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 \texttt{datatype} declarations. Most \texttt{datatype} declarations give rise to an induction principle which may be used to prove properties of recursive functions with arguments of the given type.

\textbf{Notation Reminder:} We write $e \Rightarrow^{k} e'$ (or $e \Rightarrow^{k} e'$) for a computation of $k$ steps, $e \Rightarrow e'$ (or $e \Rightarrow^{*} e'$) for a computation of any number of steps (including 0), $e \leftrightarrow v$ 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 \textit{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 \textit{has no value} if its evaluation either raises an exception or loops forever.

Observe: If $e_1 \Rightarrow e$ and $e_2 \Rightarrow e$, then $e_1 \cong e_2$. In other words, “reduction implies equivalence”. \textbf{CAUTION:} The opposite implication need not hold: two expressions can be equivalent without one reducing to the other. For instance, $(1 + 9) \cong (2 \ast 5)$, 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 \texttt{[step, 2nd clause of f]}. This means that either $e_1 \Rightarrow e_2$ or $e_2 \Rightarrow e_1$, 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.)
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

\[
\text{datatype PrimColor} = \text{Red} \mid \text{Green} \mid \text{Blue}
\]

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

\[
\text{datatype bool} = \text{true} \mid \text{false}
\]

For example, it is easy to see that

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

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 \(e \leadsto v\) for some \(v\), and for every \(e'\),

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

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

\[
\begin{align*}
\text{if } e \text{ then } e' \text{ else } e' \\
\implies \text{if } v \text{ then } e' \text{ else } e' \quad \text{[by assumption on } e]\end{align*}
\]

Now either \(v = \text{true}\) or \(v = \text{false}\) by cases on the structure of 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:

\[
\text{datatype list} = \text{nil} \mid :: \text{ of int \times list}
\]

\[
\text{infixr} ::
\]

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 \text{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, as follows:

\[
\begin{align*}
[] & = \text{nil} \\
[e_1, e_2, \ldots, e_n] & = e_1 :: (e_2 :: (\cdots (e_n :: \text{nil})))
\end{align*}
\]
The recursive nature of the declaration of list means that the corresponding induction principle is not just a proof by cases. It reads:

<table>
<thead>
<tr>
<th>If:</th>
<th>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),</th>
</tr>
</thead>
<tbody>
<tr>
<td>then:</td>
<td>the property holds for all values of type list.</td>
</tr>
</tbody>
</table>

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

\[
(*) \quad \text{@ : list * list -> list *)
\]

\[
\text{fun @ (nil, k) = k}
\]

\[
| \text{@ (x::l, k) = x :: @l,k) |
\]

\text{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 \( \text{@} \) 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:** \( l = \text{nil} \).

We need to show that for any list value \( k \), \( \text{nil} \ @ k \hookrightarrow \text{v} \) for some \( v \). Well,

\[
\text{nil} \ @ k \hookrightarrow k \quad \text{[by the first clause of } \text{@]}
\]

**Induction Step:** \( l = x :: l' \) for some values \( x \) and \( l' \).

Induction hypothesis: Assume \( l' \ @ k \hookrightarrow v' \) for some \( v' \).

We need to show that: \( l \ @ k \hookrightarrow v \) for some \( v \).

Evaluating code, we see that:

\[(x :: l') @ k \]

\[\hookrightarrow x :: (l' @ k) \quad \text{[by the second clause of } \text{@]}
\]

\[\hookrightarrow x :: v' \quad \text{[by induction hypothesis on } l']\]

\[= v \quad \text{(writing } v \text{ for the value } x :: v')
\]

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,

$$(l_1 \circ l_2) \circ l_3 \cong l_1 \circ (l_2 \circ l_3)$$

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

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

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 $l_3$, $(\text{nil} \circ l_2) \circ l_3$ is extensionally equivalent to a value $l_{23}$ iff $\text{nil} \circ (l_2 \circ l_3)$ is extensionally equivalent to the value $l_{23}$. Well,

$$(\text{nil} \circ l_2) \circ l_3 \cong l_2 \circ l_3 \quad \text{[step, first clause of } \circ \text{]}$$

and

$$\text{nil} \circ (l_2 \circ l_3) \cong \text{nil} \circ l_{23} \quad \text{[by totality of } \circ \text{]}$$

Induction Step: $l_1 = x :: l'_1$ for some values $x$ and $l'_1$.

Induction hypothesis: Assume

$$(l'_1 \circ l_2) \circ l_3 \cong l'_{12} \circ l_3 \cong l'_{123} \text{ iff } l'_1 \circ (l_2 \circ l_3) \cong l'_1 \circ l_{23} \cong l'_{123}$$

We need to show that:

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

Evaluating code, for the left expression we obtain:

$$(x :: l'_1) \circ l_2 \circ l_3 \cong (x :: (l'_1 \circ l_2)) \circ l_3 \quad \text{[step, second clause of } \circ \text{]}$$

For the right expression we obtain:

$$(x :: l'_1) \circ (l_2 \circ l_3) \cong (x :: (l'_1 \circ l_2)) \circ l_{23} \quad \text{[for some value } l_{23} \text{, by totality of } \circ \text{]}$$

$$(x :: l'_1) \circ l_{23} \cong (x :: (l'_1 \circ l_2)) \circ l_{23} \quad \text{[step, second clause of } \circ \text{]}$$

$$x :: (l'_1 \circ l_{23}) \quad \text{[for some value } l'_{123} \text{, by totality of } \circ \text{]}$$

