows:



These three ghost variables are updated along the descent. Thus we have the following loop invariant:

```
(zip (Node !subtree !pp !ppr) !zipper) = it
```

Since the memory is not mutated within the loop, showing the preservation of this loop invariant is straightforward.

The difficult part of the proof lies in the final statement, once we have exited the loop and mutated the memory. The new tree is built from the zipper using

```
zip (right !subtree) (Left !zipper !pp !ppr)
```

We have to show that the inorder traversal of that tree is the tail of the inorder traversal of the initial tree. But this only holds thanks to the lack of sharing in the tree, which is provided by the precondition `distinct (inorder it)`. Otherwise, more than one element could have disappeared from the list. We move that key property into the following lemma:

```
lemma main_lemma:
  forall m: memory, t pp p: loc,
         ppr pr: tree loc, z: zipper loc.
  let it = zip (Node (Node Empty p pr) pp ppr) z in
  istree m t it → distinct (inorder it) →
  let m' = m[pp←{m[pp] with left = m[p].right}] in
  istree m' t (zip (Node pr pp ppr) z)
```

It is proved interactively using the Coq proof assistant. The proof introduces 8 sub-lemmas and requires 114 lines of Coq tactics, including the use of the `why3` tactic to call external SMT solvers, as already mentioned for the two first challenges. The main reason for this proof to be that long is the lack of separation logic in Why3 (no notion of footprints, few lemmas about `distinct`, etc.).

It is worth pointing out that the use of zippers is only an artefact of our proof. Zippers do not appear at all in our specification.

5.4 Proof Statistics

Detailed proof results are given in Fig. 21 on page 16. (This does not include five auxiliary functions, for which proof details are available at URL `http://toccata.lri.fr/gallery/verifythis_fm2012_treedel.en.html`). The table below summarizes these results. The time limit given to automated provers is 5 seconds (apart from one goal, for which it is 60 seconds).

Function	# VCs	automatically proved
auxiliary functions	5	5 (100%)
lemmas	3	2 (67%)
`search_tree_delete_min`	30	29 (97%)
Total	38	36 (95%)

Two VCs are discharged using Coq (128 lines of tactics).

The table below shows the results per prover, among the 36 VCs that are proved automatically.

Prover	number of VCs proved
Alt-Ergo 0.95.2	35
CVC3 2.4.1	34
CVC4 1.3	38
Z3 3.2	34
Z3 4.2	34

Since 36 VCs are discharged automatically, it is clear from this table that the cooperative use of several ATPs is a true asset.

5.5 Lessons Learned

It is slightly unsatisfactory that we cannot handle this challenge in a direct way: Since WhyML does not allow arbitrary mutable data structures, we have to use an explicit encoding of a memory heap. Although we are able to handle this challenge successfully, it is clear that Why3 is not the language of choice to specify and prove pointer-heavy programs. It would be more natural to use verification tools dedicated to Java or C code, in particular since there exist such tools that are built upon Why3, using WhyML as an intermediate language: Krakatoa [13] for Java and Frama-C [10] and its Jessie plug-in [14] for C. Yet we think that doing the verification with Why3 has real benefits: it is easier to write and debug the specification and proof when VCs are not clobbered with a complex encoding of the memory heap. Once the specification with Why3 is completed and verified, one can adapt it to a Java or C implementation. In particular, we think that our idea of using a zipper in the proof can be readily reused.

6 Conclusions

We found the Why3 environment adequate for the given challenges. The specification language provides advanced tools that proved to be useful: algebraic data types, inductive predicates, and a rich standard library.

To perform the proofs, we needed several back-end provers; in particular, the more complex lemmas had to be proved interactively, using Coq. Even when considering only the VCs that were proved automatically, there is no prover among Alt-Ergo, CVC3, CVC4, and Z3 that was able to discharge all of them. Though the ability to use several provers is a clear advantage, it also makes the maintenance of proof sessions a difficult task. Why3 provides a mechanism for storing proof sessions that records which transformations and provers were used to prove each VC, so that a complete verification project can be replayed if needed [5]. Moreover, it

is possible to dump such a session, *e.g.*, Fig. 21 is automatically generated from the proof session of challenge 3. Similar tables for problems 1 and 2 are available online, in Why3's gallery.

