Some Notes on Structural Induction

Frank Pfenning
(edited by John Lafferty)
September 9, 1998

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

We write \( e \xrightarrow{k} e' \) for a computation of \( k \) steps, \( e \xrightarrow{\ast} e' \) 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 define \( e \equiv e' \) (\( e \) is operationally equivalent to \( e' \)) to hold if for any value \( v \), \( e \leftrightarrow v \) iff \( e' \leftrightarrow v \), that is, if \( e \) and \( e' \) either both have values (in which case it must be the same), or neither has a value. This notion will have to be refined when the language is extended by effects.

Structural inductions in ML arises 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.

1 Proof By Cases

A very simple form of “structural induction” arises if the \texttt{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

\begin{verbatim}
datatype PrimColor = Red | Green | Blue;
\end{verbatim}

We can now prove properties of all primitive colors by distinguishing the cases of \texttt{Red}, \texttt{Green}, and \texttt{Blue}.

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

\begin{verbatim}
datatype bool = true | false;
\end{verbatim}

For example, it is easy to see that

\begin{verbatim}
if e then e' else e' \neq e'
\end{verbatim}

since \( e \) might not terminate, while \( e' \) could. However, if \( e \) has a value, then the two expressions are operationally equivalent.
Theorem 1  For every expression \( e \) (of type bool) such that \( e \leftrightarrow v \) for some \( v \) and for every \( e' \) we have

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

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

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

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

2 Structural Induction on Lists

The pervasive type of 'a list is defined by

\[
\text{datatype } 'a \text{ list } = \text{nil} \mid :: \text{ of } 'a \ast 'a \text{ list}; \\
\ast \text{infixr } ::;
\]

The last declaration 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. ML provides an alternative syntax for lists defined by

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

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

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

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

\[
(* \ @ : 'a \text{ list } * 'a \text{ list } \rightarrow 'a \text{ list } *) \\
\text{fun } @ (\text{nil}, k) = k \\
\mid @ (x::l, k) = x :: @l,k); \\
\ast \text{infixr } @;
\]

Appending two lists always terminates in ML. 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  For any values \( l \) and \( k \) of type \( t \) list, \( l @ k \leftrightarrow v \) for some \( v \).

Proof: By structural induction on \( l \).

Induction Basis \( l = \text{nil} \). Then

\[
\text{nil } @ k \Rightarrow k
\]
**Induction Step** \( l = x::l' \). Then

\[
(x :: l') @ k \\
\implies x :: (l' @ k) \\
\implies x :: v' \quad \text{by induction hypothesis on } l'
\]

\[\square\]

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

**Lemma 3** For any values \( l_1, l_2, \) and \( l_3 \) of type \( \text{list} \),

\[
(l_1 @ l_2) @ l_3 \equiv l_1 @ (l_2 @ l_3)
\]

**Proof:** We reformulate this slight to simplify the presentation of the proof:

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

The proof is by structural induction on \( l_1 \).

**Induction Basis** \( l_1 = \text{nil} \). Then

\[
(\text{nil} @ l_2) @ l_3 \\
\implies l_2 @ l_3 \\
\implies l_{23} \quad \text{by termination of } @
\]

and

\[
\text{nil} @ (l_2 @ l_3) \\
\implies \text{nil} @ l_{23} \quad \text{by termination of } @ \\
\implies l_{23}
\]

**Induction Step** \( l_1 = x :: l'_1 \) for some \( x \). Then we compute from the left expression:

\[
((x :: l'_1) @ l_2) @ l_3 \\
\implies (x :: (l'_1 @ l_2)) @ l_3 \\
\implies (x :: l'_{12}) @ l_3 \\
\implies x :: (l'_{12} @ l_3) \\
\implies x :: l'_{123}
\]

For the right expression we obtain:

\[
(x :: l'_1) @ (l_2 @ l_3) \\
\implies x :: (l'_{12} @ l_3) \\
\implies x :: (l'_{12} @ l_{23}) \\
\implies x :: l'_{123}
\quad \text{by induction hypothesis on } l'_1
\]

The intermediate values all exist since \( @ \) terminates by Lemma 2.
We actually have the stronger and often useful result that $\oplus$ is associative even for expressions which are not necessarily values. 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 only terminating $\oplus$ computations on the resulting values in between.

