15-150 Lecture 4:
Lists

Lecture by Dan Licata and Ian Voysey

26 January 2012

1 What Are Lists?

A list of integers (a value of type `int list`) is either

`[]`, or

`x :: xs` where `x : int` and `xs : int list`.

And that's it!

`[]` is pronounced "nil" or "the empty list"; `::` is pronounced "cons". Therefore, the values of type `int list` are lists like these:

```
1 :: (2 :: (3 :: (4 :: [])))
```

This can also be written without the parens as

```
1 :: 2 :: 3 :: 4 :: []
```

because `::` is right-associative. For a particular list with a fixed number of elements, you can also write the elements inside of square-brackets separated by commas, as in

```
[1,2,3,4]
```

This is just a convenient notation from SML; it's short hand for the above form.

The operation on lists is case analysis (and recursion):

```
case l of
    [] => <branch1>
  | x :: xs => <branch2, with (x : int) and (xs : int list) in scope>
```

giving a branch for `[]` and a branch for `::`. In the body of the `::` branch, the variable `x` stands for the first element of the list, and the variable `xs` stands for the rest of the list.

Note that `::` is being used both to create lists, in value declarations, and take lists apart, in the part of the `::` branch to the left of `=>`.

2 Structurally Recursive Functions on Lists

2.1 length

Let's write a function compute the length of a list, which is to say the number of elements in the list, or the number of times that `::` appears in the structure of the list.

```
(* Purpose: compute the length of the list l *)
fun length (l : int list) : int =
    case l
     of [] => 0
      | x :: xs => 1 + length xs

val 5 = length (1 :: (2 :: (3 :: (4 :: (5 :: [])))))
```

The length of the empty list is `0`: it has no elements, and we know that because `::` does not appear in `[]`. If we assume that the recursive call is correct inductively, the length of `x :: xs` is one more than the length of the tail `xs`: we saw an instance of `::`, so we add 1 and recur.

Here's an example of `length` running on a small list:

```
length (1 :: (2 :: []))
|-> case (1 :: (2 :: [])) of [] => 0 | x :: xs => 1 + length xs
|-> 1 + length (2 :: [])
|-> 1 + (case 2 :: [] of [] => 0 | x :: xs => 1 + length xs)
|-> 1 + (1 + (length []))
|-> 1 + (1 + (case [] of [] => 0 | x :: xs => 1 + length xs))
|-> 1 + (1 + (0))
|-> 1 + 1
|-> 2
```

`length` transforms a list into an integer by walking down the list, replacing each element with `1`, and each `::` with `+`, and then evaluating up all of the additions once the recursion finishes. So above, we start with the int list expression

```
1 :: 2 :: []
```

and we get the int expression

```
1 +  1 +  0
```

2.2 sum

We can write a function to compute the sum of the elements in a list very similarly:

```
(* Purpose: sum the numbers in the list *)
fun sum (l : int list) : int =
    case l
     of [] => 0
      | x :: xs => x + sum xs

val 15 = sum [1,2,3,4,5]
```

`sum` transforms a list into an integer by walking down the list, leaving each element alone, replacing each `::` with a `+`, and evaluating up all of the additions once the recursion finishes.

So, for example, in the trace

```
sum (1 :: (2 :: []))
|-> case (1 :: (2 :: [])) of [] => 0 | x :: xs => x + sum xs
|-> 1 + sum (2 :: [])
|-> 1 + (case 2 :: [] of [] => 0 | x :: xs => x + sum xs)
|-> 1 + (2 + (sum []))
|-> 1 + (2 + (case [] of [] => 0 | x :: xs => x + sum xs))
|-> 1 + (2 + (0))
|-> 1 + 2
|-> 3
```

we transform the int list expression

```
1 :: 2 :: []
```

into the int expression

```
1 +  2 +  0
```

2.3 General Form

These two functions suggest a general template for a function on lists:

```
(* Purpose: operate on every element of a the list *)
fun f (l : int list) : A =
    case l of
        [] => < expression of type A >
      | x :: xs => < expression of type A in terms of x : int,
                                                    xs : int list,
                                             and (f xs) : A >
```

In the cons case, you can use the first element of the list `x`, the rest `xs`, and a recursive call to `f` on the smaller list `xs`. Note that any call to `f` on any expression equivalent to `l` that gets evaluated in the recursive case causes non-termination.

