Csci 2041. Homework 3.

Kris Swann

February 23, 2017

1 Question 1: Power function, over natural numbers

We first define our property as

$$P(n) := \forall x \in \mathbb{N}, \quad \texttt{power n x} = x^n.$$

And it is assumed that $\mathbb{N} = \{0, 1, 2, \dots\}$. And that in the call to `power n x`, both n and x are converted to the appropriate decimal form. We approach the solution via induction on n.

Base. n = 0.
P(0) = $\forall x \in \mathbb{N}$,
`power 0.0 x`
= 1.0 By definition of `power`.
= x^0 By properties of exponential arithmetic.
So then, the base case holds.

Hypothesis.
Suppose that for some n $\in \mathbb{N}$, P(n) = $\forall x \in \mathbb{N}$, `power n x` $= x^n$.

Step.
Suppose our hypothesis holds, then consider
P(n+1) = $\forall x \in \mathbb{N}$, `power (n+1) x`
= `x *. power n x` By definition of `power`.
= `x *.` x^n By the inductive hypothesis.
= $x \cdot x^n$ By how multiplication works in Ocaml.
= x^{n+1} By properties of exponential arithmetic.
And so then, $P(n) \implies P(n+1)$, so by induction, P(n) holds $\forall n \in \mathbb{N}$, as required. ■

2 Question 2: Power over structured members

First, we note that the principle of induction over `nat` is that for some property P,

$$\text{If } P(n) \implies P(\texttt{Succ n}) \text{ and } P(\texttt{Zero}) \text{ holds, then P(n) holds } \forall n \in \texttt{nat}$$

We define our property as

$$P(n) := \forall x \in \mathbb{N}, \quad \texttt{power n x} = x^{\texttt{toInt n}}.$$

Again we assume that x is converted to the appropriate floating point representation when it is passed into `power`. We proceed via induction on n.

Base. n = Zero.
 P(`Zero`) = $\forall x \in \mathbb{N}$,
 `power 0 x`
 = 1.0 By definition of power.
 = x^0 By properties of exponential arithmetic.
 = $x^{\texttt{toInt Zero}}$ By definition of `toInt`
So then, the property holds for n = `Zero`.

Hypothesis.
 For some n $\in$ `nat`, P(n) = $\forall x \in \mathbb{N}, \quad \texttt{power n x} = x^{\texttt{toInt n}}$ holds.

Step.
Suppose our hypothesis holds, then consider
 P(Succ n) = $\forall x \in \mathbb{N}$,
 `power (Succ n) x`
 = `x *. power n x` By definition of `power`.
 = $x \cdot \texttt{power n x}$ By how multiplication in OCaml works.
 = $x \cdot x^{\texttt{toInt n}}$ By the hypothesis.
 = $x^{\texttt{toInt n}+1}$ By exponential arithmetic.
 = $x^{\texttt{Succ n}}$. By definition of `toInt`.
So then, by induction P(n) holds $\forall n \in$ `nat`. ■

3 Question 3: Length of lists

First, we note that the principle of induction over lists is that for some property P,

$$\text{If } P(l) \implies P(h :: t) \text{ and } P(\texttt{[]}) \text{ holds, then P(n) holds } \forall n \in \texttt{'a list}$$

We define our property as

$$P(l) := \forall r \in \texttt{'a list}, \quad \texttt{length (l @ r) = length l + length r}.$$

We proceed via induction on l.

Base. `l = []`.
P(l) $= \forall r \in$ `'a list`
`length ([] @ r)`
= `length r`   By properties of list appending in OCaml.
= 0 + `length r`   Since 0 is the additive identity.
= `length [] + length r`   By definition of `length`.
So then, the property holds for `l = []`.

Hypothesis.
For some $l \in$ `'a list`, P(l) $= \forall r \in$ `'a list` `length (l @ r)` holds.

