ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

D. Basin and P. Müller

Formal Methods and Functional Programming

Solutions of Exercise Sheet 8: Motivation and Induction

Assignment 1

Here is a solution written in C:

```
int main(x, y)
{
  int z = 0;
  int v = 0;

  while (v < y) {
    z++;
    v = pow(z, x);
  }

  return z;
}

int pow(z, x)
{
  int r = 1;
  int i = x;

  while (i > 0) {
    r = r * z;
    i--;
  }

  return r;
}
```

We split our argument of correctness in various sub-arguments.

Functional correctness of power routine

Firstly, we prove that the routine `pow` is correct, *assuming that it terminates*. The argument for termination is orthogonal (independent) and follows below. In formal methods, a proof that *assumes* termination is called a *partial correctness* proof.

We aim to show that the routine `pow` computes the power z^x, where z is its first argument and x its second, assuming both x and z are positive.

The problem that we face is that, even when we assume termination, the calculating loop body may in principle be executed an arbitrary number of times n. One way to deal with this (which can also be used to reason about infinite loops, if termination is not a requirement) is to make an inductive argument on the number of loop iterations n so far. In particular, we find a condition on the state that is meant to hold after any finite number of iterations of the loop body, and then prove that it actually does, by induction on n. This condition is called the *loop invariant*.

In the base case, we considering 0 iterations, while the inductive case justifies that if the invariant holds after n iterations, then it holds after $n + 1$ iterations. This boils down to two proof obligations:

- Induction base: the code before the loop ensures that the invariant holds when the loop is reached
- Induction step: the code of the body *preserves* the invariant, assuming that the control condition of the loop is true.

We choose our loop invariant to be

$$r = z^{x-i} \ \wedge \ x \geq i \geq 0$$

and make the inductive proof as follows:

Induction base (n = 0): Since the variable r is initialised to 1, and the variable i to the value of x, we need to show that:

$$1 = z^{x-x} \ \wedge \ x \geq x$$

which follows by arithmetic.

Induction step: Assume the truth of the loop condition (otherwise there will be no $n + 1$-th iteration):

$$i > 0 \quad \text{(LC)}$$

and the loop invariant at the beginning of the execution of the n-th iteration (our inductive hypothesis):

$$r = z^{x-i} \ \wedge \ x \geq i \geq 0 \quad \text{(IH)}$$

Note that we use i, x here etc. to denote the values of the corresponding program variables at the *start* of the loop iteration in question.

We need to prove that the invariant holds at the end of the $n+1$-th iteration; i.e., for the new values of the variables after executing the loop body. By examining the loop body, we can see that the new value of i will be the old $i-1$, while the new value of r is the old $r*z$. So, in terms of the old values, to justify that the loop invariant now holds after the loop body execution, we need to justify that:

$$r * z = z^{x-(i-1)} \ \wedge \ x \geq i - 1 \geq 0 \quad \text{(PO)}$$

The first conjunct of (PO), is simplified (since $z > 0$) into $r = z^{x-i}$, which is proved by (IH).

We split the second conjunct of (PO) into two separate proof obligations: $x \geq i - 1$, proved by (IH), and $i - 1 \geq 0$, proved by (LC).

Correctness argument: We have now proved that when the loop terminates, the invariant is guaranteed to be true, i.e., $r = z^{x-i} \ \wedge \ x \geq i \geq 0$ holds. We also know that the negation of the loop condition is true at that point, i.e., $i \leq 0$. Together with the loop invariant, this implies $i = 0$, which implies $r = z^x$ as required.

Termination of power routine

Note that we also need to show that the execution of `pow` terminates for any valid pair of arguments. To do that, we observe that the expression i is always decreased by the loop body. We also observe that the loop condition does not allow this expression to become negative, and that it is initially non-negative. Thus, the loop terminates.

The technique that we used to prove termination is to find an expression that is decreased by the loop body, but cannot decrease forever (for example, because its value is always a natural number). This expression is i in our case, and it is called the *loop variant*.

