15-150 Fall 2020
Lectures 3 and 4
Specifications and proofs

©Stephen Brookes

1 Overview

This chapter consists of two parts, the first dealing with integers and the second with lists. We focus first on functions from (tuples of) integers to integers. The ideas and techniques generalize, as we will see later.

We introduce *inductive* techniques, for proving properties of recursive functions, and for showing that code terminates. We will see that program structure guides us in our search for an appropriate way to analyze behavior. We will start with "simple" or mathematical induction and progress quickly to "complete" (or "strong") induction. These methods are designed to prove statements that can be expressed in the form "For all non-negative integers n, some property holds". These methods are actually special cases of a more general proof method known as "well-founded induction". Later we will discuss "structural" induction principles for datatypes, including lists and trees; these are also special kinds of well-founded induction. We won't talk much about well-founded orderings in their full generality, as that's more usually regarded as a topic for a math course. Don't worry if these terms seem highfalutin or esoteric (yes, I like big words). You'll get used to thinking inductively, and you will soon see the need for more sophisticated (but still natural and intuitive) forms of induction.

Some of our proofs will deal with math and some deal with ML functions; we want you to see that programs are susceptible to the kind of proof techniques that you should have seen earlier in math classes.

For the moment it is enough to know that the set of non-negative integers, with the usual less-than ordering, has a least element 0, and for every positive integer n we have $0 < 1 < \cdots < n-1 < n$. Crucially, if you start with a non-negative integer and keep decreasing it, you can only make finitely any steps before you reach 0.

By starting with simple induction and gradually working up to more general kinds of induction, we hope you find it easier to get used to formal reasoning. You should also have had some experience with inductive proofs in a pre-requisite math class, so this should not come as a complete surprise.

Examples discussed here may not correspond exactly with those from class. For exercise, you might find it worthwhile to re-work the classroom examples using these techniques.

Tricks of the trade

The art of doing inductive proofs can be summarized as follows:

- Find a suitable way to formulate your problem. It may be easier to tackle a more general problem from which your desired result follows.
- Use the function definition to guide your proof strategy.
- Use terminology and notation accurately and set out your proof as an easily followable sequence of logically sound steps.

In the rest of these notes we'll explain how to put this methodology into practice and introduce a template to use when building proofs.

Some comments about values

Specifications about functions typically say what the function "does" when applied to "values" with certain assumed properties. Remember that we're talking about *expressions* in a programming language, which may or may not evaluate to (syntactic) *values*; and a syntactic value like `fn x:int => x+1` can be said to *denote* or represent a mathematical thing called a partial function (here, from the integers to the integers). Some expressions *are* values, and an expression that has a value (ie. its evaluation terminates) may be called *valuable*. We should be careful not to confuse ourselves about valuability. Don't accidentally assume that you're only dealing with valuable expressions when your spec doesn't say so. The kind of reasoning we do about functional programs is subtly more complex than typical math reasoning, because of this issue of non-termination.

In math we might say something like "For all integers n" some property holds. When talking about an ML function `F:int -> int` we might say "For all values `v:int`, `F(v)`" has some property, or "For all valuable expressions `e`, `F(e)`" has some property, but to be pedantic we have to be careful like this: these two assertions don't say the same thing [1] Even more confusingly, "For all expressions `e:int`, `F(e)` has some property" says yet another thing — it includes information about what happens when `e` is an integer expression

[1] Actually, the second property (about "all valuable `e`) is deducible from the first (for all values `v`) by compositionality: if `e` has value `v` then `F(e) = F(v)`. So if `F(v)` has a property, so does `F(e)`.

whose evaluation doesn't terminate. (And in a call-by-value language like L there's not much of interest that could be said about `F(e)` in such a case, except that `F(e)` doesn't terminate.)

One reason values are so important shows up when trying to reason using equations, about an application of a function to a given argument. For example, one of the equations that we discussed earlier (let's call it "value reduction") is:

```
(fn x => e) e2 = [e2/x]e     (if e2 is a value)
```

On the right-hand side we write `[e2/x]e` for the expression obtained by substituting `e2` for (free occurrences of) `x` in `e`. To see why this syntactic constraint is necessary, consider the declaration

```
fun loop(y:int):int = loop y
```

In the scope of this declaration we have

$$\texttt{loop}\ 42 \Longrightarrow (\texttt{fn y => loop y})\ 42 \Longrightarrow \texttt{loop}\ 42$$

so `loop 42` has an infinite execution, doesn't terminate, and definitely isn't a value! We also have

$$(\texttt{fn x => 0})\ (\texttt{loop}\ 42) \Longrightarrow^{(2)} (\texttt{fn x => 0})\ (\texttt{loop}\ 42)$$

showing that `(fn x => 0)(loop 42)` also doesn't terminate, so it definitely doesn't evaluate to `0`. Of course the result of the substitution (`loop 42` for `y` in the expression `0`) is just `0`. So the "value reduction" rule would be unsound without the constraint.

We mention all this to emphasize that you must learn to deal properly with values and valuability. We aim to show you how, but sometimes may forget to remind you why it's so important.

By the way, now that we've re-introduced the value-substitution equation for applications, it's worth noting the following fact.

If `e2` is valuable, then `(fn x => e) e2` = `[e2/x]e`.

The reason: if `e2` has a value, say `v2`, we have
`(fn x => e) e2 = (fn x => e) v2 = [v2/x]e = [e2/x]e`.
(This chain of equalities is justified using a combination of value substitution and compositionality of `=` or referential transparency.)

Totality

Here is a good place to emphasize the importance of functions that have a special property: *total* functions. We often include totality as an assumption in specifications; or we may want to prove that a function actually is total. It's worth setting out clearly (or reiterating, if you've already seen this) what it means for a function (say, of type `t1 -> t2`) to be total. Again, it's all about values.

> A function `F : t1->t2` is said to be *total* if and only if for all values `v1` of type `t1`, `F(v1)` evaluates to a value [2].

Some examples:

- `fn (x:int):int => x+1` is total
- `fn (y:int):int => y+y` is total
- The function `f:int -> bool` defined by

  ```
  fun f(x:int):bool = case Int.compare(x, 0) of
                        LESS => f(x+1)
                      | EQUAL => true
                      | GREATER => f(x-1)
  ```

  is total. (How could we show this? You'll see soon.)

Some useful properties of total functions are worth summarizing before we continue.

First, another (equivalent) way to define totality: the property as written above can be rephrased as: `F:t1->t2` is total iff for all expressions `e1` of type `t1`, if `e1` is valuable, so is `F(e1)`. Check that you understand why this version says essentially the same thing.

