INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Binary Heaps Formally Verified in Why3

Asma Tafat — Claude Marché

N° 7780

October 2011

ISSN 0249-6399 ISRN INRIA/RR--7780--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

Binary Heaps Formally Verified in Why3

Asma Tafat* †, Claude Marché†*

Thème : Programmation, vérification et preuves
Équipe-Projet ProVal

Rapport de recherche n° 7780 — October 2011 — 30 pages

Abstract: The VACID-0 benchmarks is a set of small programs which pose challenges for formal verification of their functional behavior. This paper reports on the formal verification of one of these challenges: binary heaps. The solution given here is performed using the Why3 environment for program verification. The expected behavior of the program is specified in Why3 logic, structured using the constructs for building hierarchies of theories provided by Why3. The proofs are achieved by a significant amount of automation, using SMT solvers for a large majority of the verification conditions generated, whereas the remaining verification conditions are discharged by interactive constructions of proof scripts using the Coq proof assistant.

The general aim of this case study is to demonstrate the usability and efficiency of both the Why3 specification language and the accompanying tools, which offer a fairly advanced environment for specification while keeping a significant amount of automation of proofs.

Key-words: Formal Specification, Modularity, Abstraction, Theories, Binary Heap, Heapsort, Why3

* Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405
† INRIA Saclay - Île-de-France, F-91893

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Vérification formelle des tas binaires en Why3

Résumé : Les benchmarks VACID-0 forment une collection de petits programmes qui posent des défis pour la vérification formelle de leur comportement fonctionnel. Ce rapport présente la vérification formelle de l'un de ces exemples: les tas binaires. La solution présentée utilise l'environnement pour la vérification Why3. Le comportement attendu est spécifié dans la logique de Why3, de façon structurée grâce aux constructions hiérarchiques de théories proposées par Why3. Les preuves sont effectuées de façon largement automatiques, car les prouveurs SMT disponibles en sortie de Why3 résolvent un pourcentage significatif des obligations de preuves engendrées, le reste étant prouvé interactivement avec l'assistant de preuve Coq.

La motivation de cette étude de cas est de démontrer l'utilisabilité et l'efficacité à la fois du langage de spécification de Why3 et des outils associés, qui fournissent un langage puissant de spécification tout en permettant une automatisation importante des preuves.

Mots-clés : Spécification formelle, Modularité, Abstraction, Théories, Tas binaires, Tri par tas, Why3

Contents

List of Figures

1 Introduction

Formal specification can express complex properties on the expected behavior of programs. When specifications are expressed in an expressive logical language, say at least first order logic, formal proof techniques must be used to prove that a program meets its specifications.

In the case of purely applicative programs, proof techniques are relatively well-known, because the programming language is close to the logic. Very expressive logics such as the calculus of inductive constructions, implemented in Coq [4], allow to write programs and specifications in the same language, but these programs should be purely applicative.

The Floyd-Hoare logic [14, 16] and Dijkstra's weakest precondition calculus [10] are well-known approaches to prove properties of programs containing side effects. However, these approaches make a implicit assumption, often misidentified, that is the references (or mutable variables, i.e modifiable in place) can not be *aliased*, i.e. the sharing of references is forbidden. This hypothesis must be well understood especially in the case of programs consisting of several subroutines. This can be illustrated by the following simple example (in OCaml syntax) with a post-condition.

```
let f (x:int ref) (y:int ref) =
  x := !x + 1;
  y := !y + 1
{ !x = old(!x) + 1 and !y = old(!y) + 1 }
```

The validity of this post-condition is established under the implicit assumption that x and y are distinct. Thus a call to f of the form

```
let z = ref 0 in f z z
```

should be banned (otherwise the post-condition of f say that z is incremented by 1 instead of by 2). The Why tool [13] treats such cases with a type system that rejects the call to f above. Under this explicit assumption, Filliâtre [11] showed the soundness of Hoare logic by reducing it to pure programs.

Proof of programs with aliases is long remained less well studied and less well understood than for programs without alias. But the needs in this area have come forward with the development of tools to handle programs like Java or C: ESC/Java [8], Spec# [1], Key [3], VCC [9], Why [13], Frama-C [15], etc. In this context, the generation of proof obligations is carried out by reducing to a case without alias, via memory models: for example, the heap memory can be seen as a large array indexed by the pointers, or more subtly, the model *component-as-array* [6] interprets every field structure as a large table. The main difficulty appearing then is the need to specify the potential pointer aliasing. The challenges to solve have been well defined by Leavens, Leino and Muller in 2007 [17]. One challenge is able to reason about complex data structures, modifiable in-place with data invariants to preserve.

In 2010, a collection of programs was proposed by Leino and Moskal [19]: *VACID benchmarks (Verification Ample of Correctness of Invariants of Data-structures)* available on the web page `http://vacid.codeplex.com/`. These examples are short programs that illustrate the challenges for modular proof approaches claiming to support sharing of data.

The goal of this paper is to expose a solution to one of these challenges: Binary Heaps, using the Why3 verification platform [5]. We aim at showing the usability and efficiency of both the Why3 specification language and the accompanying tools, which offer a fairly advanced environment for specification while keeping a significant amount of automation of proofs.

In section 2, we present this case study in details, and the challenges it exposes. Section 3 presents our modelling of the expected properties together with how they are proved. Section 5 concludes by comparing with related work and presents a few perspectives.

2 Presentation of the Challenge

The Binary Heap challenge is a pseudo-code in Java-like syntax which consists in (1) an abstract declaration of a class implementing the so-called binary heap structure ; (2) a static method which sorts an array of integers by first inserting each of its elements in a heap and second extracting them in increasing order ; and (3) a test method which sort a particular array of integers.

The following details the challenge in almost the same wording as in the original VACID paper [19].

2.1 Binary Heaps

A binary min-heap is a nearly full binary tree, where the nodes maintain the `heap property`, that is, each node is smaller than each of its children. The heap should be stored in an integer-indexed collection (e.g, an array). The three following operations should be provided:

```
class Heap {
    static Heap create (uint sz);
    void insert (int e);
    int extractMin ();
}
```

Notice that the type **`uint`** is supposed to denote C-style unsigned integers. The `create(sz)` method creates a new heap of maximum capacity `sz`. The `insert(e)` method should allow inserting element `e` multiple times so that `extractMin()` will return it multiple times.

2.2 Heap Sort and Test Harness

A simple implementation of heap sort is as follows

```
void heapSort(int[] arr, uint len) {
    uint i;
    Heap h = create(len);
    for (i = 0; i < len; ++i) h.insert(arr[i]);
    for (i = 0; i < len; ++i) arr[i] = h.extractMin();
}
```

and the test harness is as follows

```
void heapSortTestHarness() {
    int[] arr = { 42, 13, 42 };
    heapSort(arr, 3);
    assert(arr[0] <= arr[1] && arr[1] <= arr[2]);
    assert(arr[0] == 13 && arr[1] == 42 && arr[2] == 42);
}
```

where two assertions are added after calling heapSort, to explicitate what is supposed to hold after sorting.

2.3 Verification Tasks

The verification task that are requested in the original paper are:

1. To verify that the heap sort returns an array that is sorted (in particular, verify the first assertion in the harness)
2. To verify that the heap represents a multiset, and thus the heap sort produces a permutation of the input (in particular, verify the second assertion).