Correctness of main routine

The loop invariant is now

$$z \geq 0 \ \wedge \ v = z^x \ \wedge \ (z > 0 \Rightarrow (z-1)^x < y) \wedge \ y \geq 0$$

Induction base: The proof obligation is:

$$0 \geq 0 \ \wedge \ 0 = 0^x \ \wedge \ (0 > 0 \Rightarrow (0-1)^x < y) \wedge \ y \geq 0$$

which is true (note that we assumed x to be strictly positive; not zero).

Induction step: The proof obligation (using also the correctness of the auxiliary routine) becomes (in terms of the old values of the variables):

$$z + 1 \geq 0 \ \wedge \ (z+1)^x = (z+1)^x \ \wedge \ (z + 1 > 0 \Rightarrow z^x < y) \wedge \ y \geq 0$$

In order to discharge the proof obligation we may assume the induction hypothesis

$$z \geq 0 \ \wedge \ v = z^x \ \wedge \ (z > 0 \Rightarrow (z-1)^x < y) \wedge \ y \geq 0 \quad \text{(IH)}$$

and the loop condition

$$v < y \quad \text{(LC)}$$

Only the third conjunct is non-trivial; we need to show that $z^x < y$. But we know $v = z^x$ (proved by IH) and so from the loop condition (LC), we get $z^x < y$.

Correctness argument: After the loop body terminates, we have:

$$z \geq 0 \ \wedge \ v = z^x \ \wedge \ (z > 0 \Rightarrow (z-1)^x < y) \wedge \ y \geq 0 \wedge \ y \leq v$$

If $z = 0$, then we have $v = 0$ and so $y = 0$, and therefore the returned value 0 is correct.

If $z > 0$, then we have $(z-1)^x < y \leq z^x$, which implies:

$$z - 1 < \sqrt[x]{y} \leq z$$

as required.

Termination: The loop variant is $\max(y - v, 0)$, and it is always decreased by the loop body. Furthermore, we know it starts off non-negative, and remains so each time the loop iterates. Therefore the loop terminates.

Assignment 2

For our convenience, we name the two definition cases for `plus`

```
plus Zero n      = n                  (P1)
plus (Succ m) n  = Succ (plus m n)    (P2)
```

The proof is as follows:

Proposition: $\forall x, y, z \in$ `Nat`$\cdot$ `plus (plus` x y`)` z `= plus` x `(plus` y z`)`

Proof: Let $P(x) \equiv \forall y, z \in$ `Nat`$\cdot$ `plus (plus` x y`)` z `= plus` x `(plus` y z`)`.
We prove $\forall x \in$ `Nat` $\cdot P(x)$ by structural induction on x.

Case $x \equiv$ `Zero`:

To show: $P(x)$, i.e. $\forall y, z \in$ `Nat` $\cdot$ `plus (plus Zero` y`)` z `= plus Zero (plus` y z`)`.

We show it by showing '`plus (plus Zero` y`)` z `= plus Zero (plus` y z`)`' for arbitrary y and z.

```
plus (plus Zero y) z  =(P1)  plus y z
                      =(P1)  plus Zero (plus y z)
```

Case $x \equiv$ `Succ` x':

To show: $P(x)$, i.e.
$\forall y, z \in$ `Nat`$\cdot$ `plus (plus (Succ` x'`)` y`)` z `= plus (Succ` x'`) (plus` y z`)`.

IH: $P(x')$, i.e. $\forall y, z \in$ `Nat`$\cdot$ `plus (plus` x' y`)` z `= plus` x' `(plus` y z`)`

We show it by showing
`plus (plus (Succ` x'`)` y`)` z `= plus (Succ` x'`) (plus` y z`)`
for arbitrary y and z.

$$\begin{aligned}
\texttt{plus (plus (Succ } x'\texttt{) } y\texttt{) } z &\overset{P2}{=} \texttt{plus (Succ (plus } x' \texttt{ } y\texttt{)) } z \\
&\overset{P2}{=} \texttt{Succ (plus (plus } x' \texttt{ } y\texttt{) } z\texttt{)} \\
&\overset{IH}{=} \texttt{Succ (plus } x' \texttt{ (plus } y \texttt{ } z\texttt{))} \\
&\overset{P2}{=} \texttt{plus (Succ } x'\texttt{) (plus } y \texttt{ } z\texttt{)}
\end{aligned}$$