2.4 Raise Salaries

Let's do something a bit more practical. I have here a list of the TA's salaries and—since Dan isn't around to stop me—I'm going to give everyone a big raise.

```
(* Purpose: add amount to each salary in the list *)
fun rB (l : int list, amount : int) : int list =
    case l
     of [] => []
      | x::xs => (x + amount) :: rB (xs,amount)

val [1001, 1002, 1003, 1005] = rB ([1, 2, 3, 5], 1000)
```

Note that this function takes two arguments, like the `add` function you wrote in lab; see the end of the Lecture 3 notes if you need to review pairs. Also note that this function transforms integer lists into other integer lists, which is slightly more interesting than the examples above; this is a case where we've picked the type `A` in the general form to be the type `int list`.

3 Structural Induction on Lists

3.1 Proof of Fusion Property

Let's prove that two nested calls of `raiseBy` can be fused into one call without changing the result. Specifically, we'll prove Theorem 1. This is saying that we can optimize and traverse the list once instead of twice, and no program will be able to tell the difference.

Theorem 1 (Fusion)**.** *For all values* $\texttt{l} : \texttt{int list}, \texttt{a} : \texttt{int}, \texttt{b} : \texttt{int}$*,*

$$\texttt{rB}\ (\texttt{rB}\ (\texttt{l}, \texttt{a}), \texttt{b}) \cong \texttt{rB}\ (\texttt{l}, \texttt{a} + \texttt{b})$$

Proof. The proof is by structural induction on l. In all of the cases below, let `a` and `b` be any values of type `int`.

Case for `[]` To show:

$$\texttt{rB}\ (\texttt{rB}\ (\texttt{[]}, \texttt{a}), \texttt{b}) \cong \texttt{rB}\ (\texttt{[]}, \texttt{a} + \texttt{b})$$

Proof:

$$\begin{aligned}
& \texttt{rB}\ (\texttt{rB}\ (\texttt{[]}, \texttt{a}), \texttt{b}) && \\
\cong\ & \texttt{rB}\ (\texttt{case []\ of\ []} \Rightarrow \texttt{[]} \mid \texttt{x} :: \texttt{xs} \Rightarrow (\texttt{x} + \texttt{a}) :: \texttt{rB}\ (\texttt{xs}, \texttt{a}), \texttt{b}) && \text{step} \\
\cong\ & \texttt{rB}\ (\texttt{[]}, \texttt{b}) && \text{step} \\
\cong\ & \texttt{case []\ of\ []} \Rightarrow \texttt{[]} \mid \texttt{x} :: \texttt{xs} \Rightarrow (\texttt{x} + \texttt{b}) :: \texttt{rB}\ (\texttt{xs}, \texttt{b}) && \text{step} \\
\cong\ & \texttt{[]} && \text{step} \\
\cong\ & \texttt{case []\ of\ []} \Rightarrow \texttt{[]} \mid \texttt{x} :: \texttt{xs} \Rightarrow (\texttt{x} + (\texttt{a} + \texttt{b})) :: \texttt{rB}\ (\texttt{xs}, (\texttt{a} + \texttt{b})) && \text{step, sym} \\
\cong\ & \texttt{rB}\ (\texttt{[]}, \texttt{a} + \texttt{b}) && \text{step, sym}
\end{aligned}$$

By transitivity of $\cong$, this concludes this case.