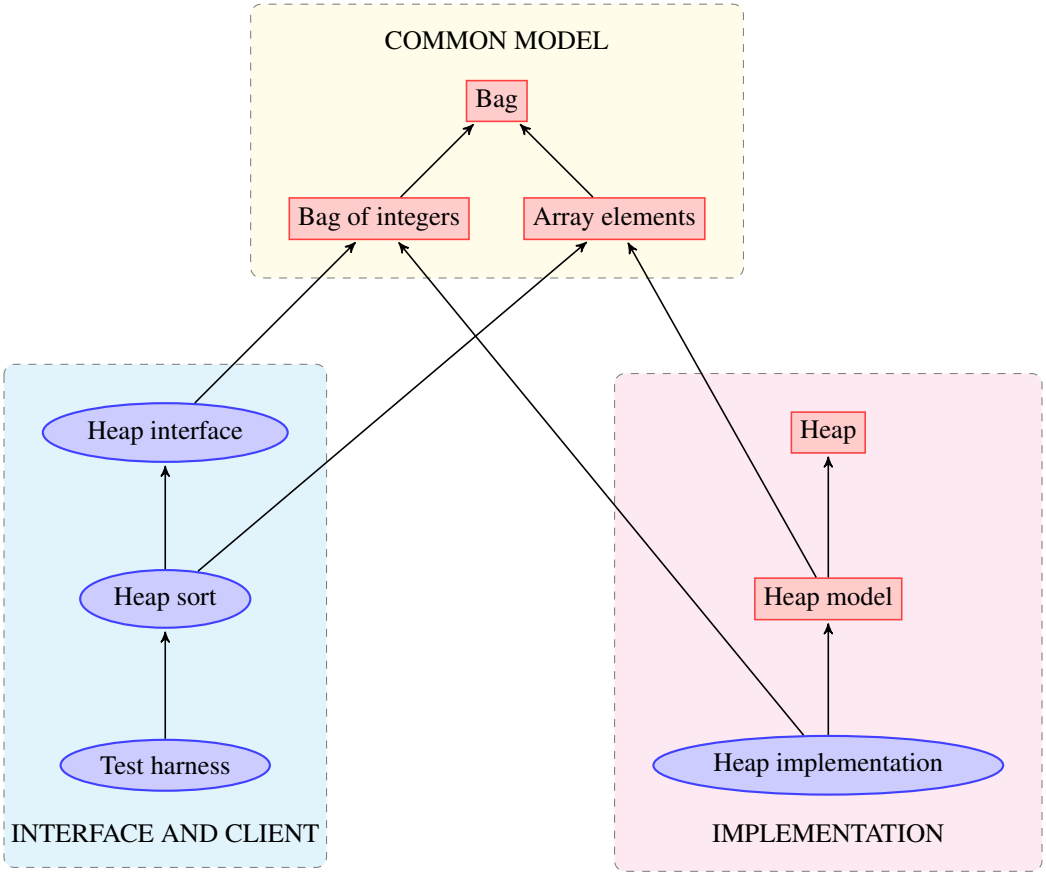

Figure 1: Architecture of the solution.

As we will see, these are not the most difficult verification to perform. We are adding to these tasks the verification of a particular, although classical, implementation of binary heaps: a heap is a nearly full binary tree stored in an array, where the tree root is stored at index 0 and for any index i, the node at index i has for left child the node at index $2i + 1$ and for right child the node at index $2i + 2$.

3 Solution

We present here our solution step by step, following what should be a standard software engineering process, starting from abstraction and obtaining an implementation. This solution is written using the Why3 verification platform [5]. A few logic theories are defined in order to specify heaps and bags structures involved in the expected behavior. Then the code presented in Subsection 2.2 is written under an equivalent form in the Why3 ML programming language on which the Why3 VC generator is run to produce the proof obligations.

The development is split into three main parts which are schematized on Figure 1. The first part is made of Why3 theories formalizing the notion of generic bags, bag of integers and finally defining the set of elements of an array. The second one, procures an *interface* for a binary heap module in accordance to the challenge as described in Section 2.1, and Why3 programs for heap sort and the test harness corresponding to Section 2.2. And finally the third part provides an implementation of that module which includes first a formalization of heaps.

```
type bag 'a

(* the most basic operation is the number of occurrences *)
function nb_occ (x: 'a) (b: bag 'a): int
axiom occ_non_negative : forall b: bag 'a, x: 'a. nb_occ x b >= 0

(* equality of bags *)
predicate eq_bag (a b : bag 'a) =
  forall x:'a. nb_occ x a = nb_occ x b
axiom bag_extensionality: forall a b : bag 'a. eq_bag a b -> a = b
```

Figure 2: Signature for `bag` logical data type

Each of these parts of the development is detailed in a separate subsection below.

3.1 Common model part: bags and such

3.1.1 Bags theory

Since we can have several occurrences of the same integer in a heap, then the natural data type to consider is `multisets` or `bags`, which is introduced in the logic side.

For modularity, we are, first, interested in polymorphic bags, whose signature is given in Figure 2. In practice, this formalization is declared as `theory` of bags. So, the type `bag 'a` representing multisets of elements of type `'a`, is abstract and characterized by the function `nb_occ` that associates to each element `x` in the bag, the number of its occurrences in it. We say that the number of occurrences characterize bags in the sense that two bags having the same number of occurrences of every elements are equal. This is formalized by the predicate `eq_bag` and the axiom `bag_extensionality`. We also need to ensure that the number of occurrences of an element is always non negative. This is formalized by the axiom `occ_non_negative`.

Then, we continue our formalization in Figure 3 by giving basic constructors for bags which are more standard in the algebraic settings: `empty bag`, `singleton` and `union` of bags. These constructors are declared and axiomatized in terms of number of occurrences. Additional classical lemmas are given too, such as associativity and commutativity of bags. We also provide a function `add` as a shortcut for `union` of a `singleton` and a `bag`.

Next, we introduce a function to denote the cardinal of bag. This function is incompletely specified by axioms giving its values on `empty bag`,`singleton` an `union` of bags, as you can remark it in Figure 4. This implicitly means that we consider only finite bags.

Finally, we declare a function `diff` denoting the difference of two bags. It is axiomatized in terms of `nb_occ` as shown on Figure 5. Few lemmas that we need in the following proofs are also defined.

The results of applying theorem provers, Alt-ergo, CVC3, Vampire and Z3, on the specified lemmas in this theory are summarized in Figure 6. Each column, in the table, corresponds to a prover and each row corresponds to one proof obligation. Hence, each cell contains the execution time the prover took to discharge the proof obligation. Empty cells correspond to the case where the corresponding prover was not tested on the associated verification condition.

