Lecture at DigiCosme Spring School 2013

Deductive Program Verification with Why3
A Tutorial

Jean-Christophe Filliâtre

April 2013

Information

This lecture has been prepared using Why3 version 0.81 (March 2013). The material related to the lecture (slides, exercises, source code from this document, etc.) is available at

`http://why3.lri.fr/digicosme-spring-school-2013/`

In addition, one may consult Why3's documentation and examples from Why3's web site `http://why3.lri.fr`.

Author:

Jean-Christophe Filliâtre

LRI - bâtiment 650
Université Paris-Sud
91405 Orsay Cedex
France

Email : `filliatr@lri.fr`
Web : `http://www.lri.fr/~filliatr/`

Contents

Chapter 1

Introduction

Any computer scientist knows John McCarthy's 91 function [?]. It is the recursive function defined as follows:

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

One may wonder whether this function always return 91, as suggested by its name, or, otherwise, under which condition it does. One may also want to prove its termination, which is not obvious, or to prove that it is equivalent to the following iterative code:

```
$e \leftarrow 1$
while $e > 0$ do
  if $n > 100$ then
    $n \leftarrow n - 10$
    $e \leftarrow e - 1$
  else
    $n \leftarrow n + 11$
    $e \leftarrow e + 1$
return n
```

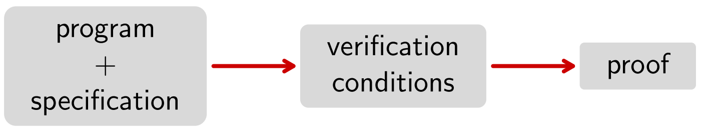
These are questions related to *program verification*. The programs above are given in pseudo-code, but we can pose similar questions for programs written in mainstream programming languages. For instance, one may wish to prove that the following Java code indeed sorts an array of Boolean values

```
int i = 0, j = a.length - 1;
while (i < j)
  if (!a[i]) i++;
  else if (a[j]) j-;
  else swap(a, i++, j-);
```

or that the following C program, although purposely obfuscated, indeed computes the number of solutions to the N-queens problem:

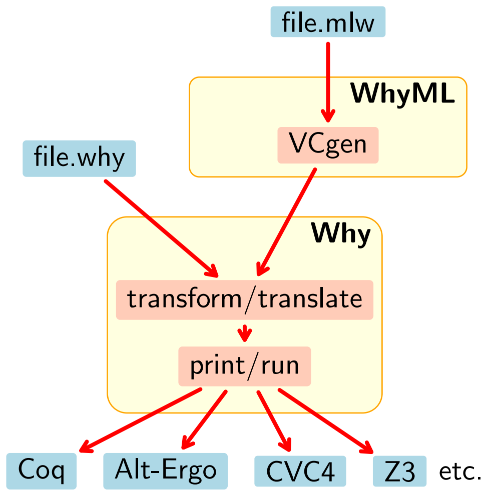
```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0«q),0,0));}
```

This lecture is an introduction to Why3 [?], a set of tools to perform *deductive program verification* [?]. Deductive verification means that we express the correctness of a program as a mathematical statement and then we prove it, which can be summarized as follows:



This idea is as old as computer science [?, ?], but it is making a giant leap today thanks to recent progress in *automated theorem provers*, especially those of the SMT family[1]. SMT solvers are able to discharge a huge amount of the verification tasks, if not all.

The Why3 platform provides a logic language (called Why) and a programming language (called WhyML). The logic is an extension of first-order logic with polymorphism, algebraic data types, and inductive predicates. The programming language is a first-order ML-like language with imperative features, pattern matching, and exceptions. The two languages are intimately tied: any logic symbol can be used in programs, and the logic is used to specify programs (via traditional pre- and postconditions, loop invariants, etc.). A standard weakest precondition calculus [?] is used to extract verification conditions from programs. The resulting formulas are then going through several transformations to be sent to a wide set of external theorem provers. The data flow may be depicted as follows:



There are basically three ways of using Why3.

for its logic only

It is perfectly fine using Why3 without writing a single program. It means you use the logic only (declaring, defining, axiomatizing, stating goals) and you see Why3 as a mere front-end for dozens of theorem provers. Yet you save the burden of learning the input languages of all these tools, and the various ways of calling them. You can also use Why3 to manipulate formulas by implementing your own transformations (there is an OCaml API).

[1]A computer scientist recently coined the concept of *SMT revolution*.

to verify algorithms and data structures

Though WhyML has limited imperative features (aliasing is not allowed), a lot of programs and data structures can be implemented in WhyML. Even when some data structure cannot be implemented, it is usually easy to model it and the remaining of the verification can be carried out. WhyML programs can be translated to executable OCaml code.

as an intermediate language

To verify programs written in a mainstream programming language, it is convenient to use Why3 as an intermediate language. A suitable memory model is designed in Why3 logic and then program constructs are compiled into WhyML constructs. The task of extracting verification conditions (and to pass them to theorem provers) is left to Why3.

So far, Why3 has been successfully used that way to verify Java programs [?], C programs [?, ?], Ada programs [?], probabilistic programs [?], and cryptographic programs [?].

The following three chapters describe these three ways of using Why3.

Many technical aspects are not covered in this tutorial, such as the transformation process to encode Why3's logic into SMT logic, the type checking rules to exclude aliasing, or the weakest precondition calculus used to extract verification conditions. We invite to reader to get more information from Why3's web site (`http://why3.lri.fr`) and from publications related to Why3 [?, ?, ?, ?, ?, ?, ?].

Chapter 2

Logic

In this chapter, we introduce Why3 logic. Program verification will be covered in the next two chapters.

2.1 First Contact with Why3

In a file `demo_logic.why`, let us define the algebraic data type of polymorphic lists. To do so, we introduce a theory `List` containing the definition of type `list ’a`[1].

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

This file can be processed in a batch way using the following command:

```
  why3 demo_logic.why
```

The contents of file `demo_logic.why` is checked, then printed on standard output. Let us add the definition of a recursive predicate `mem` checking for the presence of an element `x` in a list `l`.

```
  predicate mem (x: ’a) (l: list ’a) = match l with
    | Nil -> false
    | Cons y r -> x = y \/ mem x r
  end
```

The system automatically checks for the terminations of `mem`, looking for a subset of its arguments that ensures a lexicographic structural decreasing. In this case, it is limited to argument `l`. If we had written `mem x l` instead of `mem x r`, then the definition of `mem` would have been rejected:

```
  why3 demo_logic.why
  File "demo_logic.why", line 10, characters 2-6:
  Cannot prove the termination of mem
```

Let us add a third declarations to theory `List`, namely a goal to show that 2 belongs to the list [1;2;3]. We state it as follows:

[1]The type variable α is written `’a` in the concrete syntax, as in OCaml.

```
  goal G1: mem 2 (Cons 1 (Cons 2 (Cons 3 Nil)))
```

It is then possible to attempt discharging this goal using one of the theorem provers supported by Why3. Let us assume that the SMT solver Alt-Ergo is installed and recognized by Why3 (this can be checked with `why3 -list-provers`). Then we pass goal `G1` to Alt-Ergo using option `-P`.

```
  why3 -P alt-ergo demo_logic.why
  demo_logic.why List G1 : Valid (0.03s)
```

As we see, the goal is declared valid, without surprise. We can use the graphical user interface `why3ide` instead.

```
  why3ide demo_logic.why
```

Then we can launch theorem provers interactively, including proof assistants such as Coq.

