15-414: Bug Catching: Automated Program Verification

Lecture Notes on Logical Contracts

Frank Pfenning

Carnegie Mellon University
Lecture 2
February 4, 2021

1 Introduction

How do we know our programs are correct? Well, usually we don't. But there are a number of time-honored techniques to at least catch *some* bugs, most prominently *testing*. While indispensible, testing can almost never prove that our code is actually correct, and with distributed, parallel, concurrent, and probabilistic programs on the rise, bugs are more likely than ever to survive even rigorous testing.

It seems better to avoid bugs in the first place. Fortunately, there are a number of techniques around to help with that. One is code review, again indispensible for larger code bases, but it suffers from human limitations. By far the most pervasively used is *static typing* which catches many simple bugs before a program is ever executed. It is meaningful to the programmer and machine alike, predictable, and compositional in the sense that we can check small fragments of code independently from each other. But static typing with conventional type systems has its limitations. Consider, for example, the collection of functions of type $\mathsf{int} \to \mathsf{int}$. The question then is how we express and check *deeper* properties of programs.

Since most deep properties about programs are undecidable, there is no silver bullet that prevents all bugs and can be used as easily and routinely as a static type system. Perhaps most conservative are systems of *type refinement* that extend expressiveness somewhat while trying to retain the predictability and compositionality of conventional types. Systems of *dependent types* go further in that they allow the formal expression of proofs in addition to that of programs. This puts more of a burden on the programmer, who now either has to write correctness proofs (as in Agda) or write programs to search for correctness proofs (as in Coq).[1]

[1] Since these are lecture notes we take the liberty of providing links instead of full citations.

Dependent types in one form or other are a prevalent technique in functional programming. For imperative programs, there is a line of research on *program logics*, that is, logics specifically designed to reason about the correctness of programs. A program logic must allow us to express the programs we are reasoning about as well as the specification they are supposed to satisfy. A proof in a program logic then is evidence that the program matches its specification. Depending on the programming language and the complexity of the specification this may require some programmer interaction, or we may be able to discharge the proof obligations automatically.

In this course we will focus on the program logics and methods to automatically prove that programs satisfy their specifications. In order for this to be feasible for non-trivial programs and specifications we need to able to break the problem down into smaller units than whole programs, that is, achieve a degree of compositionality. This is the role of *logical contracts*. Logical contracts specify under which circumstances a function may be called (the *precondition*) and what it guarantees for its result (the *post-condition*). The intuition is that the correctness proof for the rest of the program may only rely on the pre- and post-condition of a function and not its definition. Additionally, contracts allow us to express why functions or loops terminate (the *variant*), and which properties are preserved in a loop or data structure (the *invariant*). All of these are critical in making verification feasible. The other critical component is the theorem prover in the background that eventually verifies that the program obeys its contracts. Techniques for automated theorem proving will be the focus of the latter part of this course.

The content of the course is divided into three parts.

Part I: Reasoning about Programs: From 122 and 150 to 414. We gain an intuitive understanding how the *executable contracts* from the course on Imperative Programming (122) and the informal mathematical contracts from Functional Programming (150) can be translated into logical contracts and then proved. It also serves as an introduction to the WhyML language and Why3 verification toolchain we'll use throughout the semester.

Part II: From Informal to Formal Reasoning. We develop a program logic (specifically *dynamic logic*) in which reasoning about programs can be carried out formally and therefore mechanized. We carefully justify the logical axioms and rules with respect to the operational meaning of the programs to show that they are *sound*. Therefore, formal proofs do indeed guarantee properties of the programs that are executed.

Part III: Mechanizing Reasoning. We explore decision procedures for tractable fragments of program logics in the form of propositional satisfiability checkers (SAT), including those with built-in theories (SAT modulo theories, or SMT), and model checking.

In this first technical lecture we will look back at *executable contracts* from the C0 language, how they are recast into logical form, and how we can use them to reason about

the correctness of programs. We will exemplify how the same techniques for specification and reasoning can be applied to functional programs. We cast the programs into WhyML and formally verify our first small programs.

