15–150: Principles of Functional Programming

Some Notes on Structural Induction

Michael Erdmann[*]

Spring 2024

These notes provide a brief introduction to structural induction for proving properties of SML programs. We assume that the reader is already familiar with SML and the notes on evaluation and natural number induction for pure SML programs.

Structural inductions in SML often arise as inductions over the structure of values defined by `datatype` declarations. Most `datatype` declarations give rise to an induction principle which may be used to prove properties of recursive functions with arguments of the given type.

Notation Reminder: We write $e \stackrel{k}{\Longrightarrow} e'$ (or $e \Longrightarrow^k e'$) for a computation of k steps, $e \Longrightarrow e'$ (or $e \Longrightarrow^* e'$) for a computation of any number of steps (including 0), $e \hookrightarrow v$ for a complete computation of e to a value v, and $n = m$ or $e = e'$ for mathematical equality.

We say that two expressions e and e' are *extensionally equivalent*, and write $e \cong e'$, whenever one of the following is true: (i) evaluation of e produces the same value as does evaluation of e', or (ii) evaluation of e raises the same exception as does evaluation of e', or (iii) evaluation of e and evaluation of e' both loop forever. In other words, evaluation of e appears to behave just as does evaluation of e'. We say that an expression *has no value* if its evaluation either raises an exception or loops forever.

Observe: If $e_1 \Longrightarrow e$ and $e_2 \Longrightarrow e$, then $e_1 \cong e_2$. In other words, "reduction implies equivalence". CAUTION: The opposite implication need not hold: two expressions can be equivalent without one reducing to the other. For instance, $(1 + 9) \cong (2 * 5)$, but neither expression reduces to the other.

In proofs that establish equivalences, we frequently justify an equivalence $e_1 \cong e_2$ by writing something like `[step, 2nd clause of f]`. This means that either $e_1 \Longrightarrow e_2$ or $e_2 \Longrightarrow e_1$, as established by the function clause being cited. (Observe that such a reduction does indeed imply $e_1 \cong e_2$, by the previous paragraph.)

[*]Modified from a draft by Frank Pfenning.

1 Proof By Cases

A very simple form of "structural induction" arises if the datatype declaration is not recursive, but provides a finite number of data constructors. For such datatypes we can prove theorems by cases, which may also be viewed as an induction with only base cases. As an example, consider the declaration

```
datatype PrimColor = Red | Green | Blue
```

We can now prove properties of all primitive colors by distinguishing the cases of `Red`, `Green`, and `Blue`.

Another form for proof by cases arises for the Booleans, since there is a pervasive definition

```
datatype bool = true | false
```

For example, it is easy to see that

$$\texttt{(if } e \texttt{ then } e' \texttt{ else } e'\texttt{)} \ncong e'$$

since, for instance, evaluation of e might loop forever, whereas e' could reduce to a value. However, if e reduces to a value, then the two expressions are extensionally equivalent, as we now prove:

Theorem 1 *For every expression e (of type `bool`) such that $e \hookrightarrow v$ for some v, and for every e',*

$$\textit{(if } e \textit{ then } e' \textit{ else } e'\textit{)} \;\cong\; e'$$

Proof: By cases on the value of e.

$$\begin{array}{ll} & \texttt{if } e \texttt{ then } e' \texttt{ else } e' \\ \Longrightarrow & \texttt{if } v \texttt{ then } e' \texttt{ else } e' \quad \text{[by assumption on } e\text{]} \end{array}$$

Now either $v = \texttt{true}$ or $v = \texttt{false}$ by cases on the structure of `bool`. In either case, the expression above reduces to e'. □

2 Structural Induction on Lists

Here is one way to define SML lists that contain integers:

```
datatype list = nil | :: of int * list
infixr ::
```

Caution: The predefined list type `int list` in SML is defined slightly differently, but in a way that should not concern us today. In this document we will simply speak of the type `list` defined above, which is equivalent to SML's type `int list`.