Let us define now the length of a list. We do that in a second theory `Length`, whose first two declarations import the previous theory `List`, as well as the theory of integer arithmetic from Why3 standard library, namely `int.Int`.

```
theory Length
  use import List
  use import int.Int
```

In theory `List`, we did not have to import arithmetic, since only integer literals were used; but here we need addition and order relation as well. We define a recursive function `length` over lists, similarly to what we did for predicate `mem`.

```
  function length (l: list 'a) : int = match l with
    | Nil -> 0
    | Cons _ r -> length r + 1
  end
```

Once again, the system automatically checks for termination. Then we state a lemma saying that a list length is always nonnegative.

```
  lemma length_nonnegative: forall l:list 'a. length(l) >= 0
```

The automated theorem prover Alt-Ergo is not able to discharge it. More specifically, it quickly aborts with message `Unknown`, which means it could not check the validity of the goal.

```
  why3 -P alt-ergo demo_logic.why
  demo_logic.why List G1 : Valid (0.01s)
  demo_logic.why Length length_nonnegative : Unknown: Unknown (0.01s)
```

Indeed, proving lemma `length_nonnegative` requires induction, which is out of reach of SMT solvers. One may use the Coq proof assistant instead to do that proof, by calling it from `why3ide` for instance. However, Alt-Ergo is able to discharge the following goal, using lemma `length_nonnegative` as an hypothesis:

```
  goal G3: forall x: int, l: list int. length (Cons x l) > 0
```

This is the distinction between declarations `goal` and `lemma`: the latter introduces a goal that may be used in the following. It is easy to check that lemma `length_nonnegative` is indeed an hypothesis of goal `G3` from the graphical user interface (in the top-right window).

Let us now define the notion of sorted list. We start with lists of integers. We introduce a new theory `SortedList` for that purpose, containing the declarations of an inductive predicate `sorted`[2].

```
theory SortedList
  use import List
  use import int.Int

  inductive sorted (list int) =
  | sorted_nil:
      sorted Nil
  | sorted_one:
      forall x: int. sorted (Cons x Nil)
  | sorted_two:
      forall x y: int, l: list int.
      x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
end
```

Such a declaration defines `sorted` as the smallest predicate satisfying the three "axioms" `sorted_nil`, `sorted_one`, and `sorted_two`. We can state a goal saying that the list [1;2;3] is sorted:

```
  goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
```

and it is easily discharged by automated theorem provers. However, defining `sorted` over lists of integers only, and using the order relation `<=` only, is not satisfactory. As much as possible, we could prefer defining `sorted` in a generic way, once and for all. The logic of Why3 being a first-order logic, it is not possible to pass the order relation as an argument to predicate `sorted`. Though, there is an alternative. We modify theory `SortedList` to use an abstract data type `t` instead of integers and an uninterpreted binary predicate `<=` over `t`.

```
theory SortedList
  use import List
  type t
  predicate (<=) t t
```

To make it right, we add three axioms to express that `<=` is an order relation.

[2]We could have defined predicate `sorted` recursively, as follows:

```
predicate sorted (l: list int) = match l with
  | Nil | Cons _ Nil -> true
  | Cons x (Cons y _ as r) -> x <= y /\ sorted r
end
```

But the purpose here is to introduce inductive predicates.

```
  axiom le_refl:  forall x: t. x <= x
  axiom le_asym:  forall x y: t. x <= y -> y <= x -> x = y
  axiom le_trans: forall x y z: t. x <= y -> y <= z -> x <= z
```

The definition of `sorted` is mostly unchanged: we simply substitute `t` for `int` and relation `<=` over type `t` for relation `<=` over type `int`.

```
  inductive sorted (list t) =
  | sorted_nil:
      sorted Nil
  | sorted_one:
      forall x: t. sorted (Cons x Nil)
  | sorted_two:
      forall x y: t, l: list t.
      x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
end
```

To handle the particular case of lists of integers, we first set up a new theory in which we import lists and integers:

```
theory SortedIntList
  use import int.Int
  use import List
```

Then we *instantiate* the generic theory `SortedList` with type `int` and with order relation `<=` over integers. To do so, we use the `clone` command, rather than `use`, as follows:

```
  clone import SortedList with type t = int, predicate (<=) = (<=),
    lemma le_refl, lemma le_asym, lemma le_trans
```

This command makes a copy of theory `SortedList`, while substituting type `int` to type `t` and the order relation `<=` over integers to the uninterpreted predicate `<=` (in this case, the two predicates have the same name but this is not a requirement). This command introduces the declaration of a *new* inductive predicate `sorted`, with an argument of type `list int`. We can use it to state the following goal:

```
  goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Additionally, we notice that the `clone` command above also included `lemma le_refl`, `lemma le_asym`, and `lemma le_trans`. It means that these three axioms must be proved[3]. They are turned into lemmas. We can easily check this out by looking at the set of goals contained in file `demo_logic.why`:

```
  why3 -P alt-ergo demo_logic.why
  ...
  demo_logic.why SortedIntList le_refl : Valid (0.00s)
  demo_logic.why SortedIntList le_asym : Valid (0.00s)
  demo_logic.why SortedIntList le_trans : Valid (0.00s)
  demo_logic.why SortedIntList sorted123 : Valid (0.00s)
```

[3]The system cannot automatically infer axioms that could become lemmas; this is not decidable. Thus it is up to the user to indicate the axioms to be turned into lemmas.

As illustrated by this example, the concept of cloning allows the design of generic theories to be instantiated later in various ways. This is analogous to generic classes in Java or functors in ML. But this is also slightly more flexible, as parameters do not have to be set once and for all. A theory is thus parameterized in various ways simultaneously.

In the example above, we can introduce another generic theory, namely that of order relation. It contains type `t`, order relation `<=`, and the three axioms.

```
theory Order
  type t
  predicate (<=) t t

  axiom le_refl:  forall x: t. x <= x
  axiom le_asym:  forall x y: t. x <= y -> y <= x -> x = y
  axiom le_trans: forall x y z: t. x <= y -> y <= z -> x <= z
end
```

Then we simply clone it inside theory `SortedList`, to get the exact same theory as before:

```
theory SortedList
  use import List
  clone export Order
  ...
end
```

The benefit here is the ability to reuse theory `Order` in other contexts. Figure **??** summarizes this new version of theory `SortedList` and its use on lists of integers. The Why3 standard library is built this way. Beside the obvious benefit of factorization, splitting the declarations into small theories also allows a fine grain control of the context of each goal. With SMT solvers, in particular, limiting the size of the logical context may improve performances significantly.

Summary. The logic of Why3 is an extension of first-order logic with polymorphism, mutually recursive algebraic data types, mutually recursive function/predicate symbols, mutually inductive predicates, and constructs `let-in`, `match-with`, and `if-then-else`. Logical declarations are of four different kinds:

- type declaration;
- function or predicate declaration;
- inductive predicate declaration;
- axiom, lemma, or goal declaration.

Logical declarations are organized into *theories*. A theory T_1 may be

- used (`use`) in another theory T_2. In that case, symbols from T_1 are *shared*, axioms from T_1 remain axioms, lemmas from T_1 become axioms, and goals from T_1 are discarded.
- or cloned (`clone`) in another theory T_2. In that case, declarations from T_1 are *copied* or *substituted*, axioms from T_1 remain axioms or become lemmas/goals, lemmas from T_1 become axioms, and goals from T_1 are discarded.

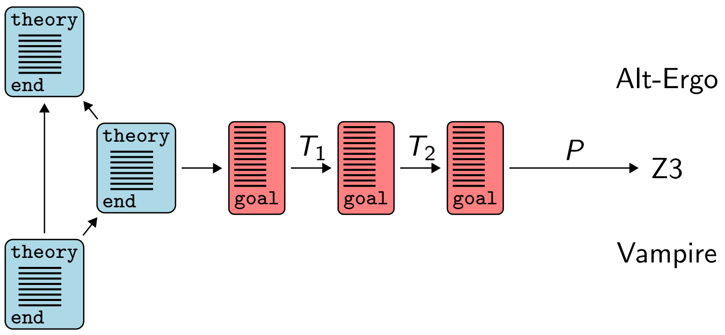
```
theory List
  type list 'a = Nil | Cons 'a (list 'a)
end