Learning goals. After this lecture, you should be able to:

- Write logical contracts (that is, pre- and post-conditions for functions, loop invariants, assertions, and variants) for small imperative programs over integers;
- Extract verification conditions for small imperative programs over integers with logical contracts;
- Translate simple imperative programs to functional analogues, including the contracts;
- Relate the verification of loop invariants to verification of corresponding tail-recursive functions;
- Write logical contracts for small functional programs over simple immutable data types such as lists;
- Browse the Why3 Standard Library;
- Use the why3 command line interface to examine verification conditions and prove them with one of the standard automated provers.

2 A Mystery Function

We start with a small puzzle. Consider the following function, written in C0 (although equally meaningful in C). Try it out or simulate it by hand to see the values it produces and come to a conjecture about which function it represents. Also give some thought to how you might go about proving your conjecture. If you didn't attend lecture, please give it a serious try before moving on to the next page.

```
1  int f(int n)
2  {
3    int i = 0;
4    int a = 0;
5    int b = 1;
6    while (i < n)
7      {
8        b = a + b;
9        a = b - a;
10       i = i+1;
11     }
12   return a;
13 }
```

It probably didn't take you long to conjecture that $f(n)$ computes the nth Fibonacci number. As is often the case, the key insight into the correctness of the function will be in the *loop invariant*. Before we get there, we follow the methodology of the *executable contracts* from 15-122 *Principles of Imperative Computation* and then translate them into logical form.

2.1 Preconditions

A *precondition* for a function imposes a requirement upon any caller, namely that the precondition should be true. For executable contracts, this means that the precondition is a pure function and *evaluates* to true. For logical contracts it will mean that the condition *is* true. In this case, the precondition is the $n \geq 0$. We use the C0 syntax `//@requires` to express the precondition and elide the remainder of the function.

```
1 int f(int n)
2 //@requires n >= 0;
3 {...}
```

2.2 Postconditions

A *postcondition* for a function expresses what the caller may assume about its result (and, later, about its effects). It is important that the caller cannot "look inside" the function to reason about it behavior, but it must rely only on the postcondition. This is an important principle allowing us to *localize* the reasoning in individual functions. In essence, functions represent an abstraction boundary that greatly aids the feasibility of program verification.

In this example, the postcondition states that f computes the Fibonacci function, but how can we actually say this? In C0, there is no recourse except to define a simple (if highly inefficient) function which we view as the specification. This function must be *pure*, that is, it may not have any externally observable effect. This is important because a program with executable contracts should behave the same whether contracts are actually checked or not.

```
1  int fib(int n)
2  //@requires n >= 0;
3  {
4    if (n == 0) return 0;
5    else if (n == 1) return 1;
6    else /* n >= 2 */ return fib(n-2) + fib(n-1);
7  }
8
9  int f(int n)
10 //@requires n >= 0;
11 //@ensures \result == fib(n);
12 {...}
```

2.3 Loop Invariants

The verification task requiring the most creativity is determining a suitable loop invariant. Here are the critical properties of loop invariants, when we consider them merely executable contracts to be checked dynamically, as the program runs.

Location: The loop invariant is always checked *just before the guard condition*. This is to ensure we can rely on the loop invariant in case the loop guard is false. The invariant is always checked with respect to the current values of all variables appearing in them.

Initialization: Just before the loop is entered the first time, the loop invariant must be true.

Preservation: When we jump back to the beginning of the loop, the loop invariant must be true.

From a logical perspective, this means

Initialization: We can prove the loop invariant from all the assumptions we have when reaching the beginning of the loop.

Preservation: We may assume the loop invariant and the guard condition are both true, before descending into the body of the loop. By the time we jump back, we then have to prove the loop invariant for the new values of all the variables at the end of the loop. Also important is that we lose all the assumptions we had accumulated before we entered the loop.

Exit: Upon exit of the loop, we may assume the loop invariant and the negation if the loop guard.