As you can constate, none of these automatic theorem provers are able to prove the lemmas `Union_comm`, `Union_assoc`, `bag_simpl`, `Diff_add`, `Diff_comm` and `Add_diff`. So, we made an interactive proof of each of them using the interactive proof assistant `Coq`. These proofs are simple, not more than 10 lines of Coq tactics and can be replayed quickly as shown in Figure 6. They need to apply the axiom `bag_extensionality` which is probably the reason why automatic provers can't solve them.

```
(* basic constructors of bags *)
function empty_bag : bag 'a
axiom occ_empty : forall x: 'a. nb_occ x empty_bag = 0
lemma is_empty : forall b: bag 'a.
      (forall x: 'a. nb_occ x b = 0) -> b = empty_bag

function singleton (x: 'a) : bag 'a
axiom occ_singleton: forall x y: 'a.
  (x = y ∧ (nb_occ y (singleton x)) = 1) ∨
  (x <> y ∧ (nb_occ y (singleton x)) = 0)
lemma occ_singleton_eq :
  forall x y: 'a. x = y -> (nb_occ y (singleton x)) = 1
lemma occ_singleton_neq :
  forall x y: 'a. x <> y -> (nb_occ y (singleton x)) = 0

function union (bag 'a) (bag 'a) : bag 'a
axiom occ_union :
  forall x: 'a, a b : bag 'a.
     nb_occ x (union a b) = (nb_occ x a) + (nb_occ x b)
(* union commutative, associative with identity empty_bag *)
lemma Union_comm : forall a b:bag 'a. (union a b) = (union b a)
lemma Union_identity : forall a:bag 'a. (union a empty_bag) = a
lemma Union_assoc : forall a b c:bag 'a.
   (union a (union b c)) = (union (union a b) c)
lemma bag_simpl : forall a b c: bag 'a.
      (union a b) = (union c b) -> a = c
lemma bag_simpl_left : forall a b c: bag 'a.
      (union a b) = (union a c) -> b = c

(* add operation *)
function add (x : 'a) (b: bag 'a) : bag 'a =  union (singleton x) b
lemma occ_add_eq : forall b: bag 'a, x y: 'a.
  x = y -> nb_occ x (add x b) = (nb_occ x b) + 1
lemma occ_add_neq : forall b: bag 'a, x y: 'a.
  x <> y -> nb_occ y (add x b) = (nb_occ y b)
```

Figure 3: Signature for basic constructors of bags

```
(* cardinality of bags *)

function card (bag 'a) : int

axiom Card_empty : card (empty_bag: bag 'a) = 0
axiom Card_singleton : forall x:'a. card (singleton x) = 1
axiom Card_union: forall x y: bag 'a. card (union x y) = (card x) + (card y)
axiom Card_zero_empty : forall x: bag 'a. card (x) = 0 -> x = empty_bag
```

Figure 4: Logic function for cardinal of bags

```
function diff (bag 'a) (bag 'a) : bag 'a

axiom Diff_occ: forall b1 b2:bag 'a, x:'a.
  nb_occ x (diff b1 b2) = max 0 (nb_occ x b1 - nb_occ x b2)
lemma Diff_empty_right: forall b:bag 'a. diff b empty_bag = b
lemma Diff_empty_left: forall b:bag 'a. diff empty_bag b = empty_bag
lemma Diff_add: forall b:bag 'a, x:'a. diff (add x b) (singleton x) = b
lemma Diff_comm: forall b b1 b2:bag 'a.
   diff (diff b b1) b2 = diff (diff b b2) b1
lemma Add_diff: forall b:bag 'a, x:'a.
  nb_occ x b > 0 -> add x (diff b (singleton x)) = b
```

Figure 5: Logic function for difference of bags

Proof obligations	**Alt-Ergo 0.93**	**CVC3 2.2**	**Coq 8.3pl2**	**Eprover 1.4 Namring**	**Simplify 1.5.4**	**Vampire 0.6**	**Yices 1.0.25**	**Z3 2.19**
is_empty		0.00		0.00		0.04		0.24
occ_singleton_eq	0.01	0.00		0.00	0.00	0.00	0.00	0.01
occ_singleton_neq	0.01	0.00		0.00	0.00	0.01	0.00	0.00
Union_comm			0.48					
Union_identity			0.50			0.18		
Union_assoc			0.50					
bag_simpl			0.50					
bag_simpl_left	0.01	0.00	0.51	0.09	0.02		0.00	0.01
occ_add_eq	0.01			0.24	0.10	0.05		1.15
occ_add_neq	0.01			0.37	0.07	0.85		0.17
Diff_empty_right			0.52			1.06		
Diff_empty_left			0.52			0.81		
Diff_add			0.54					
Diff_comm			1.00					
Add_diff			0.56					

Figure 6: Proof results for `Bag` theory

```
theory Bag_integers

function min_bag (bag int) : int
axiom Min_bag_singleton : forall x:int. min_bag (singleton x) = x
axiom Min_bag_union : forall x y:bag int.
      min_bag (union x y) = min (min_bag x) (min_bag y)
lemma Min_bag_union1 : forall x y: bag int, a: int.
     x = (add a y) -> min_bag x = min a (min_bag y)
lemma Min_bag_union2 : forall x : bag int, a: int.
     a <= min_bag x -> a <= min_bag (add a x)
end
```

Figure 7: `Bag of integers` theory

Proof obligations	Alt-Ergo 0.93	CVC3 2.2	Eprover 1.4 Namring	Simplify 1.5.4	Vampire 0.6	Z3 2.19
Min_bag_union1	0.02		0.06	0.01	1.12	0.02
Min_bag_union2	0.02	0.01	0.66		0.08	0.29

Figure 8: Proof results for `Bag of integers` theory

3.1.2 Bag of integers, minimum element

Since we deal binary heap for integers, we declare a logical function `min_bag` that returns the smallest element of a bag (Figure 7). This function is axiomatized using two axioms stating that: `min_bag` of a singleton is the element contained in this singleton, and `min_bag` of a union of two bags `x` and `y` is the minimum of `min_bag` of `x` and `min_bag` of `y`. We also proposed two lemmas. The first one specifies what is the minimum of the addition of an element to a bag. And the second one says that, if a given integer `a` is smaller that `min_bag` of a bag `b`, then it's also smaller that `min_bag` of the bag resulting from adding `a` to `b`.

These lemmas are automatically discharged by Alt-ergo, CVC3, Vampire and Z3 (Figure 8).

3.1.3 Bag of Elements of an Array

The third theory we describe, allows to define the bag of all elements occurring in an array.

In Why3, arrays are not built-in data types. In the standard library, there is a theory of polymorphic maps $map\ \alpha\ \beta$ from some type α to another one β. Thus we start by declaring the type $array\ \alpha$ as the type of maps from integers to α, as presented in Figure 9. Beware that, we use our own definition of type array different from the arrays of Why3's standard library, which are modifiable in-place.

Next we declare and axiomatize a logical function `elements`, as it shown in Figure 10. This function associates to an array `a`, and two integers `i` and `j`, a bag of elements between indexes `i` and `j-1` in `a`. The axiomatization of this function use two axioms. The first one `Elements_empty` states that `elements a i j` for an array a when i is greater or equal to j is an empty bag, since there are no elements between these two indexes. The second axiom `Elements_add` says that, if

```
use import map.Maps as A

type array 'a = A.map int 'a
```

Figure 9: Declaration of the type `array`

```
use import bag.Bag

(* [elements a i j] is the bag of elements in a[i..j[ *)
function elements (a:array 'a) (i j:int) : bag 'a
axiom Elements_empty : forall a:array int, i j:int.
     i >= j -> elements a i j = empty_bag
axiom Elements_add : forall a:array int, i j k:int.
     i < j ->
      elements a i j = add a[j-1] (elements a i (j-1))
```

Figure 10: Axiomatization of function `elements`

$i < j$ then the bag of elements between i and j in an array a is the addition of $a[j-1]$ to the bag of elements between i and $j-1$.

Furthermore, we provide a set of lemmas presented in Figure 11. And like for the two first theories, we summarize in Figure 12 which provers and how much time they need to prove these lemmas. `Elements_singleton` states that elements between two successive indexes i and $i+1$ of an array a correspond to a singleton containing $a[i]$. This lemma is a direct consequence of axiom `Elements_add`, and is automatically discharged by Z3.