theory Order
  type t
  predicate (<=) t t

  axiom le_refl:  forall x: t. x <= x
  axiom le_asym:  forall x y: t. x <= y -> y <= x -> x = y
  axiom le_trans: forall x y z: t. x <= y -> y <= z -> x <= z
end

theory SortedList
  use import List
  clone export Order

  inductive sorted (list t) =
  | sorted_nil:
      sorted Nil
  | sorted_one:
      forall x: t. sorted (Cons x Nil)
  | sorted_two:
      forall x y: t, l: list t.
      x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
end

theory SortedIntList
  use import int.Int
  use import List

  clone import SortedList with type t = int, predicate (<=) = (<=),
    lemma le_refl, lemma le_asym, lemma le_trans

  goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
end
```

Figure 2.1: A generic theory of sorted lists.

2.2 Talking to Theorem Provers

One of the first benefits of Why3 is to provide a technology to talk to provers. There are various theorem provers available and they have distinct logical languages, predefined theories, or type systems. Why3 provides a common language and a common tool to use them.

This technology is organized around the notion of *task*. A task is a logical context, that is a list of declarations, followed but exactly one goal, that is a formula. Tasks are extracted from the various theories. Given a task and a target theorem prover, Why3 performs a series of transformations on the task, so that it fits in the logic of the prover. For example, if the target prover is Z3, Why3 will apply, among other things, a transformation to get rid of inductive predicates, since Z3 knows nothing about it. Once all transformations are applied, Why3 simply prints the resulting task into the native syntax of the prover, using a pretty-printer. We can display the task flow as follows:



This journey is driven by a file (called a *driver* in Why3's terminology). This file lists the transformations to apply, the output format (that is the input syntax of the prover as well as predefined symbols and axioms), and regular expressions to diagnose messages from the prover. Such a file can be set up by the user, for instance to add support for a new prover or to conduct experiments with a prover. Drivers are provided with Why3 for the following proves: Alt-Ergo, CVC3, Yices, E-prover, Gappa, MathSAT, Simplify, Spass, Vampire, VeriT, Z3, Zenon, iProver, Mathematica, Coq, PVS.

Additionally, Why3 provides an OCaml API. One can build terms, formulas, declarations, tasks, theories (that are all guaranteed to be well-formed), and call provers. Using plug-ins, the system can be extended with new input languages, new transformations, and new pretty-printers. In particular, everything needed to add support for a new prover can be added to the system.

More details can be found in the paper *Why3: Shepherd your herd of provers* [?] as well as Why3's manual [?].

Chapter 3

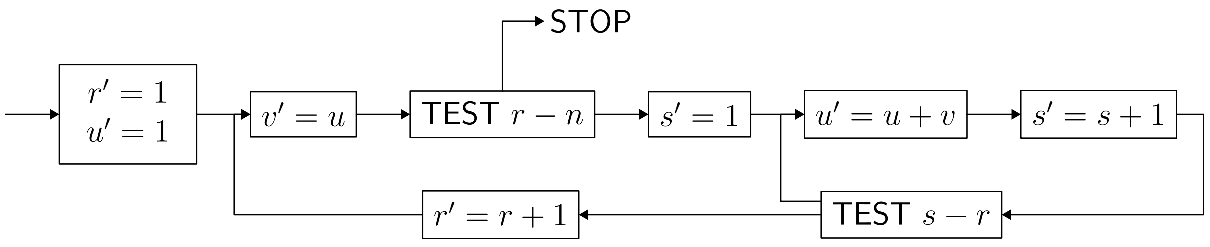
Program Verification

We now turn to program verification using Why3.

3.1 First Examples

3.1.1 Turing's Proof of a Program

As a first example, let us consider what is most likely the very first proof of a program. It is a 1949 paper by Alan Turing called *Checking a Large Routine* [?]. The program computes $n!$ using only additions. It is given as a control flow graph:



The notation $u' = u+v$ stands for the assignment of $u+v$ to variable u, that is $u \leftarrow u+v$ using a modern notation. Variable n contains the input parameter and is not modified. When the program exits, variable u contains $n!$, the factorial of n. There are two nested loops. In the outer loop, variable u contains $r!$ and its contents is copied into variable v. The inner loop computes $(r + 1)!$ in u, using repetitive additions of v. Variables r and s are loop counters, so we can easily rewrite the program using two for loops. With a slight modification of the control flow[1], the program above is thus equivalent to:

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

Let us prove the correctness of this program using Why3. In a file `demo_turing.mlw` we write the program above in the syntax of WhyML, the programming language of Why3.

[1]Finding which is left to the reader as an exercise.

Program declarations are grouped in *modules* (the programming equivalent to logical theories).

```
module CheckingALargeRoutine
  use import int.Fact
  use import ref.Ref
```

The first command imports the logical theory `int.Fact`, that is theory `Fact` from Why3 standard library. This theory provides a function symbol `fact` denoting the factorial. The second command imports a *module* from the standard library of WhyML, namely the module `Ref` providing references. As in ML, references are mutable variables: a reference is created with function `ref`, we access the contents of reference `x` with `!x`, and we assign it with `x := e`. We can now write a function `routine` taking an integer argument `n` as argument and computing its factorial using the algorithm above:

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

The code itself is similar to OCaml code. The specification is introduced with keywords `requires` and `ensures`. `requires` introduces the precondition `n >= 0` and `ensures` the postcondition `result = fact n`. Within the postcondition, variable `result` stands for the value returned by the function.

As we did in the previous chapter with a file containing logical declarations, we can pass this file to Why3 for type checking and proof processing.

```
why3 -P alt-ergo demo_turing.mlw
WP CheckingALargeRoutine WP_parameter routine : Unknown: Unknown (0.08s)
```

We see that Alt-Ergo was not able to discharge the verification condition. If we switch to the graphical user interface (`why3ide`) — which is quickly mandatory when performing program verification — we can have a look at the goal expressing the correctness of function `routine`. Omitting the context, it reduces to the following formula:

$$\forall n,\ n \geq 0 \Rightarrow (0 > n-1 \Rightarrow 1 = \texttt{fact}\ n) \wedge (0 \leq n-1 \Rightarrow \forall u,\ u = \texttt{fact}\ n)$$

The first part of the formula corresponds to the case where we do not even enter the outer `for` loop, since $n = 0$. Then we have to show $1 = \texttt{fact}\ n$, that is provable. The other part of the formula corresponds to the case where we enter the `for` loop. Then we have to prove that the final value of variable `u`, here written u, is equal to the factorial of n. But we do not have any information over u, which makes the formula unprovable.