By the structural induction principle for lists, this completes the proof. \qed
We actually have the stronger and often useful result that \( \oplus \) is associative even for expressions which are not necessarily values, assuming sequential code evaluation. This holds even under extensions by arbitrary effects, since in \( e_1 \oplus (e_2 \oplus e_3) \) and \( (e_1 \oplus e_2) \oplus e_3 \), the expressions \( e_1 \), \( e_2 \) and \( e_3 \) are evaluated in the same order, with all \( \oplus \) 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,

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

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

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

For the right-hand side we compute:

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

\( \Box \)
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:

\[
\text{datatype tree} = \text{Leaf of int} \mid \text{Node of tree} \times \text{tree}
\]

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

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

**then:** the property holds for all values of type \text{tree}.

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

\[
\begin{align*}
(* \text{ flatten} : \text{tree} \rightarrow \text{list} \\
\text{REQUIRES: true} \\
\text{ENSURES: flatten(t) computes the inorder traversal of the leaf values.} \\
*)
\end{align*}
\]

\[
\text{fun flatten} \ (\text{Leaf}(x)) = [x] \\
| \text{flatten} \ (\text{Node}(t_1, t_2)) = \text{flatten} \ t_1 \ @ \ \text{flatten} \ t_2
\]

A more efficient alternative introduces an accumulator argument.

\[
\begin{align*}
(* \text{ flatten2} : \text{tree} \times \text{list} \rightarrow \text{list} \\
\text{REQUIRES: true} \\
\text{ENSURES: flatten2} \ (t, \text{acc}) \cong \text{flatten} \ (t) \ @ \ \text{acc} \\
*)
\end{align*}
\]

\[
\text{fun flatten2} \ (\text{Leaf}(x), \text{acc}) = x::\text{acc} \\
| \text{flatten2} \ (\text{Node}(t_1, t_2), \text{acc}) = \\
\quad \text{flatten2} \ (t_1, \text{flatten2} \ (t_2, \text{acc}))
\]

\[
(* \text{ flatten'} : \text{tree} \rightarrow \text{list } *)
\]

\[
\text{fun flatten'} \ (t) = \text{flatten2} \ (t, \text{nil})
\]

We would like to prove that \text{flatten} and \text{flatten'} define the same function. In order to do that, we need to prove a lemma about \text{flatten2}, which requires a generalization of the induction hypothesis: We cannot prove directly by induction that \text{flatten2}(t, \text{nil}) \cong \text{flatten}(t) since recursive calls in \text{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},

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

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

We need to show that, for all values \( acc \) of type list,
\[
\text{flatten2}(\text{Leaf}(x), acc) \equiv \text{flatten}(\text{Leaf}(x)) \@ acc.
\]

Showing:
\[
\begin{align*}
\text{flatten2}(\text{Leaf}(x), acc) & \\
\equiv & \quad x :: acc & \quad \text{[step, first clause of flatten2]} \\
\equiv & \quad x :: (\text{nil} \@ acc) & \quad \text{[step, first clause of \@]} \\
\equiv & \quad (x :: \text{nil}) \@ acc & \quad \text{[step, second clause of \@]} \\
\equiv & \quad [x] \@ acc \\
\equiv & \quad \text{flatten}(\text{Leaf}(x)) \@ acc & \quad \text{[step, first clause of flatten]} \\
\end{align*}
\]

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

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

We need to show that for any value \( acc \) of type list,
\[
\text{flatten2}(t, acc) \equiv \text{flatten}(t) \@ acc.
\]

Showing:
\[
\begin{align*}
\text{flatten2}(\text{Node}(t_1, t_2), acc) & \\
\equiv & \quad \text{flatten2}(t_1, \text{flatten2}(t_2, acc)) & \quad \text{[step, second clause of flatten2]} \\
\equiv & \quad \text{flatten2}(t_1, \text{flatten}(t_2) \@ acc) & \quad \text{[by the IH for } t_2] \\
\equiv & \quad \text{flatten}(t_1) \@ (\text{flatten}(t_2) \@ acc) & \quad \text{[by the IH for } t_1 \text{ and totality of both flatten and } \@ (\text{Lemma } 2)] \\
\equiv & \quad (\text{flatten}(t_1) \@ \text{flatten}(t_2)) \@ acc & \quad \text{[by associativity of } \@ \text{ (Lemma } 4)] \\
\equiv & \quad \text{flatten}(\text{Node}(t_1, t_2)) \@ acc & \quad \text{[step, second clause of flatten]} \\
\end{align*}
\]

By the structural induction principle for trees, this completes the proof. \( \square \)

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 tree,
\[
\text{flatten'}(t) \equiv \text{flatten}(t).
\]

**Proof:** We compute directly:
\[
\begin{align*}
\text{flatten'}(t) & \\
\equiv & \quad \text{flatten2}(t, \text{nil}) & \quad \text{[by definition of flatten']} \\
\equiv & \quad \text{flatten}(t) \@ \text{nil} & \quad \text{[by Lemma } 5] \\
\equiv & \quad \text{flatten}(t) \\
\end{align*}
\]

Where the last equivalence holds by a property of \@ which is left as an exercise. \( \square \)

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.