Lectures Notes on Verification of Functional Programs

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

7 March, 2022

During lecture, we briefly touched upon the different traditions and techniques of program verification in *functional* and *imperative* programming. We now delve a bit further into the verification of functional programs as a stepping stone to the verification of imperative programs.

The contents of Section 2 are inspired by notes by Michael Erdmann and Frank Pfenning.

1 Reasoning about imperative programs

As we will see throughout the semester, it is often the case that we specify the *result* of an imperative computation by appealing to a functional definition that we wish to prove equivalent to the outcome of the imperative procedure. For instance, consider the following imperative implementation of the factorial function:

```
method factImp(n:int) returns (r:int)
{
    r := 1;
    var m := n;
    while (m > 0) {
        r := r*m;
        m := m-1;
    }
}
```

While we have not yet developed the techniques that will allow us to prove that this method computes the factorial of its (non-negative) integer argument, we now take the first step in any formal proof of program correctness: its *specification*. Since the method aims to implement the factorial function, it is reasonable to *require* its argument be a natural number (i.e. `n≥0`). To specify what the value computed by the method, stored in `r`, actually is the factorial of `n`, we must devise a specification-level construct that is somehow equal to $n!$. As hinted above, we do this by defining a *mathematical function*:

```
function fact(n:nat) : nat
{
    if n=0 then 1 else n*fact(n-1)
}
```

Note the use of the term *mathematical*. While the definition above is a piece of code, we must convince ourselves (and, eventually, Dafny) that the code actually is equivalent (in some sense) to a proper mathematical function (i.e., an assignment of exactly one natural number to each natural number), that can be used in a logical specification. For a recursive piece of code such as `fact` above, this means that we must be sure that `fact` is both *pure* (i.e. does not make use of nor mutate any state) and *terminates* for all its possible arguments.

Informally, it is fairly easy to see that this is indeed the case: `fact` is pure since it does not rely on any state. As for termination, `fact` takes arguments of type **`nat`** (i.e. integer numbers greater than or equal to 0). If the argument `n` is `0`, `fact` terminates immediately. Otherwise, the recursive call structure will act on successive *smaller* values (`n-1`, `n-2`, ...) until eventually reaching `0`, at which point we know the function terminates. A bit more formally, there is a *well-founded ordering* on the recursive calls to `fact(n)`, which is the mathematical term for a binary relation with a minimal element. Since such an ordering exists, the number of calls must be finite and so the function terminates. In this course we will not address the issue of proving termination in a fully formal sense, although we will often have to convince our verification tools that certain procedures do, in fact, terminate.

At this point the reader may wonder what would happen if our specification logic could refer to functions that are not mathematical in the sense above. If such were indeed the case, our logical statements would be meaningless since they would not denote properties of values. For instance, it would be logically meaningless to even state `∀ n. fact(n)≥1`, regardless of its truth value, if `fact(n)` were not in fact *equal to* some (in this case natural) number. A related issue would arise if the definition of `fact` were not a *pure* function (i.e., one that does not depend on state), since there would be no unique number that is equal to `fact(n)`, for any given `n`. In essence, we require that the functions we use in our specifications be *referentially transparent*: any function call `f(x)` can be replaced with its corresponding value (and vice-versa) without changing the meaning of the specification.

Now that we have convinced ourselves that our definition of `fact` above can indeed be used in specifications, we tentatively specify our imperative implementation:

```
method factImp(n:int) returns (r:int)
ensures r=fact(n)
{...}
```

This specification is not quite right: `factImp` is defined for arguments of type **`int`**, whereas `fact` is defined for natural numbers. The correct specification is:

```
method factImp(n:int) returns (r:int)
requires n≥0
ensures r=fact(n)
{...}
```

Unfortunately, while our specification is indeed correct, our tools cannot prove that `factImp` satisfies the specification without further assistance. We will flesh out these ideas further in the following weeks.