On the use of Coq. For each of the three challenges, we had to use Coq to discharge a few VCs that were not proved by automated provers. Writing a proof script for such VCs may seem to be a complex task that requires a fair knowledge of Coq. However, the `why3` tactic helped us to keep such tasks reasonable: a proof starts with some powerful tactic that generates a few subgoals, and after a very little number of tactics, the `why3` tactic is able to complete the proof. A typical example is a proof that requires induction. Another typical case is when explicit witnesses have to be provided for existential quantifiers, as in challenge 1. Although this process is reasonably quick and painless, it is necessary to know a little bit of Coq to use it. It is thus desirable to propose alternatives. Two recent features added to Why3 may help: first, a transformation that is able to perform induction on algebraic data types; second, a mechanism of "lemma functions", similar to that of VeriFast and Dafny, that allows the user to write a recursive program to simulate an induction scheme of his choice. Lemma functions can also be used to generate witnesses for existentially quantified assertions. Still, the ability to perform an induction over an inductive predicate, as we did in challenges 1 and 2, remains to be studied. Last but not least, we are now planning to extend Why3 with a dedicated lightweight interactive prover that would simplify proofs even further.

On the possible use of Frama-C/Jessie or Krakatoa. Why3 is indeed an intermediate verification language which is used by the front-ends Frama-C/Jessie (for C) and Krakatoa (for Java). Thus one might ask why we chose Why3 for the competition: after all, Java implementations were given for the first two challenges, and a language with pointers was mandatory for the third challenge. The first reason is that we decided, prior to the competition, that we were going to use Why3, because it is the tool we are developing. An important improvement that we developed in Why3 relies in the expressiveness of the specification language, that allows us to structure logical models into theories [7], and thus to design a well-structured, reusable standard library. This feature showed itself important during the competition. In the Jessie and Krakatoa front-ends, there is no such a large standard library of specifications yet. Thus, another lesson we learned is that we should now improve the specification languages of the front-ends. A non-trivial issue is how to reuse the same generic standard library for both Why3 and the front-ends. Another point is that the modeling of the heap memory used by the front-ends results in VCs that are sometimes obscure. Finally, we believe that, when one wants to verify a given program, the first step is the design of adequate specifications and adequate proof elements such as loop invariants, auxiliary lemmas, assertions in the code, etc. This work is easier to carry out on a simple language such as WhyML. On a language such as C or Java, one immediately faces the extra burden of showing the absence of runtime errors such as integer overflow and invalid pointer dereferencing.

Acknowledgments We gratefully thank the editors and the anonymous referees for providing us very valuable comments and suggestions to improve the quality of this paper.

References

1. 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, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.
2. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, Berlin, Germany, July 2007. Springer.
3. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.
4. F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.
5. F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. Preserving user proofs across specification changes. In E. Cohen and A. Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments (5th International Conference VSTTE)*, volume 8164 of *Lecture Notes in Computer Science*, pages 191–201, Atherton, USA, May 2013. Springer.
6. F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *The Why3 platform, version 0.82*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.82 edition, Dec. 2013. `http://why3.lri.fr/download/manual-0.82.pdf`.
7. 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*, pages 53–64, Wrocław, Poland, August 2011.
8. L. de Moura and N. Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
9. J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.
10. The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`.
11. G. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, Sept. 1997.
12. K. R. M. Leino and M. Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.
13. C. Marché. The Krakatoa tool for deductive verification of Java programs. Winter School on Object-Oriented Verification, Vinistu, Estonia, Jan. 2009. `http://krakatoa.lri.fr/ws/`.
14. Y. Moy and C. Marché. *The Jessie plugin for Deduction Verification in Frama-C — Tutorial and Reference Manual*. INRIA & LRI, 2011. `http://krakatoa.lri.fr/`.