The declaration `infixr ::` changes the lexical status of the constructor `::` to be a right-associative infix operator. That is, `1::2::3::nil` should be read as `1::(2::(3::nil))` which in turn would correspond to `::(1,::(2,::(3,nil)))` if `::` had not been declared infix. SML provides an alternative syntax for lists, as follows:

$$\begin{array}{rcl} \texttt{[]} & = & \texttt{nil} \\ \texttt{[}e_1\texttt{, } e_2\texttt{, } \ldots\texttt{, } e_n\texttt{]} & = & e_1 \texttt{ :: } (e_2 \texttt{ :: } (\cdots\ (e_n \texttt{ :: nil}))) \end{array}$$

The recursive nature of the declaration of `list` means that the corresponding induction principle is not just a proof by cases. It reads:

If:	1. a property holds for the empty list `nil` **and** 2. whenever the property holds for a value l of type `list`, it also holds for v `::` l (for any value v of type `int`),
then:	the property holds for all values of type `list`.

As a very simple example, consider the definition of a function to append two lists:

```
(* @ : list * list -> list *)
fun @ (nil, k) = k
  | @ (x::l, k) = x :: @(l,k)
infixr @
```

Appending two lists that are values always reduces to a value in SML. While this may seem trivial, it is actually not the case for some other functional languages such as Haskell in which values may be defined recursively.

Lemma 2 *The function* `@` *is total.*
In other words, for any values l and k of type `list`*, l `@` $k \hookrightarrow v$ for some v.*

Proof: By structural induction on l.

Base Case: $l =$ `nil`.
We need to show that for any value k `: list`, `nil @` k reduces to some value. Well,

$$\texttt{nil @ } k \Longrightarrow k \qquad \text{[by the first clause of @]}$$

Inductive Case: $l = x$ `::` l' for some values x `: int` and l' `: list`.

Induction hypothesis: Assume, for any value k `: list`, l' `@` $k \hookrightarrow v'$ for some v'.

We need to show that, for any value k `: list`, l `@` $k \hookrightarrow v$ for some v.

Evaluating code, we see that:

$$\begin{array}{lll} & (x \texttt{ :: } l') \texttt{ @ k} & \\ \Longrightarrow & x \texttt{ :: } (l' \texttt{ @ k}) & \text{[by the second clause of @]} \\ \Longrightarrow & x \texttt{ :: } v' & \text{[by induction hypothesis on } l'\text{]} \\ = & v & \text{(writing } v \text{ for the value } x \texttt{ :: } v'\text{)} \end{array}$$

By the structural induction principle for lists, this completes the proof. □

One can also prove that l `@` k takes $O(|l|)$ steps, where $|l|$ is the length of the list l. From this observation one can see that `(`l `@` k`) @` m takes $O(2|l| + |k|)$ steps, while l `@ (`k `@` m`)` takes only $O(|l| + |k|)$ steps. This is the basis for a number of simple efficiency improvements one can make in SML programs. It is formalized in the following lemma.

Lemma 3 *For any values l_1, l_2, and l_3 of type `list`,*

$$(l_1 \texttt{ @ } l_2) \texttt{ @ } l_3 \;\cong\; l_1 \texttt{ @ } (l_2 \texttt{ @ } l_3)$$

Proof: We reformulate this slightly to simplify the presentation of the proof:

$$(l_1 \texttt{ @ } l_2) \texttt{ @ } l_3 \cong l_{12} \texttt{ @ } l_3 \cong l_{123} \qquad \text{iff} \qquad l_1 \texttt{ @ } (l_2 \texttt{ @ } l_3) \cong l_1 \texttt{ @ } l_{23} \cong l_{123}$$

The proof is by structural induction on l_1.

Base Case: $l_1 = \texttt{nil}$.

