Lecture 4 - Lists and Structural Induction

David M Kahn *

Summer 2022

1 Lists

Lists in SML are a built-in data type for storing arbitrary finite sequences of one type of data. Most languages have some sort of data type like this, like Java's Arrays and Python's lists. However, each language treats this sort of data type a little differently. For SML, this means:

- Everything in the list must be the same type. This keeps the type sytem clean and elegant.[1]
- There is no arbitrary access; no indexing into the list. In many languages you can write something like `lst[3]` to access the 4th element of the list `lst`, but not in SML. This is because index systems are more complicated than we want our basic data types to be. Instead, SML opts for a more naive data structure: the *linked list*. This simplicity will turn out to be beneficial when it allows us to make use of *structural induction*.

Type t `list` for any element type t (there is no type that is just `list` without a type parameter for the element)

Values

- `[]` the empty list; it may also be called "nil".
- `v1::v2::v3::...::vn::[]` a list of n other values "cons"-ed together; this may also be written as `[v1, v2, v3,...,vn]`

*Adapted from Michael Erdmann's Spring 2022 notes, in turn adapted from a draft of Frank Pfenning

[1] This retriction to a single type is not actually that restrictive, as we will see later in the course.

Expressions

- Make: The expressions that make lists include all list values, any use of the *cons* operation `::` which adds an element to the front of another list, or the append operation `@` which sticks two lists together
- Use: Casing is the preferred way to break up lists, like `case lst of [] =>` e_1`| hd::tl =>` e_2, since `[]` and `::` are also patterns. (`@` is not a pattern)

Typing

- Make:
  - `[]`:t `list` for any type t
  - e_1`::`e_2:t `list` if e_1:t and e_2:t `list`
  - e_1 `@` e_2:t `list` if e_1 and and e_2 are both typed t `list`
- Use: Lists can be used by breaking them up with case statements in the normal way. For example, `case` e_1 `of [] =>` e_2`| hd::tl =>` e_3:t if
  - e_1:t' `list`
  - e_2:t
  - e_3:t assuming `hd`:t' and `tl`:t' `list`

Reduction left-to-right for binary operators, as usual

Special Notes Unlike most binary operators which are left associative, cons `::` is right associative. So `a::b::c` implicitly gets parenthesized as `a::(b::c)`. If you pay attention to the typing of cons, this is the only associativity that makes sense.

2 Structural Induction

Structural induction is a special kind of induction that is performed on inductively structured data, to prove that a property holds for all such data. For data to be inductively structured, it must have at least one base value (like nil `[]`), and also some number of ways of constructing new values out of old ones (like cons `::`) – we call the collection of both of these the data type's *(inductive) constructors*. Because lists have such a set of constructors, this means that lists are a great target for structural induction. Later we will see that we can structurally induct over other data types too.

We can actually derive structural induction from our usual induction over natural numbers[2]. This can be done by having the number we induct on count

[2]We can also go the other direction, and derive induction over natural numbers from structural induction. To go in this direction, note that natural numbers may be structured with a base value of 0 and an inductive constructor of incrementation by 1 (successor).

the number of constructors used in the data type (plus one). Thus, for lists, the base case of 0 corresponds to the case for the empty list, and the inductive step going from n to $n+1$ corresponds to the step going from a list tl of length n to a list $hd::tl$ of length $n+1$. For lists, this is just induction on the list's length!

However, it is useful to shortcut this process, and simply consider inductive cases for each constructor directly. Each base value forms a base case, and each way of making new values forms a different inductive step[3]. For the type t `list`, this leaves us with the inductive principle:

for some predicate P, if

- P(`[]`) holds
- P($hd::tl$) holds assuming P(tl) for values $hd:t$ and $tl:t$ `list`

then P(lst) holds for all values $lst:t$ `list`

There also exist the same variants we had before for induction: strong induction and induction on a generalized predicate.

2.1 Example

Here is a nice proof example for showing how to use structural induction alongside other proof principles.

First, let us assume that the following is the implementation of `@`, and assume that this implementation is total[4]. Consider proving its totality yourself as an exercise!

```
1  (*
2   * @ : int list * int list -> int list
3   * REQUIRES: true
4   * ENSURES: a @ b ==> a list containing all and only
5   *  the elements of a followed by those b, maintaining order
6   *)
7  infix @
8  fun (lst1:int list) @ (lst2:int list) : int list =
9    (case lst1 of
10       [] => lst2
11     | hd::tl => hd :: (tl @ lst2)
12   )
```

Theorem 1. `@` *is associative, i.e., for* `int list` *values* a, b, c

$$(a \mathbin{@} b) \mathbin{@} c \cong a \mathbin{@} (b \mathbin{@} c)$$

[3]If there are only base values and no way of building bigger values from smaller, then structural induction is proof by cases!

