Some Notes on Structural Induction

Michael Erdmann*
Spring 2018

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.

We write $e \xrightarrow{k} e'$ (or $e \xrightarrow{\geq k} e'$) for a computation of $k$ steps, $e \Rightarrow e'$ (or $e \Rightarrow^* e'$) for a computation of any number of steps (including 0), $e \rightarrow v$ 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 $e \equiv e'$, 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.

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.

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

```plaintext
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

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

For example, it is easy to see that

$$(\text{if } e \text{ then } e' \text{ else } e') \not\equiv e'$$

*Modified from a draft by Frank Pfenning.
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.

Theorem 1 For every expression \( e \) (of type \( \text{bool} \)) such that \( e \rightarrow v \) for some \( v \) and for every \( e' \) we have

\[
\text{if } e \text{ then } e' \text{ else } e' \equiv e'
\]

Proof: By cases on the value of \( e \).

\[
\text{if } e \text{ then } e' \text{ else } e' \\
\quad \Rightarrow \quad \text{if } v \text{ then } e' \text{ else } e' \quad \text{[by assumption on } e \text{]}
\]

Now either \( v = \text{true} \) or \( v = \text{false} \) by cases on the structure of \( \text{bool} \). In either case, the expression above reduces to \( e' \). \( \Box \)

2 Structural Induction on Lists

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

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

Caution: The predefined list type \( \text{int list} \) 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 \( \text{list} \) defined above, which is equivalent to SML's type \( \text{int list} \).

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

\[
[ ] = \text{nil} \\
[e_1, e_2, \ldots, e_n] = e_1 :: (e_2 :: (\cdots (e_n :: \text{nil})))
\]

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 \( \text{nil} \) and 2. whenever the property holds for a value \( l \) of type \( \text{list} \), it also holds for \( v :: l \) (for any value \( v \) of type \( \text{int} \)), then: the property holds for all values of type \( \text{list} \). |

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

```sml
(* @ : 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 ← v for some v.*

**Proof:** By structural induction on l.

**Base Case:** $l = \text{nil}$.  
We need to show that for any list k, $\text{nil} \ @ k \implies v$ returns some value. Well,  
\[\text{nil} \ @ k \implies k\quad \text{[by the first clause of @]}\]

**Induction Step:** $l = x :: l'$ for some $x$ and $l'$.  
Induction hypothesis: Assume $l' \ @ k \implies v'$ for some $v'$.  
We need to show that: $l \ @ k \implies v$ for some $v$.  
Evaluating code, we see that:  
\[
(x :: l') \ @ k \\
\implies x :: (l' \ @ k)\quad \text{[by the second clause of @]} \\
\implies x :: v'\quad \text{[by induction hypothesis on l']} \\
= v\quad \text{(writing v for the value x :: v')}
\]

By the structural induction principle for lists, this completes the proof. $\square$

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,  
\[l_1 \ @ l_2 \ @ l_3 \implies l_1 \ @ (l_2 \ @ l_3)\]

**Proof:** We reformulate this slightly to simplify the presentation of the proof:  
\[l_1 \ @ l_2 \ @ l_3 \implies l_{12} \ @ l_3 \implies l_{123}\quad \text{iff}\quad l_1 \ @ (l_2 \ @ l_3) \implies l_1 \ @ l_{23} \implies l_{123}\]

The proof is by structural induction on $l_1$.

**Base Case:** $l_1 = \text{nil}$.  
We need to show that for any pair of list values $l_2$ and $l_3$, $(\text{nil} \ @ l_2) \ @ l_3$ is extensionally equivalent to a value $l_{23}$ iff $\text{nil} \ @ (l_2 \ @ l_3)$ is extensionally equivalent to the value $l_{23}$. Well,  
\[
\text{nil} \ @ l_2 \ @ l_3 \implies l_2 \ @ l_3\quad \text{[by first clause of @]} \\
\implies l_{23}\quad \text{[for some value l_{23}, by totality of @, i.e., Lemma 2]}\]

and  
\[
\text{nil} \ @ (l_2 \ @ l_3) \implies \text{nil} \ @ l_{23}\quad \text{[by totality of @]} \\
\implies l_{23}\quad \text{[by first clause of @]}
\]
**Induction Step:** \( l_1 = x :: l'_1 \) for some \( x \) and \( l'_1 \).