```
module CheckingALargeRoutine

  use import int.Fact
  use import ref.Ref

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

end
```

Figure 3.1: Computing the factorial using additions (Turing, 1949).

Indeed, we have to first set a *loop invariant* to get information regarding the final value of `u`. In our case, this invariant states that `u` contains the factorial of `r`. We write it as follows:

```
    for r = 0 to n-1 do invariant { !u = fact r }
```

Using this invariant, it is now possible to establish the postcondition when exiting function `routine`. To see it, we can use the *Split* command in the graphical user interface, which results in four sub-goals. The validity of the postcondition is the fourth one and is easily discharged by a solver. On the contrary, we cannot discharge the third goal, which states the preservation of the loop invariant we just added. Without any surprise, we have to equip the inner loop with an invariant as well, namely the following:

```
      for s = 1 to r do invariant { !u = s * fact r }
```

It is now easy for a solver to show the correctness of the program. There is no termination to prove here, as it is automatically guaranteed in the case of `for` loops. The whole file is given figure ??.

3.1.2 McCarthy's 91 Function

Let us consider now another example, namely the aforementioned McCarthy's 91 function [?]. It is the following recursive function:

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

In WhyML syntax, we define such a function using the `let rec` construct.

```
let rec f (n:int) : int =
  if n <= 100 then f (f (n + 11)) else n - 10
```

We intend to prove the termination of this function. To do so, we must equip it with a *variant*, that is a term that decreases at each recursive call for a well-founded order relation. The simplest solution is to use an integer term and the following relation:

$$x \prec y \overset{\text{def}}{=} x < y \land 0 \leq y.$$

In the case of function 91, the variant is quite easy to figure out. It is `100 - n`. In WhyML syntax, we give it as follows:

```
let rec f (n:int) : int variant { 100 - n } =
  if n <= 100 then f (f (n + 11)) else n - 10
```

As a result, we get a verification condition to show that the variant indeed decreases in each of the two recursive calls. In the first case, the sub-goal is easily discharged. Indeed we have $100 - (n + 11) \prec 100 - n$ when $n \leq 100$. On the contrary, the second sub-goal cannot be discharged, for we don't have enough information regarding the value returned by the first recursive call. Thus we have to give function `f` a specification, namely a postcondition:

```
let rec f (n: int) : int variant { 100-n }
  ensures { result = if n <= 100 then 91 else n-10 }
  =
  if n <= 100 then
    f (f (n + 11))
  else
    n - 10
```

Then we get a verification condition that is easily discharged by automated provers. It expresses both termination and correctness. This is an example where proving termination requires proving a behavioral property as well. (Another example is Floyd's algorithm for cycle detection, also know as tortoise and hare algorithm; see [?].)

Let us consider now a non-recursive version of function 91, namely the following iterative program:

```
let iter (n0: int) =
  let n = ref n0 in
  let e = ref 1 in
  while !e > 0 do
    if !n > 100 then begin
      n := !n - 10;
      e := !e - 1
    end else begin
      n := !n + 11;
      e := !e + 1
    end
    done;
    !n
```

The idea behind this program is the following. It is not necessary to maintain a stack of all recursive calls to `f`. The total number of such calls is enough. It is maintained in variable `e`, and initialized to 1. If `n` > 100 we subtract 10 from `n` and we decrement `e`, to account for one completed call to `f`. Otherwise, we add 11 to argument `n` and we increment `e` to account for a completed call to `f` and to two new calls to be performed.

Proving termination of this program is both simpler and more complex than proving the termination of the recursive function. It is simpler as we do not have to prove anything regarding the values computed by the program. But it is also more complex, as the variant is now a pair, namely

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

that decreases for the lexicographic order, each component being ordered by relation $\prec$ above. Such a lexicographic order relation is defined in Why3 standard library and we simply need to import it:

```
  use import int.Lex2
```

We can now set this variant to the `while` loop, using the following syntax:

```
    while !e > 0 do
      variant { (100 - !n + 10 * !e, !e) with lex }
      ...
```

where keyword `with` is used to specify the order relation. The verification condition is discharged automatically, without any other annotation in the program.

We could stop here. But proving the behavioral correctness of this iterative code appears to be interesting as well. Indeed, this is not obvious that this new code computes $f(n)$. The loop invariant is

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

where n_0 stands for the initial value of n. Therefore, when we exit the loop, with $e = 0$, we get $n = f(n_0)$ and we return this value. To write down such an invariant, we first introduce the logical function f.

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

This is not the recursive function we proved above, but rather the specification of function 91. Then we define $f^k(x)$. A logical function cannot be defined recursively over integers (only structural recursion is allowed) thus we cannot define $f^k(x)$ recursion over k. As a consequence, we turn to an axiomatization:

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

More generally, Why3 standard library contains a theory `int.Iter` that axiomatizes the iteration of a given function. Thus we can conveniently replace the three declarations above with the appropriate cloning of that theory:

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

function f (n: int) : int = if n <= 100 then 91 else n-10

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

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

Figure 3.2: Non-recursive version of McCarthy's 91 function.

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

The annotated code is given figure ??. As for the recursive version, the verification condition is automatically discharged.

3.2 Arrays

Up to now, we had programs using only integers. Let us consider now programs using arrays.

Why3 standard library provides arrays in module `array.Array`.

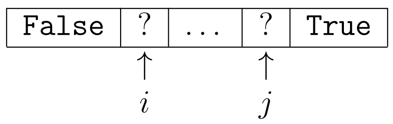
```
use import array.Array
```

This module declares a polymorphic type `array 'a`, an access operation written `a[e]`, an assignment operation written `a[e1] <- e2`, and various operations such as `create`, `length`, `append`, `sub`, or `copy`.

3.2.1 Two-Way Sort

As an example, let us consider proving the correctness of the following algorithm that sorts an array a of n Boolean values according to `False` < `True`.

```
$i, j \leftarrow 0, n-1$
while $i < j$
  if $\neg a[i]$ then $i \leftarrow i+1$
  else if $a[j]$ then $j \leftarrow j-1$
  else swap $a[i]$ and $a[j]$; $i \leftarrow i+1$; $j \leftarrow j-1$
```



We first set up the program specification. We have to state that array a is sorted when the program exits. We define a predicate `sorted` to do so:

```
predicate (<<) (x y: bool) = x = False \/ y = True
predicate sorted (a: array bool) =
  forall i1 i2: int. 0 <= i1 <= i2 < a.length -> a[i1] << a[i2]
```

We also have to specify that the final contents of array a is ane *permutation* of its initial contents (otherwise, a program setting all elements to `False` would be accepted as a valid sort). There are multiple ways to define the permutation property. We could count the number of `False` (or `True`) values and assert that it is unchanged, or, equivalently, assert that the two multi-sets of elements are the same. Another solution, well-suited for the program above, consists in defining the permutation property as the smallest equivalence relation containing transpositions (swaps to two elements). Thus we define `permut` as an inductive predicate

```
inductive permut (a1 a2: array bool) =
```

with two clauses for reflexive and transitive properties

```
| permut_refl:
    forall a: array bool. permut a a
| permut_trans:
    forall a1 a2 a3: array bool.
    permut a1 a2 -> permut a2 a3 -> permut a1 a3
```

and a third clause to indicate that `permut` contains transpositions:

```
| permut_swap:
    forall a1 a2: array bool. forall i j : int.
    length a1 = length a2 -> 0 <= i < length a1 -> 0 <= j < length a2 ->
    a1[i] = a2[j] -> a1[j] = a2[i] ->
    (forall k: int. 0 <= k < length a1 -> k <> i -> k <> j -> a1[k]=a1[k]) ->
    permut a1 a2
```

Note that we have no clause to say that `permut` is symmetric (which is, in a general case, required to define an equivalence relation). There are several reasons: first, we can prove that relation `permut` is symmetric; second, this extra property is not needed for our proof. The generic definition of `permut` for arrays with elements of any type is provided by Why3 standard library, in module `array.ArrayPermut`.

We can now write the specification. It is as follows:

```
let two_way_sort (a: array bool) : unit
  ensures { sorted a }
  ensures { permut (old a) a }
  =
  ...
```

Notation `old a` in the postcondition stands for the initial value of `a`, that is its values at the function entry. We still need a loop invariant. It decomposes into four parts. First, we have two arithmetic relations over i and j.