[4]In this course, we will either tell you that a function is total, or you will have to prove it yourself.

Proof. We prove by structural induction over a.

Base Case $a = \texttt{[]}$

For this case, we want to show $([] \mathbin{@} b) \mathbin{@} c \cong [] \mathbin{@} (b \mathbin{@} c)$.

First we show that $([] \mathbin{@} b) \mathbin{@} c \cong v_1$ for some value v_1 by showing the reduction $([] \mathbin{@} b) \mathbin{@} c \implies v$, since extensional equivalence is closed under reduction.

$$\begin{aligned} ([] \mathbin{@} b) \mathbin{@} c &\implies b \mathbin{@} c && \text{clause 1} \\ &\implies v_1 \text{ (for some value } v_1) && \text{totality of @} \end{aligned}$$

Then we show that $[] \mathbin{@} (b \mathbin{@} c) \cong v_1$ as well, again via reduction.

$$\begin{aligned} [] \mathbin{@} (b \mathbin{@} c) &\implies [] \mathbin{@} v_1 \text{ (for the same } v_1 \text{ as above)} && \text{totality of @} \\ &\implies v_1 && \text{clause 1} \end{aligned}$$

Finally, because extensional equivalence is an equivalence relation, these two facts mean that $([] \mathbin{@} b) \mathbin{@} c \cong [] \mathbin{@} (b \mathbin{@} c)$, completing the case.

Inductive Step $a = hd :: tl$ for `int list` values hd, tl

For this case, we want to show $((hd :: tl) \mathbin{@} b) \mathbin{@} c \cong (hd :: tl) \mathbin{@} (b \mathbin{@} c)$, assuming that $(tl \mathbin{@} b) \mathbin{@} c \cong tl \mathbin{@} (b \mathbin{@} c)$.

First, we show that $((hd :: tl) \mathbin{@} b) \mathbin{@} c \cong hd :: ((tl \mathbin{@} b) \mathbin{@} c)$ via a sequence of reductions and extensional equivalences. Note that the extensional equivalence of the last line follows due the reduction that was ensured by totality.

$$\begin{aligned} ((hd :: tl) \mathbin{@} b) \mathbin{@} c &\implies (hd :: (tl \mathbin{@} b)) \mathbin{@} c && \text{clause 2} \\ &\implies (hd :: v_2) \mathbin{@} c \text{ (for some value } v_2) && \text{totality of @} \\ &\implies hd :: (v_2 \mathbin{@} c) && \text{clause 2} \\ &\cong hd :: ((tl \mathbin{@} b) \mathbin{@} c) && tl \mathbin{@} b \cong v_2 \end{aligned}$$

Then, we show that $(hd :: tl) \mathbin{@} (b \mathbin{@} c) \cong hd :: ((tl \mathbin{@} b) \mathbin{@} c)$ similarly as before.

$$\begin{aligned} (hd :: tl) \mathbin{@} (b \mathbin{@} c) &\implies (hd :: tl) \mathbin{@} v_1 \text{ (for some value } v_1) && \text{totality of @} \\ &\implies hd :: (tl \mathbin{@} v_1) && \text{clause 2} \\ &\cong hd :: (tl \mathbin{@} (b \mathbin{@} c)) && b \mathbin{@} c \cong v_1 \end{aligned}$$

Finally, using the inductive hypothesis that $(tl \mathbin{@} b) \mathbin{@} c \cong tl \mathbin{@} (b \mathbin{@} c)$, we can relate the above two conclusions as $hd :: ((tl \mathbin{@} b) \mathbin{@} c) \cong hd :: (tl \mathbin{@} (b \mathbin{@} c))$. Since extensional equivalence is an equivalence relation, this means we can conclude $((hd :: tl) \mathbin{@} b) \cong (hd :: tl) \mathbin{@} (b \mathbin{@} c)$, completing the case.

□

Note that we needed to use the totality of `@` in the above proof. Without it, we have no way of knowing that expressions like b @ c actually are valuable, and we need to turn them into values to use our reduction rules. We also made repeated use of referential transparency to swap around extensionally equivalent subexpressions.