Lemma `Elements_union` says that union of the bag of elements between indexes i and j and the bag of those between j and k is simply equal to the bag of elements between indexes i and k, for any integers i, j and k such that $i \leq j \leq k$. To prove this lemma, we need to make an induction on k and thus to use Coq.

Lemma `Elements_add1` states for any integers i and j such that $i < j$, that the bag of elements of an array a between i and j represent the addition of $a[i]$ to the bag of elements of the same array between $i+1$ and j. This lemma is proved in Coq simply by rewriting the term using the precedent lemmas.

Lemma `Elements_remove_last` express that, if $i < j-1$ for integers i and j, then the bag of elements of an array a between i and j represent the addition of $a[j-1]$ to the bag of elements of a between i and $j-1$. As you should expect, this lemma is automatically discharged by theorem provers: Alt-ergo, CVC3 and Z3.

Lemma `Occ_element` states that each element of the array between i and n occurs at least once in `elements a i n`, for any integers i and n. The proof of this lemma is done with Coq.

Lemma`Elements_set_outside` says that modifying an array at in dice k, does not modify the bag of element of this array between i and j if k is outside the interval $[i, j[$.

Lastly, two others lemmas allow to express what will be the bag of elements of an array a between i and n when a is modified at index j such that $i \leq j < n$. Thus, the lemma `Elements_set_inside2` states that; for i, j, and n, inserting a value e at $a[j]$, has as consequence that bag of element of a between i and n will contain a new occurrence of e, and $a[j]$ will be deleted. This lemma is automatically discharged by Z3. The second one, which is proved by Coq, is another formulation of the same property: if bag of elements of an array a is the addition of $a[j]$ to a bag b, then after insertion of the value e at $a[j]$, the bag of elements of a between i and n is equal to the addition of e to the bag b.

```
lemma Elements_singleton : forall a:array 'a, i j:int.
     j = i + 1 ->
       (elements a i j) = (singleton a[i])
lemma Elements_union : forall a:array 'a, i j k:int.
     i <= j <= k  ->
      (elements a i k) = union (elements a i j) (elements a j k)
lemma Elements_add1 : forall a:array 'a, i j :int.
      i < j ->
      (elements a i j) = add a[i] (elements a (i+1) j)
lemma Elements_remove_last: forall a:array 'a, i j :int.
      i < j-1 ->
      (elements a i (j-1)) =
         diff (elements a i j) (singleton a[j-1])
lemma Occ_elements: forall a:array 'a, i j n:int.
       i <= j < n ->
         nb_occ a[j] (elements a i n) > 0
lemma Elements_set_outside : forall a:array 'a, i j:int.
       i <= j -> forall k : int. (k < i || k >= j) ->
       forall e:'a.
         (elements (a[k<-e]) i j) = (elements a i j)
lemma Elements_set_inside :
  forall a:array 'a, i j n: int, e:'a, b:bag 'a.
    i <= j < n ->
    (elements a i n) = add a[j] b ->
    (elements a[j<-e] i n) = add e b
lemma Elements_set_inside2 : forall a:array 'a, i j n: int, e:'a.
       i <= j < n ->
       elements (a[j<-e] i n =
          add e (diff (elements a i n) (singleton a[j]))
```

Figure 11: `Elements` theory

Proof obligations	Alt-Ergo 0.93	CVC3 2.2	Coq 8.3pl2	Eprover 1.4 Namring	Yices 1.0.25	Z3 2.19
Elements_singleton						0.02
Elements_union			0.58			
Elements_add1			0.51		0.48	
Elements_remove_last	0.02	0.02		4.80		0.02
Occ_elements			0.56			
Elements_set_outside			0.66			
Elements_set_inside			0.60			
Elements_set_inside2	0.02					0.12

Figure 12: Proof results for `Elements` theory

```
module AbstractHeap

(**** logic declarations *****)

use import int.Int
use import bag.Bag
use import bag_of_integers.Bag_integers

(* abstract interface for heaps *)

type logic_heap
function model (h:logic_heap): (bag int)
function capacity (h:logic_heap): int
...
end
```

Figure 13: Abstract declaration of a binary heap

```
val create : sz:int ->
  { 0 <= sz }
  ref logic_heap
  { model !result = empty_bag ∧ capacity !result = sz }
```

Figure 14: Contract for `create` method

```
val insert : this:ref logic_heap -> e:int ->
  { card (model !this) < capacity !this }
  unit writes this
  { model !this = add e (model (old !this)) ∧
    capacity !this = capacity (old !this) }
```

Figure 15: Contract for `insert` method

3.2 Abstract interface for heaps, heap sort and test

In the second step, we look at the client part (the blue area in Figure 1). So we provide a formal specification for the abstraction of a binary heap. Then we propose a specification and a Why3 version of `heapsort` implementation, and then we prove the proposed `testHarness`.

Note that in this section we are not dealing with logical component, but with programs. Thus, each subsection in the following speaks about `module` and not `theory`.

3.2.1 Heap interface

Providing a formal specification for the abstraction consists to describe the contract of each method defined in the interface. Thus, as it is shown in Figure 13, in the module `Abstraction`, we firstly import the required external modules. After, we declare an abstract type `logic_heap`, together with two logical functions that defines an appropriate abstraction: `model` associates to an instance of `logic_heap` a bag of integers, `capacity` denotes its maximal capacity.

Then, we describe the methods' public profiles and contracts as follows.

The `create` method's post-condition tells that its result, which is a reference to a `logic_heap`, contains no elements, by saying that its model is an empty bag (Figure 14). Its capacity is set to the given parameter `sz`.

```
val extractMin : this:ref logic_heap ->
  { model !this <> empty_bag }
  int writes this
  { result = min_bag (model (old !this)) ∧
    model (old !this) = add result (model !this) ∧
    card (model (old !this)) = (card (model !this)) + 1 ∧
    capacity !this = capacity (old !this) }
```

Figure 16: Contract for `extractMin` method

For `insert` method, in one hand, the `writes` clause ensures that the input parameter `this` (reference to `logic_heap`) changes. In the other hand, the post-condition gives a relation between old and new value of `this`: model of `this` in the post state is the result of addition of the input parameter e to the model of `this` in the pre-state (Figure 15). There is also a pre-condition stating that the logic heap should not already be full. The post-condition additionally specifies that the capacity is unchanged.

As in the previous method, the `writes` clause of `extractMin` method ensures that the input parameter `this` is modified. The pre-condition states that model of the input parameter `this` should not be empty. Finally, the post-condition ensures that the returned value is the smallest element in the heap, and gives a relation between its old and its new value, too (Figure 16), by stating that the bag of elements contains one element less and thus the cardinality of the bag in the pre-state is equal to its cardinality in the post-state plus one.

This complete the building of formally specified interface for binary heaps.

3.2.2 Heap Sort Method

The second step is to annotate `heapSort` method as shown in Figure 17. Since we use `elements` and `min_bag`, we have to "import" theories `Elements` and `Bag_of_integers`, and by transitivity `Bag`. Note that this dependencies are represented in Figure 1 by arrows between `heapsort` and, `Array Elements` and `Bag_of_integers`. We also need to import some Why3 modules for dealing with maps and references.

As it's shown in Figure 17, too, the proposed code contains two loops. The first allows to create a heap from elements of an array a by invoking `insert` method. The loop invariant states that at iteration i the constructed heap contains exactly i elements which are the i first elements of a. The second loop, make inverse. That is extracting elements from the heap and inserting them in the array. Since `extractMin` returns the smallest element of the heap, then the resulting array is sorted. The loop invariant is not much more complicated. At each iteration, we need to precise what is the cardinality of model of the heap, and that it contains one element less than in precedent iteration. We also need to "say" that the i first elements of a are sorted, and that the $i - 1$ elements already extracted are smaller than the minimum in the heap.