Case for `x::xs` inductive hypothesis:

$$\texttt{rB}\ (\texttt{rB}\ (\texttt{xs}, \texttt{a}), \texttt{b}) \cong \texttt{rB}\ (\texttt{xs}, \texttt{a} + \texttt{b})$$

To show:

$$\texttt{rB}\ (\texttt{rB}\ (\texttt{x} :: \texttt{xs}, \texttt{a}), \texttt{b}) \cong \texttt{rB}\ (\texttt{x} :: \texttt{xs}, \texttt{a} + \texttt{b})$$

Proof:

$$
\begin{array}{lll}
 & \texttt{rB}\ (\texttt{rB}\ (\texttt{x} :: \texttt{xs}, \texttt{a}), \texttt{b}) & \\
\cong & \texttt{rB}\ (\texttt{case}\ \texttt{x} :: \texttt{xs}\ \texttt{of}\ [] \Rightarrow [] \mid \texttt{x} :: \texttt{xs} \Rightarrow (\texttt{x} + \texttt{a}) :: \texttt{rB}\ (\texttt{xs}, \texttt{a}), \texttt{b}) & \text{step} \\
\cong & \texttt{rB}\ ((\texttt{x} + \texttt{a}) :: \texttt{rB}\ (\texttt{xs}, \texttt{a}), \texttt{b}) & \text{step} \\
\cong & \texttt{case}\ (x + a) :: \texttt{rB}\ (\texttt{xs}, \texttt{a})\ \texttt{of}\ [] \Rightarrow [] \mid \texttt{x} :: \texttt{xs} \Rightarrow (\texttt{x} + \texttt{b}) :: \texttt{rB}\ (\texttt{xs}, \texttt{b}) & *** \\
\cong & ((x + a) + b) :: \texttt{rB}\ (\texttt{rB}\ (\texttt{xs}, \texttt{a}), \texttt{b}) & \text{step} \\
\cong & ((x + a) + b) :: \texttt{rB}\ (\texttt{xs}, \texttt{a} + \texttt{b}) & \text{IH} \\
\cong & (x + (a + b)) :: \texttt{rB}\ (\texttt{xs}, \texttt{a} + \texttt{b}) & \text{math} \\
\cong & \texttt{case}\ x :: xs\ \texttt{of}\ [] \Rightarrow [] \mid \texttt{x} :: \texttt{xs} \Rightarrow (\texttt{x} + (\texttt{a} + \texttt{b})) :: \texttt{rB}\ (\texttt{xs}, (\texttt{a} + \texttt{b})) & \text{step, sym} \\
\cong & \texttt{rB}\ (\texttt{x} :: \texttt{xs}, \texttt{a} + \texttt{b}) & \text{step, sym}
\end{array}
$$

By transitivity of $\cong$, and taking *** on faith, this concludes this case and the proof.

□

3.2 Valuability

To finish the proof of Theorem 1 above, we need to give a justification for why

$$\texttt{rB}\ ((\texttt{x} + \texttt{a}) :: \texttt{rB}\ (\texttt{xs}, \texttt{a}), \texttt{b})$$

and

$$\texttt{case}\ (x + a) :: \texttt{rB}\ (\texttt{xs}, \texttt{a})\ \texttt{of}\ [] \Rightarrow [] \mid \texttt{x} :: \texttt{xs} \Rightarrow (\texttt{x} + \texttt{b}) :: \texttt{rB}\ (\texttt{xs}, \texttt{b})$$

are equivalent.

Why do we need to be careful here? We make a very similar assertion in the base case when we said that

$$\texttt{rB}\ ([], \texttt{b}) \cong \texttt{case}\ []\ \texttt{of}\ [] \Rightarrow [] \mid \texttt{x} :: \texttt{xs} \Rightarrow (\texttt{x} + \texttt{b}) :: \texttt{rB}\ (\texttt{xs}, \texttt{b})$$

In both cases, we're substituting an argument for a parameter in the body of a function; the critical difference is that in the base case the arguments to that function are both values, but in the inductive case one of them is an expression but not a value.

So we need a rule stating to what a function application is equivalent that works in both situations. We seem to want to be able to say that, when given any function

```
fun f (x : A) : B = e1
```

and any expression `e2 : A`,

$$(\texttt{f}\ \texttt{e2}) \cong [\texttt{e2}/\texttt{x}]\texttt{e1}$$

where $[\texttt{e2}/\texttt{x}]\texttt{e1}$ is the expression that results from substituting `e2` for all instances of `x` in `e1`.