Proof obligations	Alt-Ergo (0.95.2)	CVC3 (2.4.1)	CVC4 (1.3)	Coq (8.4pl2)	Eprover (1.6)	Vampire (0.6)	Z3 (3.2)	Z3 (4.3.1)
`inorder_zip`								
transformation `induction_ty_lex`								
1.								
transformation `split_goal_wp`								
1.	0.01	0.02	0.03		0.00	0.01	0.02	0.04
2.	0.06	2.93	0.13		0.42	0.77	(5s)	(5s)
`VC for tree_left`	0.02	0.02	0.03		0.02	0.03	0.02	0.01
`VC for tree_right`	0.01	0.02	0.02		0.01	0.03	0.02	0.01
`main_lemma`	(5s)	(5s)	(5s)	15.94	(5s)	5.10	(5s)	(5s)
`VC for search_tree_delete_min`								
transformation `split_goal_wp`								
1. precondition	0.01	0.02	0.02		0.01	0.00	0.02	0.04
2. precondition	0.02	0.02	0.02		0.01	0.00	0.00	0.00
3. precondition	0.01	0.01	0.02		0.01	0.00	0.00	0.00
4. unreachable point	0.02	0.04	0.03		0.05	0.06	0.02	0.06
5. assertion	0.02	0.06	0.03		0.79	2.61	0.04	0.06
6. postcondition	0.25	0.04	0.04		(5s)	3.42	0.02	4.98
7. precondition	0.01	0.03	0.03		0.01	0.01	0.00	0.00
8. precondition	0.01	(5s)	0.03		0.04	0.05	0.02	0.06
9. precondition	0.02	0.03	0.02		0.01	0.00	0.01	0.02
10. loop invariant init	0.02	0.03	0.03		0.01	0.00	0.01	0.02
11. loop invariant init	0.02	0.01	0.02		0.01	0.01	0.01	0.02
12. loop invariant init	0.08	0.14	0.04		1.09	2.81	0.10	0.12
13. assertion	0.03	0.06	0.04		(5s)	5.10	0.02	0.05
14. precondition	0.02	0.09	0.04		0.08	0.06	0.05	0.04
15. precondition	0.02	0.02	0.03		0.01	0.00	0.07	0.03
16. precondition	0.01	0.02	0.02		0.01	0.00	0.02	0.03
17. loop invariant preservation	0.02	0.02	0.03		0.01	0.01	0.02	0.03
18. loop invariant preservation	0.02	0.02	0.03		0.01	0.01	0.02	0.03
19. loop invariant preservation	0.10	0.05	0.06		2.16	2.87	(5s)	(5s)
20. loop variant decrease	0.02	0.03	0.04		0.02	0.02	0.02	0.01
21. assertion	0.03	0.06	0.04		(5s)	5.10	0.02	0.04
22. assertion	0.02	0.02	0.02		0.01	0.00	0.02	0.05
23. precondition	0.01	0.01	0.03		0.01	0.00	0.01	0.04
24. precondition	0.02	0.02	0.02		0.01	0.00	0.00	0.00
25. precondition	0.05	0.05	0.04		0.06	0.05	0.04	0.05
26. assertion	0.22	0.16	0.05		1.17	2.63	0.09	0.11
27. precondition	0.02	0.04	0.02		0.01	0.00	0.00	0.02
28. postcondition								
transformation `split_goal_wp`								
1.	0.10	(5s)	0.05		(5s)	2.82	(5s)	(5s)
2.	(5s)	(5s)	0.09		0.03	2.58	0.11	0.02
3.	(5s)	7.04	0.43		(5s)	3.44	0.08	0.08
4.	(5s)	(5s)	0.10	1.38	0.11	0.64	(5s)	(5s)

Fig. 21. Proof results on challenge 3. An answer on green (light) background indicates that the prover succeeded to discharge the VC, in the given number of seconds. An answer on red (dark) background indicates that the prover reached the time limit given between parentheses.

15. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga Springs, NY, June 1992. Springer.
16. A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 292–296, Trento, Italy, July 1999. Springer.
17. S. Schulz. System description: E 0.81. In D. A. Basin and M. Rusinowitch, editors, *Second International Joint Conference on Automated Reasoning*, volume 3097 of *Lecture Notes in Computer Science*, pages 223–228. Springer, 2004.
18. A. J. Summers and P. Mueller. Freedom before commitment: a lightweight type system for object initialisation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 1013–1032, New York, NY, USA, 2011. ACM.
19. C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. A. Schmidt, editor, *22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.

```
theory Map

  type map α β

  function get (map α β) α : β
  function set (map α β) α β : map α β

  (** syntactic sugar *)
  function ([])   (a : map α β) (i : α) : β = get a i
  function ([←]) (a : map α β) (i : α) (v : β) :
    map α β = set a i v

  axiom Select_eq :
    forall m : map α β. forall a1 a2 : α.
    forall b : β [m[a1 ← b][a2]].
    a1 = a2 → m[a1 ← b][a2]  = b

  axiom Select_neq :
    forall m : map α β. forall a1 a2 : α.
    forall b : β [m[a1 ← b][a2]].
    a1 ≠ a2 → m[a1 ← b][a2] = m[a2]

  function const β : map α β

  axiom Const : forall b:β, a:α. (const b)[a] = b

end
```

Fig. 22. Theory of polymorphic, purely applicative maps.

A Relevant Theories from the Why3 Standard Library

In this Appendix, we provide details on relevant parts of the Why3 standard library, notably those used in the first challenge. The full contents of that library is available online at `http://why3.lri.fr/stdlib-0.81/`.

A.1 Library `map`

In Why3, arrays are modelled as references to purely applicative maps. Such maps are specified in the theory `Map` presented in Fig. 22. These maps are parametric in both the type of their indexes and of their values, so the Why3 type for such maps is a polymorphic type `(map α β)`. The main functions operating on maps are `set` and `get`, that respectively modify a map at a given index (purely applicatively, that is by returning a new map) and read a map at a given index. Both functions are axiomatized by `Select_eq` and `Select_neq`. This is the classical *theory of arrays* in the context of satisfiability modulo theories. The additional operation `const` returns a constant map.

Theory `MapSorted` (Fig. 23) specifies what it means for a map to be sorted in increasing order. This notion is restricted to maps indexed by integers. The theory is written with a type parameter `elt` and a predicate parameter `le`, so that it can be reused, by cloning it, for an arbitrary order relation.

```
(** {2 Sorted Maps (indexed by integers)} *)
theory MapSorted

  use import int.Int
  use import Map

  type elt

  predicate le elt elt

  (** [sorted_sub a l u] is true whenever the array
      segment [a(l..u-1)] is sorted w.r.t order
      relation [le] *)
  predicate sorted_sub (a : map int elt) (l u : int) =
    forall i1 i2 : int. l ≤ i1 ≤ i2 < u →
      le a[i1] a[i2]

end
```

Fig. 23. Theory of sorted maps.

```
theory MapPermut

  use import int.Int
  use export Map

  (** [exchange m1 m2 i j] is true when the maps [m1]
      and [m2] are identical except at indexes [i]
      and [j], where the values are exchanged *)
  predicate exchange (a1 a2 : map int α) (i j : int) =
    a1[i] = a2[j] ∧ a2[i] = a1[j] ∧
    forall k:int. (k ≠ i ∧ k ≠ j) → a1[k] = a2[k]

  (** [permut_sub m1 m2 l u] is true when the segment
      [m1(l..u-1)] is a permutation of the segment
      [m2(l..u-1)]

      It is defined inductively as the smallest
      equivalence relation that contains the
      exchanges *)
  inductive permut_sub (map int α) (map int α) int int =
  | permut_refl :
      forall a : map int α. forall l u : int.
        permut_sub a a l u
  | permut_sym :
      forall a1 a2 : map int α. forall l u : int.
      permut_sub a1 a2 l u → permut_sub a2 a1 l u
  | permut_trans :
      forall a1 a2 a3 : map int α. forall l u : int.
      permut_sub a1 a2 l u → permut_sub a2 a3 l u →
        permut_sub a1 a3 l u
  | permut_exchange :
      forall a1 a2 : map int α. forall l u i j : int.
      l ≤ i < u → l ≤ j < u → exchange a1 a2 i j →
        permut_sub a1 a2 l u

end
```