For the annotated code presented in Figure 17, Why3 generates 16 proof obligations that are, all, automatically discharged by theorem provers. Actually, Why3 generates 11 verification conditions. However, some of them represent a conjunction of simpler formulas and are not directly proved by theorem provers. Thus, Why3 allows to split these proof obligations and provide a set of simpler ones that are automatically discharged by theorem provers.

The lemma `Min_of_sorted` described in Figure 18 and stating that, for a given map a sorted between 0 and n, the smallest element in the bag containing elements between `i` and `n` is `a[i]`, for any `i` between `0` and `n`, is required to help theorem provers to automatically discharge proof obligations generated by Why3. Nevertheless, it remains some verification conditions that necessitate the use of Coq, as it is detailed in Figure 19.

```
let heapSort (a : array int) =
  { length a >= 0 }
  'Init:
  let len = length a in
  let h = create len in

  for i = 0 to len-1 do
    invariant
      { 0 <= i <= len ∧
        card (model !h) = i ∧
        capacity !h = capacity (at !h 'Init) ∧
        model !h = elements a.elts 0 i }
    insert h a[i]
  done;

  for i = 0 to len-1 do
    invariant
      { 0 <= i <= len ∧
        card (model !h) = len - i ∧
        capacity !h = capacity (at !h 'Init) ∧
        elements (at a.elts 'Init) 0 len =
          union (model !h) (elements a.elts 0 i) ∧
        sorted_sub a 0 i ∧
        forall j:int. 0 <= j < i -> a[j] <= min_bag (model !h)
      }
    a[i] <- extractMin h;
    assert { a[i] <=  min_bag (model !h) }
  done

  { sorted a ∧
    elements a.elts 0 (length a) =
      elements (old a.elts) 0 (length a) }
```

Figure 17: Specification of `heapSort` method

```
lemma Min_of_sorted:    forall a:M.map int int, i n :int.
0 <= i < n -> (M.sorted_sub a 0 n) ->
min_bag (elements a i n) = M.get a i
```

Figure 18: Lemma `Min_of_sorted`

Proof obligations		Alt-Ergo 0.93	CVC3 2.2	Coq 8.3pl2	Eprover 1.4 Namring	Vampire 0.6	Yices 1.0.25	Z3 2.19
Min_of_sorted				0.81				
parameter heapSort	*precondition*	0.02	0.01			4.95	0.02	0.00
	normal postcondition	0.02	0.06				0.51	0.02
	for loop initialization	0.02	0.02		3.07		0.02	0.00
	for loop preservation	0.02	0.02				0.02	0.00
	normal postcondition	0.02	0.02		3.14		0.02	0.00
	for loop initialization	0.02	0.02			6.91	0.02	0.02
	for loop preservation	0.02	0.02				0.02	0.02
		0.02						
		0.02	0.02		0.81	0.91	0.02	0.02
		0.02	0.02				0.02	0.02
		0.02			2.35			0.84
		0.01						
		0.03	0.02				0.02	0.02
	normal postcondition	0.02	0.02				0.02	0.01
	for loop initialization		0.03		3.42		0.02	0.54
	for loop preservation	0.03	0.02		3.42	1.13	0.02	0.03
		0.02	0.02		3.46		0.03	0.03
		9.75				1.72		0.04
		0.02	0.03		1.06	1.03	0.02	0.02
		0.02	0.02			1.36	0.02	0.03
		0.02	0.02				0.02	0.03
		0.02						
								0.04
			0.17					
		0.04	0.04				0.03	
	normal postcondition	0.07	0.04		3.36		1.18	

Figure 19: Proof results for `heapSort` method

```
let testHarness () =
  let arr = Array.make 3 0 in
  arr[0] <- 42;
  arr[1] <- 13;
  arr[2] <- 42;
  heapSort arr;
  assert { arr[0] <= arr[1] <= arr[2] };
  assert { (elements arr.elts 0 3) =
        union (singleton 13) (union (singleton 42) (singleton 42)) };
  assert { arr[0] = min_bag (elements arr.elts 0 3) };
  assert { arr[0] = 13 };
  assert { arr[1] = min_bag (elements arr.elts 1 3) };
  assert { arr[1] = 42 };
  assert { arr[2] = 42 }
```

Figure 20: Specification of the `testHarness` method

Proof obligations		Alt-Ergo 0.93	CVC3 2.2	Vampire 0.6	Yices 1.0.25	Z3 2.19
parameter testHarness	*precondition*	0.02	0.00	3.22	0.01	0.00
	precondition	0.02	0.01	5.62	0.02	0.00
	precondition	0.02	0.01		0.02	0.00
	precondition	0.02	0.01		0.02	0.00
	precondition	0.01	0.01		0.02	0.00
	assertion	0.05	0.06			
	assertion					0.05
	assertion	0.05	0.02		0.02	5.82
	assertion	0.89	0.04			0.05
	assertion	0.02	0.02		0.02	
	assertion		0.04			0.03
	assertion		0.04			0.03

Figure 21: Proof results for the `testHarness` method

3.2.3 Testing Harness

Let us now try to prove a harness test presented in Section 2. As it is shown in Figure 20, we need additional assertions to help theorem provers to automatically discharge all the proof obligations.

Thus, in addition to these assertions, Why3, generates 8 verification conditions that are automatically discharged by Alt-ergo, CVC3 and Z3 as you can see it in Figure 21.

3.3 Proved implementation of binary heaps

The third and last part of our solution (red area in Figure 1), provide an implementation of the binary heap. Since heaps are complete binary trees where each node has a value greater or equal to those of its two sons, or that of its unique son, we propose an efficient implementation using the fact that a binary tree can be represented inside an array where only the values of the nodes are stored, and where the two sons of a node stored an index i are stored at indexes $2i + 1$ and $2i + 2$.

```
use map.Map as A
type map = A.map int int
type logic_heap = (map, int, int)

function model (h:logic_heap): (bag int) =
   let (a,n,c) = h in elements a 0 n

function capacity (h:logic_heap): int =
   let (a,n,c) = h in c
```

Figure 22: Concrete definition of the model of a heap

```
lemma Model_empty :
   forall a: array int, c:int. model (a,0,c) = empty_bag

lemma Model_singleton :
   forall a: array int, c:int. model (a,1,c) = singleton(a[0])

lemma Model_set :
  forall a : array int,v: int, i n c: int.
    0 <= i < n ->
        add (a[i]) (model (a[i <- v],n,c)) =
        add v (model (a, n, c))

lemma Model_add_last:
 forall a: array int, n c: int. n >= 0 ->
   model (a, n+1, c) = add (a[n]) (model (a, n, c))
```

Figure 23: Heap model theory

3.3.1 Model of heap

We give a concrete definition of heaps as shown in Figure 22. A heap h is a triple of an array a, and two integers n and c. c denotes the capacity of the heap and n the number of elements stored in it. The array a stores the binary tree as explained above. The logical function `model` which was just abstractly declared before, is now completely defined as the bag of elements of a between 0 and n. Same for function `capacity` that returns the third parameter of the triple representing the heap. We follow here the guidelines of the refinement approach for developing programs [21].