An extremely useful fact: if `f:t1->t2` and `g:t2->t3` are total functions, so is the composite function `(g o f):t1->t3` (the result of composing `f` with `g`). Indeed, the composite function has the property that

```
(g o f) = fn (x:t1):t3 => g(f x)
```

[2] By the way, the type guarantee implies that the value of `F(v1)` will have type `t2`.

Suppose `f` and `g` are total, and let `v1` be a value of type `t1`. By totality of `f`, there is a value `v2` of type `t2` such that `f v1` evaluates to `v2`. By totality of `g`, there is a value `v3` of type `t3` such that `g v2` evaluates to `v3`. So we have

$$(\texttt{g o f})\ \texttt{v1} \Longrightarrow^* \texttt{g v2} \Longrightarrow^* \texttt{v3}$$

Now back to induction...

2 Mathematical induction

Suppose we want to prove:

For all non-negative integers n, $P(n)$ holds.

Here $P(n)$ is some property of n.

Mathematical (or simple) induction is the following proof strategy:

(a) Show that $P(0)$ holds.

(b) Assume that $n > 0$ and $P(n-1)$ holds; show from these assumptions that $P(n)$ also holds.

We call $n = 0$ the "base case" here; we also refer to $n > 0$ as the "inductive case" here, and $P(n-1)$ as the "induction hypothesis".

If you can prove (a) and (b), it follows that for all non-negative integers k, $P(k)$ holds. This is because (a) tells us that $P(0)$ holds; so by (b) we get that $P(1)$ holds; by (b) again we get that $P(2)$ holds; and so on. For each non-negative integer k, after k steps of this proof process we get that $P(k)$ holds. (This paragraph is not part of the proof strategy, but should serve to convince you why the strategy works!)

Depending on your choice of property P, simple induction may or may not work as a proof strategy[3]. Once you get familiar with using this technique you will learn to identify which kinds of problems are susceptible to this strategy.

[3]It's pretty obvious that you can only use simple induction to prove $\forall n \geq 0.P(n)$ if for each $k > 0$ you can use $P(k-1)$ to prove $P(k)$. Sometimes the information conveyed by $P(k-1)$ just isn't strong enough for this purpose.

An example using mathematical induction

Here is a simple example, and we'll present its proof using our standard template, which has five parts.

(i) State the theorem to be proven:

(THM): For all $n \geq 0$, $2 \cdot n = n + n$.

(We have labelled this theorem statement so we can refer to it later.)
Let us write $P(n)$ for the property "$2 \cdot n = n + n$".
So we're trying to prove (THM) : $\forall n \geq 0.\ P(n)$.

(ii) State what the proof method is going to be:

Proof: By mathematical induction on n.

(iii) State and prove the base case:

Base case: $n = 0$.
Need to show: $P(0)$, i.e. $2 \cdot 0 = 0 + 0$.
Proof: This is obvious, since $2 \cdot 0 = 0$ and $0 + 0 = 0$.

(iv) State the inductive case and the induction hypothesis, then prove the inductive step:

Inductive case: $n > 0$.
Induction hypothesis: $P(n-1)$, i.e. that

$$2 \cdot (n-1) = (n-1) + (n-1).$$

Inductive step: We need to show, assuming $n > 0$ and that $P(n-1)$ holds, that $P(n)$ holds, i.e. that

$$2 \cdot n = n + n.$$

Proof: Again this is easy, using well known properties of addition and multiplication. Here are the details:

$$\begin{array}{rcll} 2n & = & 2((n-1)+1) & \text{since } n = (n-1)+1 \\ & = & 2(n-1)+2 & \text{by distributivity} \\ & = & (n-1)+(n-1)+2 & \text{by } P(n-1) \\ & = & n+n & \text{by arithmetic} \end{array}$$

(We write "by arithmetic" to indicate use of standard facts about arithmetic operations on the integers.)

(v) By (iii) and (iv) it follows that (THM) holds.

By clearly identifying stages (i) through (v) as shown in this template we hope you will learn to write well organized proofs. Note also how we included justifications for proof steps. In practice we may often omit some or most of the justificational details, especially when they rely on "obvious" or standard mathematical facts. Nevertheless the sign of a "good" proof is when missing details can indeed be filled in very easily, without the need to guess what the proof's author had in mind.

A variation on simple induction

Sometimes it is convenient to use a slight variation on the above form of simple induction, by isolating more than one "base" cases. In the standard strategy we deal with $P(0)$ as base case and the inductive step can be used to generate proofs of $P(1)$, $P(2)$, and so on. It is equally valid to begin with $k > 1$ base cases $P(0), \ldots, P(k)$ for each of which a direct proof can be given, and to rely on the inductive step to get proofs for $P(k+1), P(k+2)$, and so on. The "best" choice of k will obviously depend on the problem.

Example

Consider the following recursive function definition:

```
fun zero(n:int):int = if n<42 then 0 else zero(n-1)
```

(a) Using simple induction (with base case `0`) prove that

$$\forall n \geq 0.\, \texttt{zero}(n) = 0$$

Remember, this means the same as:
For all values `n` of type `int`, `zero n` evaluates to `0`.

(b) Use simple induction with base cases 0 through 41 to prove this.

(c) Prove that `zero` is total.

Simple recursive functions

Simple induction can be useful for reasoning about recursive functions with a simple structural property. For example, suppose we have a function `f` of type `int -> int`, and we want to prove that for all non-negative integer values `n`, `f(n)` evaluates to an integer value, say `v`, with a certain property, say `Q(v)`. For example, if `Q(v)` is trivial (i.e. `true`), we're trying to prove that `f(n)` terminates. Or `Q(v)` may say that the value of `v` is equal to some quantity that depends on the value of `n`.

Suppose that the function definition for `f` has a clause for `f(0)` giving the result directly, and a clause for `f(x)` that makes a recursive call to `f(x-1)`. We can try using *simple induction* to prove that for all non-negative integer values `n`, `P(n)` holds, where we let `P(n)` be the property: `f(n)` evaluates to an integer value `v` such that `Q(v)` holds.

A proof of this result, using simple induction on `n`, has the following form:

(i) State the theorem to be proven:

(THM): For all `n` ≥ 0, `f(n)` evaluates to a value `v` such that `Q(v)` holds.

(We have labelled this theorem statement so we can refer to it later.)

(ii) State what the proof method is going to be:

Proof: By mathematical induction on n.

(iii) State and prove the base case:

Base case: `n=0`.
To show: `P(0)`, i.e. `f(0)` evaluates to a value v_0 such that `Q(`v_0`)` holds.
Proof: (Use the function definition for this!)

(iv) State the inductive case and the induction hypothesis, then prove the inductive step:

Inductive case: `n > 0`.
Induction Hypothesis: `P(n-1)`, i.e.

`f(n-1)` evaluates to a value v' such that `Q(`v'`)` holds.