**Lemma 4** For arbitrary expressions $e_1$, $e_2$ and $e_3$,

$$(e_1 \oplus e_2) \oplus e_3 \equiv e_1 \oplus (e_2 \oplus e_3)$$

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

For the right-hand side we compute:

$$e_1 \oplus (e_2 \oplus e_3)$$

$$\implies l_1 \oplus (e_2 \oplus e_3) \text{ or } e_1 \text{ has no value}$$

$$\implies l_1 \oplus (l_2 \oplus e_3) \text{ or } e_2 \text{ has no value}$$

$$\implies l_{12} \oplus e_3 \text{ by termination of } \oplus$$

$$\implies l_{12} \oplus l_3 \text{ or } e_3 \text{ has no value}$$

$$\implies l_{123} \text{ by termination of } \oplus$$

For the right-hand side we compute:

$$(e_1 \oplus e_2) \oplus e_3$$

$$\implies (l_1 \oplus e_2) \oplus e_3 \text{ or } e_1 \text{ has no value}$$

$$\implies (l_1 \oplus l_2) \oplus e_3 \text{ or } e_2 \text{ has no value}$$

$$\implies l_{12} \oplus e_3 \text{ by termination of } \oplus$$

$$\implies l_{12} \oplus l_3 \text{ or } e_3 \text{ has no value}$$

$$\implies l_{123} \text{ by termination of } \oplus$$

For the right-hand side we compute:

$$(e_1 \oplus e_2) \oplus e_3$$

$$\implies (l_1 \oplus e_2) \oplus e_3 \text{ or } e_1 \text{ has no value}$$

$$\implies (l_1 \oplus l_2) \oplus e_3 \text{ or } e_2 \text{ has no value}$$

$$\implies l_{12} \oplus e_3 \text{ by termination of } \oplus$$

$$\implies l_{12} \oplus l_3 \text{ or } e_3 \text{ has no value}$$

$$\implies l_{123} \text{ by termination of } \oplus$$

\[ \square \]

### 3 Structural Induction on Other Types

As an example for structural induction over other types we use binary trees, where the leaves are labeled.

```
datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a tree;
```

The structural induction principle for trees then reads:

*If a property holds for every leaf Leaf(x) and whenever the property holds for values $t_1$ and $t_2$ of type $s$ tree it also holds for Node (t1, t2), then the property holds for all values of type $s$ tree.*

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

```
(* val flatten : 'a tree -> 'a list *)
fun flatten (Leaf(x)) = [x]
  | flatten (Node(t1,t2)) = flatten t1 @ flatten t2;
```

A more efficient alternative introduces an accumulator argument.
(* val flatten2 : 'a tree * 'a list -> 'a list *)
(* flatten2 (t, acc) == flatten (t) @ acc *)
fun flatten2 (Leaf(x), acc) = x::acc
  | flatten2 (Node(t1,t2), acc) = 
      flatten2 (t1, flatten2 (t2, acc));
(* val flatten' : 'a tree -> 'a 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) ≡ 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 s tree and acc of type s list we have

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

Proof: By structural induction on t.

Induction Basis \( t = \text{Leaf} \ (x) \). We compute the value of both sides.

\[
\text{flatten2} (\text{Leaf}(x), \text{acc}) \\
\implies x :: \text{acc}
\]

and

\[
\text{flatten} (\text{Leaf}(x)) @ \text{acc} \\
\implies [x] @ \text{acc} \\
\equiv (x :: \text{nil}) @ \text{acc} \\
\implies x :: \text{acc}
\]

Induction Step \( t = \text{Node} \ (t_1, t_2) \). We compute the value of both sides, using Lemma 4.

\[
\text{flatten2} (\text{Node}(t_1, t_2), \text{acc}) \\
\implies \text{flatten2} (t_1, \text{flatten2} (t_2, \text{acc})) \\
\implies \text{flatten2} (t_1, l_2) \\
\implies l_{12}
\]

and

\[
\text{flatten} (\text{Node}(t_1, t_2)) @ \text{acc} \\
\implies (\text{flatten} (t_1) @ \text{flatten} (t_2)) @ \text{acc} \\
\equiv \text{flatten} (t_1) @ (\text{flatten} (t_2) @ \text{acc}) \text{ by associativity of @ (Lemma 4)} \\
\implies \text{flatten} (t_1) @ l_2 \text{ by induction hypothesis on } t_2 \\
\implies l_{12} \text{ by induction hypothesis on } t_1
\]

The theorem now follows directly.
Theorem 6  For any value $t$ of type $s$ list we have

$$ \text{flatten}'(t) \equiv \text{flatten}(t) $$

Proof: We compute directly:

$$ \begin{align*}
\text{flatten}'(t) & \Rightarrow \text{flatten2}(t, \text{nil}) \\
& \equiv \text{flatten}(t) \odot \text{nil} \quad \text{by Lemma 5} \\
& \Rightarrow l \odot \text{nil} \\
& \Rightarrow l
\end{align*} $$

Where the last equality holds by a property of $\odot$ which is left as an exercise.

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