However, what do we mean when we say that `factImp` satisfies the specification? In this case, satisfying the specification means that for all possible `n:` **`int`** such that `n≥0`, it is the case that the call to `factImp(n)` returns a value that is equal to `fact(n)`. But what do we know about `fact(n)`? We have argued that it must be equivalent to a mathematical function, but how do we know that it is actually *the* factorial function? Proving that our pure functions are correct is the goal of the remainder of this document.

2 Reasoning about functional programs

As showcased above, Dafny includes both pure functions and imperative code. In this section we use the term Dafny to refer exclusively to its functional fragment, which is *pure*.

When proving the correctness of a functional program we must be able to refer to some underlying definition of the programming language and its operational behavior (or *operational semantics*). While a precise and complete definition of the functional core of Dafny is out of the scope of this course, we will make use of the following assumptions and notation:

- We will generally not distinguish between a mathematical object (such as an integer or a natural number) and its representation as an object in Dafny.
- When reasoning about functional programs, we will ignore any machine-based limits and so integers and reals will be mathematical integers and real numbers and not machine integers or reals.
- We write e for arbitrary expressions and v for values, a special kind of expression.
- We write $e \hookrightarrow v$ to mean that e *evaluates* to value v.
- We write $e \overset{1}{\Longrightarrow} e'$ to mean that expression e *reduces* to expression e' in a single step.
- We write $e \overset{k}{\Longrightarrow} e'$ to mean that expression e *reduces* to expression e' in k steps.
- We write $e \Longrightarrow e'$ to mean that expression e *reduces* to expression e' in 0 or more steps.

Our notion of *step* in the operational semantics is kept abstract and does not necessarily coincide with the actual operations performed by an implementation. Since our main concern is proving correctness rather than proving complexity bounds, the concrete number of steps is mostly irrelevant and we will frequently use the notation $e \Longrightarrow e'$ for reduction. Evaluation and reduction are related in the sense that if $e \hookrightarrow v$ then either e is equal to v already or $e \overset{1}{\Longrightarrow} e_1 \overset{1}{\Longrightarrow} \ldots \overset{1}{\Longrightarrow} v$ (and vice-versa). Values are special forms of expression in that they evaluate to themselves "in 0 steps". For any value v, there is no expression e such that $v \overset{1}{\Longrightarrow} e$. We assume that function application reduces in a single-step to its function body with the formal parameters substituted for the function arguments, as in math. We will assume that all primitive operations such as addition, subtraction, conditional branching, etc., all reduce in a single-step when acting on values (i.e. $3 + 2 \overset{1}{\Longrightarrow} 5$ and **if true then** e_1 **else** $e_2 \overset{1}{\Longrightarrow} e_1$). For example, we have that:

$$\texttt{fact(0)} \overset{1}{\Longrightarrow} \texttt{(}\textbf{if}\ \texttt{0=0}\ \textbf{then}\ \texttt{1}\ \textbf{else}\ \texttt{0*fact(0-1))} \overset{1}{\Longrightarrow} \texttt{1}$$

2.1 Equivalence of Expressions and Referential Transparency

In a pure functional language, we say that two expressions e and e' of the same (non-function) type are equivalent, written $e \equiv e'$, (a) whenever the evaluation of e produces the same value as the evaluation of e'; or, alternatively, (b) if the evaluation of e and of e' both loop forever. Note that this notion of equivalence is potentially distinct from mathematical equality, written $e = e'$.

As mentioned above, the principle of *referential transparency* means that in any functional program we may replace any expression with any other equivalent expression without affecting the value of the program. This is a powerful principle since it enables reasoning about functional programs. Roughly speaking, this is substitution of "equals for equals", a notion so familiar from mathematics that one does it all the time without any worries. While this may sound obvious, this principle is extremely useful in practice, and it can lend support to program optimization or simplification steps that help develop better programs.