We need to show that for any pair of values l_2 and l_3 of type `list`, $(\texttt{nil @ } l_2) \texttt{ @ } l_3$ is extensionally equivalent to a value l_{23} iff $\texttt{nil @ } (l_2 \texttt{ @ } l_3)$ is extensionally equivalent to the value l_{23}. Well,

$$\begin{array}{lll} & (\texttt{nil @ } l_2) \texttt{ @ } l_3 & \\ \cong & l_2 \texttt{ @ } l_3 & \text{[step, first clause of @]} \\ \cong & l_{23} & \text{[for some value } l_{23}\text{, by totality of @, i.e., Lemma 2]} \end{array}$$

and

$$\begin{array}{lll} & \texttt{nil @ } (l_2 \texttt{ @ } l_3) & \\ \cong & \texttt{nil @ } l_{23} & \text{[by totality of @]} \\ \cong & l_{23} & \text{[step, first clause of @]} \end{array}$$

Inductive Case: $l_1 = x \texttt{ :: } l_1'$ for some values $x \texttt{ : int}$ and $l_1' \texttt{ : list}$.

Induction hypothesis: Assume, for any values l_2 and l_3 of type `list`,

$$(l_1' \texttt{ @ } l_2) \texttt{ @ } l_3 \cong l_{12}' \texttt{ @ } l_3 \cong l_{123}' \qquad \text{iff} \qquad l_1' \texttt{ @ } (l_2 \texttt{ @ } l_3) \cong l_1' \texttt{ @ } l_{23} \cong l_{123}'$$

We need to show that, for any values l_2 and l_3 of type `list`,

$$(l_1 \texttt{ @ } l_2) \texttt{ @ } l_3 \cong l_{12} \texttt{ @ } l_3 \cong l_{123} \qquad \text{iff} \qquad l_1 \texttt{ @ } (l_2 \texttt{ @ } l_3) \cong l_1 \texttt{ @ } l_{23} \cong l_{123}$$

Evaluating code, for the left expression we obtain:

$$\begin{array}{lll} & ((x \texttt{ :: } l_1') \texttt{ @ } l_2) \texttt{ @ } l_3 & \\ \cong & (x \texttt{ :: } (l_1' \texttt{ @ } l_2)) \texttt{ @ } l_3 & \text{[step, second clause of @]} \\ \cong & (x \texttt{ :: } l_{12}') \texttt{ @ } l_3 & \text{[for some value } l_{12}'\text{, by totality of @]} \\ \cong & x \texttt{ :: } (l_{12}' \texttt{ @ } l_3) & \text{[step, second clause of @]} \\ \cong & x \texttt{ :: } l_{123}' & \text{[for some value } l_{123}'\text{, by totality of @]} \end{array}$$

For the right expression we obtain:

$$\begin{array}{lll} & (x \texttt{ :: } l_1') \texttt{ @ } (l_2 \texttt{ @ } l_3) & \\ \cong & (x \texttt{ :: } l_1') \texttt{ @ } l_{23} & \text{[for some value } l_{23}\text{, by totality of @]} \\ \cong & x \texttt{ :: } (l_1' \texttt{ @ } l_{23}) & \text{[step, second clause of @]} \\ \cong & x \texttt{ :: } l_{123}' & \text{[by the induction hypothesis on } l_1'\text{]} \end{array}$$

By the structural induction principle for lists, this completes the proof. □

We actually have the stronger and often useful result that `@` is associative even for expressions which are not necessarily values, assuming sequential code evaluation. This holds even under extensions by arbitrary effects, since in e_1 `@` `(`e_2 `@` e_3`)` and `(`e_1 `@` e_2`)` `@` e_3, the expressions e_1, e_2 and e_3 are evaluated in the same order, with all `@` computations in between reducing to values whenever the arguments are values.