Assignment 3

We use the following labels for the definition cases:

```
plus Zero n      = n                (P1)
plus (Succ m) n  = Succ (plus m n)  (P2)

leq Zero n             = true     (L1)
leq (Succ m) Zero      = false    (L2)
leq (Succ m) (Succ n)  = leq m n  (L3)
```

The proof is:

Proposition: $\forall m, n \in$ `Nat`$\cdot$ `(leq (plus` m n`)` m `= true)` $\Rightarrow$ n `= Zero`

Proof: Let $P(m) \equiv \forall n \in$ `Nat`$\cdot$ `(leq (plus` m n`)` m `= true)` $\Rightarrow$ n `= Zero`.
We prove $\forall m \cdot P(m)$ by structural induction on m.

Case $m \equiv$ `Zero`:

To show:$P(m)$, i.e.
$\forall n \in$ `Nat`$\cdot$ `(leq (plus Zero` n`) Zero = true)` $\Rightarrow$ n `= Zero`

We show it by showing '`(leq (plus Zero` n`) Zero = true)` $\Rightarrow$ n `= Zero`' for arbitrary n.

We assume the antecedent of the implication, i.e.
`leq (plus Zero` n`) Zero = true` (AS1)
and need to prove the consequence, i.e. n `= Zero`.

We argue by contradiction: suppose instead that
$n \neq$ `Zero` (AS2).
Then, by the definition of `Nat`, n `= Succ` n' for some $n' \in$ `Nat`. However, we then find that:

$$\begin{array}{ll} \texttt{leq (plus Zero } n\texttt{) Zero} & \overset{P1}{=} \texttt{ leq (Succ } n'\texttt{) Zero} \\ & \overset{L2}{=} \texttt{ false} \end{array}$$

which contradicts our assumption (AS1). This contradiction shows that the assumption (AS2) was false, and we conclude $n =$ `Zero` as required.

Case $m \equiv$ `Succ` m':

To show: $P(m)$, i.e.
$\forall n \in$ `Nat` $\cdot$ `(leq (plus (Succ` m'`) ` n`) (Succ` m'`) = true)` $\Rightarrow$ n `= Zero`

IH: $P(m')$, i.e. $\forall n \in$ `Nat`$\cdot$ `(leq (plus` m' n`) ` m' `= true)` $\Rightarrow$ n `= Zero`

We show it by showing
`(leq (plus (Succ` m'`) ` n`) (Succ` m'`) = true)` $\Rightarrow$ n `= Zero`
for arbitrary n.

We assume
`leq (plus (Succ` m'`) ` n`) (Succ` m'`) = true`
and aim to deduce that n `= Zero`:

$$\begin{array}{ll} & \texttt{(leq (plus (Succ } m'\texttt{) } n\texttt{) (Succ } m'\texttt{) = true)} \\ \overset{P2}{\Rightarrow} & \texttt{(leq (Succ (plus } m' \texttt{ } n\texttt{)) (Succ } m'\texttt{) = true)} \\ \overset{L3}{\Rightarrow} & \texttt{leq (plus } m' \texttt{ } n\texttt{) } m' \texttt{ = true} \\ \overset{IH}{\Rightarrow} & n \texttt{ = Zero} \end{array}$$

Assignment 4

We would like to apply structural induction on the type of lists of integers. Using $P(l)$ as an induction hypothesis does not seem to be enough however. For example, assume $P(l)$ and try to prove $P(x : l)$

$$\texttt{sum } (x : l) \ = \ \texttt{sumaux } (x : l) \ 0 \ = \ ?$$

and we are already stuck! The induction hypothesis $P(l)$ does not say anything about `sumaux`, so we cannot use it.

Our failure to use $P(l)$ was to be expected: `sum` is not recursively defined. It is `sumaux` that is recursively defined, which means that the induction should apply to the definition of `sumaux`.