For *pure* functional programs such as those we will be reasoning about, because evaluation causes no side-effects, if we evaluate an expression twice, we obtain the same result. Moreover, the relative order in which one evaluates (non-overlapping) sub-expressions of a program makes no difference to the value of the program, so one may in principle use parallel evaluation strategies to speed up code while being sure that this does not affect the final value.

Our setting is further restricted: not only are we considering pure functional programs but we will also restrict our attention to *terminating* programs (this is also known as *total functional programming*). This means that two expressions are equivalent if-and-only-if they evaluate to the same value.

2.2 Proving a Simple Function Correct

The correctness property of a function corresponds to a lemma we can use in proving the correctness of later functions. As a first, simple example, consider the function:

```
function square(n:int) : int { n*n }
```

It is trivial to conclude that the function above implements the squaring function $f(n) = n^2$. Throughout the course we will take such "obvious" properties for granted, but it is instructive to see what a formal proof of the correctness of the `square` looks like:

Lemma 2.1. *For every integer value n, `square`$(n) \hookrightarrow n^2$.*

Proof. We prove this statement directly, relying on the operational semantics of the language:

$$\begin{array}{lll} & \texttt{square}(n) & \\ \overset{1}{\Longrightarrow} & n \texttt{*} n & \text{by evaluation of function application} \\ \overset{1}{\Longrightarrow} & n \times n & \text{by evaluation of } * \\ = & n^2 & \text{by math} \end{array}$$

$\square$

Note how each reasoning step above is justified by some basic principle or assumption. Again, we will not generally prove such trivial functions in this way. Dafny's proof automation mechanisms effectively take care of most of this administrative reasoning, including in more sophisticated settings. However, this kind of reasoning is in fact happening "somewhere" under the hood, and we should be aware of it.

3 Proofs by Induction

The simplest form of induction, which you may recall from math or calculus classes, is induction over the naturals $0, 1, \ldots$. This is known by a few names: *mathematical induction*, *standard induction*, *simple induction* or *weak induction*. The idea is simple but powerful: assume we want to prove a property for every natural number n. We first prove that the property holds for the first natural number, 0 (the base case). Then, we assume the property holds for an arbitrary natural n and establish it for the successor of n (i.e. $n + 1$ – the inductive step). The two steps together guarantee the property holds for all natural numbers – why? We know it holds for 0, and because of the inductive step, it must hold for its successor 1. By the same account, if it holds for 1 then it must hold for its successor 2, and so on, for any given natural number.

There are many small variations of this general scheme which can be easily justified and which we will also call mathematical induction. For instance, induction might start at 1 if we want to prove a property of all positive integers rather than the naturals. As a side note, the attentive reader may have noticed that the kind reasoning that justifies how induction "works" is similar to the one used to intuitively justify how recursive functions behave. This is more than coincidence: proofs by induction and recursive functions are related in a strong and precise sense. In fact, they can be seem as *the same thing*. This is known as the *Curry-Howard correspondence*, but we will leave it at that.

As you may have guessed, proofs by induction are the main workhorse when reasoning formally about (recursive) functional programs. To see how it all works, consider the following recursive function:

```
function power(n:int, m:int) : int {
  if m=0 then 1 else n*power(n,m-1)
}
```

The function above (inefficiently) computes n^m for an arbitrary integer n and natural number m by multiplying n with itself m times. We take $0^0 = 1$. In fact, we should actually write the function as:

```
function power(n:int, m:nat) : int {
  if m=0 then 1 else n*power(n,m-1)
}
```

The function is only well-defined if its second argument is a natural number and not an arbitrary integer. If m could be a negative integer, evaluation would not terminate (and so our code would not represent a mathematical function).

Lets now prove that $\texttt{power}(n,m)$ does indeed calculate n^m, using induction over the naturals.

Theorem 3.1. $\texttt{power}(n,m) \hookrightarrow n^m$ *for all natural numbers m and all integers n.*

Proof. By mathematical induction on m.

