Inductive Program Correctness

Consider the length function over lists:

```ocaml
let rec length (l: 'a list) : int =
  match l with
  | [] -> 0
  | _ :: l' -> 1 + length l'
```

We would like to prove the correctness of this function, namely, the following proposition.

**Claim 1 (Correctness of Length).** For all lists \( l \), \( \text{length} \ l \) produces the length of \( l \).

We can try to prove this claim via case analysis on the input to \( \text{length} \):

*Proof attempt.* Consider the possible shapes of \( l \):

- \( l = [] \), then \( \text{length} \ [] \rightarrow 0 \) which is the expected length of the empty list.
- \( l = x :: l' \), then \( \text{length} \ (x :: l') \rightarrow 1 + \text{length} \ l' \).

In the cons case of the proof, we need to prove \( 1 + \text{length} \ l' \) produces the length of the list. Intuitively, we know this is the case because we expect \( \text{length} \) to correctly produce the length of \( l' \), the tail of \( l \). To factor in \( x \)'s contribution to the length, we only need to add one to the length of \( l' \).

However, how do we formalize this intuition? We do so using *induction*, a proof principle that allows us to reason about recursively or inductively-defined structures.

**Inductive Reasoning**

One way that we can proceed with the proof above, is to perform case analysis on \( l' \). If we do so, we obtain the following cases:

- \( l' = [] \), then \( \text{length} \ [] \rightarrow 0 \) so \( 1 + \text{length} \ l' \rightarrow 1 \) which is the length of the list containing one element.
- \( y :: l'' \), then \( \text{length} \ (y :: l'') \rightarrow 1 + \text{length} \ l'' \). Plugging this result back into the original expression yields: \( 1 + \text{length} \ l' \rightarrow 1 + 1 + \text{length} \ l'' \).

At this point, we're stuck again except we've unfolded the function by one recursive call. If we keep going, we'll unfold the function an arbitrary number of times, always resulting in one more recursive call left to unfold. We know that this line of reason ends in the base case—when the list is empty—because we know
that our lists are always finite in length. However, because we do not know exactly what our list looks like, we don’t know in the proof how many times we need to unfold the function.

We need some alternative way of reasoning about the recursive call other than unfolding it an arbitrary number of times. Induction provides us with the means of reasoning about these recursive calls. A proof by induction follows by case analysis on a given object, e.g., a list. However, in the recursive case, we may utilize an inductive hypothesis to help us complete the proof in this case.

Concretely, when we use induction to prove our claim about the length function, we gain the following inductive hypothesis in the recursive case of the proof:

**Inductive Hypothesis (length)**. length 1' produces the length of 1'.

Our inductive hypothesis is simply the claim that we wish to prove instantiated to the tail of the list. Intuitively, our inductive hypothesis formalizes the intuition that we should “assume that our recursive call just works”. With this assumption, we can complete the proof, reproduced below:

**Proof.** We prove the claim by induction on l. Consider the possible shapes of l:

- l = [], then length [] ⟶ * 0 which is the expected length of the empty list.
- l = x :: l' and our inductive hypothesis states that length l' produces the length of l'. Then length (x :: l') ⟶ 1 + length l'. By our inductive hypothesis, we know that length l' computes the length of l'. Because l = x :: l' we obtain the length of l by adding one to this length to account for x.

The inductive proof is identical to our proof by cases except that we gain our inductive hypothesis as an assumption to help us finish the recursive case of the proof. This reasoning consists of (a) using our inductive hypothesis to reason about the recursive call and (b) arguing that the extra work done by the program on top of this recursive call leads to a correct result for the overall list. In our proof, we state our inductive hypothesis explicitly as a reminder to ourselves what it is as well as a note to the reader about what our assumptions are regarding the inductive hypothesis.

**Another Example: append**

As another example, consider the append function:

```plaintext
let rec append (l1: 'a list) (l2: 'a list) : 'a list =
  match l1 with
  | [] -> l2
  | x :: l1' -> x :: append l1' l2
```

append ought to produce a list that contains the elements of l1 followed by the elements of l2, but how do we formalize this property? Because it is difficult to state this property precisely, we’ll instead state a weaker proposition that implies this property:

**Claim 2 (Append Preserves Length).** \( \forall l1, l2. \text{length} (\text{append} \ l1 \ l2) = \text{length} \ l1 + \text{length} \ l2 \)

This property claims that the length of the output list from append is the sum of the lengths of the two input lists. This property alone does not imply that our function is correct, but if the property holds, it is difficult to come up with a reasonable program that has this behavior but is still incorrect\(^1\).