To show: `P(n)`, i.e. `f(n)` evaluates to a value `v` such that `Q(v)` holds.
Proof: show this, using `P(n-1)`, the assumption that `n>0`, and the function definition.

(v) By (iii) and (iv) it follows that (THM) holds.

Comment: we deliberately used different "meta-variables" $\mathtt{v}, \mathtt{v}_0, \mathtt{v}'$ in the different subtasks above, to avoid any confusion; if we had written the same `v` everywhere you might have gotten mixed up. You don't have to label the theorem or the induction hypothesis, but doing so makes it easy to refer to them in your proof.

In practice, there may be more than one reasonable way to write down a theorem statement, or to state an induction hypothesis. For example, it may be convenient to write an equation like `f(n)=v` instead of saying "`f(n)` evaluates to `v`". This is especially common when we are trying to prove that `f(n)` evaluates to a specific value (that is expressible explicitly and usually depends on `n`). Recall that since the return type of `f` here is `int`, saying that `f(n)` evaluates to `v` means the same as saying that "the value of `f(n)` is `v`", or just $\mathtt{f}(\mathtt{n}) = \mathtt{v}$.

And if the function definition is not as simple as the assumptions we made above about `f`, for example if `f(n)` makes more than one recursive call, or has more than one explicit "base" clauses in which no recursive calls are needed, we'll need to adapt the proof strategy and use a more sophisticated form of induction (coming soon!).

A simple recursive example

```
(* exp : int -> int *)
fun exp (n:int):int = if n=0 then 1 else 2 * exp (n-1)
```

This function is *not* total: it doesn't terminate when applied to a negative integer. Your task: Prove that for all $n \geq 0$, `exp` $n = 2^n$.

The definition of `exp` gives the value `exp(0)` directly, without making any recursive calls; so `n=0` is a "base" clause. Otherwise when `n` > 0 it's clear that `exp(n)` makes a recursive call to `exp(n-1)`. So this function definition fits the pattern above and we can adapt the proof template from above for simple induction.

(i) The theorem to be proven:

(EXP): For all $\mathtt{n} \geq 0$, $\mathtt{exp}(n) = 2^n$.

(ii) Proof: By mathematical induction on n.

(iii) Base case: `n=0`.
To show: `exp(0)` $= 2^0$.
Proof: This is clear from the function definition, since $2^0 = 1$ and

```
exp 0
  = if 0=0 then 1 else 2 * exp(0-1)
  = if true then 1 else 2 * exp(0-1)
  = 1.
```

(We use implicitly the obvious math fact that expressions which are both equal to the same value are equal.)

(iv) Inductive case: `n` > 0.
Induction Hypothesis:

(IH): `exp(n` $-$ `1)` $= 2^{\mathtt{n}-1}$.

To show: with these assumptions, `exp(n)` $= 2^{\mathtt{n}}$.

Proof: Using the function definition for `exp`, plus obvious facts about if-then-else expressions, the induction hypothesis (IH), and standard arithmetic facts, we have

```
exp n
  = if n=0 then 1 else 2 * exp(n-1)        by definition of exp
  = if false then 1 else 2 * exp(n-1)      since n>0
  = 2 * exp(n-1)
  = 2 * (2^(n-1))                          by IH
  = 2^n                                    by math
```

as required.

(v) By (iii) and (iv) it follows that (EXP) holds.

That completes the proof. Note that we used "equational reasoning" and we formulated the theorem and induction hypothesis in terms of "equality". And we took advantage of referential transparency, even though we didn't say so explicitly! (Make sure you can see where we did this.)

Variation

We could have done this proof using evaluation rather than equality, but we would need to be careful in writing out the details because mathematical notation like 2^n isn't legal ML syntax. (You can't just say "`exp(n)` evaluates to `2^n`", because we're talking here about evaluation to an ML value, so using ML notation, and in ML the `^` symbol means string concatenation, not exponentiation!) Just for the sake of illustration, here is how we could have done the correctness analysis this way. Before we start, note that in order to maintain distinction between numerals (syntax) and integer values we can use phrases like *the numeral for* $2^{\mathtt{n}}$ instead of *the numeral* $2^{\mathtt{n}}$, and this is better since $2^{\mathtt{n}}$ is really a mathematical notation rather than a piece of program syntax like a numeral; for example, the numeral for 2^3 is `8`. We could also use the phrase "the value of $2^{\mathtt{n}}$" for the same purpose.

(i) The theorem to be proven:

> (THM): For all non-negative integers `n`,
> for all expressions `e` such that `e =>* n`,
> `exp(e) =>* v`, where `v` is the numeral for $2^{\mathtt{n}}$.

(ii) Proof: By simple induction on n.

(iii) Base case: `n=0`. To show: For all `e` such that `e =>* 0`, `exp(e) =>* 1`. (Clearly, `1` is the numeral for 2^0.)

Proof: Suppose `e` is an expression such that `e =>* 0`. We have, using some obvious facts about expression evaluation:

```
exp e
  =>* (fn n => if n=0 then 1 else 2 * exp(n-1)) e
  =>* (fn n => if n=0 then 1 else 2 * exp(n-1)) 0  since e =>* 0
  =>* if 0=0 then 1 else 2 * exp(0-1)
  =>* if true then 1 else 2 * exp(0-1)    since 0=0 =>* true
  =>* 1.
```

(iv) Inductive case: `n` > 0
Induction Hypothesis:

> (IH): For all expressions `e'` such that `e' =>* n-1`,
> `exp(e') =>* v'`, where `v'` is the numeral for $2^{\mathtt{n}-1}$.

To show: Let `e` be an expression such that `e =>* n`. We must show that `exp(e) =>* v`, where `v` is the numeral for 2^n.

```
exp e
   =>* (fn n => if n=0 then 1 else 2 * exp(n-1)) e
   =>* (fn n => if n=0 then 1 else 2 * exp(n-1)) n
                                          since e =>* n
   =>* if n=0 then 1 else 2 * exp(n-1)
   =>* if false then 1 else 2 * exp(n-1)
                                          since n=0  =>* false
   =>* 2 * exp(n-1)
   =>* 2 * v', where v' = 2^(n-1)
                                          by induction hypothesis
   =>* v, where v = 2^n
```

We rely here on the math fact that for all integers n, $2 \times 2^{n-1} = 2^n$.

(v) By (iii) and (iv) it follows that (THM) holds.

3 Strong induction

Simple induction may work for functions whose definitions are simple enough, as outlined above. (There is still the need to master the subtle art of picking a sensible induction hypothesis, one that actually enables you to complete the inductive step in the proof method.) However, there are many functions that need a more complex kind of recursive formulation, and simple induction is (as its name suggests) not general enough. Next we introduce a more widely applicable technique known as *complete induction*, *strong induction*, or *course-of-values induction*.