It is a case where we need to *generalize* the formula that we want to prove, to take into account all possible inputs to `sumaux` (formally this means that we prove a *stronger* property than what is expected). The formula to be proved is:

$$Q(l) \equiv \forall i \in \texttt{Int} \cdot \left(\texttt{sumaux}\ l\ i\ =\ i + \sum_{j=0}^{(\texttt{length}\ l)-1} l \,!!\, j \right)$$

We can now prove $\forall l \in \texttt{[Int]} \cdot Q(l)$ by structural induction on l.

Case $l \equiv$ `[]`: We need to prove:

$$\forall i \in \texttt{Int} \cdot\ \texttt{sumaux}\ \texttt{[]}\ i\ =\ i + \sum_{j=0}^{-1} \texttt{[]} \,!!\, j$$

where the summation over an empty range that appears here is equal to 0. The formula to be proven is equivalent to

$$\forall i \in \texttt{Int} \cdot\ \texttt{sumaux}\ \texttt{[]}\ i\ =\ i$$

which can be derived from the definition of `sumaux`, since the definition is given in turns of an arbitrary i.

Case $l \equiv (x : l')$: We need to prove

$$\forall i \in \texttt{Int} \cdot\ \texttt{sumaux}\ (x : l')\ i\ =\ i + \sum_{j=0}^{(\texttt{length}\ (x:l'))-1} (x : l') \,!!\, j$$

IH:

$$\forall i \in \texttt{Int} \cdot\ \texttt{sumaux}\ l'\ i\ =\ i + \sum_{j=0}^{(\texttt{length}\ l')-1} l' \,!!\, j$$

Let i be an arbitrary integer, and we prove:

$$\texttt{sumaux}\ (x : l')\ i\ =\ i + \sum_{j=0}^{(\texttt{length}\ (x:l'))-1} (x : l') \,!!\, j$$

By the definition of `sumaux` this is equivalent to

$$\texttt{sumaux}\ l'\ (x + i)\ =\ i + \sum_{j=0}^{(\texttt{length}\ (x:l'))-1} (x : l') \,!!\, j$$

We now apply our IH, where we instantiate the quantified i with $x+i$. This gives:

$$x + i + \left(\sum_{j=0}^{(\texttt{length}\ l')-1} l' !! j \right) \;=\; i + \sum_{j=0}^{(\texttt{length}\ (x:l'))-1} (x : l') !! j$$

or, equivalently (subtracting i on both sides):

$$x + \left(\sum_{j=0}^{(\texttt{length}\ l')-1} l' !! j \right) \;=\; \sum_{j=0}^{(\texttt{length}\ (x:l'))-1} (x : l') !! j$$

To prove the last formula, we can start from the right-hand side, as follows:

$$\begin{aligned}
\sum_{j=0}^{(\texttt{length}\ (x:l'))-1} (x : l') !! j &= (x : l') !! 0 + \left(\sum_{j=1}^{(\texttt{length}\ (x:l'))-1} (x : l') !! j \right) \\
&= x + \left(\sum_{j=0}^{(\texttt{length}\ (x:l'))-2} (x : l') !! (j+1) \right) \\
&= x + \left(\sum_{j=0}^{(\texttt{length}\ l')-1} l' !! j \right)
\end{aligned}$$

This concludes our structural induction proof.

Original Problem. The original problem is now proved as follows. Assume l is an arbitrary list. Our proof obligation is:

$$\texttt{sum}\ l \;=\; \sum_{i=0}^{(\texttt{length}\ l)-1} l !! i$$

By the definition of `sum`, this is equivalent to:

$$\texttt{sumaux}\ l\ 0 \;=\; \sum_{i=0}^{(\texttt{length}\ l)-1} l !! i$$

By the lemma that we have proved (specialized for $i = 0$):

$$0 + \left(\sum_{j=0}^{(\texttt{length}\ l)-1} l !! j \right) \;=\; \sum_{i=0}^{(\texttt{length}\ l)-1} l !! i$$

which is true.