Usually, one or more loop invariants pertain to the value of a variable that is incremented, decremented, or otherwise drives the iteration of the loop. Here, this is i. It is easy to see that i ranges from 0 to n.

```
1  int f(int n)
2  //@requires n >= 0;
3  //@ensures \result == fib(n);
4  {
5    int i = 0;
6    int a = 0;
7    int b = 1;
8    while (i < n)
9      //@loop_invariant 0 <= i && i <= n;
10     //@loop_invariant ...
11     {
12       b = a + b;
13       a = b - a;
14       i = i+1;
15     }
16   ...
17 }
```

Even though it may be a bit counterintuitive at first, we need to specify $i \leq n$ and not just $i < n$ because before the loop guard is checked after the last iteration we have $i = n$.

The other missing invariant in this case is concerned with the value we are computing. Finding such an invariant may require considerable ingenuity—here a little experimentation with pencil and paper tells us that $a = \mathsf{fib}(i)$ and $b = \mathsf{fib}(i+1)$.

```
1  int f(int n)
2  //@requires n >= 0;
3  //@ensures \result == fib(n);
4  {
5    int i = 0;
6    int a = 0;
7    int b = 1;
8    while (i < n)
9      //@loop_invariant 0 <= i && i <= n;
10     //@loop_invariant a == fib(i) && b == fib(i+1);
11     {
12       b = a + b;
13       a = b - a;
14       i = i+1;
15     }
16   ...
17 }
```

Logically, we conclude the following facts upon loop exit:

- $0 \leq i$ and $i \leq n$ (loop invariant at line 9)
- $a = \mathsf{fib}(i)$ and $b = \mathsf{fib}(i+1)$ (loop invariant at line 10)
- $i \not< n$ (loop guard at line 8 is false)

2.4 Assertions

From the information we have when exiting the loop, we can see that $i = n$ must be true (because $i \leq n$ and $i \not< n$). In the absence of formal proof, we may not be fully confident of that fact, so we can insert an assertion that is checked dynamically and would raise an exceception if $i \neq n$.

Logically, too, it is often helpful to state what we know after the loop in the form of an *assertion*. An assertion must be known to be true, and is can therefore be assumed subsequently. While theoretically redundant (adding an assumption that is already entailed by our knowledge doesn't change what we can prove), it may be an important lemma for the theorem prover and can make the difference between success and failure.

```
1  int f(int n)
2  //@requires n >= 0;
3  //@ensures \result == fib(n);
4  {
5    int i = 0;
6    int a = 0;
7    int b = 1;
```

```
8   while (i < n)
9     //@loop_invariant 0 <= i && i <= n;
10    //@loop_invariant a == fib(i) && b == fib(i+1);
11    {
12      b = a + b;
13      a = b - a;
14      i = i+1;
15    }
16  //@assert i == n;
17  return a;
18 }
```

2.5 Postcondition Revisited

As we see in the code, we return a. We have already concluded that $i = n$ and we also know that $a = \mathsf{fib}(i)$, so $a = \mathsf{fib}(n)$. Since we return a, we substitute it for \result in the postcondition and verify that, indeed, $a = \mathsf{fib}(n)$

The postcondition is always placed in the preamble of a function. That is so that a caller can see what it may assume about the value that is returned. But it is actually checked at every return statement inside the function.

2.6 Testing with Executable Contracts

In C0, we can exploit the contracts adding some testing code (in a function main), compiling it with the `-d` flag (for *dynamic checking*), and executing it. For example:

```
1 int main() {
2   for (int i = 0; i < 10; i++) {
3     printf("f(%d) = %d\n", i, f(i));
4   }
5   return 0;
6 }
```

and then

```
% cc0 -d -x mystery.c0
f(0) = 0
f(1) = 1
f(2) = 1
f(3) = 2
f(4) = 3
f(5) = 5
f(6) = 8
f(7) = 13
f(8) = 21
f(9) = 34
0
%
```

where 0 is the value return by the main function. The fact that no exception was raised means that our executable contracts always evaluated to true. To check that, we may purposely introduce a bug into our code and run the program and observe the exception.

Unfortunately, when we move to logical contracts we lose the option of *testing the contracts* because they can no longer be executed. Presumably, it would be possible to call the theorem prover dynamically on the instantiated logical conditions in case we are unable to prove the correctness of our code and are trying to determine why.

You can find the C0 program for the this example in the file mystery.c0.