To see why this isn't good enough, consider the function

```
fun f (x : int) : int = 7
```

and the expression

```
f (1 div 0)
```

By our proposed rule, we can substitute for all zero occurrences of `x` in `7` and get

$$\texttt{f (1 div 0)} \cong \texttt{7}$$

But SML has a call-by-value evaluation semantics! So if we actually evaluate the expression, it raises an error because you can't divide by zero.

Because $\cong$ is an equivalence relation, allowing both interpretations would mean that

$$\texttt{7} \cong \texttt{raise Div}$$

which this directly contradicts the definition of $\cong$, where we said that programs that raise exceptions are not equivalent to programs that do not raise exceptions. The evaluation semantics of SML aren't going to change, so something must be wrong with our rule about equivalence of function application expressions.

To patch up our rule, we need a couple of definitions:

1. *Definition.* An expression e is valuable if and only if there exists some value v such that $e \cong v$. Specifically to this proof,
   - *(pairs)* If $e = (e_1, e_2)$ then e is valuable iff e_1 is valuable and e_2 is valuable.
   - *(sums)* If $e = e_1 + e_2$ then e is valuable iff e_1 is valuable and e_2 is valuable.
   - *(cons)* If $e = e_1 :: e_2$ then e is valuable iff e_1 is valuable and e_2 is valuable.
   - *(app)* If $e = (f\ e_1)$ then e is valuable iff f is total and e_1 is valuable
2. *Definition.* A function $f : \alpha \to \beta$ is total if and only if for all values $v : \alpha$, $(f\ v)$ valuable.

Now, we can give the following rule:

> Given any function
>
> ```
> fun f (x : A) : B = e1
> ```
>
> and any expression `e2 : A`, if `e2` is valuable then
>
> $$(\texttt{f e2}) \cong [\texttt{e2}/\texttt{x}]\texttt{e1}$$

So, finally, the justification that we really wanted at *** is

1.
   - `+` is total
   - `x` is assumed to be a value
   - `a` is assumed to be a value
   - Therefore, `x+a` is valuable by the rule for sums.
2.
   - `xs` is assumed to be a value
   - `a` is assumed to be a value
   - Therefore, `(xs,a)` is valuable by the rule for pairs.
3.
   - `rB` is total (by Theorem 2 proven below)
   - Therefore, `rB(xs,a)` is valuable by the rule for applications.

4. • Therefore, `(x+a)::(rB(xs,a))` is valuable by the rule for cons.

5. • `b` is assumed to be a value
   • Therefore, `rB((x+a)::(rB(xs,a)), b)` is valuable by the rule for pairs.

6. Finally, then,

$$\texttt{rB}\ ((\texttt{x} + \texttt{a}) :: \texttt{rB}\ (\texttt{xs}, \texttt{a}), \texttt{b})$$

   is contextually equivalent to

$$\texttt{case}\ (x + a) :: \texttt{rB}\ (\texttt{xs}, \texttt{a})\ \texttt{of}\ \texttt{[]} => \texttt{[]} \mid \texttt{x} :: \texttt{xs} => (\texttt{x} + \texttt{b})::\texttt{rB}\ (\texttt{xs}, \texttt{b})$$

   by the valuability argument above and a step.

This is very verbose and we'll often suppress most of these concerns—like the totality of simple SML built-ins, explicitly citing assumptions, and totality lemmas about structurally recursive functions such as `rB`.

The main point, though, is that you can only treat an expression like a name for a value and step through function application with it when *you know it will actually produce a value.*

3.3 Template for Structural Induction on Lists

Induction is applicable if you're trying to prove a theorem of the form

"for all `l : int list`, [some statement about l is true]"

Here's the format that any proof by structural induction on lists should have:

Proof. The proof is by structural induction on l.

Case for `[]` To show: [substitute `[]` into the statement for every instance of l]
Proof: ...

Case for `x::xs`. Inductive hypothesis: [substitute `xs` into the predicate].
To show: [substitute `x::xs` into the statement for every instance of l].
Proof: ...

□

3.3.1 Critical Observation

It's critical to note that following this schema does *not* produce a proof by induction on the length of the list: we're arguing about the structure of the list directly and length is never involved. It happens to be the case that lists and natural numbers have a similar structure, but that's incidental. We never mentioned length in our proof of Theorem 1 above or Theorem 2 below.