Fig. 24. Theory of map permutations (excerpt).

```
theory MapInjection

  use import int.Int
  use export Map

  (** [injective a n] is true when [a] is an injection
      on the domain [(0..n-1)] *)
  predicate injective (a: map int int) (n: int) =
    forall i j: int. 0 ≤ i < n → 0 ≤ j < n →
      i ≠ j → a[i] ≠ a[j]

  (** [surjective a n] is true when [a] is a surjection
      from [(0..n-1)] to [(0..n-1)] *)
  predicate surjective (a: map int int) (n: int) =
    forall i: int. 0 ≤ i < n →
      exists j: int. (0 ≤ j < n ∧ a[j] = i)

  (** [range a n] is true when [a] maps the domain
      [(0..n-1)] into [(0..n-1)] *)
  predicate range (a: map int int) (n: int) =
    forall i: int. 0 ≤ i < n → 0 ≤ a[i] < n

  (** main lemma: an injection on [(0..n-1)] that
      ranges into [(0..n-1)] is also a surjection *)
  lemma injective_surjective:
    forall a: map int int, n: int.
    injective a n → range a n → surjective a n

end
```

Fig. 25. Theory of injective and surjective maps.

Theory `MapPermut` (Fig. 24) specifies what it means for two maps to be permutations of each other, that is, to contain the same elements with the same number of occurrences. This is restricted to maps indexed by integers. Predicate (`permut_sub` m_1 m_2 l u) holds when map segments $m_1[l \dots u-1]$ and $m_2[l \dots u-1]$ are permutations of each other. This is defined as the smallest equivalence relation containing transpositions of elements.

Theory `MapInjection` (Fig. 25) defines several notions for maps on the domain $0 \dots n-1$ for some integer n: being injective, being a surjection into $0 \dots n-1$, and ranging into $0 \dots n-1$. Lemma `injective_surjective` is the classical mathematical result, more or less equivalent to the pigeon-hole principle, saying that any injection that ranges into $0 \dots n-1$ is also surjective. This result is used to prove the first challenge. This is not the first time we had to use such a result to prove a program: we already used it to prove the challenge *Sparse Arrays* from the VACID-0 benchmarks [12].

A.2 Module `Array`

Fig. 26 contains an excerpt of the Why3 module that defines arrays. This module is discussed in Section 2.2. The full contents of that library is available online at `http://why3.lri.fr/stdlib-0.81/array.mlw.html`.

```
module Array

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

  type array α
    model { length : int; mutable elts : map int α }
    invariant { 0 ≤ self.length }

  function get (a: array α) (i: int) : α =
    M.get a.elts i

  function set (a: array α) (i: int) (v: α) : array α =
    { a with elts = M.set a.elts i v }

  (** syntactic sugar *)
  function ([]) (a: array α) (i: int) : α = get a i
  function ([←]) (a: array α) (i: int) (v: α)
    : array α = set a i v

  val ([]) (a: array α) (i: int) : α
    requires { 0 ≤ i < a.length }
    reads    { a }
    ensures  { result = M.get a.elts i }

  val ([]←) (a: array α) (i: int) (v: α) : unit
    requires { 0 ≤ i < a.length }
    writes   { a }
    ensures  { a.elts = M.set (old a.elts) i v }

  val length (a: array α) : int
    ensures { result = a.length }

end
```

Fig. 26. Module `Array` (excerpt).