**Induction hypothesis:** Assume

\[
(l'_1 \odot l_2) \odot l_3 \equiv l'_{12} \odot l_3 \equiv l'_{123} \quad \text{iff} \quad l'_1 \odot (l_2 \odot l_3) \equiv l'_1 \odot l_{23} \equiv l'_{123}
\]

We need to show that:

\[
(l_1 \odot l_2) \odot l_3 \equiv l_{12} \odot l_3 \equiv l_{123} \quad \text{iff} \quad l_1 \odot (l_2 \odot l_3) \equiv l_1 \odot l_{23} \equiv l_{123}
\]

Evaluating code, for the left expression we obtain:

\[
(x :: l'_1) \odot (l_2 \odot l_3) \equiv (x :: (l'_1 \odot l_2)) \odot l_3 = (x :: l'_{12}) \odot l_3 \quad \text{[by the second clause of \( \odot \)]}
\]

\[
(x :: l'_{12}) \odot l_3 \equiv (x :: l'_{123}) \quad \text{[for some value \( l'_{12} \), by totality of \( \odot \)]}
\]

\[
x :: (l'_1 \odot l_3) \equiv x :: l'_{123} \quad \text{[by the second clause of \( \odot \)]}
\]

For the right expression we obtain:

\[
(x :: l'_1) \odot (l_2 \odot l_3) \equiv (x :: l'_{12}) \odot l_{23} \quad \text{[for some value \( l_{23} \), by totality of \( \odot \)]}
\]

\[
x :: (l'_1 \odot l_{23}) \equiv x :: l'_{123} \quad \text{[by the induction hypothesis on \( l'_1 \)]}
\]

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

We actually have the stronger and often useful result that \( \odot \) is associative even for expressions which are not necessarily values, assuming sequential code evaluation. This holds even under extensions by arbitrary effects, since in \( e_1 \odot (e_2 \odot e_3) \) and \( (e_1 \odot e_2) \odot e_3 \), the expressions \( e_1 \), \( e_2 \) and \( e_3 \) are evaluated in the same order, with all \( \odot \) 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 \textit{list},

\[
(e_1 \odot e_2) \odot e_3 \equiv e_1 \odot (e_2 \odot e_3)
\]

**Proof:** By straightforward computation and Lemma 3:

\[
\begin{align*}
(e_1 \odot e_2) \odot e_3 & \quad \Rightarrow \quad (l_1 \odot e_2) \odot e_3 \quad \text{or } e_1 \text{ has no value} \\
& \quad \Rightarrow \quad (l_1 \odot l_2) \odot e_3 \quad \text{or } e_2 \text{ has no value} \\
& \quad \Rightarrow \quad l_{12} \odot e_3 \quad \text{[by totality of \( \odot \)]} \\
& \quad \Rightarrow \quad l_{12} \odot l_3 \quad \text{or } e_3 \text{ has no value} \\
& \quad \Rightarrow \quad l_{123} \quad \text{[by totality of \( \odot \)]}
\end{align*}
\]

For the right-hand side we compute:

\[
\begin{align*}
e_1 \odot (e_2 \odot e_3) & \quad \Rightarrow \quad l_1 \odot (e_2 \odot e_3) \quad \text{or } e_1 \text{ has no value} \\
& \quad \Rightarrow \quad l_1 \odot (l_2 \odot e_3) \quad \text{or } e_2 \text{ has no value} \\
& \quad \Rightarrow \quad l_1 \odot (l_2 \odot l_3) \quad \text{or } e_3 \text{ has no value} \\
& \quad \Rightarrow \quad l_1 \odot l_{23} \quad \text{[by totality of \( \odot \)]} \\
& \quad \Rightarrow \quad l'_{123} \quad \text{[by totality of \( \odot \)]} \\
& \quad \Rightarrow \quad l_{123} \quad \text{[by proof of Lemma 3]}
\end{align*}
\]

\[\square\]
3 Structural Induction on Other Types

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

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

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

<table>
<thead>
<tr>
<th>If:</th>
<th>1. a property holds for every leaf ( \text{Leaf}(x) ), with ( x ) of type ( \text{int} ), and</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2. whenever the property holds for values ( t_1 ) and ( t_2 ) of type ( \text{tree} ), it also holds for ( \text{Node}(t_1, t_2) ),</td>
</tr>
<tr>
<td>then:</td>
<td>the property holds for all values of type ( \text{tree} ).</td>
</tr>
</tbody>
</table>