Base Case: $m = 0$

We need to show that $\texttt{power}(n,0) \hookrightarrow n^0$, for all n.

$$\begin{array}{lll} & \texttt{power}(n,0) & \\ \overset{2}{\Longrightarrow} & 1 & \text{by evaluation of function application and the conditional} \end{array}$$

Induction Step: Assume that, for some $m \geq 0$, and all integers n, $\texttt{power}(n,m) \hookrightarrow n^m$. Prove that the property holds for $m + 1$.

We need to show that $\texttt{power}(n,m+1) \hookrightarrow n^{m+1}$.

$$\begin{array}{lll} & \texttt{power}(n,m+1) & \\ \overset{2}{\Longrightarrow} & n\texttt{*power}(n,m+1-1) & \text{by evaluation of function application and the conditional} \\ \overset{2}{\Longrightarrow} & n\texttt{*power}(n,m) & \text{by math} \\ \Longrightarrow & n\texttt{*}n^m & \text{by the induction hypothesis} \\ \overset{1}{\Longrightarrow} & n \times n^m & \text{by evaluation of } * \\ = & n^{m+1} & \text{by math} \end{array}$$

which completes the proof.

□

The level of detail of a proof typically depends on the context in which the proof is carried out (and the mathematical sophistication of the expected reader). In this course, you should feel free to omit the number of computation steps and combine obvious reasoning steps. Appeals to the induction hypothesis and other non-obvious steps (like appealing to lemmas) should be justified as in the example above.

You should convince yourself that the reasoning above does indeed prove the power function is correct: we show that `power(`n`,0)` is correct, for all n (base case); we show that, for some m and all n, if we assume `power(`n`,`m`)` is correct (the inductive hypothesis) then `power(`n`,`$m+$`1)` is correct (inductive case). Then, we can carry out this reasoning for an arbitrary m: the function is correct for 0; by the inductive step, it is correct for 1; by the inductive step, it is correct for 2, and so on up-to m. This is often the intuitive reasoning one performs when writing recursive functions.

4 Generalizing the Induction Hypothesis

From the example above it may seem that proofs by induction are always straightforward. While this is the case for many proofs, there is the occasional function whose correctness proof turns out to be more difficult. This is often because the statement we are trying to prove is *too weak*, and what we have to prove is something more general than the final result we are aiming for. This is referred to as *generalizing the induction hypothesis*. A common symptom of the need to generalize the induction hypothesis (i.h.) is when an appeal to the i.h. results in a statement that is overly specialized and does not actually allow us to proceed with our reasoning.

While it can be shown that there is no general recipe (i.e., an algorithm) for this that will always work, we can isolate a common case. Consider the following alternative implementation of the power function, through the use of a helper function:

```
function pow(n:int, m:nat,r:int) : int {
  if m=0 then r else pow(n,m-1,r*n)
}
function powerAlt(n:int,m:nat) : int {
  pow(n,m,1)
}
```

The `pow` function uses its third argument `r` as an *accumulator*, that is then used as the final result for `pow` when m is 0. You should convince yourself that the definition of `powerAlt` above and the definition of `power` from before are indeed equivalent.

As an aside, this is a commonly used trick in functional programming. In the earlier `power` function, if we want to calculate `power(2,4)` we must calculate `power(2,3)` and multiply it by 2, which in turn requires calculating `power(2,2)` and multiplying that intermediate result by 2, and so on. Compilers for functional languages must make arrangements for storing the intermediate computations, so that the final result can be correctly "unrolled" back. On the other hand, the `pow` function above does not have this kind of recursive structure – it features so-called *tail recursion*. For instance, to calculate `pow(2,4,1)` we need only calculate `pow(2,3,2)`, and so `pow(2,2,4)`, and so `pow(2,1,8)` and finally `pow(2,0,16)`, which is `16`. No "unrolling" back of intermediate results is needed. For functions of this shape, compilers can optimize away any special arrangements for storing intermediate computations, which can make the execution of tail-recursive functions such as `pow` much more efficient when compared to the execution of functions like `power`. This is called *tail-call optimization*.