Suppose we want to prove:

For all non-negative integers n, $P(n)$ holds.

Strong induction is the following proof strategy:

(a) Base case: Show $P(0), P(1), \ldots, P(k)$, for some non-negative integer k.

(b) Let $n > k$ and assume that $P(m)$ holds, for all m such that $0 \leq m < n$. Show that $P(n)$ also holds, under these assumptions.

We call $n = 0$ through $n = k$ the "base case" here; we refer to $n > k$ as the "inductive case", and $P(0), P(1), \ldots, P(n-1)$ as the "induction hypotheses".

This technique can be used to reason about the applicative behavior of a function `f` of type `int -> int` that has a finite number of base cases (for `0`, `1`, up to some fixed number, say `k`) and some recursive clauses in which `f(x)` makes recursive calls that apply `f` to *smaller* non-negative integer arguments.

The classic example is the Fibonacci sequence, represented as the ML function `fib` as in:

```
fun fib 0 = 1
 |  fib 1 = 1
 |  fib n = fib(n-1) + fib(n-2)
```

Here we have two base cases and one recursive clause in which `fib(n)` makes two recursive calls; since this clause will only ever be used when the value of `n` is bigger than 1, the values of `n-1` and `n-2` in the recursive calls are guaranteed to be non-negative, and they are also obviously smaller than the value of `n`.

Here is a template for complete induction. Suppose as above that we have a function definition for `f` of type `int -> int`, with a finite number of base cases (`n=0` up to `n=k`) for some non-negative integer `k`, and with some recursive clauses in which `f(x)` makes recursive calls that apply `f` to *smaller non-negative* integer arguments.

(i) State the theorem to be proven:

(THM): For all `n` ≥ 0, `f(n)` evaluates to a value `v` such that `Q(v)` holds.

(ii) State what the proof method is going to be:

Proof: By complete induction on n.

(iii) State and prove the base cases:

Base cases: `n=0` through `n=k`.
Show that for each `i` in the range `0` through `k`,
`f(i)` evaluates to a value $\mathtt{v}_i$ such that `Q(`$\mathtt{v}_i$`)` holds.
(Use the function definition!)

(iv) State the inductive case(s), the induction hypothesis, and prove the inductive step:

Inductive case: `n > k`
Induction Hypothesis:

(IH): For all `m` such that $0 \leq m < n$,
`f(m)` evaluates to a value v' such that `Q(v')` holds.

Show, using the function definition for `f`, and (IH), that
`f(n)` evaluates to a value `v` such that `Q(v)` holds.

(v) By (iii) and (iv) it follows that (THM) holds.

An example using strong induction

Let's use complete induction to prove something about the `fib` function. First we need some math. Let ϕ be the real number $\frac{1+\sqrt{5}}{2}$ and let ψ be the real number $\frac{1-\sqrt{5}}{2}$. Then it is easy to check that $\phi^2 = \phi + 1$ and $\psi^2 = \psi + 1$. Hence for $n \geq 1$, $\phi^{n+1} = \phi^n + \phi^{n-1}$ and $\psi^{n+1} = \psi^n + \pi^{n-1}$. (Multiply both sides of the equations by ϕ^{n-1} and ψ^{n-1}, respectively.)

Assuming these results about ϕ and ψ, we will prove that for all $\mathtt{n} \geq 0$, $\mathtt{fib}(\mathtt{n}) = \frac{1}{\sqrt{5}}(\phi^{n+1} - \psi^{n+1})$.

Recall that we already mentioned that the definition of `fib` has base cases `0` and `1`, and when `x>1` we noted that `fib(x)` makes recursive calls to `fib(x-1)` and `fib(x-2)`, both arguments being non-negative. So the `fib` function definition fits the template for complete induction, with `k=1`.

(i) State the theorem to be proven:

(FIB): For all $\mathtt{n} \geq 0$, $\mathtt{fib}(\mathtt{n}) = \frac{1}{\sqrt{5}}(\phi^{n+1} - \psi^{n+1})$.

(ii) State what the proof method is going to be:

Proof: By complete induction on n.

(iii) State and prove the base cases:

Base cases: `n=0` and `n=1`.

- For `n=0`:

```
fib(0) = 1 by definition
```

Recall the values of ϕ and ψ from above. Since $\phi^{0+1} - \psi^{0+1} = \phi - \psi = \sqrt{5}$, we have $\texttt{fib}(0) = 1 = \frac{1}{\sqrt{5}}(\phi^1 - \psi^1)$, as required.

– For `n=1`:

```
fib(1) = 1 by definition
```

Clearly $\phi - \psi = \sqrt{5}$, $\phi + \psi = 1$, so $\phi^{1+1} - \psi^{1+1} = \phi^2 - \psi^2 = (\phi - \psi)(\phi + \psi) = \sqrt{5}$, and we have $\texttt{fib}(1) = 1 = \frac{1}{\sqrt{5}}(\phi^2 - \psi^2)$, as required.

(iv) State the inductive case, the induction hypothesis, and prove the inductive step:

Inductive case: $\texttt{n} > 1$
Induction Hypothesis:

(IH): For all $\texttt{m}$ such that $\texttt{0} \leq \texttt{m} < \texttt{n}$, $\texttt{fib}(\texttt{m}) = \frac{1}{\sqrt{5}}(\phi^{\texttt{m}+1} - \psi^{\texttt{m}+1})$.

We show that $\texttt{fib}(\texttt{n}) = \frac{1}{\sqrt{5}}(\phi^{\texttt{n}+1} - \psi^{\texttt{n}+1})$, as follows.

By (IH),

$$\begin{aligned} \texttt{fib}(\texttt{n}-1) &= \tfrac{1}{\sqrt{5}}(\phi^{\texttt{n}} - \psi^{\texttt{n}}) \\ \texttt{fib}(\texttt{n}-2) &= \tfrac{1}{\sqrt{5}}(\phi^{\texttt{n}-1} - \psi^{\texttt{n}-1}) \end{aligned}$$

Hence, by the function definition,

$$\begin{aligned} \texttt{fib}(\texttt{n}) &= \texttt{fib}(\texttt{n}-1) + \texttt{fib}(\texttt{n}-2) \\ &= \tfrac{1}{\sqrt{5}}(\phi^{\texttt{n}} - \psi^{\texttt{n}}) + \tfrac{1}{\sqrt{5}}(\phi^{\texttt{n}-1} - \psi^{\texttt{n}-1}) \\ &= \tfrac{1}{\sqrt{5}}(\phi^{\texttt{n}} + \phi^{\texttt{n}-1} - \psi^{\texttt{n}} - \psi^{\texttt{n}-1}) \\ &= \tfrac{1}{\sqrt{5}}(\phi^{\texttt{n}+1} - \psi^{\texttt{n}+1}) \end{aligned}$$

(v) By (iii) and (iv), (FIB) holds.