2.2 A Mathematical Digression

To be more aware of what we are doing mathematically when we induct, it is key to realize that all forms of induction rely on the structure they are inducting over being the *least fixed point* of its constructors. For natural numbers, this means we take the the smallest set containing 0 and closed under adding 1, which is the standard collection of natural numbers. For integer lists, this means we take the the smallest set of values containing the empty list and closed under cons-ing another `int` onto the front, which is the standard collection of finite integer lists.

Note that if we added "infinity" to the above set of natural numbers, the set is still closed under its constructors – infinity plus 1 is still infinity. Likewise, adding infinite (`v1::v2::v3::v4::...`) or looped (`lst = v::lst`) "lists" to our set of list values would still give a set closed under the list constructors. However, both these sets are *larger* than the previous sets named, since they are strict supersets, and so are *not* what we consider when we induct.

Such a focus when inducting over SML lists is appropriate because the type of lists is an *inductive* type, as are most types that you will see. This means that the values of that type are only considered to be those in the smallest set closed under its constructors, which are the same sets that induction acts on. As a result, those infinite and looped "lists" are not actually lists, and it is perfectly OK that our proof technique of induction misses them.

Thus, we can say that the length function for a list given below is total, even though it would never terminate on an infinite or looped "list".

```
1  (*
2   * len: int list -> int
3   * REQUIRES: true
4   * ENSURES: len lst evaluates to the length of lst
5   *)
6  fun len lst =
7    (case lst of
8        [] => 0
9      | _::tl => 1 + len tl
10   )
```

Do not take it for granted that every type you see in the wild is inductive. For example, in OCaml, a cousin of SML, one can make looped lists. This can make it difficult to reason fully about some languages, since induction would not be sufficient. However, for right now with SML, we are only using inductive types.

Some languages also have features to allow treating the set of values for a type as the *greatest* fixed point of its constructors – such types are called *co*inductive types. These correspond to a dual proof technique to induction called *co*induction. You need not worry about this technique right now, but it is out there if you wanted to go beyond the course.

3 Tail Recursion

Tail recursion is an important optimization tool in the arsenal of a functional programmer. This is a way of ordering recursive calls so that the machine executing the code does not need to remember to do any additional computation after the recursive call returns. This has a number of consequences for the efficiency of the code.

3.1 Definitions

Tail Call Within the body of a function f, a function call is a tail call if f does not inspect or compute with the result of that call; instead that call is the "last" computation f does, and f returns the call's result.

Tail Recursion A function is tail recursive if every call it makes to a recursive function is a tail call, and each of those called recursive functions can themselves be considered tail recursive.

A prototypical template (but not the only possible template) for a tail recursive function is the following, where `e1` and `e2` are arbitrary expressions (that do not make additional function calls), and `p1` and `p2` are appropriate patterns.

```
1  fun f x =
2    (case x of
3        p1 => e1
4      | p2 => f (e2)
5    )
```

Notice how our reduction rules would cause `e2` to be computed *before* applying `f`, so the call to `f` is a tail call.

3.2 Machine View

While we do not normally care to think about the implementation details of the machine running our code, and it is not strictly-speaking necessary for the course, doing so for tail recursion can be enlightening.

The machine running our code must track the environment to resolve variables, and also something called the *return address* which tells the machine where in the code to return with any computed values. These are usually stored in an area of memory called the *call stack*. Every time a function is called, a new *frame* is usually added to call stack to store any new local variables that

call of the function uses, as well as the return address. When a function call would return a value, it simply stores that value in a return register, pops the code pointer back to its return address, deallocates its frame, and restores the environment stored in the previous frame.

However, for tail recursive functions, the machine can make an optimization. By definition, no computation is done with the return of the tail calls inside a tail recursive function – instead the value returned from the tail call is just itself returned by the function. This means the machine can forget the environment it would normally store in a stack frame, since no variables are needed if no computation will be done. This also means that the machine only needs to remember the single return address of original function caller – when the tail recursive function returns, by definition there is no computation remaining to do in any level of its recursion, so the next computation to occur will be outside of the tail recursive function, wherever it was originally called. Together, these two facts mean there is nothing that needs to be stored in the call stack frames, so the machine can save time and memory by just never allocating them. Instead, the machine can treat the tail recursion like an imperative loop, which is a code pattern it can execute efficiently.