We would now like to prove that `powerAlt` $(n,m) \hookrightarrow n^m$, for all integers n and naturals m. This requires proving something about `pow`, which takes one more argument. Intuitively, we would like to prove `pow` $(n,m,1) \hookrightarrow n^m$. Unfortunately, proving this statement directly by induction will fail. Lets see where it breaks down.

Suppose we have assumed the induction hypothesis:

$$\texttt{pow}\,(n,m,1) \hookrightarrow n^m$$

and try to prove

$$\texttt{pow}\,(n,m+1,1) \hookrightarrow n^{m+1}$$

We proceed as we did in the inductive case for `power`:

$$\begin{array}{ll} & \texttt{pow}\,(n,m+1,1) \\ \overset{2}{\Longrightarrow} & \texttt{pow}\,(n,m+1-1,1*n) \\ \overset{1}{\Longrightarrow} & \texttt{pow}\,(n,m,1*n) \\ \overset{1}{\Longrightarrow} & \texttt{pow}\,(n,m,n) \end{array}$$

but now we are stuck. We cannot apply the induction hypothesis since the last argument in the call to `pow` is not `1`. The solution is to generalize the property to allow any `r: int` in such a way that the desired result follows easily. The generalization is as follows:

Lemma 4.1. `pow` $(n,m,r) \hookrightarrow r \times n^m$*, for all* r*, all* $n \geq 0$*, and all* m *natural.*

Proof. By mathematical induction on m.

Base Case: $m = 0$

We need to show that `pow` $(n,0,r) \hookrightarrow r \times n^0$, for all n.

$$\begin{array}{lll} & \texttt{pow}\,(n,0,r) & \\ \overset{2}{\Longrightarrow} & r & \text{by evaluation of function application and the conditional} \\ = & r \times n^0 & \text{by math} \end{array}$$

Induction Step: Assume that, for some $m \geq 0$, and all integers n and r', `pow` $(n,m,r') \hookrightarrow r' \times n^m$. Prove that the property holds for $m+1$.

We need to show that `pow` $(n,m+1,r) \hookrightarrow r \times n^{m+1}$.

$$\begin{array}{lll} & \texttt{pow}\,(n,m+1,r) & \\ \overset{2}{\Longrightarrow} & \texttt{pow}\,(n,m+1-1,r*n) & \text{by evaluation of function application and the conditional} \\ \overset{2}{\Longrightarrow} & \texttt{pow}\,(n,m,r\times n) & \text{by math and evaluation of } * \\ \Longrightarrow & (r\times n)\times n^m & \text{by the induction hypothesis, with } r' = r \times n \\ = & r \times n^{m+1} & \text{by math} \end{array}$$

which completes the proof.

□

The correctness of `powerAlt` can now be established directly.

Theorem 4.2. `powerAlt` $(n,m) \hookrightarrow n^m$*, for all integers* n *and natural numbers* m*.*

Proof.

$$\begin{array}{lll} & \texttt{powerAlt}(n,m) & \\ \overset{1}{\Longrightarrow} & \texttt{pow}(n,m,1) & \text{by evaluation rule for application} \\ \Longrightarrow & 1 \times n^m & \text{by Lemma 4.1} \\ = & n^m & \text{by math} \end{array}$$

□

5 Beyond Natural Numbers

Up to this point we have seen how to reason precisely about functions that manipulate and produce (natural) numbers through the proof method of mathematical induction. However, as you may recall from any course you may have participated in that featured functional programming, most interesting or realistic recursive functions manipulate more sophisticated data structures such as lists and trees – themselves often defined as recursive (or inductive) data types.