3 From Executable to Logical Contracts

In the preceding section we have been writing *executable contracts* but explained their meaning in terms of *logical contracts*. In this section we translate from C0 to WhyML, the language used in the Why3 tool chain and the main language in the remainder of the course. Rather than formally introducing WhyML, we present the original C0 and the WhyML code side by side and then explain some of the differences.

```
1
2  int fib(int n)                                  function fib (n:int) : int
3  //@requires n >= 0;
4  {
5    if (n == 0) return 0;                         axiom fib0 : fib 0 = 0
6    else if (n == 1) return 1;                    axiom fib1 : fib 1 = 1
7    else /* n >= 2 */                             axiom fibn : forall n:int. n >= 0
8    return fib(n-2) + fib(n-1);                     -> fib n + fib (n+1) = fib (n+2)
9  }
10
11 int f(int n)                                    let f(n:int) : int =
12 //@requires n >= 0;                             requires { n >= 0 }
13 //@ensures \result == fib(n);                   ensures { result = fib n }
14 {
15   int i = 0;                                    let ref i = 0 in
16   int a = 0;                                    let ref a = 0 in
17   int b = 1;                                    let ref b = 1 in
18   while (i < n)                                 while i < n do
19     //@loop_invariant 0 <= i && i <= n;           invariant { 0 <= i /\ i <= n }
20     //@loop_invariant a == fib(i)                 invariant { a = fib i
21     //@             && b == fib(i+1);                       /\ b = fib (i+1) }
22     {                                             variant { n - i }
23       b = a + b;                                  b <- a + b ;
24       a = b - a;                                  a <- b - a ;
25       i = i+1;                                    i <- i + 1
26     }                                           done ;
27   //@assert i == n;                             assert { i = n } ;
28   return a;                                     a
29 }
```

At the top, we see that the pure function fib is replaced by an *undefined* function fib. Instead of giving its definition we state some axioms about its properties so the

theorem prover can reason about it. Among other things, it means that fib cannot be used computationally, only in contracts. In the third axiom we see a universal quantifier `'forall'` and we also see implication `'->'`.

The translation of the requires and ensures clauses in the definition of f is straightforward, except that we use `'='` for equality instead of `'=='`.

A definition such as `int i = 0;` is translated to a binding `let ref i = 0 in`. This pattern makes i assignable in its scope, using the notation `i <- ...`. Formally, i will have type `ref int`, but uses of the dereference operator `'!'` remains implicit to make the code more visually appealing and closer to the imperative counterpart.

The while loop and invariants translate in a straightforward manner, but we note that the short-circuiting conjunction `'&&'` is replaced by the logical conjunction `'/\'`.

New on the side of WhyML is a *variant* declaration. It contains a quantity that serves as a termination measure for the loop (or, as we will see later, recursive function). If we give an integer quantity, it should be nonnegative and strictly decreasing during each loop iteration.

You can find the live code for this with some additional comments in fib.mlw.

We can now see if Why3 can prove the correctness of this function using the `alt-ergo` prover.

```
% why3 prove -P alt-ergo fib.mlw
fib.mlw Mystery f'vc: Valid (0.01s, 17 steps)
%
```

Yes it can! This is our first example of a verified program, all the way from straightforward C0 code to an analogous verified function in WhyML.

3.1 Loop Invariant Holds Initially

Just to test our understanding, we now walk through the code and track what we may assume, and what we have to prove. First we note that the axioms about fib will be available to the prover from the ambient environment. Further, we name each of the assumptions so we can refer to them when needed. We start by reasoning about how the loop invariant is satisfied initially.

```
1  let f(n:int) : int =
2  requires { n >= 0 }              (* assume: n >= 0    [H0] *)
3  ensures { result = fib n }       (* skip for now...       *)
4  let ref i = 0 in                 (* assume: i = 0     [H1] *)
5  let ref a = 0 in                 (* assume: a = 0     [H2] *)
6  let ref b = 1 in                 (* assume: b = 1     [H3] *)
7  while i < n do                   (* prove: [H0-3] -> 0 <= i /\ i <= n *)
8    invariant { 0 <= i /\ i <= n }
9    invariant { a = fib i          (* prove: [H0-3] -> a = fib i *)
10             /\ b = fib (i+1) }   (*                  /\ b = fib (i+1) *)
11   variant { n - i }
12   b <- a + b ;
13   a <- b - a ;
```

```
14   i <- i + 1
15 done ;
16 assert { i = n } ;
17 a
```