3.3 Example

The following implementation `len` of the length function for lists is *not* tail recursive, because `len` is not called as a tail call – after the recursive call to `len` returns, the code performs the additional computation of adding one.

```
1  (*
2   * len : int list -> int
3   * REQUIRES: true
4   * ENSURES: len lst ==> the length of lst
5   *)
6  fun len (lst:int list) : int =
7    (case lst of
8        [] => 0
9      | _::tl => 1 + len tl
10   )
```

However, the following implementation `lenT` *is* tail recursive. Note how it achieves tail recursion by introducing an accumulator and making use of a helper function. This is a common pattern for achieving tail recursion.

```
1  (*
2   * lenHelp : int list * int -> int
3   * REQUIRES: true
4   * ENSURES lenHelp (lst, a) evaluates to a + length of lst
5   *)
6  fun lenHelp (lst:int list, acc:int) : int =
7    (case lst of
8        [] => acc
9      | _::tl => lenHelp (tl, acc + 1)
```

```
    )

  (*
   * lenT : int list -> int
   * REQUIRES: true
   * ENSURES: lenT lst ==> the length of lst
   *)
  fun lenT (lst:int list) : int =
    lenHelp (lst, 0)
```

We can visually see the effect of tail recursion by observing the execution trace of each implementation.

For the non-tail recursive `len`, the execution trace bulges out in the middle, since it must remember to increment for every element in the list. Recording all of these increments requires an amount of extra memory proportional to the length of the input list[5]. Just look how much of each line is devoted to something other than the list!

```
  len [1, 2, 3, 4, 5, 6, 7, 8]
==> 1 + len [2, 3, 4, 5, 6, 7, 8]
==> 1 + (1 + len [3, 4, 5, 6, 7, 8])
==> 1 + (1 + (1 + len [4, 5, 6, 7, 8]))
==> 1 + (1 + (1 + (1 + len [5, 6, 7, 8])))
==> 1 + (1 + (1 + (1 + (1 + len [6, 7, 8]))))
==> 1 + (1 + (1 + (1 + (1 + (1 + len [7, 8])))))
==> 1 + (1 + (1 + (1 + (1 + (1 + (1 + len [8]))))))
==> 1 + (1 + (1 + (1 + (1 + (1 + (1 + (1 + len []))))))))
==> 1 + (1 + (1 + (1 + (1 + (1 + (1 + (1 + 0)))))))
==> 1 + (1 + (1 + (1 + (1 + (1 + (1 + 1))))))
==> 1 + (1 + (1 + (1 + (1 + (1 + 2)))))
==> 1 + (1 + (1 + (1 + (1 + 3))))
==> 1 + (1 + (1 + (1 + 4)))
==> 1 + (1 + (1 + 5))
==> 1 + (1 + 6)
==> 1 + 7
==> 8
```

However, the tail recursive `lenT` orders its computations to perform the incrementation earlier using an accumulator. Notice that this time, the amount of each line devoted to something other than the list remains bounded by a constant, and each line never makes reference to anything from a previous recursive call's function body.

```
  lenT [1, 2, 3, 4, 5, 6, 7, 8]
==> lenHelp ([1, 2, 3, 4, 5, 6, 7, 8], 0)
==> lenHelp ([2, 3, 4, 5, 6, 7, 8], 0 + 1)
==> lenHelp ([2, 3, 4, 5, 6, 7, 8], 1)
==> lenHelp ([3, 4, 5, 6, 7, 8], 1 + 1)
```

[5]Specifically, each increment needs to be pointed to by a return address, and each return address needs its own stack frame.

```
 6 ==> lenHelp ([3, 4, 5, 6, 7, 8], 2)
 7 ==> lenHelp ([4, 5, 6, 7, 8], 2 + 1)
 8 ==> lenHelp ([4, 5, 6, 7, 8], 3)
 9 ==> lenHelp ([5, 6, 7, 8], 3 + 1)
10 ==> lenHelp ([5, 6, 7, 8], 4)
11 ==> lenHelp ([6, 7, 8], 4 + 1)
12 ==> lenHelp ([6, 7, 8], 5)
13 ==> lenHelp ([7, 8], 5 + 1)
14 ==> lenHelp ([7, 8], 6)
15 ==> lenHelp ([8], 6 + 1)
16 ==> lenHelp ([8], 7)
17 ==> lenHelp ([], 7 + 1)
18 ==> lenHelp ([], 8)
19 ==> 8
```