Consider for instance the following piece of code which defines a (generic) inductive data type `List` and a recursive function that adds all the elements of a list of integers:

```
  datatype List<T> = Nil | Cons(T,List<T>)

  function add(l : List<int>) : int {
    match l
    case Nil ⇒ 0
    case Cons(x,xs) ⇒ x + add(xs)
}
```

The **datatype** declaration specifies a type `List` with a type parameter `T`. The type has exactly two constructors, `Nil` and `Cons`. The `Nil` constructor codifies the empty list. The `Cons` constructor represents a (singly-linked) cell containing an element of type `T` and (a reference to) the rest of the list. The type definition above is, in all practical terms, equivalent to the OCaml definition of a (polymorphic) list:

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

The `add` function takes an argument `l` of type `List<int>` (note the instantiated type) and produces a value of type **int** which is the sum of all elements of `l`. It does so by *pattern matching* on the structure of the given list. Since `l` is of type `List<int>` it can only be constructed using a `Nil` or a `Cons`. In the former case, it means we are trying to add the elements of the *empty* list and so the result must be 0. In the latter case, we are dealing with a list with at least one element and a (possibly empty) tail. Pattern matching allows us to *name* the components of the Cons cell so that we may use them directly, thus we write **case** `Cons(x,xs) ⇒x + add(xs)` to denote that when the list is a `Cons`-cell carrying an integer value `x` and rest of the list `xs`, the `add` function is defined by adding `x` to the result of `add`ing the rest of the list.

While it may be slightly harder to wrap your head around why, informally, the function `add` above is correct, intuitively we can see that it must produce the correct result: for empty lists it will produce 0, for one element lists it will add that element to 0, and so on. Moreover, since each call of `add` deconstructs one element of the list, and there can only be finitely many such elements, we can also convince ourselves that the function always terminates.

5.1 Structural Induction

We will now make the correctness argument precise (i.e., formal) using a generalization of the technique from the previous sections: *structural induction*. The idea is that we can reason using induction on the possible *structure* of so-called inductive data types such as lists and trees in the sense given above. Instead of just postulating the induction principle over lists, we will attempt to construct it by analogy with the natural numbers, which can themselves be seen as an inductive data type:

```
  datatype Nat = Zero | Succ(Nat)
```

The type of natural numbers is, in a precise sense, equivalent to the inductive type `Nat` above: 0 (`Zero`) is a natural number; and, the successor of a natural number n (`Succ(n)`) is itself a natural number. For instance we can define addition over `Nat` as:

```
function addNat(n:Nat,m:Nat) : Nat
{
    match n
    case Zero ⇒ m
    case Succ(n') ⇒ add(n',Succ(m))
}
```

When the first argument of `addNat` is `Zero`, the result is the second argument (i.e., $0+m = m$). Otherwise, the first argument must be of the form `Succ(n')`, for some `n'`, and the result of adding the successor of n' to m is the same as adding n' to the successor of m (i.e., $(n'+1)+m = n' + (m+1)$).

Now let us revisit the induction principle for natural numbers: to show that a property holds for all natural numbers we first prove that it holds for zero (i.e., prove for `Zero`) and then, we assume the property holds for an arbitrary natural n' and prove the property holds for its sucessor (i.e., assume property for `n'`, prove for `Succ(n')`). By analogy, this means that to show that a property holds for all values of type `Nat`, we must show the property holds for `Zero` and then, assuming the property holds for some `n' : Nat`, show that the property holds for `Succ(n')`. The reason why this is a valid proof principle is the same as that for mathematical induction: values of type `Nat` can only be either `Zero` or finitely generated by repeated application of `Succ` to a `Nat`. Thus, for any given `Nat`, if its `Zero`, the "base case" shows us the property holds; if its the successor of `Zero` (`Succ(Zero)`, the inductive step together with the base case shows us the property holds; if its `Succ(Succ(Zero))`, the base case and two instances of the inductive step discharge the proof, and so on.