Step.
Suppose our hypothesis holds, then consider for some $h \in$ `'a`
P(`h::l`) $= \forall r \in$ `'a list`,
`length ((h::l) @ r)`
= `length (([h] @ l) @ r)`   By the second provided property of lists.
= `length ([h] @ (l @ r))`   By the first provided property of lists.
= `length (h::(l @ r))`   By the basic properties of lists in OCaml.
= `1 + length (l @ r)`   By definition of `list`.
= `1 + length l + length r`   By the hypothesis.
= `length (h::l) + length r`   By definition of `list`.
So then, by induction since $P(l) \implies P(h :: l)$, then by induction, P(l) holds $\forall l \in$ `'a list`. ■

4 Question 4: List length and reverse

We define our property as

$$P(l) := \texttt{length (reverse l) = length l}$$

We proceed via induction on l.

Base. `l = []`.
P(`[]`) =
`length (reverse [])`
= `length []` By definition of `reverse`.
So then, the property holds for `l = []`.

Hypothesis.
For some $l \in$ `'a list`, P(l): `length (reverse l) = length l` holds.

Step.
Suppose our hypothesis holds, then consider for some $h \in$ `'a`
P(`h::l`) =
`length (reverse (h::l))`
= `length (reverse l @ [h])` By definition of `reverse`.
= `length (reverse l) + length [h]` By the property proven in 3.
= `length l + length [h]` By the hypothesis.
= `length [h] + length l` By assosiativity of addition.
= `length ([h] @ l)` By the property proven in 3.
= `length (h::l)` By the basic properties of lists in OCaml.
So then, by induction, P(l) holds $\forall l \in$ `'a list` as required. ■

5 Question 5: List reverse and append

We define our property as

$P(l1) := \forall l2 \in$ `'a list`, `reverse (append l1 l2) = append (reverse l2) (reverse l1)`

We proceed via induction on l1.

Base. `l1 = []`.
P(`[]`) $= \forall l2 \in$ `'a list`,
`reverse (append [] l2)`
= `reverse l2` By definition of `append`.
= `append (reverse l2) []` By Lemma 5.1.
= `append (reverse l2) (reverse [])` By definition of `reverse`.
So `P([])` holds.

Hypothesis.
For some $l1 \in$ `'a list`, $P(l1) := \forall l2 \in$ `'a list`,
`reverse (append l1 l2) = append (reverse l2) (reverse l1)` holds.

Step.
Suppose our hypothesis holds, then consider for some $h \in$ `'a`
P(`h::l1`) $= \forall l2 \in$ `'a list`
`reverse (append (h::l1) l2)`
= `reverse (h::(append l1 l2))` By definition of `append`.
= `(reverse (append l2 l2)) @ [h]` By definition of `reverse`.
= `(append (reverse l2) (reverse l1)) @ [h]` By the hypothesis.
= `append (reverse l2) ((reverse l1) @ [h])` By Lemma 5.3.
= `append (reverse l2) (reverse (h::l2))` By definition of `reverse`.
So then, by induction, P(l1) holds $\forall l1 \in$ `'a list` as required. ■

5.1 Lemma 1.

We define our property as

$$P(l) := \texttt{reverse l = append (reverse l) []}$$

We proceed via induction on l.

Base. `l = []`.
`P([]) = reverse []`
= `[]` By definition of `reverse`.
= `append [] []` By definition of `append`.
= `append (reverse []) []` By definition of `reverse`.
So `P([])` holds.

Hypothesis.
For some $l \in$ `'a list`, P(l) := `reverse l = append (reverse l) []` holds.

Step.
Suppose our hypothesis holds, then consider for some $h \in$ `'a`

P(`h::l`) =
`reverse (h::l)`
= `reverse l @ [h]`   By definition of `reverse`.
= `(append (reverse l) []) @ [h]`   By definition of `append`.
= `append ((reverse l) @ [h]) []`   By Lemma 5.2.
= `append (reverse (h::l)) []`   By definition of `reverse`.
So then, by induction, P(l) holds $\forall l \in$ `'a list` as required. ■

5.2 Lemma 2.

We define our property as

$$P(l1) := \forall l2 \in \texttt{'a list}, \quad \texttt{(append l1 []) @ l2 = append (l1 @ l2) []}$$

We proceed via induction on l1.