Assignment 5

Useful Lemmas

Apart from Lemma 1 and Lemma 2, given in the exercise sheet, our proof uses the following lemmas:

Lemma 3: List concatenation is associative:

$$\forall a, b, c \in \texttt{[Int]} \cdot (a \texttt{++} b) \texttt{++} c \;=\; a \texttt{++} (b \texttt{++} c)$$

Lemma 4: Length of concatenation:

$$\forall a, b, c \in \texttt{[Int]} \cdot \texttt{length}(a \texttt{++} b) \;=\; \texttt{length}\ a + \texttt{length}\ b$$

Lemma 5: Length of comprehension:

$$\begin{array}{l}\forall l \in \texttt{[Int]}, P \in \texttt{Int} \to \texttt{Bool} \cdot \\ \quad \texttt{length}(\texttt{[x | x <- } l\texttt{, } P(\texttt{x})\texttt{]}) + \texttt{length}(\texttt{[x | x <- } l\texttt{, } \neg P(\texttt{x})\texttt{]}) \;=\; \texttt{length}\, l\end{array}$$

Lemma 6: Indexing concatenation:

$$\begin{array}{l}\forall a, b \in \texttt{[Int]}, p \in \texttt{Int}, k \in \mathbb{N} \cdot k < \texttt{length}\ a \;\Rightarrow\; (a \texttt{++} [p] \texttt{++} b)\,\texttt{!!}\,k \;=\; a\,\texttt{!!}\,k \\ \forall a, b \in \texttt{[Int]}, p \in \texttt{Int} \cdot (a \texttt{++} [p] \texttt{++} b)\,\texttt{!!}\,(\texttt{length}\ a) \;=\; p \\ \forall a, b \in \texttt{[Int]}, p \in \texttt{Int}, k \in \mathbb{N} \cdot k > \texttt{length}\ a \;\Rightarrow\; (a \texttt{++} [p] \texttt{++} b)\,\texttt{!!}\,k \;=\; b\,\texttt{!!}\,(k - \texttt{length}\ a - 1)\end{array}$$

Main Proof

First, we need to define what it means for a list l to be sorted. Our predicate *sorted* is defined as follows:

$$\mathit{sorted}(l) \;\stackrel{\text{def}}{=}\; \forall i, j \in \mathbb{N} \cdot i < j < \texttt{length}\ l \Rightarrow\; l\,\texttt{!!}\,i \leq l\,\texttt{!!}\,j$$

Apart from sortedness, we need the property that `qsort` l has the same length as l.

We are going to prove the property by *strong induction* (Noetherian induction) on the length of l. That is, we assume that

$$\begin{array}{l}\forall m \in \texttt{[Int]} \cdot \texttt{length}\ m < \texttt{length}\ l \\ \quad \Rightarrow\; \mathit{sorted}(\texttt{qsort}\ m) \;\wedge\; \texttt{length}\ m = \texttt{length}(\texttt{qsort}\ m) \quad (\mathsf{IH})\end{array}$$

and our proof obligation is now

$$\textit{sorted}(\texttt{qsort}\ l)\ \wedge\ \texttt{length}\ l = \texttt{length}(\texttt{qsort}\ l) \quad \text{(PO1)}$$

We split our proof into two cases. Note, that these two cases happen to coincide with the two constructors of our algebraic data type `[Int]`, but that the case distinction is done on the length of the list that is yielded by applying these constructors, and not on the constructors themselves. Remember, that we are doing induction on the length of l!

- $\texttt{length}(l) = 0 \Rightarrow l = \texttt{[]}$

  In this case, by the definition of `qsort`, and that of `length`, (PO1) becomes

  $$\textit{sorted}(\texttt{[]})\ \wedge\ 0 = 0$$

  By the definition of *sorted*, (PO1) is equivalent to a vacuous universal quantification, and therefore true.