\(^1\)This is partially due to the fact that our lists are polymorphic. That is, the type of elements of our lists are abstract. Because of this, we can’t make up arbitrary values to put into our list; they must instead be drawn from l1 and l2 instead.
At a high-level, we know that append obeys this property because in the base case, the output is simply \( l_2 \) as \( l_1 \) is empty. In the recursive case, we know that if we are able to successfully extend the tail of \( l_1' \) to \( l_2 \), we can extend this to \( l_1 \) by consing \( x \) onto the front of the list.

We formalize this intuition with an appropriate formal proof of the property:

Proof. We prove the claim by induction on \( l_1 \). Consider the possible shapes of \( l_1 \):

- \( l_1 = [] \), then \( \text{length} (\text{append} \ [\,] \ 12) \rightarrow^* \text{length} 12 \) and \( \text{length} [] + \text{length} 12 \rightarrow^* \text{length} 12 \).
- \( l_1 = x :: l_1' \) and our inductive hypothesis states that \( \text{length} (\text{append} \ l_1' \ 12) = \text{length} \ l_1' + \text{length} 12 \). Then \( \text{length} (\text{append} \ (x :: l_1') \ 12) \rightarrow^* \text{length} (x :: \text{append} \ l_1' \ 12) \). From our inductive hypothesis, we know that \( \text{append} \ l_1' \ 12 \) produces a list of length \( \text{length} \ l_1' + \text{length} 12 \). Because \( 1 = x :: l_1' \), we know that we obtain \( 1 \) by consing one more element onto \( l_1' \), thus we know that \( \text{length} (x :: \text{append} \ l_1' \ 12) = \text{length} \ l_1 + \text{length} 12 \). □

Both our inductive proofs follow the given skeleton:

- We state that we are performing an induction on a particular list mentioned into the proposition.
- The proof proceeds by case analysis on the possible shapes of that list.
- The base case is usually trivial and proceeds as if we were performing normal case analysis with no induction.
- In the inductive case, we:
  - State our inductive hypothesis explicitly.
  - Evaluate the program to the point where a recursive call occurs.
  - Use our inductive hypothesis to reason about the recursive call.
  - Then argue that the additional work that we do in the inductive case allows us to conclude the proposition holds for the overall list.

We will generalize this inductive proof structure later as we encounter other inductive data types, but for now, we can use this skeleton as the basis for all of our inductive proofs.

A Note on Termination

How do we know that induction is sound? It turns out that we can think of induction, the proof principle, itself as a proposition. Formally stated, the inductive proof principle for lists is:

\[
\forall P, P([\,]) \rightarrow (\forall x, P(1) \rightarrow P(x :: 1)) \rightarrow \forall l. P(l)
\]

In English, this proof principle says:

- \( \forall P \): “for all propositions (over lists)”
- \( P([\,]) \): “if you show that the proposition holds for the empty list”
- \( \forall x, l. P(1) \rightarrow P(x :: 1) \): “and you show, given the proposition holds for an arbitrary list, that the proposition holds for that list extended with an arbitrary element”
- \( \forall l. P(l) \): “you may conclude that the proposition holds over all lists”.

3
When we say that induction is sound, we specifically mean that this logical proposition is provable. This has been done for us already, so we can just assume that the proof principle holds. However, there is one subtlety in the principle that is worth mentioning. Consider this bad implementation of the length function:

```ml
let rec length' (l:'a list): int =
  match l with
  | [] -> 0
  | _ :: l' -> length l
```

Rather than recurring on the tail of `l`, we recur on `l` itself. This, of course, leads to an infinite loop for any non-empty list, but look at this proof of correctness for this version of `length'`:

**Claim 3 (Length' is Correct).** *For all lists* `l`, *length’* `l` *produces the length of* `l`.

**Proof.** By induction on `l`. Consider the possible shapes of `l`:

1. If `l = []`, then `length' []` converges to `0` which is the expected length of the empty list.
2. If `l = x :: l'`, then `length' (x :: l')` converges to `length (x :: l')`. By our inductive hypothesis, we know that `length (x :: l')` produces the length of `l = x :: l'`, so we are done!

What is wrong with this proof? We erroneously assumed the inductive hypothesis applies to the call `length` `l` when it only applies to a recursive call to the tail of `l`! (And we conveniently hid this fact by not stating what we assumed our inductive hypothesis was—a good reason to state it explicitly in your own proofs!)