We stop when we reach the loop. We now have to prove the two loop invariants from the accumulated assumptions [H0–H3]. The fact that we have to prove this for all n, i, a, b remains implicit. Of course, we have assumptions such as $i = 0$ which means we really only need to prove it for this value and we can reason by substitution. What we end up having to show then is

$$n \geq 0 \to (0 \leq 0 \wedge 0 \leq n)$$
$$n \geq 0 \to (0 = \mathsf{fib}\ 0 \wedge 1 = \mathsf{fib}\ (0+1))$$

which is easy, remembering the axioms about fib. The parentheses in these two verification conditions are optional, since the convention is that conjunction '$\wedge$' binds more tightly than implication '$\to$'.

3.2 Loop Invariants are Preserved

Next we have to show that the loop invariant is preserved. This means we have to traverse the loop, knowing only the loop invariants themselves. We lose assumptions [H1–H3] because variables i, a, and b are changed inside the loop. However, we retain the assumption about n because n is not changed. Actually, it couldn't: it has type int rather then ref int and is therefore not assignable.

When we assign to a variable in the loop we need to create a fresh generation of the variable to be used in the verification condition. This is because the variable will hold different values at different points in the program and this must be reflected in the logic.

```
1  let f(n:int) : int =
2  requires { n >= 0 }              (* assume: n >= 0    [H0] *)
3  ensures { result = fib n }
4  let ref i = 0 in
5  let ref a = 0 in
6  let ref b = 1 in
7  while i < n do                     (* assume: i < n                [H4] *)
8    invariant { 0 <= i /\ i <= n } (* assume: 0 <= i /\ i <= n    [H5] *)
9    invariant { a = fib i          (* assume: a = fib i                 *)
10             /\ b = fib (i+1) }   (*       /\ b = fib (i+1)       [H6] *)
11   variant { n - i }              (* skip for now *)
12   b <- a + b ;                   (* assume: b' = a + b           [H7] *)
13   a <- b - a ;                   (* assume: a' = b' - a          [H8] *)
14   i <- i + 1                     (* assume: i' = i + 1           [H9] *)
15 done ;                           (* prove: [H0,H4-H9] -> ...          *)
16 assert { i = n } ;
17 a
```

Note that we had to be careful in [H8] to refer to the correct generation of b, namely the one that holds the value of the variable b at that point in the program. That's b'. When we reach the end of the loop (done) we have to prove the loop invariants, but *for*

the current generation of all variables appearing in them. In this loop, we have to substitute i', a', and b' for i, a, and b, respectively. In other words, we have to prove

$$[\text{H0,H4–H9}] \to 0 \leq i' \wedge i' \leq n$$
$$[\text{H0,H4–H9}] \to a' = \mathsf{fib}\ i' \wedge b' = \mathsf{fib}\ (i'+1)$$

The first one follows immediately since $i' = i + 1$ and $0 \leq i$ [H5] and $i < n$ [H4]. The second one follows similarly in a couple of steps, using the axiom instance $\mathsf{fib}\ i + \mathsf{fib}\ (i+1) = \mathsf{fib}\ (i+2)$.

3.3 Loop Variant

It is easy to prove from the invariants that $0 \leq n - i$ and that $n - i' < n - i$ because $i' = i + 1$. That is, the loop terminates because the integer $n - i$ is always nonnegative and strictly decreases during each iteration.