- $\texttt{length}(l) > 0 \Rightarrow l = (p : xs)$

  In this case, we define:

  $$\begin{aligned} l_1 &= \texttt{[x|\ x<-}xs\texttt{,\ x}\leq p\texttt{]} \\ l_2 &= \texttt{[x|\ x<-}xs\texttt{,\ x}> p\texttt{]} \end{aligned}$$

  and observe that the definition of `qsort`, we have

  $$\texttt{qsort}(l)\ =\ \texttt{qsort}\ l_1\ \texttt{++}\ \texttt{[}p\texttt{]}\ \texttt{++}\ \texttt{qsort}\ l_2$$

  By Lemma 1, and the definition of `length`, we have for both $i = 1, 2$:

  $$\texttt{length}\ l_i \leq \texttt{length}\ xs = (\texttt{length}\ l) - 1 < \texttt{length}\ l$$

  Now (IH) gives us for both $i = 1, 2$:

  $$\textit{sorted}(\texttt{qsort}\ l_i)\ \wedge\ \texttt{length}(\texttt{qsort}\ l_i) = \texttt{length}\ l_i$$

  We break (PO1) into two proof obligations:

  $$\begin{aligned} &\textit{sorted}(\texttt{qsort}\ l) \quad \text{(PO1.1)} \\ &\texttt{length}(\texttt{qsort}\ l) = \texttt{length}\ l \quad \text{(PO1.2)} \end{aligned}$$

  and we perform the following computation:

$$\begin{aligned}
\texttt{length}(\texttt{qsort}\ l) &\overset{L3,L4}{=} \texttt{length}(\texttt{qsort}\ l_1) + \texttt{length}[p] + \texttt{length}(\texttt{qsort}\ l_2)\\
&\overset{IH}{=} \texttt{length}\ l_1 + \texttt{length}[p] + \texttt{length}\ l_2\\
&= \texttt{length}[p] + \texttt{length}\ l_1 + \texttt{length}\ l_2\\
&\overset{L5}{=} \texttt{length}[p] + \texttt{length}\ xs\\
&\overset{L4}{=} \texttt{length}(p : xs)\\
&= \texttt{length}\ l
\end{aligned}$$

which proves (PO1.2).

To prove (PO1.1), we use quantifier elimination. That is, we assume that i, j are arbitrary integers, such that

$$0 \le i < j < \texttt{length}\ l \quad \text{(AS)}$$

and prove:

$$l\,!!\,i \le l\,!!\,j \quad \text{(PO1.2a)}$$

We split into cases.

- Case 1: $j < \texttt{length}\ l_1$. By Lemma 6 and (AS), we have

  $$l\,!!\,i = l_1\,!!\,i \ \wedge\ l\,!!\,j = l_1\,!!\,j$$

  Together with (IH) (instantiate $m = l_1$) and (AS), this proves (PO1.2a).
- Case 2: $j = \texttt{length}\ l_1$. By Lemma 6 and (AS), we have

  $$l\,!!\,i = l_1\,!!\,i \ \wedge\ l\,!!\,j = p$$

  Lemma 2 proves (PO1.2a).
- Case 3: $i < \texttt{length}\ l_1 < j$. By Lemma 6, we have

  $$l\,!!\,i = l_1\,!!\,i \ \wedge\ l\,!!\,j = l_2\,!!\,(j - \texttt{length}\ l_1 - 1)$$

  Lemma 2 now gives us $l\,!!\,i \le p < l\,!!\,j$ which proves (PO1.2a).
- Case 4: $i = \texttt{length}\ l_1 < j$. By Lemma 6, we have

  $$l\,!!\,i = p \ \wedge\ l\,!!\,j = l_2\,!!\,(j - \texttt{length}\ l_1 - 1)$$

  Lemma 2 proves (PO1.2a).

– Case 5: $\texttt{length}\ l_1 < i < j$. By Lemma 6, we have

$$l\,!!\,i == l_2\,!!\,(i - \texttt{length}\ l_1 - 1)\ \wedge\ l\,!!\,j = l_2\,!!\,(j - \texttt{length}\ l_1 - 1)$$

(IH) (instantiate $m = l_2$) proves (PO1.2a).

By (AS), all cases are covered. This completes the proof.