```
        invariant { 0 <= !i /\ !j < length a }
```

Then we state that all values left to i are all `False`.

```
        invariant { forall k: int. 0 <= k < !i -> a[k] = False }
```

Similarly, we state that all values right to j are all `True`.

```
        invariant { forall k: int. !j < k < length a -> a[k] = True }
```

Finally, we have to state the permutation property, that is that array `a` is always a permutation of its initial contents. To do so, we introduce a label `Init` at the end of the function body, with construct `'Init:`, then we use that label in the loop invariant to refer to the value of `a` at this program point, with notation `at a 'Init`.

```
        invariant { permut (at a 'Init) a }
```

This completes the loop invariant. Besides, we prove termination in a trivial way, using $j - i$ as variant. The code is summarized in figure **??**.

Exercise: Dijkstra's Dutch national flag problem. Prove the correctness of a program sorting an array with three different values, standing for the three colors of the Dutch national flag (`type color = Blue | White | Red`). The code is given on the lecture web site.

3.2.2 Boyer-Moore's Majority Algorithm

Let us consider a slightly more complex algorithm using arrays, known as majority algorithm. N people are voting for candidates. We are given their votes in an array. For instance, with three candidates `A`, `B`, and `C`, and $N = 13$, we may have the following set of votes:

A	A	A	C	C	B	B	C	C	C	B	C	C

The problem is to determine whether one candidate get absolute majority, that is more than $N/2$ votes. In the example above, candidate `C` gets 7 votes, and thus absolute majority since $7 > 13/2$.

There is a very nice algorithm to solve this problem, due to Boyer and Moore [?]. It runs in time $O(N)$, using constant extra space (three integer variables). There are two passes. In the first pass, we maintain a potential winner for absolute majority (initialized with the first vote) and a counter (initialized with 1). When we encounter the same vote,

```
use import int.Int
use import bool.Bool
use import ref.Refint
use import array.Array
use import array.ArrayPermut

predicate (<<) (x y: bool) = x = False \/ y = True

predicate sorted (a: array bool) =
  forall i1 i2: int. 0 <= i1 <= i2 < a.length -> a[i1] << a[i2]

let two_way_sort (a: array bool) : unit
  ensures { sorted a }
  ensures { permut (old a) a }
  =
  'Init:
  let i = ref 0 in
  let j = ref (length a - 1) in
  while !i < !j do
    invariant { 0 <= !i /\ !j < length a }
    invariant { forall k: int. 0 <= k < !i -> a[k] = False }
    invariant { forall k: int. !j < k < length a -> a[k] = True }
    invariant { permut (at a 'Init) a }
    variant   { !j - !i }
    if not a[!i] then
      incr i
    else if a[!j] then
      decr j
    else begin
      let tmp = a[!i] in
      a[!i] <- a[!j];
      a[!j] <- tmp;
      incr i;
      decr j
    end
  done
```

Figure 3.3: Sorting an array of Boolean values.

we increment the counter; otherwise we decrement it. Whenever the counter reaches zero, we simply pick up the next element in the array as potential winner. The key idea is that, when we are done with all array elements, only our potential winner may indeed have absolute majority. Thus we simply make a second pass to count the actual number of votes for that candidate and then decide whether it is greater than $N/2$.

We start by introducing an uninterpreted type for candidates

```
type candidate
```

and two exceptions that are going to be used to signal failure and success during the search:

```
exception Not_found
exception Found
```

The first pass of the algorithm uses two references `cand` and `k` and a `for` loop:

```
let mjrty (a: array candidate) : candidate =
  let n = length a in
  let cand = ref a[0] in
  let k = ref 0 in
  for i = 0 to n-1 do
    if !k = 0 then begin cand := a[i]; k := 1 end
    else if !cand = a[i] then incr k else decr k
  done;
```

As a slight optimization, we immediately signal a negative search if `k` is 0 at the end of the loop.

```
  if !k = 0 then raise Not_found;
```

To implement the second pass efficiently, we use the exception `Found` to signal a successful search. Thus we have the following structure:

```
  try
    ...
    raise Not_found
  with Found ->
    !cand
  end
```

If exception `Found` is raised at some point, we return `!cand`. Otherwise, we raise `Not_found`. The second pass itself is coded as follows. If the count for `cand` is already over $N/2$, we are done:

```
    if 2 * !k > n then raise Found;
```

Otherwise, we reset the counter to 0 and we count the actual number of votes for `cand` using a `for` loop. If at any moment the counter exceeds $N/2$, we raise `Found`.

```
    k := 0;
    for i = 0 to n-1 do
      if a[i] = !cand then begin
        incr k;
```

```
          if 2 * !k > n then raise Found
        end
      done;
```

To specify this program, we make use of library `array.NumOfEq`. It provides a predicate `numof`, such that `numof` a v l u is the number of values in the sub-array $a[l..u[$ that are equal to v. This way, the postcondition for a successful search is as simple as

```
    ensures  { 2 * numof a result 0 (length a) > length a }
```

Note that we avoid using a division: we use $2 \times a > b$ rather than $a > b/2$. This is equivalent and results in simpler verification conditions. In case of an unsuccessful search, we state that any candidate c gets no more than $N/2$ votes.

```
    raises   { Not_found ->
      forall c: candidate. 2 * numof a c 0 (length a) <= length a }
```

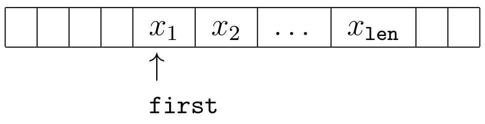
The code, its specification, and the loop invariants are given in figure ??. All verification conditions are discharged automatically.

3.2.3 Ring Buffer

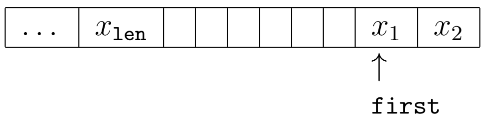
Let us illustrate now the notions of *ghost code* and *type invariant* using one of the verification challenges from the *2nd Verified Software Competition* [?]. A circular buffer is implemented within an array, using the following record type:

```
  type buffer 'a = {
    mutable first: int;
    mutable len  : int;
            data : array 'a;
  }
```

Elements are stored in array `data` starting at index `first` and there are `len` of them:



This data structure has a queue interface: if the queue is not full, a new element can be added to the right of $x_{\mathtt{len}}$; if the queue is not empty, x_1 can be popped. Elements may wrap around the array bounds, as follows:



To specify the various operations over this data structure, it is convenient to model its contents as a list, namely the list $[x_1, x_2, \ldots, x_{\mathtt{len}}]$, and to store it inside the record itself, in a *ghost* field:

```
  type buffer 'a = {
    mutable first: int;
    mutable len  : int;
```

```
exception Not_found
exception Found

type candidate

clone import array.NumOfEq with type elt = candidate

let mjrty (a: array candidate) : candidate
  requires { 1 <= length a }
  ensures  { 2 * numof a result 0 (length a) > length a }
  raises   { Not_found ->
    forall c: candidate. 2 * numof a c 0 (length a) <= length a }
= let n = length a in
  let cand = ref a[0] in
  let k = ref 0 in
  for i = 0 to n-1 do
    invariant { 0 <= !k <= numof a !cand 0 i }
    invariant { 2 * (numof a !cand 0 i - !k) <= i - !k }
    invariant { forall c:candidate. c <> !cand -> 2 * numof a c 0 i <= i - !k }
    if !k = 0 then begin
      cand := a[i];
      k := 1
    end else if !cand = a[i] then
      incr k
    else
      decr k
  done;
  if !k = 0 then raise Not_found;
  try
    if 2 * !k > n then raise Found;
    k := 0;
    for i = 0 to n-1 do
      invariant { !k = numof a !cand 0 i /\ 2 * !k <= n }
      if a[i] = !cand then begin
        incr k;
        if 2 * !k > n then raise Found
      end
    done;
    raise Not_found
  with Found ->
    !cand
  end
```

Figure 3.4: Boyer and Moore's majority algorithm.

```
          data : array 'a;
    ghost mutable sequence: list 'a;
  }
```

A ghost field, and more generally any ghost code, is to be used in specification only. It cannot interfere with regular code in the following sense:

- ghost code cannot modify regular data (but it can access it in a read-only way);
- ghost code cannot modify the control flow of regular code;
- regular code cannot access or modify ghost data.

Said otherwise, ghost data and ghost code can be removed without affecting the execution of the program. On the example above, it means that only ghost code will be able to access and assign field `sequence`.

To relate the contents of the regular fields and of the ghost field, we equip the record type `buffer 'a` with a *type invariant*. A type invariant is valid at function boundaries: it is assumed at function entry for function arguments and global variables, and must be established at function exit for the function result, if any, and for values modified by the function. A type invariant is introduced after the type definition, using the keyword `invariant`. In our case, it first says that `data` contains at least one element and that `first` is a valid index in `data`:

```
  invariant {
    let size = Array.length self.data in
    0 <= self.first < size /\ ... }
```

The identifier `self` is used to refer to the value of type `buffer 'a`. Then we add that field `len` cannot be greater than the number of elements in `data`:

```
    { ... 0 <= self.len <= size /\ ... }
```

Finally, we relate the contents of `data` with the list in field `sequence`. First, we say that `sequence` has length `len`:

```
    { ... self.len = List.length self.sequence /\ ... }
```

Finally, we way that elements in `sequence` are exactly those in array `data` at indices `first`, `first` $+1, \ldots$ To account for the wrapping around the array bound, we make two cases:

```
    { ...
    forall i: int. 0 <= i < self.len ->
      (self.first + i < size ->
         nth i self.sequence = Some self.data[self.first + i]) /\
      (0 <= self.first + i - size ->
         nth i self.sequence = Some self.data[self.first + i - size]) }
```

(Function `nth` is imported from Why3 standard library. It returns the i-nth element of a list, as `Some` x when it exists and `None` otherwise.)

Given this type invariant, we can now use field `sequence` to specify operations over the ring buffer. For instance, function `push` has a postcondition stating that `sequence` has been extended to the right with a new element:

```
val push (b: buffer 'a) (x: 'a) : unit
  ...
  ensures  { b.sequence = (old b.sequence) ++ Cons x Nil }
```

The full specification, including other operations, is given in figure ??.

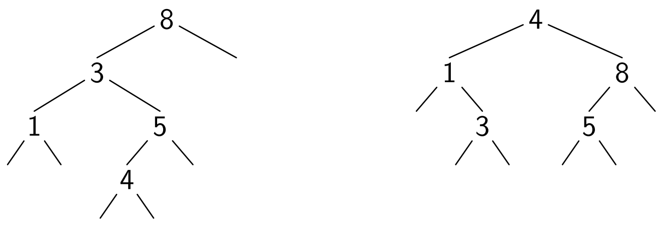
Exercises. Implement operations `create`, `clear`, `push`, `head`, and `pop` over the ring buffer and prove them correct. The template file is given on the lecture web site.

3.3 Algebraic Data Types

A key idea of Hoare logic [?] is that all symbols from the logic can be used in programs. In particular, we already used type `int` from the logic in many programs. Similarly, we can use algebraic data types in programs. In Why3 standard library, we find the following algebraic data types:

```
type bool = True | False                          (in bool.Bool)
type option 'a = None | Some 'a                   (in option.Option)
type list 'a = Nil | Cons 'a (list 'a)            (in list.List)
```

The user is free to define other algebraic data types and to use them in programs. Let us illustrate this on an example. This is the famous *same fringe problem*: given two binary trees holding values at nodes, we want to check whether they have the same elements when traversed in inorder. The problem is subtle, as the two trees may have different shapes. For instance, the two trees



both have elements $1, 3, 4, 5, 8$ when traversed in inorder. This problem has practical application: it provides an order relation over binary search trees (BST) that is shape-insensitive. This is useful to build other BSTs whose elements are BSTs (to implement sets of sets).

Let us write a WhyML program to solve the same fringe problem and let us prove its correctness. We introduce some uninterpreted type `elt` for the elements (the nature of the elements is irrelevant here) and a type `tree` for binary search trees.

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

For the purpose of the specification, we introduce a function `elements` returning the result of an inorder traversal of a tree.

```
function elements (t: tree) : list elt = match t with
| Empty -> Nil
```

```
type buffer 'a = {
  mutable first: int;
  mutable len  : int;
          data : array 'a;
  ghost mutable sequence: list 'a;
}
invariant {
  let size = Array.length self.data in
  0 <= self.first <  size /\
  0 <= self.len   <= size /\
  self.len = List.length self.sequence /\
  forall i: int. 0 <= i < self.len ->
    (self.first + i < size ->
       nth i self.sequence = Some self.data[self.first + i]) /\
    (0 <= self.first + i - size ->
       nth i self.sequence = Some self.data[self.first + i - size])
}

val create (n: int) (dummy: 'a) : buffer 'a
  requires { n > 0 }
  ensures  { length result.data = n }
  ensures  { result.sequence = Nil }

val clear (b: buffer 'a) : unit
  writes  { b.len, b.sequence }
  ensures { b.len = 0 }
  ensures { b.sequence = Nil }

val push (b: buffer 'a) (x: 'a) : unit
  requires { b.len < length b.data }
  writes   { b.data.elts, b.len, b.sequence }
  ensures  { b.len = (old b.len) + 1 }
  ensures  { b.sequence = (old b.sequence) ++ Cons x Nil }

val head (b: buffer 'a) : 'a
  requires { b.len > 0 }
  ensures  { match b.sequence with Nil -> false | Cons x _ -> result = x end }