Then we provide, for this function four lemmas in Figure 23. The first two, allow to state that the model of an empty heap is an empty bag and a model of a heap containing only one element is a singleton of this element. The other two lemmas establish a relation between model of heap before its modification an its model after it is modified. These lemmas are automatically discharged by Alt-ergo, except `Model_set` which is proved with `Coq` (Figure 24).

3.3.2 Heap theory

This step consists to define a theory related to heap data structure. But before, we define some useful functions that are presented in Figure 25. These functions are just a syntactic sugar to simplify the formulas in the following. They define for a given node i, what is its left and right child, and its parent node.

Then we provide a few lemmas for these functions. These are shown in Figure 26. These are just arithmetic properties and are all discharged automatically by theorem provers as it will detailed later in Figure 29.

Proof obligations	**Alt-Ergo 0.93**	**CVC3 2.2**	**Coq 8.3pl2**	**Eprover 1.4 Namring**	**Vampire 0.6**	**Yices 1.0.25**	**Z3 2.19**
Model_empty	0.02	0.02		0.03	0.17	0.50	0.03
Model_singleton	0.02	0.02		6.57	0.51	0.04	0.04
Model_set			0.71				
Model_add_last		0.02				0.29	0.04

Figure 24: Proof results for Heap model theory

```
function left(i:int) : int = 2*i+1
function right(i:int) : int = 2*i+2
function parent(i:int) : int = div (i−1) 2
```

Figure 25: Syntactic sugar

```
lemma Parent_non_neg: forall i:int. 0 < i −> 0 <= parent i
lemma Parent_inf: forall i:int. 0 < i −> parent i < i
lemma Left_sup: forall i:int. 0 <= i −> i < left i
lemma Right_sup: forall i:int. 0 <= i −> i < right i
lemma Parent_right:forall i:int. 0 <= i −> parent (right i) = i
lemma Parent_left:forall i:int. 0 <= i −>  parent (left i) = i
lemma Child_parent:forall i:int. 0 < i −>
      left (parent i) = i || right (parent i) = i
lemma Inf_parent: forall i j: int. 0 < j <= right i −>
      parent j <= i
lemma Parent_pos: forall j: int. 0 < j −> 0 <= parent j
```

Figure 26: Lemmas about `left`, `right` and `parent` functions

```
predicate parentChild (i: int) (j: int) =
    0 <= i < j −> (j = left i) || (j = right i)

predicate is_heap_array (a: map) (idx: int) (sz: int) =
    0 <= idx −> forall i j: int.
      idx <= i < j < sz −>
      parentChild i j −>
      a[i] <= a[j]

predicate is_heap (h : logic_heap) =
   let (a, sz) = h in sz >= 0 ∧ is_heap_array a 0 sz
```

Figure 27: Predicates `ParentChild`, `Is_heap_array` and `is_heap`

```
lemma Is_heap_when_no_element :
  forall a:map, idx  n: int.  0 <= n <= idx -> is_heap_array a idx n

lemma Is_heap_sub :
  forall a:map, i n :int.
    is_heap_array a i n ->
     forall j: int.  i <= j <= n -> is_heap_array a i j

lemma Is_heap_sub2 :
  forall a:map, n :int.
    is_heap_array a 0 n ->
     forall j: int.  0 <= j <= n -> is_heap_array a j n

lemma Is_heap_when_node_modified :
  forall a:map, n e idx i:int. 0 <= i < n ->
     is_heap_array a idx n ->
     (i > 0 -> a[parent i] <= e ) ->
     (left i < n -> e <= a[left i]) ->
     (right i < n -> e <= a[right i]) ->
     is_heap_array (a[i <- e]) idx n

lemma Is_heap_add_last :
  forall a:map, n e:int. n > 0 ->
    is_heap_array a 0 n ∧ (e >= a[parent n]) ->
    is_heap_array (a[n <- e]) 0 (n + 1)

lemma Parent_inf_el:
     forall a: map, n: int.
       is_heap_array a 0 n ->
        forall j:int. 0 < j < n  -> a[parent j] <= a[j]

lemma Left_sup_el:
     forall a: map, n: int.
       is_heap_array a 0 n ->
        forall j: int. 0 <= j < n  -> left j < n ->
          a[j] <= a[left j]

lemma Right_sup_el:
     forall a: map, n: int.
       is_heap_array a 0 n ->
        forall j: int. 0 <= j < n -> right j < n ->
           a[j] <= a[right j]

lemma Is_heap_relation:
  forall a:map, n :int. n > 0 ->
    is_heap_array a 0 n ->
        forall j: int. 0 <= j -> j < n ->  a[0] <= a[j]
```

Figure 28: Heap theory

Then, we propose three predicates, shown in Figure 27. `ParentChild i j` holds if j is either the left or the right child of i. `Is_heap_array a idx sz` holds if the array $a[idx \ldots sz-1]$ is weakly ordered, i.e. each node is smaller (or equal) than its children. Finally, the predicate `is_heap h` is true whenever h has the heap property.

Proof obligations	Alt-Ergo 0.93	CVC3 2.2	Coq 8.3pl2	Eprover 1.4 Namring	Simplify 1.5.4	Vampire 0.6	Yices 1.0.25	Z3 2.19
Parent_inf	0.01	0.00			0.00		0.00	0.01
Left_sup	0.01	0.00			0.00		0.00	0.01
Right_sup	0.01	0.00			0.00		0.00	0.01
Parent_right	0.02	0.00			0.00		0.00	0.01
Parent_left	0.01	0.00			0.06		0.00	0.01
Inf_parent	0.02	0.01			0.00		0.00	0.01
Child_parent	0.12	1.81			0.24		0.00	0.01
Parent_pos	0.01	0.00		5.22	0.00		0.00	0.01
Is_heap_when_no_element	0.01	0.00			0.00	1.14	0.01	0.01
Is_heap_sub	0.01	0.00			0.00		0.01	0.02
Is_heap_sub2	0.02	0.00			0.00			0.02
Is_heap_when_node_modified	0.42	0.31					0.02	0.05
Is_heap_add_last		0.05					0.01	0.02
Parent_inf_el		0.01		3.95			0.01	0.02
Left_sup_el	0.02	0.01		4.74	0.01	4.85	0.01	0.02
Right_sup_el	0.02	0.01		4.85	0.00	4.76	0.00	0.02
Is_heap_relation			0.59					

Figure 29: Proof results for Heap theory

We propose a set of lemmas to present properties we expect from binary heaps. For example, we need to state that; if we add a new element e to a heap h, with respecting the condition that a node is greater than its parent node, stated in lemma `Parent_inf_el`, and smaller than its children, stated in `Left_sup_el` and `Right_sup_el` lemmas. Then h remains a heap. This property is specified in lemma `Is_heap_when_node_modified`. This is detailed in Figure 28.

As shown in Figure 29, all these lemmas are discharged automatically by Alt-ergo and CVC3, except the last one, `Is_heap_relation` which states that the root of a binary heap is smaller than all other elements of this heap. The proof of this lemma need an induction on integers and is proved in Coq.