3.4 Loop Invariants and Negated Loop Guard Imply Postcondition

To complete the verification we need to traverse the remainder of the function and show that whatever we know when we exit from the loop (and any further assumptions we might make) imply the postcondition.

```
1  let f(n:int) : int =
2  requires { n >= 0 }   (* assume: n >= 0                          [H0] *)
3  ensures { result = fib n }
4  let ref i = 0 in
5  let ref a = 0 in
6  let ref b = 1 in
7  while i < n do
8    invariant { 0 <= i /\ i <= n }
9    invariant { a = fib i /\ b = fib (i+1) }
10   variant { n - i }
11   b <- a + b ;
12   a <- b - a ;
13   i <- i + 1
14   done ;              (* assume: not (i < n)                  [H10] *)
15                       (* assume: 0 <= i /\ i <= n             [H11] *)
16                       (* assume: a = fib i /\ b = fib (i+1)   [H12] *)
17 assert { i = n } ;    (* prove: [H0,H10-H12] -> i = n *)
18                       (* assume: i = n                        [H13] *)
19 a                     (* prove: [H0,H10-H12,H13] -> a = fib n      *)
```

Fortunately, both the intermediate assertion and the postcondition (where we substitute the return value a for result) are easy to prove.

3.5 Examining the Verification Condition

The verification condition that is passed to the back-end provers is constructed along the lines we have shown in this section. It accounts for a number of features of WhyML

we did not introduce yet, so the algorithm is relatively complicated. For small examples or portions of programs it may be useful to examine the verification condition. In the command line interface to Why3 this can be accomplished with command `why3 prove <file>.mlw` without providing a prover. Below is the (relevant portion) of the output from this command on the file developed in this section.

```
% why3 prove mystery.mlw
  ...
  goal f'vc :
    forall n:int.
     n >= 0 ->
     ((0 <= 0 /\ 0 <= n) && 0 = fib 0 /\ 1 = fib (0 + 1)) /\
     (forall b:int, a:int, i:int.
       (0 <= i /\ i <= n) /\ a = fib i /\ b = fib (i + 1) ->
       (if i < n
        then forall b1:int.
              b1 = (a + b) ->
              (forall a1:int.
                a1 = (b1 - a) ->
                (forall i1:int.
                  i1 = (i + 1) ->
                  (0 <= (n - i) /\ (n - i1) < (n - i)) /\
                  (0 <= i1 /\ i1 <= n) && a1 = fib i1 /\ b1 = fib (i1 + 1)))
        else i = n && a = fib n))
```

We see that quantifiers are explicit, substitutions have often (but not always) been carried out already, and that the verification condition is nested rather than being presented as a collection of independent propositions. In the integrated development environment (IDE) for why3, this goal can be split into subgoals to be proved independently. We will show this in a future lecture.

4 From Imperative to Functional Programs

Of course, verification is not confined to imperative programs. In fact, one might suspect functional programs are easier to verify since the they are closer to logic (or at least those without effects).

The manner in which we constructed the verification condition is already some form of "purification" since the verification condition itself is effect-free. To start with, we translate the loop to a recursive function, abstracted over all variables occurring in the loop (which are: a, b, i, n). At the end of the loop we make a tail-recursive call. Note that the function header specifies `let rec` because the function is recursive.

```
1                                       let rec fib_rec a b i n =
2   while i < n do                      if i < n
3     b <- a + b ;                      then let b' = a + b in
4     a <- b - a ;                           let a' = b' - a in
5     i <- i + 1                             let i' = i + 1 in
6   done                                     fib_rec a' b' i' n
7                                       else ...
```

Unlike the variable in the imperative code on the left, the variables on the right are *immutable*, which is one of the benefits of this translation.

But what happens to the loop invariants? Consider this question before you move on to the next page . . .

Answer: the loop invariants become a *precondition* to the function implementing the loop. This means the precondition must be satisfied in the initial call (from the top-level function we haven't written yet), but it must also be satisfied when the tail call is made, which is exactly what is needed to show that the loop invariant is preserved. The variant also carries over unchanged, because the reason the recursion terminates is the same as the reason for the termination of the loop. We have also filled in some (redundant) types in the function header.

```
let rec fib_rec (a : int) (b : int) (i : int) (n : int) : int =
  requires { 0 <= i /\ i <= n }
  requires { a = fib i /\ b = fib (i+1) }
  variant { n - i }
  ensures { ... }
  if i < n
  then let b' = a + b in
       let a' = b' - a in
       let i' = i + 1 in
       fib_rec a' b' i' n
  else ...
```