We can now reconstruct the induction principle for lists defined as:

```
  datatype List<T> = Nil | Cons(T,List<T>)
```

by following a similar recipe. We have as many cases as there are constructors in the data type: a base (or non-inductive case) for `Nil` and an inductive case for `Cons`. Spelling it out, to show a property holds for all values of type `List` we show the property holds for `Nil` and then, assuming it holds for an arbitrary list `l'`, we show that it holds for `Cons(n,l')`, for an arbitrary `n`. Lets show the `add` function over lists of integers is correct in the following sense:

Theorem 5.1. *For all* `l : List<int>`, `add(l)` $\hookrightarrow l_0 + \cdots + l_n$, *where* l_i *denotes the* i*-th element of the list, and the length of* `l` *given by* n.

Proof. By structural induction on `l`.

Case: `l` is `Nil`

We need to show that `add(Nil)` $\hookrightarrow 0$

$$\begin{array}{lll} & \texttt{add(Nil)} & \\ \overset{2}{\Longrightarrow} & 0 & \text{by evaluation of function application and pattern matching} \end{array}$$

Case: `l` is `Cons(n,l')`, for arbitrary `n` and `l'`

Assume that, `add(l')` $\hookrightarrow l_0 + \dots l_m$ where the length of `l'` is m (i.e, the induction hypothesis).

We need to show that `add(Cons(n,l'))` $\hookrightarrow n + l_0 + \dots l_m$.

$$\begin{array}{lll} & \texttt{add(Cons(n,l'))} & \\ \overset{2}{\Longrightarrow} & \texttt{n+add(l')} & \text{by evaluation of function application and pattern matching} \\ \Longrightarrow & \texttt{n} + (l_0 + \dots l_m) & \text{by the induction hypothesis} \\ = & n + l_0 + \dots l_m & \text{by math} \end{array}$$

which completes the proof.

□

6 Taking stock

We have now seen some techniques that allow us to formally prove the correctness of functional programs, the most important of which is the use of induction to establish correctness properties of recursive functions. From a conceptual point of view, this is important for our verification programme because the specification of imperative code will often appeal to functional code, the meaning of which may not be entirely obvious. Crucially, within a prover such as Dafny, it is simply not possible to prove that our specification is "correct" in an absolute sense – there is no Dafny analogue of Theorems 3.1 or 5.1, which must be established through external means. However, it is possible to establish in Dafny a weaker result, related to Theorem 4.2 – that `powerAlt` and `power` are equivalent (see Exercise 3).

The more pragmatically minded reader may be wondering what is the point of all this when the main goals of the course are studying the verification of *imperative* programs. It turns out that the kind of reasoning we made in the previous section on generalizing the inductive hypothesis is *exactly* the same kind of reasoning we must often make when proving the correctness of *loops* in imperative code. As we will see in detail in later lectures, to prove that `factImp` from Section 1 is correct, we must define a *loop invariant*: a property that holds throughout all executions of the loop, from which the correctness of the procedure follows. Devising the appropriate loop invariant and generalizing the inductive hypothesis are instances of the same principle.

7 Exercises

1. Prove, using the style of Section 3, that the function `fact` from Section 1 is correct. That is, prove that `fact(`n`)` $\hookrightarrow n!$, for all natural numbers n.

2. Consider the following alternative formulation of the factorial function:

```
function factAcc(n:nat,a:int) : int
{
  if (n = 0) then a else factAcc(n-1,n*a)
}
function factAlt(n:nat) : int { factAcc(n,1) }
```

Prove that `factAlt` is a correct implementation of the factorial function.

3. Using Dafny, define the functions `pow`, `powerAlt` and `power` as above. Prove, using Dafny, that `power` and `powerAlt` are equivalent. That is, define in Dafny:

   ```
   lemma powerEquiv(n:int,m:nat)
       ensures power(n,m) == powerAlt(n,m)
   { ... }
   ```

   Hint: Dafny will not be able to prove this lemma automatically, just as we could not establish the result directly by induction. Define another lemma which relates the result of `pow` to the result of `power`, that we can then use to prove `powerEquiv`. Note that using a lemma in Dafny uses the same syntax as calling a function (ended with `;`).