Remarks

This proof might seem a bit weird, talking about real numbers like ϕ and mixing mathematical arithmetic notation with program specifications. But it's all OK as long as we're accurate.

This `fib` function isn't total. It's useful when we know we want to apply it to a non-negative integer. That's what we just proved. To turn it into a total function we *could* invent a value to be returned on negative arguments, or maybe replace it by a function of type `int -> int option`. See if you van figure out what we mean, and if there would any advantages in doing so.

Warning

Be careful! Make sure whenever you attempt an inductive proof that the problem obeys the general rules mentioned in the preamble above. Make sure you choose an appropriate version of induction. Use the function's definition to guide you!

Here, for example, is a recursive function definition that doesn't have all of the required structural properties, and for which strong induction is *not* going to be usable:

```
fun foo 0 = 0
 |  foo x = foo(x-2)
```

Even though for `x>0` we see that `foo(x)` makes a recursive call to `foo(x-2)`, and the value of `x-2` is less than the value of `x`, the value of `x-2` may not be a non-negative integer! (One of the requirements included above was that the recursive calls must be on smaller but still non-negative integers.) In fact, for `x=1` we get `x-2<0`. And `foo(x)` will loop forever if `x>0` and `x` is odd.
So it is just as well that we aren't allowed (because of the rules violation) to use this proof method to deduce that "for all `n` ≥ 0, `foo`(`n`) terminates"! If we attemp[t suh a proof, what happens?

For further exploration consider the following exercises:

1. Use simple induction to prove that for all `n` ≥ 0, `foo(2 * n) = 0`, where `foo` is the function given above.

2. Is it true that for all `n` ≥ 0, `foo`(`n`) $= 0$? What goes wrong when you try to use strong induction to prove this property?

4 Lists

Now we will illustrate how to use inductive techniques to reason about functions that manipulate or return lists. For simplicity, we choose examples involving lists of integers. These ideas and techniques work just as well for lists built from other types of value. Later we will introduce a more general formulation of *structural induction* applicable to more general datatypes in ML, and this will be a nice generalization of the list techniques in this chapter.

Suppose we want to prove:

> For all integer lists L, $P(L)$ holds.

Here $P(L)$ is some property of lists.

How might we go about this? The key is to realize that every list value is *constructible* from the empty list using a finite number of "cons" operations. For example, the list `[1,2,3]` is expressible as `1::(2::(3::nil))`. Actually, since the ML cons operator is right-associative, we can suppress the parentheses and say `[1,2,3] = 1::2::3::nil`. And every integer list value is either empty (equal to `nil`), or non-empty and of the form `x::R` for some integer value `x` and integer list value `R`.

List induction is the following proof strategy:

(a) Show that $P(\texttt{nil})$ holds.

(b) Let `L=x::R` be an arbitrary integer list value, and assume as induction hypothesis that $P(\texttt{R})$ holds. Show that $P(\texttt{L})$ also holds, under these assumptions.

We call `nil` the "base case" here; we also refer to $\texttt{x} :: \texttt{R}$ as the "inductive case" here, and $P(\texttt{R})$ as the "induction hypothesis".

If you can prove (a) and (b), it follows that for all integer lists L, $P(L)$ holds. This is because (a) tells us that $P(\texttt{nil})$ holds; so by (b) we get that $P(\texttt{x} :: \texttt{nil})$ holds, for all integer values `x`; by (b) again we get that $P(\texttt{y} :: \texttt{x} :: \texttt{nil})$ holds for all integers `x` and `y`; and so on. For each non-negative integer k, after k steps of this proof process we get that $P(L)$ holds *for all integer lists L with k elements*. (This paragraph is not part of the proof strategy, but should serve to convince you why the strategy works!)

Depending on your choice of property P, this simple form of list induction may or may not work as a proof strategy. Once you get familiar with

using this technique you will learn to identify which kinds of problems are susceptible to this strategy.

An example using list induction

Consider the ML function `length:int list -> int` defined by:

```
fun length (L:int list):int =
   case L of
         [ ]  => 0
      | (_::R) => 1 + length R
```

This function definition uses a "case expression" with pattern matching for values of type `int list`. Actually we could just as well have defined this function without a case expression, instead using function clauses, one for each list pattern in the case expression, as in:

```
fun length ([ ] :int list):int = 0
 |  length ( _::R) = 1 + length R
```

Stylistically, it is a bit awkward to decide where to place the type annotations that we are insisting you use (so far); we chose to put them in the first clause only, but it is equally acceptable to put them in each clause. In this respect the case version looks less asymmetric, since the types only need to appear once, with the function's argument.

In the rest of this question the analysis should work out for each of these alternative ways to define this function.

Your task: prove that for all integer list values `L`, `length(L)` evaluates to a non-negative integer. (Informally, "every list value has a finite length.") This is a pretty obvious result, but the proof is a nice illustration of how to use induction on list structure. And again, all this really says is that (the value of) `length` is a total function from integer lists to integers.

The details of the proof start on the next page...
We fill out the template as we go.

(i) State the theorem to be proven:

(THM): For all `L` : `int list`, `length`(`L`) evaluates to a non-negative integer.

Let us write P(`L`) for this property, i.e. "`length(L)` evaluates to a non-negative integer".

(ii) State what the proof method is going to be:

Proof: By list induction on `L`.

(iii) State and prove the base case:

Base case: `L` = `nil`.
Need to show: P(`nil`), i.e. `length(nil)` terminates.
Proof: This is obvious from the function definition, since `length(nil) =>* 0`, and `0` is a non-negative integer.

(iv) State the inductive case and the induction hypothesis, then prove the inductive step:

Inductive case: Let `L` be of form `x::R`, where `x` is an integer and `R` is a list.
Induction hypothesis: We assume that P(`R`) holds,i.e. there is a non-negative integer `n` such that `length R =>* n`.
Inductive step: We need to show, with these assumptions, P(`x` :: `R`) holds, i.e. that `length(x::R)` evaluates to a non-negative integer.
Proof: Again this is easy. Guided by the function definition we have:

```
length(x::R)
   =>* 1 + length(R)
   =>* 1 + n     for some integer value n>=0,
                by assumption P(R)
   =>* v,        where v is the numeral for n+1
```

Since we assumed `n>=0`, we also have `n+1>=0` and `v` here is a non-negative integer. So `P(x::R)` holds, as required.

(v) By (iii) and (iv) it follows that (THM) holds.

Although this example is rather obvious, it does mean we have proven the fundamental fact that every ML list value is "finite", i.e. has a finite length, where length is defined as above. This notion of length coincides with the number of items in the list. Also notice that for all `L`, `length(L)` ≥ 0; and `length(x :: R)` $>$ `length(R)`. So it should be fairly obvious that every theorem that has a proof using *list induction* also has a proof by *mathematical induction on the length of lists*.