In general, the ensures clause in such a recursive function will summarize what we know from the loop invariant and the failure of the guard condition. In this example we just return a and state that it should be fib n; no other variables inside the loop will be relevant or needed by the caller. The assertion before the return just becomes a corresponding assertion.

```
let rec fib_rec (a : int) (b : int) (i : int) (n : int) : int =
  requires { 0 <= i /\ i <= n }
  requires { a = fib i /\ b = fib (i+1) }
  variant { n - i }
  ensures { result = fib n }
  if i < n
  then let b' = a + b in
       let a' = b' - a in
       let i' = i + 1 in
       fib_rec a' b' i' n
  else assert { i = n } ;
       a
```

It remains to translate the part of the function leading up to the loop. This "top level" function fib_top retains the pre- and post-conditions from the original function f, and also translates assignable variable to variable bindings.

```
let fib_top (n : int) : int =
  requires { n >= 0 }
  ensures { result = fib n }
  let i = 0 in
  let a = 0 in
  let b = 1 in
  fib_rec a b i n
```

Note that, as indicated before, the call to fib_rec has to establish its precondition,

which is the same as establishing that the loop invariant holds initially in the imperative code.

You can find the complete (fully verified) live code for this example at fib.mlw. We can clean up the code a little bit to avoid unnecessary intermediate bindings.

```
let rec fib_rec (a : int) (b : int) (i : int) (n : int) : int =
  requires { 0 <= i /\ i <= n }
  requires { a = fib i /\ b = fib (i+1) }
  variant { n - i }
  ensures { result = fib n }
  if i < n
  then fib_rec b (a + b) (i + 1) n
  else a

let fib_top (n : int) : int =
  requires { n >= 0 }
  ensures { result = fib n }
  fib_rec 0 1 0 n
```

5 Verifying Properties of Data Structures

For the moment, we'll stay in the functional world, although in WhyML we use data structures similarly also in imperative programs. There is a clever implementation of queues in a functional language using two stacks (usually directly represented by lists), sometimes called *functional queues*. If the queue is used in a single-threaded way (that is, we don't dequeue from a queue in the same state more than once) then amortized analysis shows that both enqueue and dequeue operations have constant amortized cost.

The basic algorithmic idea is as follows. The queue is represented by two lists, the front and the back. Initially both are empty. When we enqueue, we add elements to the back, and when we dequeue, we take them from the front. If the front happens to be empty when a dequeue request comes in we *reverse* the back to become the new front. For example, after enqueuing 1, 2, 3 in that order, the front is still empty and the back will be the list $[3, 2, 1]$. If we now dequeue, the front will become $[1, 2, 3]$ (the reverse of the back) and we remove 1 from the front, leaving it $[2, 3]$ with the back empty.

To represent the queue, we see a couple of new constructs. One of them is *polymorphism* because we would like queues to be generic in the types of the elements. In particular, the type `queue 'a` will be a queue with elements of type `'a` (usually pronounced *alpha*). We also need lists, which we can find in the Why3 Standard Library. Among many other things, we find:

```
  type list 'a = Nil | Cons 'a (list 'a)

  let rec function (++) (l1 l2: list 'a) : list 'a =
    match l1 with
    | Nil        -> l2
    | Cons x1 r1 -> Cons x1 (r1 ++ l2)

```

```
let rec function reverse (l: list 'a) : list 'a =
  match l with
  | Nil       -> Nil
  | Cons x r -> reverse r ++ Cons x Nil
  end
```

We see that lists have constructors `Nil` and `Cons` and that we discriminate between lists using the expression `match ... with ... end`. We use a *record* of two elements, the front and the back, as our representation of queues.

```
type queue 'a = { front : list 'a ; back : list 'a }
```

We would like to define the following functions

```
empty () : queue 'a
enq (x : 'a) (q : queue 'a) : queue 'a
deq (q : queue a) : option ('a , queue 'a)
```

A queue might be empty, so deq returns and optional pair consisting of the first element and the remainder of the queue. For this we need the option library:

```
type option 'a = None | Some 'a
```

