15-150 Fall 2013
Lecture 3
Specifications, proofs, and induction

Stephen Brookes

Tuesday, September 3

1 Overview

In this lecture we focus on functions from (tuples of) integers to integers. The ideas and techniques generalize, as we will see later.

We introduce *induction* techniques for proving properties of functions and for proving that code terminates. We begin with "simple" or "mathematical" induction and progress 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". They are actually special cases of a more general notion 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 begin with well-founded orderings in their full generality. For the moment it is enough to remember 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 $0 < 1 < \cdots < n-1 < n$.

By starting with the simple and gradually working up to the more general, we hope that you will find it easier to get used to this kind of formal reasoning.

NOTE: Examples discussed in these notes may not correspond exactly with those from class. As an exercise, re-work the classroom examples using the techniques as explained here.

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) Let $n > 0$ and assume that $P(n-1)$ holds. Show that $P(n)$ also holds, under these assumptions.

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[1]. 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 mathematical induction

(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

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

[1]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.

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:

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

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

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) Use simple induction (with base case `0`) to prove $\forall n \geq 0.\, \texttt{zero}(n) = 0$.

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

(c) Use simple induction with base cases 0 through 42 to prove this.

Comment on the relative ease or difficulty of the three choices of k.

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 $\texttt{n} \geq 0$, `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 $\texttt{v}_0$ such that `Q(`$\texttt{v}_0$`)` 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 $\texttt{v}'$ such that `Q(`$\texttt{v}'$`)` 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 $\mathtt{f(n)} = \mathtt{v}$.

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);
```

Your task: Prove that for all $n \geq 0$, `exp` $n = 2^n$.

Notice that the definition of `exp` gives the value `exp(0)` directly, without making any recursive calls; so `n=0` is a "base" clause. Otherwise it is 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: $\texttt{exp}(0) = 2^0$.
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: $\texttt{n} > 0$.
Induction Hypothesis:

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

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

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* $2^{\texttt{n}}$ instead of *the numeral* $2^{\texttt{n}}$, and this is better since $2^{\texttt{n}}$ 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 $2^{\texttt{n}}$" 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 $2^{\texttt{n}}$.

(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 $2^{\texttt{n}-1}$.

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

```
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 IH
   =>* 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.

Exercises

(a) Consider the function `exp'` defined by:

```
(* exp' : int * int -> int *)
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`, $\mathtt{exp'(n,a)} = \mathtt{a} * 2^{\mathtt{n}}$.

(b) 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 $\mathtt{n} \geq 0$, $\mathtt{f(n)} = (\mathtt{n\ mod\ 2}) + 1$.

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 $\texttt{n} \geq 0$, `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 $\texttt{v}_i$ such that $\texttt{Q(v}_i\texttt{)}$ 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 $\texttt{0} \leq \texttt{m} < \texttt{n}$,
`f(m)` evaluates to a value $\texttt{v}'$ such that $\texttt{Q(v}'\texttt{)}$ 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 $\texttt{n} \geq 0$, $\texttt{fib(n)} = \frac{1}{\sqrt{5}}(\phi^{n+1} - \psi^{n+1})$.

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 $\mathtt{fib}(0) = 1 = \frac{1}{\sqrt{5}}(\phi^1 - \psi^1)$, 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 $\mathtt{fib}(1) = 1 = \frac{1}{\sqrt{5}}(\phi^2 - \psi^2)$, as required.

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

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

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

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

By (IH),

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

Hence, by the function definition,

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

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

Warning

Be careful! Make sure whenever you attempt an inductive proof that the problem obeys the general rules mentioned in the preamble above.

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 to use this proof method to deduce that "for all $\texttt{n} \geq 0$, `foo(n)` terminates"!

4 Exercises

Even if you don't write out complete answers to these exercises, it's worth studying them and sketching solutions.

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

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

3. Prove that for all `n` ≥ 0, `bar(n)` $= 0$, where `bar:int->int` is given by

```
fun bar (n:int):int =
    case n of
       0 => 0
     | 1 => 0
     | _ => bar(n-2);
```

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

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

   Let `n`! be defined to be 1 if `n` is 0, and equal to the product of the integers from 1 to `n` when `n` > 0. Note that we do *not* define `n`! to be the value of `n` $\times$ (`n` $- 1$)! when `n` > 0, even though this is a true fact; use the definition given above in your proof.

   (i) Prove *by simple induction on* `n` that for all `n` ≥ 0, `fact(n)` $=$ `n`!.

   (ii) Prove *by strong induction on* `n` that for all `n` ≥ 0, `fact(n)` $=$ `n`!.

   Comment on the differences between these two proofs.

5. Consider the function `comb:int*int -> int` given by:

```
fun comb(n, 0) = 1
 |  comb(n, m) = if m=n then 1 else comb(n-1, m) + comb(n-1, m-1);
```

Prove (by a suitably chosen inductive method) that
for all $\texttt{n} \geq 0$ and all $\texttt{m}$ such that $0 \leq \texttt{m} \leq \texttt{n}$,

$$\texttt{comb}(\texttt{n},\texttt{m}) = \frac{\texttt{n}!}{\texttt{m}!(\texttt{n}-\texttt{m})!}.$$

Don't get confused by our use of mathematical division symbols: the programming language notation for integer division is `div`, but we use the more standard math notation $\frac{a}{b}$ instead of a `div` b. The function definition doesn't even mention factorials or division anyway! You can rely on standard properties of integer arithmetic in your proof.

6. 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 $\texttt{m},\texttt{n} \geq 0$, $\texttt{f}(\texttt{m},\texttt{n})$ 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 $\texttt{m} * \texttt{n}$, but you must justify why this works.)

7. 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 $\texttt{n} \geq 0$, `even`$(\texttt{n})$ returns `true` if $\texttt{n} \bmod 2 = 0$ and returns `false` if $\texttt{n} \bmod 2 = 1$.

8. The logarithm (base 2) of a positive integer n is defined to be the unique integer k such that $2^k \leq n < 2^{k+1}$. (When n is positive such a k is guaranteed to exist, because $2^0 = 1$, and the powers of 2 form a strictly increasing sequence of integers: $2^0 < 2^1 < 2^2 < \cdots$, so n must lie in one of the "gaps" or actually be a power of 2.)

Clearly it follows from this definition that 0 is the logarithm of 1, because $2^0 = 1 < 2^1 = 2$.

Note that for $n > 1$ we have $1 \leq n$ `div` $2 < n$. And if $n > 1$ and k is the logarithm of n, we have $k \geq 1$ and

$$2^k \leq n < 2^{k+1},$$

so by dividing through by 2 and simplifying (2^k `div` $2 = 2^{k-1}$, etc) we also have

$$2^{k-1} \leq n \texttt{ div } 2 < 2^k,$$

which tells us that $k-1$ is the logarithm of n `div` 2. We can also reverse the argument to show that if $k-1$ is the logarithm of (n `div` 2) then k is the logarithm of n. Hence for $n > 1$ the logarithm of n is one more than the logarithm of (n `div` 2). These facts are useful in the rest of this question.

Let `log:int -> int` be the function defined by:

```
fun log (n:int):int = if n=1 then 0 else 1 + log(n div 2);
```

(a) Prove by complete induction on `n` that for all `n` ≥ 1, `log(n)` evaluates to the logarithm of `n`.
(It's OK to introduce the conventional math notation $log_2(n)$ for the base 2 logarithm of n, defined as above; but don't confuse log_2 with `log`. And don't use any properties of log_2 other than those mentioned above.)

(b) Explain why simple induction is an inappropriate strategy for proving this result.