4. Do the same as in Exercise 3 but for `fact` and `factAlt`.

5. Try to understand what the following mystery functions compute and prove them correct using Dafny. To do so, you will only need to write the appropriate ensures clause in each function (i.e., `function m1(n:nat,m:nat) :nat ensures m1(n,m) ==n*n+m*m`):

   (a)
   ```
   function mystery1(n:nat,m:nat) : nat
   { if n==0 then m else mystery1(n-1,m+1) }
   ```

   (b)
   ```
   function mystery2(n:nat,m:nat) : nat
       decreases m
   { if m==0 then n else mystery2(n+1,m-1) }
   ```

   (c)
   ```
   function mystery3(n:nat,m:nat) : nat
   { if n==0 then 0 else mystery1(m,mystery3(n-1,m)) }
   ```

   (d)
   ```
   function mystery4(n:nat,m:nat) : nat
   { if m==0 then 1 else mystery3(n,mystery4(n,m-1)) }
   ```

6. [⋆] Without using any specification for `mystery1` prove the following lemma:

   ```
   lemma mys1c(n:nat,m:nat)
   ensures mystery1(n,m) == mystery1(m,n)
   {...}
   ```

   Hint: You will likely have to prove something about `mystery1(n,0)` and the relation of `mystery1(n,m+1)` and `1+mystery1(n,m)`.

7. [⋆⋆⋆] Similarly, without using any specification for `mystery2` prove the following property:

   ```
   lemma mysEq(n:nat,m:nat)
       ensures mystery1(n,m) == mystery2(n,m)
   {...}
   ```

8. Prove, using the style of Section 3, that the following function is correct:

   ```
   function gen(n:nat) : List<nat> {
       if n=0 then Nil else Cons(n,gen(n-1))
   }
   ```

   That is, that (for any `n`), `gen(n)` produces the list $[n, n-1, n-2, \ldots, 0]$.

9. Show using the style of Section 5.1, that the following function is correct:

   ```
   function length<T>(l:List<T>) : nat {
       match l
       case Nil ⇒ 0
       case Cons(_,xs) ⇒ 1+length<T>(xs)
   }
   ```

   That is, that (for any `l`), `length<T>(l)` calculates the length of `l`.

10. [⋆] Show using the style of Section 5.1, that `lengthTL<T>(l,0)` calculates the length of list `l`:

    ```
    function lengthTL<T>(l:List<T>,acc:nat) : nat {
        match l
        case Nil ⇒ acc
        case Cons(_,xs) ⇒ lengthTL<T>(xs,1+acc)
      }
    ```

    Note that you may need to generalize the induction hypothesis for the argument to go through correctly.

11. [⋆] Use Dafny to show that `lengthTL<T>(l,0)` and `length(l)` are equivalent. That is, prove the following lemma (likely via another auxiliary lemma):

    ```
    lemma lengthEq<T>(l:List<T>)
        ensures length<T>(l) = lengthTL<T>(l,0)
    { }
    ```

12. [⋆⋆⋆] Show in Dafny that the natural numbers, when defined as an inductive data type, are an adequate representation of the natural numbers. That is, show any value n generated by the data type:

    ```
      datatype Nat = Zero | Succ(Nat)
    ```

    maps to the correct corresponding value of type **`nat`**. To do this you will likely want to define functions `code` and `decode` that map `Nat` to **`nat`** and vice-versa and show that they *commute* (i.e. `code(decode(n)) =n` and vice-versa).

13. [⋆ ⋆ ⋆ ⋆] If you completed the previous exercise, show that the `add` function defined over `Nat` is adequate with respect to addition of natural numbers. Let `decode(n:Nat) :` **`nat`** and `code(n:`**`nat`**`) : Nat` be the (correct) mappings between the two types. You want to establish two results (for all appropriately typed n and m):

    (a) `add(n,m) =code(decode(n)+decode(m))`

    (b) `n+m =decode(add(code(n),code(m)))`

    Hint: You may need to appeal to an auxiliary result that the result of `add`ing `code`s with the `code` of adding naturals.