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

```haskell
(* flatten : tree -> list  
  REQUIRES: true  
  ENSURES: flatten(t) returns 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.

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

```haskell
(* 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 \( \text{tree} \) and \( \text{acc} \) of type \( \text{list} \),

\[
\text{flatten2}(t, \text{acc}) \cong \text{flatten}(t) @ \text{acc}
\]

**Proof:** By structural induction on \( t \).
**Base Case:** \( t = \text{Leaf}(x) \), for some \( x : \text{int} \).

We need to show that, for all values \( acc \) of type \text{list},
\[
\text{flatten2}(\text{Leaf}(x), \; acc) \cong \text{flatten}(\text{Leaf}(x)) \@\; acc.
\]
Showing:
\[
\begin{align*}
\text{flatten2}(\text{Leaf}(x), \; acc) & \cong x :: acc & \text{[by first clause of flatten2]} \\
& \cong x :: (\text{nil} \@\; acc) & \text{[by first clause of \@ and referential transparency]} \\
& \equiv (x :: \text{nil}) \@\; acc & \text{[by second clause of \@]} \\
& \equiv [x] \@\; acc \\
& \equiv \text{flatten}(\text{Leaf}(x)) \@\; acc & \text{[by first clause of flatten]}
\end{align*}
\]

**Induction Step:** \( t = \text{Node}(t_1, \; t_2) \) for some \( t_1 \) and \( t_2 \), both of type \text{tree}.

Induction hypothesis: Assume that for any value \( acc \) of type \text{list},
\[
\text{flatten2}(t_1, \; acc) \cong \text{flatten}(t_1) \@\; acc \quad \text{and} \quad \text{flatten2}(t_2, \; acc) \cong \text{flatten}(t_2) \@\; acc.
\]
We need to show that for any value \( acc \) of type \text{list},
\[
\text{flatten2}(t, \; acc) \cong \text{flatten}(t) \@\; acc.
\]
Showing:
\[
\begin{align*}
\text{flatten2}(\text{Node}(t_1, \; t_2), \; acc) & \cong \text{flatten2}(t_1, \; \text{flatten2}(t_2, \; acc)) & \text{[by the second clause of flatten2]} \\
& \equiv \text{flatten2}(t_1, \; \text{flatten}(t_2) \@\; acc) & \text{[by the IH for } t_2 \text{ and referential transparency]} \\
& \equiv \text{flatten}(t_1) \@\; (\text{flatten}(t_2) \@\; acc) & \text{[by the IH for } t_1 \text{ and totality of both flatten and \@ (Lemma 2)]} \\
& \equiv (\text{flatten}(t_1) \@\; \text{flatten}(t_2)) \@\; acc & \text{[by associativity of \@ (Lemma 4)]} \\
& \equiv \text{flatten}(\text{Node}(t_1, \; t_2)) \@\; acc & \text{[by the second clause of flatten]}
\end{align*}
\]
By the structural induction principle for trees, this completes the proof.

Comment: We used totality of \text{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 \text{tree},
\[
\text{flatten}'(t) \cong \text{flatten}(t).
\]

Proof: We compute directly:
\[
\begin{align*}
\text{flatten}'(t) & \cong \text{flatten2}(t, \; \text{nil}) & \text{[by definition of flatten']} \\
& \equiv \text{flatten}(t) \@\; \text{nil} & \text{[by Lemma 5]} \\
& \equiv \text{flatten}(t) & \text{[by the last equivalence holds by a property of \@ which is left as an exercise.}}
\end{align*}
\]

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.

Page 6