Lemma 4 *For arbitrary expressions e_1, e_2 and e_3 of type `list`,*

$$(e_1 \ \texttt{@}\ e_2)\ \texttt{@}\ e_3 \cong e_1\ \texttt{@}\ (e_2\ \texttt{@}\ e_3)$$

Proof: By straightforward computation and Lemma 3:

$$\begin{array}{lll} & (e_1\ \texttt{@}\ e_2)\ \texttt{@}\ e_3 & \\ \Longrightarrow & (l_1\ \texttt{@}\ e_2)\ \texttt{@}\ e_3 & \text{or } e_1 \text{ has no value} \\ \Longrightarrow & (l_1\ \texttt{@}\ l_2)\ \texttt{@}\ e_3 & \text{or } e_2 \text{ has no value} \\ \Longrightarrow & l_{12}\ \texttt{@}\ e_3 & [\text{by totality of } \texttt{@}] \\ \Longrightarrow & l_{12}\ \texttt{@}\ l_3 & \text{or } e_3 \text{ has no value} \\ \Longrightarrow & l_{123} & [\text{by totality of } \texttt{@}] \end{array}$$

For the right-hand side we compute:

$$\begin{array}{lll} & e_1\ \texttt{@}\ (e_2\ \texttt{@}\ e_3) & \\ \Longrightarrow & l_1\ \texttt{@}\ (e_2\ \texttt{@}\ e_3) & \text{or } e_1 \text{ has no value} \\ \Longrightarrow & l_1\ \texttt{@}\ (l_2\ \texttt{@}\ e_3) & \text{or } e_2 \text{ has no value} \\ \Longrightarrow & l_1\ \texttt{@}\ (l_2\ \texttt{@}\ l_3) & \text{or } e_3 \text{ has no value} \\ \Longrightarrow & l_1\ \texttt{@}\ l_{23} & [\text{by totality of } \texttt{@}] \\ \Longrightarrow & l'_{123} & [\text{by totality of } \texttt{@}] \\ \cong & l_{123} & [\text{by proof of Lemma 3}] \end{array}$$

□

3 Structural Induction on Other Types

As an example for structural induction over other types we use binary trees in which the leaves carry integer data:

```
datatype tree = Leaf of int | Node of tree * tree
```

The structural induction principle for these types of trees then reads:

If:	1. a property holds for every leaf `Leaf(`x`)`, with x a value of type `int`, **and** 2. whenever the property holds for values t_1 and t_2 of type `tree`, it also holds for `Node(`t_1`,` t_2`)`,
then:	the property holds for all values of type `tree`.

The following function is inefficient, since the elements of `flatten t1` may end up being copied many times when the result lists are appended.

```
(* flatten : tree -> list
   REQUIRES: true
   ENSURES:  flatten(t) computes the inorder traversal of the leaf values.
*)
fun flatten (Leaf(x)) = [x]
  | flatten (Node(t1,t2)) = flatten t1 @ flatten t2
```

A more efficient alternative introduces an accumulator argument.

```
(* flatten2 : tree * list -> list
   REQUIRES: true
   ENSURES:  flatten2 (t, acc) ≅ flatten (t) @ acc
*)
fun flatten2 (Leaf(x), acc) = x::acc
  | flatten2 (Node(t1,t2), acc) =
      flatten2 (t1, flatten2 (t2, acc))

(* flatten' : tree -> list *)
fun flatten' (t) = flatten2 (t, nil)
```

We would like to prove that `flatten` and `flatten'` define the same function. In order to do that, we need to prove a lemma about `flatten2`, which requires a generalization of the induction hypothesis: We cannot prove directly by induction that `flatten2(`t`, nil)` $\cong$ `flatten(`t`)` since recursive calls in `flatten2` have a more general structure. The case of a leaf provides a clue about the proper generalization.

Lemma 5 *For any values t of type* `tree` *and acc of type* `list`*,*

`flatten2(`t`,` acc`)` $\cong$ `flatten(`t`) @` acc

Proof: By structural induction on t.