val pop (b: buffer 'a) : 'a
  requires { b.len > 0 }
  writes   { b.first, b.len, b.sequence }
  ensures  { b.len = (old b.len) - 1 }
  ensures  { match old b.sequence with
             | Nil -> false
             | Cons x l -> result = x /\ b.sequence = l end }
```

Figure 3.5: Specification of a ring buffer.

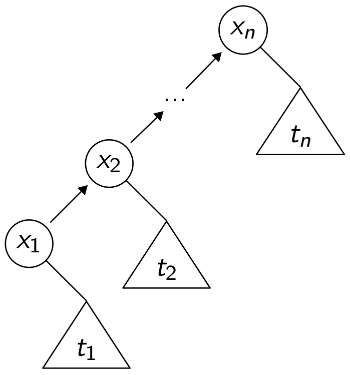
```
  | Node l x r -> elements l ++ Cons x (elements r)
  end
```

The function we seek to define and prove is thus as follows:

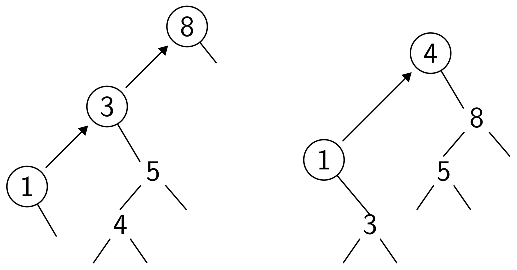
```
let same_fringe (t1 t2: tree) : bool
  ensures { result=True <-> elements t1 = elements t2 }
  =
  ...
```

Of course, we could simply use function `elements` to solve the problem. However, this would be a quite bad solution: first, list concatenations are costly (they could reach a quadratic total cost) second, it is unnecessary to build the two lists if they quickly differ (at least in the context of a strict language as WhyML).

Thus we consider a better solution, consisting in considering the leftmost branch of each tree as a list, from bottom up. Each element on this list is paired with its right sub-tree. It is thus a list of pairs (x_i, t_i), as follows:



On the example above, the two lists are the following:



The comparison then proceeds as follows. We first compare the first element of each list. If they differ, we are done. Otherwise, we replace the first element in each list by the leftmost branch of its right sub-tree, and then we recursively perform the comparison. If at some point a list is exhausted while the other is not, we answer negatively. Finally, we successfully terminate if both lists exhaust at the same time.

To write this algorithm in WhyML, we first introduce a data type `enum` for the lists of pairs (in the following, we call such a list an enumerator).

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

Then we define a recursive function `enum_elements` to get the inorder traversal of an enumerator, analogous to function `elements`, for specification purposes only.

```
function enum_elements (e: enum) : list elt = match e with
  | Done -> Nil
  | Next x r e -> Cons x (elements r ++ enum_elements e)
end
```

Now we proceed to the code itself. We start with a function `enum` that builds an enumerator of type `enum` from a tree `t` of type `tree`. We proceed recursively along the leftmost branch of `t`. To do so, we pass a second argument `e` of type `enum` to accumulate.

```
let rec enum (t: tree) (e: enum) : enum
  ensures { enum_elements result = elements t ++ enum_elements e }
  =
  match t with
  | Empty -> e
  | Node l x r -> enum l (Next x r e)
  end
```

The specification states that the traversal of the result is the concatenation of the traversal of `t` and that of `e`. Then we write the core of the algorithm as a function `eq_enum` comparing two enumerators `e1` and `e2`. We proceed recursively, until we exhaust at least one list.

```
let rec eq_enum (e1 e2: enum) : bool
  ensures { result=True <-> enum_elements e1 = enum_elements e2 }
  =
  match e1, e2 with
  | Done, Done ->
      True
  | Next x1 r1 e1, Next x2 r2 e2 ->
      x1 = x2 && eq_enum (enum r1 e1) (enum r2 e2)
  | _ ->
      False
  end
```

The specification states that we decide equality of the traversals of `e1` and `e2`. Finally, the solution to the same fringe problem is a mere call to `eq_enum` on the enumerators for the two trees.

```
let same_fringe (t1 t2: tree) : bool
  ensures { result=True <-> elements t1 = elements t2 }
  =
  eq_enum (enum t1 Done) (enum t2 Done)
```

The solution is summarized in figure **??**.

Exercise: inorder traversal. Using the same type for binary trees, we consider the following recursive function that fills an array a with the elements of a tree t, following an inorder traversal.

```
fill $t$ $start$ $\stackrel{\text{def}}{=}$
  if $t$ = Empty then return $start$
```

```
use import int.Int
use import list.List
use import list.Append

type elt
type tree = Empty | Node tree elt tree

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

type enum = Done | Next elt tree enum

function enum_elements (e: enum) : list elt = match e with
  | Done -> Nil
  | Next x r e -> Cons x (elements r ++ enum_elements e)
end

let rec enum (t: tree) (e: enum) : enum
  ensures { enum_elements result = elements t ++ enum_elements e }
  =
  match t with
  | Empty -> e
  | Node l x r -> enum l (Next x r e)
  end

let rec eq_enum (e1 e2: enum) : bool
  ensures { result=True <-> enum_elements e1 = enum_elements e2 }
  =
  match e1, e2 with
  | Done, Done ->
      True
  | Next x1 r1 e1, Next x2 r2 e2 ->
      x1 = x2 && eq_enum (enum r1 e1) (enum r2 e2)
  | _ ->
      False
  end

let same_fringe (t1 t2: tree) : bool
  ensures { result=True <-> elements t1 = elements t2 }
  =
  eq_enum (enum t1 Done) (enum t2 Done)
```

Figure 3.6: Solving the same fringe problem.

```
if $t = \texttt{Node}\ l\ x\ r$ then
    $res \leftarrow \texttt{fill}\ l\ start$
    if $res = \texttt{length}\ a$ then return $res$
    $a[res] \leftarrow x$
    $\texttt{fill}\ r\ (res + 1)$
```

The WhyML code for this program, as well as the properties to prove, are available from the lecture web site.

3.4 Other Data Structures

If we look into Why3 standard library module `array.Array` we find the following declarations for the type of arrays:

```
type array 'a model { length: int; mutable elts: map int 'a }
```

Such a declaration has two meanings. In *programs*, it is analogous to the declaration of an abstract type, that would be

```
type array 'a
```

In the *logic*, however, it is equivalent to the declaration of an immutable record type, that would be

```
type array 'a = { length: int; elts: map int 'a }
```

Said otherwise, an array is *modeled* in the logic as a record with two fields, the length of type `int`, and the contents of type `map int 'a` (a purely applicative map).

The type being abstract in programs, we cannot implement operations over type `array 'a`. But we can declare them. For instance, the access operation is declared as follows:

```
val ([]) (a: array 'a) (i: int) : 'a reads {a}
  requires { 0 <= i < length a }
  ensures  { result = a[i] }
```

This function takes `a` and `i` as arguments, together with a precondition to ensure an array access within bounds. It returns a value of type `'a`, and the postcondition states that the returned value is the value contained in the purely applicative map, that is `a.elts`. As we see, logical annotations, such as the pre- and postcondition here, have access to the record fields `length` and `elts`. It is the purpose of a model type to be available in the logic (and only in the logic). The annotation `reads {a}` indicates that the function accesses the contents of array `a` (but does not modify it). The assignment operation is declared in a similar way:

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

The main difference is the annotation `writes {a}`, which indicates that the contents of `a` is modified by a call to this function. This is allowed since the field `elts` was declared to be `mutable`.

Modeling Hash Tables. As an example, let us do a similar model of hash tables[2] More precisely, let us consider polymorphic hash tables with the following interface (in OCaml syntax):

```
type t 'a 'b
val create: int -> t 'a 'b
val clear: t 'a 'b -> unit
val add:  t 'a 'b -> 'a -> 'b -> unit
exception Not_found
val find: t 'a 'b -> 'a -> 'b
```

Function `create` returns a fresh, empty table. Function `clear` empties a given table. Function `add` inserts a new entry. Finally, function `find` returns the value mapped to a key, if any, and raises exception `Not_found` otherwise.

As for arrays above, we first declare a type `t 'a 'b` for the hash tables, modeling the contents using a purely applicative map.