Before we write code, we should decide on the specifications. The key idea is that we use a single list to represent the queue, with the first element in the queue at the front of the list. In other words, we use another data structure (a list) in the specification of the behavior of the queue. Of course, we do not want to use such a list as an *implementation* because the cost of an enqueue operation would be linear in the size of the queue (rather than have amortized constant cost). Therefore we define the function sequence that represents the state of a queue in the proper sequence. Recall that because we add the elements to the back, to represent the proper state of the queue we have to *reverse* the back.

```
function sequence (q : queue 'a) : list 'a = q.front ++ reverse q.back
```

Now let's write the postconditions for all of the functions. They have no precondition since any state of the queue is valid. The ones for empty and enq are fairly straightforward. For the enqueue operation we add the new element to the end of the sequence.

```
let empty () : queue 'a =
ensures { sequence result = Nil }
...

let enq (x : 'a) (q : queue 'a) : queue 'a =
ensures { sequence result = sequence q ++ Cons x Nil }
...

let deq (q : queue 'a) : option ('a , queue 'a) =
```

For the dequeue operation, we have to return Nil if the queue is empty and Some (x, r) if x is at the front of the queue and r is the remainder. Writing this out logically, we use an existential quantifier.

```
  let deq (q : queue 'a) : option ('a , queue 'a) =
  ensures { (result = None /\ sequence q = Nil)
            \/ ( exists x:'a. exists r:queue 'a. result = Some(x,r)
                 /\ sequence q = Cons x (sequence r)) }
```

At this point the code itself is not too difficult, just for the case of enqueue we nest two matches because if the front is empty the queue is empty only if the back is also empty. We show the code here; it can also be found in queue.mlw.

```
module Queue

  use list.List
  use list.Append
  use list.Reverse
  use option.Option

  type queue 'a = { front : list 'a ; back : list 'a }

  function sequence (q : queue 'a) : list 'a =
    q.front ++ reverse q.back

  let empty () =
  ensures { sequence result = Nil }
  { front = Nil ; back = Nil }

  let enq (x : 'a) (q : queue 'a) : queue 'a =
  ensures { sequence result = sequence q ++ Cons x Nil }
  { front = q.front ; back = Cons x q.back }

  let deq (q : queue 'a) : option ('a , queue 'a) =
  ensures { (result = None /\ sequence q = Nil)
            \/ ( exists x:'a. exists r:queue 'a. result = Some(x,r)
                   /\ sequence q = Cons x (sequence r)) }
  match q.front with
  | Nil -> match reverse q.back with
           | Nil -> None
           | Cons x b -> Some (x, { front = b ; back = Nil })
           end
  | Cons x f -> Some (x, { front = f ; back = q.back })
  end

end
```

While the code and specifications seem correct, we cannot be 100% confident that the prover will be able to verify it. In particular, the reasoning depends on the lemmas in the libraries pertaining to the properties of append ('++') and reverse. Fortunately, in this case it succeeds. Just for variety, we tried it with the CVC4 prover.

```
% why3 prove -P cvc4 queue.mlw
queue.mlw Queue empty'vc: Valid (0.04s, 6867 steps)
queue.mlw Queue enq'vc: Valid (0.05s, 7687 steps)
queue.mlw Queue deq'vc: Valid (0.08s, 13374 steps)
%
```

6 Some Examples of Verified Systems

There are many examples of impressive verification efforts, but software verification is certainly not a routine industrial practice. We give here an entirely subjective list of efforts we found particularly interesting. They use a variety of languages and theorem proving environments.

- Flyspeck, a formal proof of Kepler's Conjecture which was first stated in 1611 and final proven by Thomas Hales with the help of a computer program in 1998. The proof was questioned and subsequently formally verified in a large community effort led by Hales. It was completed in 2014.
- CompCert, a formally verified compiler for a large subset of ANSI C.
- CakeML, a formally verified compiler for a functional language in the ML family.
- CertiKOS, a certified Kit Operating System.
- Everest, a project towards a verified implementation of the HTTPS protocol ecosystem.
- CLI System Stack, an early successful effort to verify a system stack from a high-level language through a compiler all the way to hardware verified at the gate level, described in a collection of papers.