Simple list functions

List induction can be useful for reasoning about recursive functions with a simple structural property. For example, suppose we have a function `f` of type `int list -> int`, and we want to prove that for all integer lists `L`, `f(L)` evaluates to an integer value, say `v`, with a certain property, say `Q(v)`. For example, if `Q(v)` is trivial (i.e. `true`), we're trying to prove that `f(n)` terminates. Or `Q(v)` may say that the value of `v` is equal to some quantity that depends on the value of `n`.

Suppose that the function definition for `f` has a clause for `f(nil)` giving the result directly, and a clause for `f(x::R)` that makes a recursive call to `f(R)`. We can try using *list induction* to prove that for all integer lists `L`, `P(L)` holds, where we let `P(L)` be the property: `f(L)` evaluates to an integer value `v` such that `Q(v)` holds.

A proof of this result, using list induction on `L`, has the following form:

(i) State the theorem to be proven:

(THM): For all `L` : `int list`, `f(L)` evaluates to a value `v` such that `Q(v)` holds.

(ii) State what the proof method is going to be:

Proof: By list induction on `L`.

(iii) State and prove the base case:

Base case: `L=nil`.
To show: `P(nil)`, i.e. `f(nil)` evaluates to a value v_0 such that `Q(`v_0`)` holds.
Proof: (Use the function definition for this!)

(iv) State the inductive case and the induction hypothesis, then prove the inductive step:

Inductive case: `L=x::R`.
Induction Hypothesis: `P(R)`, i.e.

`f(R)` evaluates to a value $\mathtt{v}'$ such that $\mathtt{Q}(\mathtt{v}')$ holds.

To show: `P(L)`, i.e. `f(x::R)` evaluates to a value `v` such that `Q(v)` holds.
Proof: show this, using `P(R)`, the assumption that `n>0`, and the function definition.

(v) By (iii) and (iv) it follows that (THM) holds.

Comment: we deliberately used different "meta-variables" $\mathtt{v}, \mathtt{v}_0, \mathtt{v}'$ in the different subtasks above, to avoid any confusion; if we had written the same `v` everywhere you might have gotten mixed up. You don't have to label the theorem or the induction hypothesis, but doing so makes it easy to refer to them in your proof.

In practice, there may be more than one reasonable way to write down a theorem statement, or to state an induction hypothesis. For example, it may be convenient to write an equation like `f(L)=v` instead of saying "`f(L)` evaluates to `v`". This is especially common when we are trying to prove that `f(L)` evaluates to a specific value (that is expressible explicitly and usually depends on `L`). Recall that since the return type of `f` here is `int`, saying that `f(L)` evaluates to `v` means the same as saying that "the value of `f(L)` is `v`", or just $\mathtt{f}(\mathtt{L}) = \mathtt{v}$.

And if the function definition is not as simple as the assumptions we made above about `f`, for example if `f(L)` makes more than one recursive call, or has more than one explicit "base" clauses in which no recursive calls are needed, we'll need to adapt the proof strategy and use a more sophisticated form of induction (coming soon!).

A simple list example

```
(* sum : int list -> int *)
fun sum (L:int list):int =
  case L of
       [ ] => 0
    | (x::R) => x + sum(R)
```

Your task: Prove that for all `L`, `sum L` returns the sum of the integers in `L`.

Notice that the definition of `sum` gives the value `sum(nil)` directly, without making any recursive calls; so `L=nil` is a "base" clause. Otherwise it is clear that `sum(x::R)` makes a recursive call to `sum(R)`. So this function definition fits the pattern above and we can adapt the proof template from above for list induction.

(i) The theorem to be proven:

(SUM): For all `L`, `sum`(`L`) returns the sum of the integers in `L`.

(ii) Proof: By list induction on `L`.

(iii) Base case: `L=nil`.
To show: `sum`(`nil`) = `0`.
Proof: This is clear from the function definition and the fact that there are no integers in the empty list, so its sum is 0.

(iv) Inductive case: `L` = `x` :: `R`.
Induction Hypothesis:

(IH): `sum R` returns the sum of the integers in `R`.

To show: with these assumptions, `sum`(`x` :: `R`) returns the sum of the integers in `x::R`.

Proof: Using the function definition for `sum`, plus obvious facts about addition, and the induction hypothesis (IH), we have

```
sum (x::R)
  = x + sum R  by definition of sum
  = x + k      where k is the sum of the integers in R,
               by IH
  = the sum of the integers in x::R
```

as required.

(v) By (iii) and (iv) it follows that (SUM) holds.

Notice that this result also shows that `sum` is a total function.

Incrementing along a list

Consider the function

```
addtoeach : int * int list -> int list
```

defined by

```
fun addtoeach (x:int, L:int list) =
   case L of
         [ ]  => [ ]
      | (y::R) => (x+y)::addtoeach(x,R)
```

Intuitively, we might say that `addtoeach(x, L)` returns a list obtained by incrementing each item in `L` by `x`. Even though the word "increment" is sometimes used with imperative connotations, we are *not* talking about an imperative or destructive operation here – we are doing functional programming. No list is being destroyed or modified. Rather, a new list is constructed and returned by the function call.

Prove that for all integers `x` and integer lists `L`,

```
sum(addtoeach(x, L)) = sum L + x * length L.
```

Since `sum` and `length` are total, this implies that `addtoeach` is total.

Prove that for all integers `x` and `y`, and all integer lists `L`,

```
addtoeach(x, addtoeach(y, L)) = addtoeach(x+y, L).
```

Inserting into a sorted list

We surely all know what it means to say that a list of integers is *sorted* (in non-decreasing order). For example, `[1,1,2,4,4,10]` is sorted, but `[3,2,1]` is not. Technically, a list is sorted if each item is less-than-or-equal-to the items that appear later in the list. Note the key facts that the empty list is sorted; a singleton list `[x]` is sorted; and if `y::L` is sorted and $\texttt{x} \leq \texttt{y}$, then `x::(y::L)` is sorted.

Let's assume we know what it means to say that one list is a *permutation* of another. For example, `[1,3,2]` is a permutation of `[3,2,1]`, but not of

`[1,3,3,2]`. Technically a permutation of a list is another list that contains the same items possibly in a different arrangement.

Consider the function `ins:int*int list -> int list` defined by:

```
fun ins (x, [ ]) = [x]
 |  ins (x, y::R) = if x>y then y::ins(x, R) else x::(y::R)
```

By the way, we can omit the parentheses in `x::(y::R)`, because ML treats the list-building "cons" operator `::` as right-associative.