Intuitively, induction over lists is sound because we can only apply our inductive hypothesis to a *structurally smaller list* in our function. By doing so, we know that on every recursive call, the list we are performing induction over gets smaller and smaller, eventually ending in the base case.

Because of our pattern matching construct, it is obvious that the usual subject of our recursive calls, `l'` (where `l = x :: l'`), is structurally smaller than the input list `l`, so we won’t need to prove this fact explicitly. However, we need to be careful when we are performing our recursive calls on non-obvious arguments, e.g., a recursive call on the result of some helper function. In these cases, we would need to prove that the list is indeed structurally smaller than the input.

**Formality in Inductive Proof**

In this course, we have taken the viewpoint that proof is really an argument trying to convince a reader that a given proposition is true. Really, there is a spectrum of formality where a more informal proof hides all but the essential portions of the proof in the interest of brevity or clarity. In contrast, a more formal proof includes more details in the interest of completeness at the cost of readability.

As a concrete example, consider the following function over lists:

```ml
let rec stutter (l:'a list): 'a list =
  match l with
  | [] -> []
  | x :: l' -> x :: x :: stutter l'
```

which duplicates all the elements of the input list and a corresponding proposition about the function:
Claim 4 (Stutter Doubles Length). \( \forall l. \text{length} (\text{stutter } l) = 2 \times \text{length } l \).

Now let’s consider some proofs of this proposition, starting with the informal and ending with the most formal that we might consider writing in this course. Here is an informal proof:

Proof. By induction on \( l \).

This proof is hardly a proof at all! It only lists the proof strategy, but offers no other guidance as to how the proof proceeds. In many real-world scenarios, the proof in question might be trivial to write out in comparison to the rest of the prose that the proof sits in. This style of proof elides all the details of the proof and only offers how to start it if an interested reader wants to check the author’s work.

One level up from this are the high-level intuition arguments that we have made previously:

Proof. \text{stutter} doubles the length of \( l \) because in the recursive case, we add two copies of each element to the output list.

This style of proof gives us slightly more confidence that the proposition holds, although it elides important details such as the case analysis and how the program symbolically evaluates.

If we begin including these elements, we arrive at the formal inductive proof style we have been using in this reading so far:

Proof. By induction on \( l \). Consider the possible shapes of \( l \):

- \( l = [] \), then \( \text{stutter } [] \longrightarrow^* [] \) and \( 2 \times 0 = 0 \).
- \( l = x :: l' \) and our inductive hypothesis states that \( \text{stutter } l' = 2 \times \text{length } l' \). Then \( \text{stutter } (x :: l') \longrightarrow^* x :: x :: \text{stutter } l' \). By our inductive hypothesis, we know that \( \text{stutter } l' = 2 \times \text{length } l' \). For the additional element \( x \), we add two copies of \( x \) to the result, so the overall size of the list is \( 2 + 2 \times \text{length } l' = 2 \times (1 + \text{length } l') \) as desired.

Building on top of the intuitive proof, our formal inductive proof describes the possible cases in detail and checks that the function behaves as desired by performing symbolic execution, \ie, evaluation, of the function with the given inputs. It argues the correctness of the claim in terms of both the resulting program as well as the inductive hypothesis.

This is not the most formal style of proof we can write down. Indeed, we have skipped steps of evaluation by using \( \longrightarrow^* \) that we could include. Also, we could have also argued the inductive case in terms of pure program manipulation and rewrites, justifying each step in detail, for example:

\[
\begin{align*}
\text{length} (x :: x :: \text{stutter } l') &= 2 \times \text{length } l & \text{goal (1)} \\
= 2 + \text{length} (\text{stutter } l') &= 2 \times \text{length } l & \text{by definition of length (2)} \\
= 2 + 2 \times \text{length } l' &= 2 \times \text{length } l & \text{by inductive hypothesis (3)} \\
= 2 \times (1 + \text{length } l') &= 2 \times \text{length } l & \text{distributivity of multiplication (4)} \\
= 2 \times \text{length } l &= 2 \times \text{length } l & \text{by definition of length (5)}
\end{align*}
\]

This form of proof leaves even less to the imagination at the cost of burden on the proof writer.

To maintain a balance between formalities and head trauma, we’ll prefer the original formal proof above as our canonical “formal proof”. This level of detail gets across the main ideas of the proof while also demonstrating that you understand the mechanics of the processes involved.

5