```
type t 'a 'b model { mutable contents: map 'a (option 'b) }
```

We choose to map at most one value for each key. Thus the fields `contents` maps each key k of type `'a` to a value of type `option 'b`: `None` stands for the absence of value for key k, and `Some` v stands for an entry $k \mapsto v$ in the table. To ease forthcoming specifications, we set up notation $h[k]$ for the entry for key k in table h.

```
function ([]) (h: t 'a 'b) (k: 'a) : option 'b = get h.contents k
```

Function `create` takes an integer n as argument (the initial size of the bucket array) and returns a fresh, empty table.

```
val create (n:int) : t 'a 'b
  requires { 0 < n }
  ensures  { forall k: 'a. result[k] = None }
```

Function `clear` empties a given table. Thus it has the same postcondition as function `create`. But it also has specifies that table `h` is modified (`writes {h}`).

```
val clear (h: t 'a 'b) : unit
  writes  { h }
  ensures { forall k: 'a. h[k] = None }
```

Function `add` also has a `writes {h}` annotation:

```
val add (h: t 'a 'b) (k: 'a) (v: 'b) : unit
  writes  { h }
  ensures { h[k] = Some v /\ forall k': 'a. k' <> k -> h[k'] = (old h)[k'] }
```

Its postcondition states that key `k` is now mapped to value `v` in the new contents of `h` and that any other key `k'` is still mapped to the same value as before, that is `(old h)[k']`. Indeed, annotation `writes h` denotes a possible modification of *all* the contents of table `h` and thus it is here the role of the postcondition to state what is modified and what is not. Finally, we declare function `find`.

[2]Hash tables are provided in Why3 standard library since version 0.80.

```
use import option.Option
use import int.Int
use import map.Map

type t 'a 'b model { mutable contents: map 'a (option 'b) }

function ([]) (h: t 'a 'b) (k: 'a) : option 'b = get h.contents k

val create (n:int) : t 'a 'b
  requires { 0 < n }
  ensures  { forall k: 'a. result[k] = None }

val clear (h: t 'a 'b) : unit
  writes  { h }
  ensures { forall k: 'a. h[k] = None }

val add (h: t 'a 'b) (k: 'a) (v: 'b) : unit
  writes  { h }
  ensures { h[k] = Some v /\ forall k': 'a. k' <> k -> h[k'] = (old h)[k'] }

exception Not_found

val find (h: t 'a 'b) (k: 'a) : 'b
  reads { h }
  ensures { h[k] = Some result }
  raises { Not_found -> h[k] = None }
```

Figure 3.7: Modeling hash tables.

```
exception Not_found

val find (h: t 'a 'b) (k: 'a) : 'b
  reads   { h }
  ensures { h[k] = Some result }
  raises  { Not_found -> h[k] = None }
```

The keyword raises introduces a postcondition corresponding to the case where an exception is raised (Not_found in that case), while ensures declares a postcondition corresponding to the case where a value is returned. Figure ?? summarizes our model of hash tables.

It is worth mentioning that there is currently no possibility to ensure that a given WhyML implementation of hash tables conforms to this model.

Chapter 4

Using Why3 as an Intermediate Language

Using Why3 to verify programs written in a mainstream programming language (such as C or Java) requires an adequate modeling of its semantics in the logic of Why3. In this chapter, we briefly explain how to do so on two examples.

4.1 Machine Arithmetic

So far we have use arithmetic from Why3 standard library, that is mathematical integers of unbounded precision. Let us assume we need to model machine arithmetic instead (say, signed 32-bit integers), either to show the absence of arithmetic overflow in a program, or to reason about possible overflows. The main difficulty is that we do not want to loose the arithmetic capabilities of SMT solvers (which only know about mathematical arithmetic). One way to do is to introduce a new, uninterpreted type `int32` for machine integers

```
type int32
```

together with a function giving the corresponding value of type `int`:

```
function toint int32 : int
```

The idea is then to use only type `int` in program annotations, that is to apply function `toint` in a systematic way around sub-expressions of type `int32`. The range of 32-bit integers is defined with two constants

```
constant min_int: int = -2147483648
constant max_int: int =  2147483647
```

and an axiom states that any value of type `int32` is bounded by these constants:

```
axiom int32_domain:
  forall x: int32. min_int <= toint x <= max_int
```

If our purpose is to build a model to prove the absence of arithmetic overflow, we simply need to function to build a value of type `int32` from a value of type `int` with a suitable precondition:

```
val ofint (x:int) : int32
  requires { min_int <= x <= max_int }
  ensures { toint result = x }
```

Then we can translate any program manipulating machine integers into a WhyML program where each arithmetic operation uses function `ofint`. For instance, an expression such as $x + y$ is translated into `ofint` (`toint` x) (`toint` y). Equivalently, we can also declare (or define) a function performing this computation, that is

```
val (+) (x: int32) (y: int32) : int32
  requires { min_int <= toint x + toint y <= max_int }
  ensures  { toint result = toint x + toint y }
```

The lecture web site contains an application of this model to the verification of a program searching for a value in a sorted array using binary search. This way we find the famous bug related to the computation of the mean value of `l` and `u` using `(l+u)/2` [?]. Then we can fix the bug, for instance using `l+(u-l)/2` instead, and we can show that this new program is safe from any arithmetic overflow.

4.2 Memory Models

A key idea of Hoare Logic, implemented in Why3, is that the system can statically identify the various memory locations. This is *absence of aliasing*. In particular, memory locations are not first-class values. Thus to handle programs with mutable data structures such as linked lists or mutable trees, or programs with explicit pointers, one has to *model the memory heap*.

A trivial example would be that of C programs with explicit pointers of type `int*`. A simple way to model such programs would be to introduce a type for pointers and a global reference holding the contents of the memory:

```
type pointer
val memory: ref (map pointer int)
```

Then a C expression such as `*p` is translated into `!memory[p]`. If pointer arithmetic is allowed, then either the type `pointer` is defined as an alias for type `int`, or operations over type `pointer` (such as shifting, comparisons, etc.) are introduced. This is somehow a rather crude model, where the user quickly get stuck into a specification nightmare.

There are more subtle models, such as the *component-as-array* model [?, ?]. Say we want to model the following C type for simply linked lists:

```
struct List {
  int          head;
  struct List *next;
};
```

The idea behind the component-as-array model is to have *several* maps to model the various structure fields. In this case, we have (at least) two global maps for fields `head` and `next`:

```
  type pointer
  val head: ref (map pointer int)
  val next: ref (map pointer pointer)
```

An obvious benefit is that any modification to reference `head` does not have any impact on the contents of reference `next`. The benefit is even greater when there are many C fields (being they of the same type or not, of from the same structure or not). The component-as-array model may be even further refined using a region-analysis of pointers [**?**]. On the contrary, a drawback is that low-level tricks such as pointer casts or pointer arithmetic are not easily allowed anymore.