3.4 Totality

This proof was not presented in lecture, but it's included for completeness and because it's a good example of a proof by structural induction on lists.

Theorem 2 (Totality)**.** *rB is total.*

Proof. By the definitions of totality and valuability, it suffices to prove

For all values $\texttt{l} : \texttt{int list}, \texttt{a} : \texttt{int}$, $\exists v : \texttt{int list}$ such that

$$\texttt{rB}\ (\texttt{l}, \texttt{a}) \cong v$$

We will proceed, therefore, by induction on `l`. In all of the cases below, let `a` be any values of type `int`.

Case for `[]` To show: $\exists v : \texttt{int list}$ such that v is a value and $\texttt{rB}\ (\texttt{[]}, \texttt{a}) \cong v$
Proof:

$$\begin{array}{rll} & \texttt{rB}\ (\texttt{[]}, \texttt{a}) & \\ \cong & \texttt{case [] of [] => []} \mid \texttt{x :: xs => (x + a)::rB}\ (\texttt{xs}, \texttt{a}) & \text{step} \\ \cong & \texttt{[]} & \text{step} \end{array}$$

Choose v to be the value $\texttt{[]} : \texttt{int list}$. By the transitivity of $\cong$, this concludes the case.

Case for `x::xs`. Inductive hypothesis: $\exists v : \texttt{int list}$ such that v is a value and $\texttt{rB}\ (\texttt{xs}, \texttt{a}) \cong v$
To show: $\exists v : \texttt{int list}$ such that v is a value and $\texttt{rB}\ (\texttt{x :: xs}, \texttt{a}) \cong v$
Proof:

$$\begin{array}{rll} & \texttt{rB}\ (\texttt{x :: xs}, \texttt{a}) & \\ \cong & \texttt{case x :: xs of [] => []} \mid \texttt{x :: xs => (x + a)::rB}\ (\texttt{xs}, \texttt{a}) & \text{step} \\ \cong & (\texttt{x} + \texttt{a}) :: \texttt{rB}(\texttt{xs}, \texttt{a}) & \text{step} \end{array}$$

By the inductive hypothesis, $\exists v : \texttt{int list}$ such that v is a value and $\texttt{rB}\ (\texttt{xs}, \texttt{a}) \cong v$. Therefore, let $\texttt{rl} : \texttt{int list}$ be given such that $\texttt{rB}(\texttt{xs}, \texttt{a}) \cong \texttt{rl}$. Continuing from above, this gives us

$$\cong\ (\texttt{x} + \texttt{a}) :: \texttt{rl} \qquad \text{IH}$$

We assume that built in addition on values of type `int` is total, so let `rs` be a value such that

$$\texttt{rs} \cong (x + a)$$

Again continuing from above, this give us

$$\cong\ \texttt{rs} :: \texttt{rl} \qquad + \text{ total}$$

We assume that built in `::` is total, so let `rf` be a value such that

$$\texttt{rf} \cong \texttt{rs} :: \texttt{rl}$$

Again continuing from above, this give us

$$\cong\ \texttt{rf} \qquad :: \text{ total}$$

Choose v to be the value $\texttt{rf} : \texttt{int list}$. By the transitivity of $\cong$, this concludes the case.

□

3.5 Other Theorems

The three simple functions discussed above are actually enough to prove a number of interesting theorems. Here are just a couple, with the proofs left to the reader.

Theorem 3 (Self Inverse)**.** *For all values* `l : int list`, `a : int`,

$$\texttt{rB}(\texttt{rB}(\texttt{l}, \sim \texttt{a}), \texttt{a}) \cong \texttt{l}$$

Theorem 4 (Length Preservation)**.** *For all values* `l : int list`, `a : int`,

$$(\texttt{length l}) \cong \texttt{length}(\texttt{rb}(\texttt{l}, \texttt{a}))$$

Theorem 5 (Sum Scale)**.** *For all values* `l : int list`, `a : int`,

$$\texttt{sum}(\texttt{rB}(\texttt{l}, \texttt{a})) \cong (\texttt{sum l}) + ((\texttt{length l}) * \texttt{a})$$