We show first that for all integers `x` and all lists `L`, `ins(x, L)` evaluates to a permutation of `x::L`.

(i) The theorem to be proven:

> For all integer lists `L`, and all integers `x`,
> `ins`(`x`, `L`) evaluates to a permutation of `x::L`.

(ii) Proof: By list induction on `L`.

(iii) Base case: `L=nil`.
To show: for all integers `x`, `ins`(`x`, `nil`) evaluates to a permutation of `x::nil`.
Proof: This is clear from the function definition, because `ins(x, nil) =>* [x]` and this is obviously a permutation of `x::nil` (remember that `x::nil = [x]`).

(iv) Inductive case: `L` = `y` :: `R`.
Induction Hypothesis (IH):

> For all integers `x'`,
> `ins(x', R)` evaluates to a permutation of `x'::R`.

To show: with these assumptions, for all integers `x`, `ins(x, y::R)` evaluates to a permutation of `x::(y::R)`.

Proof: Using the function definition for `ins`, and (IH), we have

```
ins (x, y::R)
  = if x>y then y::ins(x,R) else x::(y::R)
```

Now we do a case analysis.

- Either `x>y`, in which case `ins(x,y::R)=y::ins(x,R)`. By IH, `ins(x,R)` evaluates to a permutation of `x::R`, say `P`. So `ins(x,y::R)` evaluates to `y::P`, which is a permutation of `y::(x::R)`. This is also a permutation of `x::(y::R)`, as required.
- Or `x` $\leq$ `y`, in which case `ins(x, y::R) = x::(y::R)` and this is clearly a permutation of `x::(y::R)`, as required.

We show next, *using the now established result that* `ins(x,L)` *returns a permutation of* `x::L`, that for all integers `x` and all sorted lists `L`, `ins(x, L)` evaluates to a sorted list.

(i) The theorem to be proven:

For all sorted integer lists `L`, for all integers `x`, `ins`$(\texttt{x}, \texttt{L})$ returns a sorted list.

(ii) Proof: By list induction on `L`.

(iii) Base case: `L=nil`.
To show: for all integers `x`, `ins`$(\texttt{x}, \texttt{nil})$ evaluates to a sorted list.
Proof: This is clear from the function definition, because `ins(x, nil) =>* [x]` and this is a sorted list.

(iv) Inductive case: `L` $=$ `y` :: `R`, a sorted list. Note that this implies that `R` is also a sorted list, and that `y` is less-than-or-equal-to every integer in `R`.
Induction Hypothesis:

(IH): For all integers `x`, `ins(x, R)` returns a sorted list.

To show: with these assumptions, for all `x`, `ins(x, y::R)` returns a sorted list.

Proof: Using the function definition for `ins`, and (IH), we have

```
ins (x, y::R)
  = if x>y then y::ins(x,R) else x::y::R
```

Now we do a case analysis.

- Either `x>y`, in which case `ins(x,y::R)=y::ins(x,R)`. By IH, `ins(x,R)` evaluates to a sorted list, say `S`. By the previous proof, `S` is a permutation of `x::R`. Since `x>y` and `y` is less-than-or-equal-to every integer in `R`, we see that `y::S` is sorted. So we have shown that `ins(x,y::R)` evaluates to `y::S`, a sorted list.
- Or `x` $\leq$ `y`, in which case `ins(x, y::R)=x::(y::R)`. And this list is sorted, because (by assumption) `y::R` is sorted and `x` is less-than-or-equal-to `y`, which is $\leq$ every integer in `R`. (We're appealing here to the fact that the $\leq$ ordering on integers is *transitive*, i.e. if `x` $\leq$ `y` and `y` $\leq$ `z`, then `x` $\leq$ `z`).

It should now be an obvious consequence of these two results that for all integers `x` and all sorted lists `L`, `ins(x, L)` evaluates to a sorted permutation of `L`. (Actually for an integer list `L` there is exactly one permutation of `L` that is sorted, so it's not just "a" sorted permutation of `L`, but "the".)

More about total functions

Recall: A function `F : t1 -> t2` is *total* iff for all values `v1` of type `t1`, `F v1` evaluates to a value (of type `t2`). Clearly this implies also that `F` has a value (or is itself a value).

Totality can be a convenient property (sometimes absolutely necessary) to assume in specifications, especially when dealing with a function that takes another function as an argument, e.g

```
(* apply_to_all : ('a -> 'b) * 'a list -> 'b list  *)
fun apply_to_all (f, [ ]) = [ ]
|   apply_to_all (f, x::L) = (f x) :: apply_to_all (f, L)

(* REQUIRES  f total  *)
(* ENSURES   apply_to_all (f, [x1,..., xn]) = [f x1, ..., f xn] *)
```

This is a valid spec, but so is this one, that doesn't assume `f` is total:

```
(* REQUIRES  f x1, ..., f xn all valuable  *)
(* ENSURES   apply_to_all (f, [x1,..., xn]) = [f x1, ..., f xn] *)
```

And even if we relax the assumption completely we still get a valid spec:

```
(* REQUIRES true  *)
(* ENSURES  apply_to_all (f, [x1,..., xn]) = [f x1, ..., f xn]  *)
```

It's still valid, because if `f` isn't valuable, or one of the `f xi` isn't valuable, both sides of the equation will be expressions that fail to terminate, so they are still "equal". However, if with different choice of ENSURES we may truly need to REQUIRE totality:

```
(* REQUIRES  f total  *)
(* ENSURES   apply_to_all total  *)
```

See why?
The point is that you shouldn't get into the habit of always assuming totality: it may not be needed as an assumption, depending on the guarantee you want.

Another rather important thing to note is that sometimes when you prove that a function satisfies a spec you are actually simultaneously proving that the function is total, even if the spec doesn't say so out loud. (We've already had examples like this – go back and find some!)

And one more thing. Sometimes, in order to prove that a function is total, you need to prove something more explicit about its applicative behavior. For example, consider

```
fun F(x:int):int = case Int.compare(x, 0) of
                              LESS => F(F(x+1))
                            | EQUAL => 0
                            | GREATER => F(F(x-1))
```

To prove `F` is total, you need to prove (by induction on $|x|$, the absolute value of `x`) that for all `x` : `int`, `F(F x) = 0`. Go ahead an do it! It's a good exercise. You'll see how the induction hypothesis gives you just enough information for the proof to work. By the way, you can't just prove that `F` is total by induction (can't induct on function values) and you can't prove by induction on |`x`|)that for all `x` : `int`, `F x` evaluates to a value; if you try this, you won't be able to justify the inductive step [4]. Try it and see that you get stuck!

[4]Just knowing that `F(x+1)` evaluates to a value isn't enough to show that `F(F(x+1))` evaluates to a value.