3.3.3 Heap implementation

As remarked before, we follow a refinment approach the develop our program. The Why3 implementations of the 3 methods `create`, `insert` and `extractMin` are now given contracts which are the same as in the abstract interface, except that we add predicate `is_heap` as a private invariant for our data structure.

create method This method create a fresh instance of `logic_heap`, x, and returns a reference on x, as shown in Figure 30. The function `A.const` from Why3 standard library returns a constant map. For this method, Why3 generates one proof obligation that is automatically discharged by theorem provers (Figure 33).

```
let create (sz: int) : ref logic_heap =
  { true }
    let x = (A.const 0, 0, sz) in ref x
  { is_heap !result ∧
    model !result = empty_bag ∧
    capacity !result = sz }
```

Figure 30: Implementation of method `create`

```
let insert (this : ref logic_heap) (e : int) : unit  =
{ is_heap !this ∧ card (model !this) < capacity !this }
  let (a,n,c) = !this in
  let arr = ref a in
  let i = ref n in
  try
  while (!i > 0) do
    invariant {
      0 <= !i <= n ∧
      (!i = n →
         is_heap_array !arr 0 n  ∧
         model (!arr,n,c) = model (a,n,c)) ∧
      (!i < n →
         is_heap_array !arr 0 (n + 1) ∧
         !arr[!i] > e ∧
         model (!arr,n+1,c) = add !arr[!i] (model (a,n,c)))
    }
    variant { !i }
    let parent = div (!i − 1) 2 in
    let p = A.get !arr parent in
    if (e >= p) then raise Break;
    arr := !arr[!i<−p];
    i := parent
  done
  with Break → ()
  end;
  arr := !arr[!i<−e];
  this := (!arr, n + 1, c);
  assert { 0 < !i < n → is_heap !this };
  assert { !i < n → model !this = add e (model (a,n,c)) }
{ is_heap !this ∧
    model !this = add e (model (old !this)) ∧
    capacity !this = capacity (old !this) }
```

Figure 31: Implementation of method `insert`

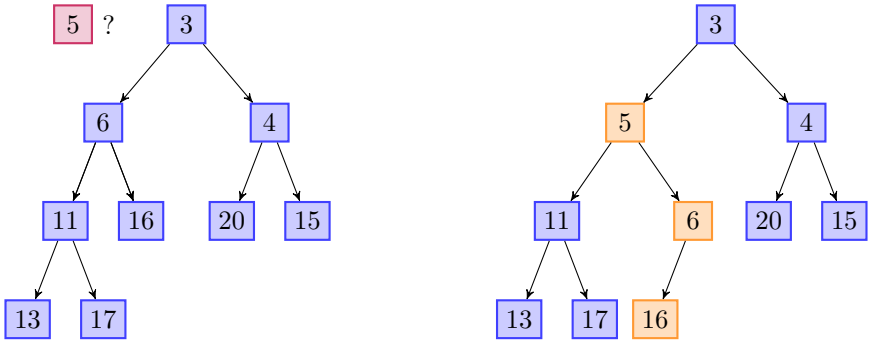

Figure 32: Insertion of the new element 5 in a heap

Proof obligations		Alt-Ergo 0.93	CVC3 2.2	Coq 8.3pl2	Eprover 1.4 Namring	Vampire 0.6	Yices 1.0.25	Z3 2.19
parameter create		0.03	0.06		0.19	0.51		0.04
parameter insert	*loop invariant init*	0.11	0.02				0.03	0.04
	assertion						1.23	
	assertion	0.32	0.17				0.03	0.05
	normal postcondition	0.21					1.40	
			0.04				0.03	0.72
		0.06						
	loop invariant preservation		1.68				1.69	
	loop variant decreases	0.08	0.04		0.22	0.04	0.03	0.04
	assertion						0.03	
	assertion	0.38	0.21				0.03	0.05
	normal postcondition	0.59	0.36					
		2.22	0.03				0.03	
		0.06	0.04		0.03		0.03	0.04

Figure 33: Proof results for `create` and `insert` implementation

Insert method For the `insert` method, we follow the classical algorithm, which tries to add the new element in the first free cell. Since this insertion may destroy the heap property, we need to move this new element up, until we reach a smaller parent. This algorithm is illustrated in Figure 32, and the corresponding code is annotated Figure 31.

For this method, 9 verification conditions are generated. But as, for `HeapSort` method, we need to split some proof obligations to help theorem provers. So, these verification conditions are automatically discharged, except one assertion as it is mentioned in Figure 33.

extractMin method Finally, for the `extractMin` method, we just need to return the root of the heap, and the last element of the array must be reinserted in the heap. The associated modifications in the heap are illustrated in Figure 35, and the code of the method is given in Figure 34.

```
let extractMin (this : ref logic_heap) : int =
{ model !this <> empty_bag ∧ is_heap !this }
  let (a,n,c) = !this in
  assert {n > 0};
  let min = a[0] in
  let n' = n−1 in
  let last = a[n'] in
  assert { n' > 0 −> nb_occ last (diff (model (a,n,c))
                        (singleton min)) > 0 } ;
  let arr = ref a in
  let i = ref 0 in
  try
    while ( !i < n') do
      invariant {
        0 <= !i ∧
        (n' > 0 −> !i < n') ∧
        is_heap_array !arr 0 n' ∧
        (!i = 0 −> !arr = a) ∧
        (n' > 0 −>
          elements !arr 0 n' =
            add !arr[!i] (diff (diff (model (a,n,c))
                 (singleton last))
               (singleton min))) ∧
        (!i > 0 −> !arr[parent !i] < last) }
      variant {n' − !i}
      let left = 2 * !i + 1 in
      let right = 2 * !i + 2 in
      if (left >= n') then raise Break;
      let smaller = ref left in
      if right < n' then
        if !arr[left] > !arr[right]
          then smaller := right;
      if last <= !arr[!smaller] then raise Break;
      arr := !arr[!i <− !arr[!smaller]];
      i := !smaller
    done;
    assert { n' = 0 }
  with Break −> ()
  end;
  if !i < n' then
    begin
      arr := !arr[!i <− last];
      assert { !i > 0 −> is_heap_array !arr 0 n' };
      assert { is_heap_array !arr 0 n' };
      assert { n' > 0 −> elements !arr 0 n' =
                  (diff (model (a,n,c)) (singleton min)) }
    end;
  this := (!arr,n',c);
  min
{ is_heap !this ∧
  result = min_bag (model (old !this)) ∧
  model (old !this) = add result (model !this) ∧
  capacity !this = capacity (old !this) }
```

Figure 34: Implementation of method `extract`

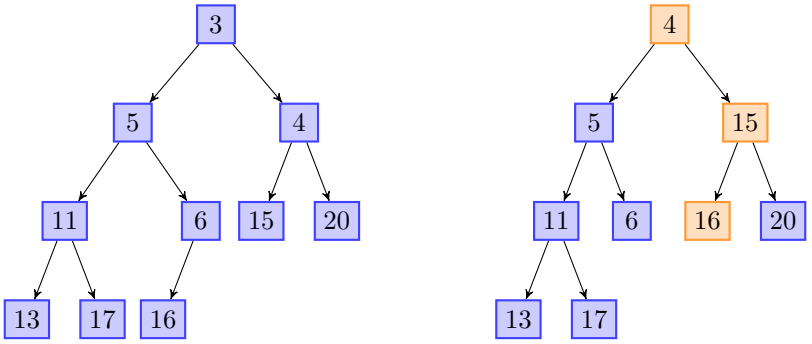

Figure 35: Extraction of the smallest element of a heap

```
lemma Is_heap_min:
  forall a:array int, n c:int. n > 0 ->
    is_heap_array a 0 n -> A.get a 0 = min_bag (model (a, n, c))
```

Figure 36: Heap_min lemma statement

In this case, Why3 generates 35 verification conditions, some of which must be split to be proved by theorem provers. See Figures 37 and 38.

To help theorem provers, we also provide a lemma `Is_heap_min`, which states that the root of a given heap is the smallest element of it's model (Figure 36).