Base. `l1 = []`.
`P([])` = $\forall l2 \in$ `'a list`,
`(append [] []) @ l2`
= `[] @ l2`   By the definition of `append`.
= `l2`   By the basic properties of list appending in OCaml.
= `append l2 []`   By the definition of `append`.
= `append ([] @ l2) []`   By the basic properties of list appnding in Ocaml.
So `P([])` holds.

Hypothesis.
For some $l1 \in$ `'a list`, $P(l1) := \forall l2 \in$ `'a list`, `(append l1 []) @ l2 = append (l1 @ l2) []` holds.

Step.
Suppose our hypothesis holds, then consider for some $h \in$ `'a`
P(`h::l1`) = $\forall l2 \in$ `'a list`
`(append (h::l1) []) @ l2`
= `h::(append l1 []) @ l2`   By definition of `append`.
= `h::((append l1 []) @ l2)`   By basic properties of lists in OCaml.
= `h::(append (l1 @ l2) [])`   By the hypothesis.
= `append (h::(l1 @ l2)) []`   By definiton of `append`.
= `append (h::l1 @ l2) []`   By basic properties of lists in OCaml.
So then, by induction, P(l1) holds $\forall l \in$ `'a list` as required. ■

5.3 Lemma 3.

We define our property as

$$P(l1) := \forall l2, l3 \in \texttt{'a list}, \quad \texttt{(append l1 l2) @ l3 = append l1 (l2 @ l3)}$$

We proceed via induction on l1.

Base. `l1 = []`.
`P([])` = $\forall l2, l3 \in$ `'a list`,

(append [] l2) @ l3
= l2 @ l3 By definition of append.
= append [] (l2 @ l3) By definition of append.
So P([]) holds.

Hypothesis.
For some $l1 \in$ 'a list, $\forall l2, l3 \in$ 'a list, (append l1 l2) @ l3 = append l1 (l2 @ l3) holds.

Step.
Suppose our hypothesis holds, then consider for some $h \in$ 'a
P(h::l1) = $\forall l2, l3 \in$ 'a list
(append (h::l1) l2) @ l3
= (h::(append l1 l2)) @ l3 By definition of append.
= h::((append l1 l2) @ l3) By basic properties of lists in OCaml.
= h::(append l1 (l2 @ l3)) By the hypothesis.
= append (h::l1) (l2 @ l3) By definition of append.
So then, by induction, P(l1) holds $\forall l \in$ 'a list as required. ■

6 Question 6: Sorted lists

We define our property as

$$P(l) := \forall e \in \texttt{'a}, \quad \texttt{sorted l} \implies \texttt{sorted (place e l)}$$

We proceed via induction on l.

Base. `l = []`.
`P([])` $= \forall e \in$ `'a`,
`sorted []`
$=$ `true` By definition of `sorted`.
$\implies$ `true` This must be the case since true must imply true, it cannot imply false.
$=$ `sorted e::[]` By definition of `sorted`.
$=$ `sorted [e]` By basic properties of lists in OCaml.
$=$ `sorted (place e [])` By definition of `place`.
So we see that `P([])` holds.

Hypothesis.
For some $l \in$ `'a list`, $\forall e \in$ `'a`, `sorted l` $\implies$ `sorted (place e l)` holds.

Step.
Suppose our hypothesis holds, then there are two cases, if `l = []`, then we have the base case, which we have proven to be valid. Othwerwise we have `l = x::xs`. Now consider for some $h \in$ `'a`, `l = x::xs`
P(`h::l`) $= \forall e \in$ `'a`
`sorted (h::l)`
$=$ `sorted (h::x::xs)` By our assumption that `l = x::xs`.
$=$ `h <= x && sorted (x::xs)` By definition of `sorted`.
$=$ `h <= x && sorted l` By our assumption that `l = x::xs`.
From this point there are two cases.
Case 1. `h > x`.
We continue on from where we left off in the equality,
= false && sorted l By evaluation of `h <= x` under our assumption that `h > x`.
= false By properties of short-circuiting of `&&`.
Then the implication holds since `false` $\implies A$ is always `true` regardless of the value of A (By logical properties of implications). So then we are done and `P(h::l)` holds in this case.
Case 2. `h <= x`.
We continue on from where we left off before Case 1 in the equality,
$=$ `true && sorted l2` By evaluation of `h <= x` under our assumption that `x <= x`.
$=$ `sorted l2` By basic properties of &&.
$\implies$ `sorted (place e l)` By the hypothesis.
$=$ `sorted (place e (h::l))` By Lemma 6.1.
So then, by induction, P(l) holds $\forall l \in$ `'a list` as required. ■