Base Case: $t = \texttt{Leaf}(x)$, for some value `x : int`.

We need to show that, for all values acc of type `list`,
$\texttt{flatten2(Leaf}(x)\texttt{,}\ acc) \cong \texttt{flatten(Leaf}(x)\texttt{) @}\ acc$.

Showing:

$$
\begin{array}{lll}
 & \texttt{flatten2(Leaf}(x)\texttt{,}\ acc) & \\
\cong & x\ \texttt{::}\ acc & \text{[step, first clause of } \texttt{flatten2}] \\
\cong & x\ \texttt{:: (nil @}\ acc) & \text{[step, first clause of } \texttt{@}] \\
\cong & (x\ \texttt{:: nil) @}\ acc & \text{[step, second clause of } \texttt{@}] \\
= & [x]\ \texttt{@}\ acc & \\
\cong & \texttt{flatten(Leaf}(x)\texttt{) @}\ acc & \text{[step, first clause of } \texttt{flatten}]
\end{array}
$$

Inductive Case: $t = \texttt{Node}(t_1\texttt{,}\ t_2)$ for some values t_1 and t_2, both of type `tree`.

Induction hypothesis: Assume that
for any value acc of type `list`, $\texttt{flatten2}(t_1\texttt{,}\ acc) \cong \texttt{flatten}(t_1)\ \texttt{@}\ acc$ and
for any value acc of type `list`, $\texttt{flatten2}(t_2\texttt{,}\ acc) \cong \texttt{flatten}(t_2)\ \texttt{@}\ acc$.

We need to show that for any value acc of type `list`,
$\texttt{flatten2}(t\texttt{,}\ acc) \cong \texttt{flatten}(t)\ \texttt{@}\ acc$.

Showing:

$$
\begin{array}{lll}
 & \texttt{flatten2(Node}(t_1\texttt{,}\ t_2)\texttt{,}\ acc) & \\
\cong & \texttt{flatten2}(t_1\texttt{, flatten2}(t_2\texttt{,}\ acc)) & \text{[step, second clause of } \texttt{flatten2}] \\
\cong & \texttt{flatten2}(t_1\texttt{, flatten}(t_2)\ \texttt{@}\ acc) & \text{[by the IH for } t_2] \\
\cong & \texttt{flatten}(t_1)\ \texttt{@ (flatten}(t_2)\ \texttt{@}\ acc) & \text{[by the IH for } t_1 \text{ and totality of both} \\
 & & \ \texttt{flatten} \text{ and } \texttt{@} \text{ (Lemma 2)]} \\
\cong & \texttt{(flatten}(t_1)\ \texttt{@ flatten}(t_2))\ \texttt{@}\ acc & \text{[by associativity of } \texttt{@} \text{ (Lemma 4)]} \\
\cong & \texttt{flatten(Node}(t_1\texttt{,}\ t_2))\ \texttt{@}\ acc & \text{[step, second clause of } \texttt{flatten}]
\end{array}
$$

By the structural induction principle for trees, this completes the proof. □

Comment: We used totality of `flatten` in the proof above. Imagine how you might prove that.

The following theorem now follows directly:

Theorem 6 *For any value t of type* `tree`*,*

$\texttt{flatten'}(t) \cong \texttt{flatten}(t)$.

Proof: We compute directly:

$$
\begin{array}{lll}
 & \texttt{flatten'}(t) & \\
\cong & \texttt{flatten2}(t\texttt{, nil}) & \text{[by definition of } \texttt{flatten'}] \\
\cong & \texttt{flatten}(t)\ \texttt{@ nil} & \text{[by Lemma 5]} \\
\cong & \texttt{flatten}(t) &
\end{array}
$$

Where the last equivalence holds by a property of `@` which is left as an exercise. □

There are also variants of structural induction analogous to strong induction, where we need to apply the induction hypothesis to some subexpression of the given value. We will not go into further details here.