4 Statistics and Scores

The machine used for playing the proofs has 8 cores (Intel Xeon W5580) at 3.20GHz, with 24Gb of RAM. To evaluate the time needed to play all the proofs in batch mode, we use the `why3replayer` command. In Why3, all the proofs made, including Coq ones, are stored in a proof session project. The `why3replayer` command reruns all the provers on these proofs. Indeed, the tables giving proofs result shown in previous section are generated with this command.

We replayed all the proofs in one batch configured such that up to four provers can be executed in parallel. Since these are independant processes, this effectively make use of the multiple cores of the machine. The overall wall-clock time used to replay is around 3 minutes and 1 second (computed indeed using the average of several executions). The CPU time is around 9 minutes and 56 seconds.

In the VACID-0 paper [19], there is a proposal for computing the score on each example. On the Binary Heap example, there is 40 points for verifying that `heapSort` returns a sorted array, and and 40 other points for verifying that the result is a permutation of the input array. Additionaly, 5 points are given if termination is proved. We verify all of that, so we get these 85 points. Additional points are given related to the time taken to replay the proofs, using the formula $5 - \frac{1}{2}\ln t$. For us, $t = 181s$. Thus, we get 2.401 points. Finally, some points are awarded depending on the ratio of size of annotations over size of programs, measured in token. We computed the number of tokens using Why3 itself, and got 2951 tokens in annotations and 752 tokens in programs. The ration is thus equal to $r = 3.924$ and the points awarded are $6 - 2\ln r = 3.266$.

The total number of points we obtain is thus 90.667.

We remind that we did more than what was required. We proved an implementation of binary heaps. The figures above include this part of the verification. If we replay everything except the implementation, the time would significantly shorter, and the ratio spec/code smaller too, so we would get more points. But indeed we believe we deserve more points for proving the implementation.

Proof obligations		Alt-Ergo 0.93	CVC3 2.2	Coq 8.3pl2	Eprover 1.4 Namring	Vampire 0.6	Yices 1.0.25	Z3 2.19
Is_heap_min				0.66				
parameter extractMin	*assertion*	0.01	0.02				0.03	0.04
	assertion			0.65				
	loop invariant init	0.02	0.02				0.03	0.00
		0.02	0.03				0.02	0.00
		0.05	0.10					0.04
		0.02	0.02		0.00	0.00	0.02	0.00
		0.03	0.18				0.03	0.05
		0.02	0.02				0.02	0.00
	assertion	2.59						
	assertion	0.03	0.04				0.03	
	assertion		0.22					0.04
	normal postcondition		0.04				0.03	
			0.04		0.23		0.02	0.07
			0.21				0.03	
		0.16						
	normal postcondition	0.03	0.03				0.03	
			0.03		0.00		0.03	0.00
			0.03				0.03	0.00
		0.02						
	assertion	0.04						
	assertion	1.02						
	assertion		0.05					0.04
	normal postcondition		1.51				0.03	
	normal postcondition		0.03				0.03	
	loop invariant preservation	0.04	0.04				0.03	0.04
		0.03	0.04		0.01	0.03	0.03	0.04
		0.05						
		0.04	0.04				0.02	0.04
		0.10	0.16					
		0.07	0.06				0.03	
	loop variant decreases	0.05	0.05				0.03	0.04
	assertion	2.27						
	assertion	0.90						
	assertion		0.24					0.04
	normal postcondition	0.72	0.05				0.03	
			0.04		0.22		0.03	0.09
			0.20				0.03	0.05
		0.25						
	normal postcondition		0.03				0.02	
			0.03		0.01		0.03	0.00
			0.03				0.02	
		0.03						

Figure 37: Proof results for `extractMin` implementation (part 1)

Proof obligations		Alt-Ergo 0.93	CVC3 2.2	Coq 8.3pl2	Eprover 1.4 Namring	Vampire 0.6	Yices 1.0.25	Z3 2.19
parameter extractMin	*loop invariant preservation*	0.04	0.04				0.03	0.04
		0.03	0.04		0.23	0.06	0.03	0.04
		0.05						
		0.03	0.04				0.03	0.03
		0.09	0.15				0.03	
		0.14	0.06				0.04	
	loop variant decreases	0.04	0.04				0.02	0.04
	assertion	2.78						
	assertion	0.04						
	assertion		0.24					0.04
	normal postcondition		1.68				0.03	
	normal postcondition	0.02	0.03				0.02	
			0.03		0.01		0.03	0.00
		0.02	0.03				0.03	0.00
		0.03						
	loop invariant preservation	0.03	0.05				0.03	0.04
		0.03	0.05		0.23	0.06	0.03	0.04
		0.06						
		0.03	0.04				0.03	0.04
		0.08	0.16				0.03	
		0.12	0.06				0.03	
	loop variant decreases	0.04	0.05				0.03	0.04
	assertion	0.02	0.03				0.03	0.04
	assertion	0.03	0.03		0.18	0.01	0.03	0.00
	assertion	0.03						
	assertion	0.02	0.03		0.01	0.01	0.03	0.00
	normal postcondition	0.02	0.02				0.03	0.00
	normal postcondition	0.36	0.06				1.07	0.15

Figure 38: Proof results for `extractMin` implementation (part 2)

5 Related Works and Conclusions

Related Works In 1999, Filliâtre and Magaud [12] present a formal proof of total correctness of the heap-sort algorithm in the system `Coq`. The implementation is an imperative program processing the given array in-place, it proceeds by transposition and is significantly different from ours.

In 2011, Burghardt et al. [7] provide various examples for the formal specification, implementation, and deductive verification of C programs using the ANSI/ISO-C Specification Language and the Jessie plug-in of Frama-C. They are interested in different algorithms and data structures like stacks and heaps. For the latter, they consider various heap-related algorithms, such that: testing at run time whether a given array is arranged as a heap, adding or removing an element to or from a given heap, etc. Finally, they use these to implement a sorting algorithm. But, in this proposition, the authors did not do the hardest part, namely the proving the preservation of heap contents (that is the result array is a permutation of the input array). They only signal this gap by placing a non-ACSL comment.

On the VACID-0 website, a solution in VCC [9] is given (and none in Dafny [18]). Only the annotated source code is given, without explanations and we are not able to compare with our solution.

Future Works in the near future we would like to specify and prove a C version of this case study, annotated in ACSL [2], using Frama-C [15] and its Jessie plugin [20]. But more importantly, we would like to arrange the specification in a way to reflect that our abstract interface is formally *refined* into the implementation, following the guidelines we presented in an earlier paper [21].

References

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

[2] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.

[3] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[5] 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*, Wrocław, Poland, August 2011.

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

[7] J. Burghardt, J. Gerlach, L. Gu, K. Hartig, H. Pohl, J. Soto, and K. Völlinger. ACSL by example, towards a verified C standard library. Technical report, Fraunhofer First, 2011. `http://www.first.fraunhofer.de/fileadmin/FIRST/ACSL-by-Example.pdf`.

[8] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[9] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009.

[10] E. W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

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

[12] J.-C. Filliâtre and N. Magaud. Certification of Sorting Algorithms in the System Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, Nice, France, 1999.

[13] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

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

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

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580 and 583, Oct. 1969.

[17] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007.

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

[19] 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.

[20] Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, Jan. 2009.

[21] A. Tafat, S. Boulmé, and C. Marché. A refinement methodology for object-oriented programs. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented Software, Papers Presented at the International Conference*, Karlsruhe Reports in Informatics, pages 143–159, Paris, France, June 2010. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083`.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399