6.1 Lemma 1.

We define our property as P(l) only when $h \leq x$ and when `l = x::xs`.

$$P(l) := \forall e, h \in \texttt{'a}, \texttt{sorted (place e (h::l))} = \texttt{sorted (place e l)}$$

We proceed via a direct proof. There are three cases.

Case 1. `e < h`.
Then we must have that
`sorted (place e (h::l))`
= `sorted (e::h::l)` By definition of `place`.
= `e <= h && sorted (h::l)` By definition of `sorted`.
= `true && sorted (h::l)` By our assumption that `e < h`.
= `sorted (h::l)` By logical properties of `&&`.
= `sorted (h::x::xs)` By our assumption that `l = x::xs`.
= `h <= x && sorted (x::xs)` By definition of `sorted`.
= `true && sorted (x::xs)` By our assumption that `h <= x`.
= `e <= x && sorted (x::xs)` By our assumptions that `e < h <= x`.
= `sorted (e::x::xs)` By definition of `sorted`.
= `sorted (place e (x::xs))` By definition of `place`.
= `sorted (place e l)` By our assumption that `l = x::xs`.
So then we are done in this case.
Case 2. `e >= h, e < x`.
Then we must have that
`sorted (place e (h::l))`
= `sorted (h::(place e l))` By definition of `place`.
= `sorted (h::(place e (x::xs)))` By our assumption that `l = x::xs`.
= `sorted (h::e::x::xs)` By definition of `place`.
= `h <= e && sorted (e::x::xs)` By definition of `sorted`.
= `true && sorted (e::x::xs)` By our assumption that `e >= h`.
= `sorted (e::x::xs)` By logical properties of &&.
= `sorted (place e (x::xs))` By definition of `place`.
= `sorted (place e l)` By our assumption that `l = x::xs`.
So then we are done in this case.
Case 3. `e >= h, e >= x`.
Then we must have that
`sorted (place e (h::l))`
= `sorted (h::(place e l))` By definition of `place`.
= `sorted (h::(place e (x::xs)))` By our assumption that `l = x::xs`.
= `sorted (h::x::(place e xs))` By definition of `place`.
= `h <= x && sorted (x::(place e xs))` By definition of `sorted`.
= `true && sorted (x::(place e xs))` By our assumption that `e >= h`.
= `sorted (x::(place e xs))` By logical properties of `&&`.
= `sorted (place e (x::xs))` By definition of `place`.
= `sorted (place e l)` By our assumption that `l = x::xs`.
So then we are done in this case.

So then in any case, P(l) holds, so then we are done. ■.

7 Question 7: Sorted Lists

We are able to make the claim that `is_elem e (place e l)` is always true even if `l` is not sorted because `place e l` will position e within the resulting list so that every xi which precedes e will be less than e. Then when `is_elem` is looking through the list, $e > xi$ will be true until it encounters e. So the computation will find e even though the rest of the list is never even looked at. This is why the list need not be sorted for `is_elem e (place e l)` to be true for all `l`.

Next, we turn our attention to the question if we can prove `sorted (place e l)` without assuming that `sorted l`. In short, no. As a counterexample consider the case when `l = x::xs`, `e < x, and sorted l = false`.
Then we can simply evaluate `sorted (place e l)`
= `sorted (place e (x::xs))` By our assumption that `l = x::xs`.
= `sorted (e::x::xs)` By definition of `place`.
= `e <= x && sorted (x::xs)` By definition of `sorted`.
= `e <= x && sorted l` By our assumption that `l = x::xs`.
= `e <= x && false` By our assumption that `sorted l = false`.
= `false` By logical properties of `&&`.
So then we have found an instance when we cannot prove `sorted (place e l)` without assuming that `sorted l = true`.