5 Closing remarks

It may seem that we are (and will continue to be) a bit pedantic, trying to be precise and accurate with our definitions, and our constant fussiness about "values". Unfortunately we really need to do this. (After all, ML is a call-by-value language, so the emphasis on values should seem justified!) There is a prevailing tendency to (over-)use the same word or terminology for many different purposes — this is commonly done in math and (especially) in the real world! (It can be a matter of convenience, but if misused may cause confusion.) So there's a good reason for our attempted precision. In math one normally uses the word "function" to mean what we call a "total function"; mathematicians would probably refer to ML functions as *partial* functions, reserving the positive-sounding adjective *total* for those whose application (to a value) always terminates. (A slightly weird consequence of math usage is that in math a partial function isn't a function with the property of being partial, and a total function is a partial function that isn't really "partial".)

Don't worry if you don't follow my philosophical musings. In case the nomenclature is getting confusing here is a summary of the various kinds of "functions" in our functional programming universe, and some connections between them:

- ML function expressions
  Expressions of type `t1 -> t2`, which may or may not evaluate to a value (of the same type).

- ML function values
  Values of type `t1 -> t2`, also known as *abstractions*, of the syntactic form `fn (x:t1):t2 => e`, the type annotations being optional.

- An ML function *denotes* a partial function from values of type `t1` to values of type `t2`. This partial function is "undefined" when applied to an argument value for which the ML function fails to terminate.

- Some ML functions are *total*, and actually denote a *total* function from values of type `t1` to values of type `t2`.

6 Self-test

Even if you don't write out complete answers, it's worth sketching solutions. Understand which inductive methods are or are not applicable. Learn to recognize what methods are best suited to the task at hand.

1. Prove that for all $\mathtt{n} \geq 0$, $\mathtt{bar}(\mathtt{n}) = \mathtt{0}$, where `bar:int->int` is given by

```
fun bar (n:int):int =
    if n <= 1 then 0 else bar(n-2)
```

2. Consider the function `exp':int * int -> int` defined by:

```
fun exp' (n:int, a:int):int = if n=0 then a else exp' (n-1, 2*a)
```

   Prove by simple induction on `n` that for all $\mathtt{n} \geq 0$,
   for all integers `a`, $\mathtt{exp'(n,a)} = \mathtt{a} * 2^{\mathtt{n}}$.

3. Consider the function `f` of type `int -> int` given by

```
fun f 0 = 1
 |  f 1 = 2
 |  f n = 3 - f(n-1)
```

   Prove that for all $\mathtt{n} \geq 0$, $\mathtt{f}(\mathtt{n}) = (\mathtt{n} \ \mathtt{mod}\ \mathtt{2}) + \mathtt{1}$.

4. Consider the function `fact:int->int` given by:

```
fun fact n = if n=0 then 1 else n * fact(n-1)
```

   For $\mathtt{n} \geq 0$ let $\mathtt{n}!$ be to the product of the integers $\mathtt{1}, \ldots, \mathtt{n}$; when `n` is `0` this product is taken to be `1`.

   (i) Prove *by simple induction on* `n` that for all $\mathtt{n} \geq 0$, $\mathtt{fact}(\mathtt{n}) = \mathtt{n}!$.

   (ii) Prove *by strong induction on* `n` that for all $\mathtt{n} \geq 0$, $\mathtt{fact}(\mathtt{n}) = \mathtt{n}!$.

   Comment on the differences between these two proofs.

5. Consider the function `f:int * int -> int` defined by:

```
fun f(0,n) = n
 |  f(m,0) = m
 |  f(m,n) = f(m,n-1) + f(m-1,n)
```

   Prove that for all $\mathtt{m}, \mathtt{n} \geq 0$, $\mathtt{f}(\mathtt{m}, \mathtt{n})$ terminates.
   (HINT: You cannot use induction on `m`, and you cannot use induction on `n`. Those methods won't work – try it and see. Instead, you can use induction on $\mathtt{m} * \mathtt{n}$, but you must justify why this works.)

6. Prove that the function `even:int -> bool` given by

```
fun even 0 = true
 |  even 1 = false
 |  even n = even(n-2)
```

   satisfies the following property:

   > For all integer values $\mathtt{n} \geq 0$, $\mathtt{even}(\mathtt{n})$ returns `true` if $\mathtt{n}\ \mathtt{mod}\ 2 = 0$ and returns `false` if $\mathtt{n}\ \mathtt{mod}\ 2 = 1$.

7. For each of the following specifications, which (if any) are satisfied by the `ins` function? No need to prove each case, but in any case where you think the answer is NO please explain briefly.

   (i)
```
(* REQUIRES true *)
(* ENSURES ins(x, L) returns a list value S                  *)
(* such that there are lists A and B for which S = A@(x::B).  *)
```

   (ii)
```
(* REQUIRES true *)
(* ENSURES ins(x, L) returns a list value S                  *)
(* such that there are lists A and B for which S = A@(x::B)   *)
(* and every item in A is less than x.                       *)
```

   (iii)
```
(* REQUIRES true *)
(* ENSURES ins(x, L) returns a list value that contains x. *)
```

   (iv)
```
(* REQUIRES L is sorted *)
(* ENSURES ins(x, L) returns a list value S                  *)
(* such that there are lists A and B for which S = A@(x::B).  *)
```

8. Consider the function `upto:int * int -> int list` given by

```
fun upto(i:int, j:int):int list = if i>j then [ ] else i::upto(i+1,j)
```

   - Show that for all integer values `i` and `j`, `upto(i,j)` terminates. (Hint: it's obvious when `i>j`. For `i` $\leq$ `j` use induction on the value of `j-i`.)
   - Prove, using equational reasoning, that for all integers `i` and `j` such that `i` $\leq$ `j`, `length(upto(i,j)) = (j-i)+1`.
   - Prove, using evaluational reasoning, that for all integers `i` and `j` such that `i` $\leq$ `j`, `length(upto(i,j)) =>* (j-i)+1`.

9. Consider the function

```
append:int list * int list -> int list
```

   defined by

```
fun append(A:int list, B:int list):int list =
  case A of
       [ ]    => B
   |  (x::A') => x:: append(A', B)
```

   Prove that for all integer lists `A` and `B`, there is a list `L` such that

```
append(A, B) =>* L and length(L) = length(A)+length(B).
```

   Using equational reasoning, show that for all integer lists `A` and `B`,

```
length(append(A, B)) = length(A) + length(B).
```

10. The built-in infix append operator of ML is `@`. For all integers `x` and all integer lists `L` and `R`, the following equations hold:

```
   [ ]@R = R
(x::L)@R = x::(L@R)
```

   (Remember that an equation like this means that the two expressions evaluate to the same value.) Using these facts, prove that for all integer lists `A` and `B`, `append(A, B) = A@B`.