15–212: Principles of Programming

Some Notes on Structural Induction

Michael Erdmann*
Spring 2011

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 $e \overset{k}{\Longrightarrow} e'$ for a computation of k steps, $e \Longrightarrow e'$ 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 define $e \cong e'$ (e is *operationally equivalent* to e') to hold if for any value v, $e \hookrightarrow v$ iff $e' \hookrightarrow v$, 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 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

```
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

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

since e might not terminate, while e' could. However, if e has a value, then the two expressions are operationally equivalent.

[*] Modified from a draft by Frank Pfenning, 1997.

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

$$\textit{\texttt{if}}\ e\ \textit{\texttt{then}}\ e'\ \textit{\texttt{else}}\ e' \cong e'$$

Proof: By cases on the value of e.

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

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

The pervasive type of `'a list` is defined by

```
datatype 'a list = nil | :: of 'a * 'a list;
infixr ::;
```

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

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

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:	1. a property holds for the empty list `nil` **and** 2. 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 list * 'a list -> 'a list *)
fun @ (nil, k) = k
  | @ (x::l, k) = x :: @(l,k);
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* `@` *$k \hookrightarrow v$ for some v.*

Proof: By structural induction on l.

Induction Basis: $l = \texttt{nil}$. Then

$$\texttt{nil @ } k \Longrightarrow k \qquad \text{by straightforward code evaluation}$$

Induction Step: $l = x \texttt{ :: } l'$ for some x.

Induction hypothesis: Assume $l' \texttt{ @ } k \hookrightarrow v'$ for some v'.

We need to show that: $l \texttt{ @ } k \hookrightarrow v$ for some v.

Evaluating code, we see that:

$$\begin{array}{ll} & (x \texttt{ :: } l') \texttt{ @ k} \\ \Longrightarrow & x \texttt{ :: } (l' \texttt{ @ k}) \\ \Longrightarrow & x \texttt{ :: } v' \qquad\qquad \text{by induction hypothesis on } l' \\ = & v \end{array}$$

□

One can also prove that $l \texttt{ @ } k$ takes $O(|l|)$ steps, where $|l|$ is the length of the list l. From this observation one can see that $(l \texttt{ @ } k) \texttt{ @ } m$ takes $O(2|l| + |k|)$ steps, while $l \texttt{ @ } (k \texttt{ @ } m)$ takes only $O(|l| + |k|)$ 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 t `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:

$$\begin{array}{l} (l_1 \texttt{ @ } l_2) \texttt{ @ } l_3 \Longrightarrow l_{12} \texttt{ @ } l_3 \Longrightarrow l_{123} \quad \text{iff} \\ l_1 \texttt{ @ } (l_2 \texttt{ @ } l_3) \Longrightarrow l_1 \texttt{ @ } l_{23} \Longrightarrow l_{123} \end{array}$$

The proof is by structural induction on l_1.

Induction Basis: $l_1 = \texttt{nil}$. Then

$$\begin{array}{ll} & (\texttt{nil @ } l_2) \texttt{ @ } l_3 \\ \Longrightarrow & l_2 \texttt{ @ } l_3 \\ \Longrightarrow & l_{23} \qquad\qquad \text{by termination of @} \end{array}$$

and

$$\begin{array}{ll} & \texttt{nil @ } (l_2 \texttt{ @ } l_3) \\ \Longrightarrow & \texttt{nil @ } l_{23} \qquad\qquad \text{by termination of @} \\ \Longrightarrow & l_{23} \end{array}$$

Induction Step: $l_1 = x \texttt{ :: } l_1'$ for some x.

Induction hypothesis: Assume

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

We need to show that:

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

Evaluating code, for the left expression we obtain:

$$\begin{array}{ll} & ((x \texttt{ :: } l_1') \texttt{ @ } l_2) \texttt{ @ } l_3 \\ \Longrightarrow & (x \texttt{ :: } (l_1' \texttt{ @ } l_2)) \texttt{ @ } l_3 \\ \Longrightarrow & (x \texttt{ :: } l_{12}') \texttt{ @ } l_3 \\ \Longrightarrow & x \texttt{ :: } (l_{12}' \texttt{ @ } l_3) \\ \Longrightarrow & x \texttt{ :: } l_{123}' \end{array}$$

For the right expression we obtain:

$$\begin{array}{lll} & (x \texttt{ :: } l_1') \texttt{ @ } (l_2 \texttt{ @ } l_3) & \\ \Longrightarrow & (x \texttt{ :: } l_1') \texttt{ @ } l_{23} & \\ \Longrightarrow & x \texttt{ :: } (l_1' \texttt{ @ } l_{23}) & \\ \Longrightarrow & x \texttt{ :: } l_{123}' & \text{by induction hypothesis on } l_1' \end{array}$$

The intermediate values all exist since @ terminates by Lemma 2.

□

We actually have the stronger and often useful result that @ is associative even for expressions which are not necessarily values. This holds even under extensions by arbitrary effects, since in $e_1 \texttt{ @ } (e_2 \texttt{ @ } e_3)$ and $(e_1 \texttt{ @ } e_2) \texttt{ @ } e_3$, the expressions e_1, e_2 and e_3 are evaluated in the same order, with only terminating @ computations on the resulting values in between.

Lemma 4 *For arbitrary expressions e_1, e_2 and e_3 (of the same list type),*

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

Proof: By straightforward computation and Lemma 3.

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

For the right-hand side we compute:

$$\begin{array}{lll} & e_1 \texttt{ @ } (e_2 \texttt{ @ } e_3) & \\ \Longrightarrow & l_1 \texttt{ @ } (e_2 \texttt{ @ } e_3) & \text{or } e_1 \text{ has no value} \\ \Longrightarrow & l_1 \texttt{ @ } (l_2 \texttt{ @ } e_3) & \text{or } e_2 \text{ has no value} \\ \Longrightarrow & l_1 \texttt{ @ } (l_2 \texttt{ @ } l_3) & \text{or } e_3 \text{ has no value} \\ \Longrightarrow & l_1 \texttt{ @ } l_{23} & \text{by termination of } \texttt{@} \\ \Longrightarrow & l_{123} & \text{by Lemma 3} \end{array}$$

□

3 Structural Induction on Other Types

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

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

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

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

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 result lists are appended.

```
(* val flatten : 'a tree -> 'a list
   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.

```
(* 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)` $\cong$ `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 s `tree` and acc of type s `list` we have*

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

Proof: By structural induction on t.

Induction Basis: $t =$ `Leaf(`x`)`. We compute the value of both sides.

$$\begin{array}{ll} & \texttt{flatten2(Leaf(}x\texttt{), } acc\texttt{)} \\ \Longrightarrow & x \texttt{ :: } acc \end{array}$$

and

$$\begin{array}{ll}
 & \texttt{flatten(Leaf(}x\texttt{)) @ } acc \\
\Longrightarrow & \texttt{[}x\texttt{] @ } acc \\
\equiv & \texttt{(}x\texttt{ :: nil) @ } acc \\
\Longrightarrow & x\texttt{ :: } acc
\end{array}$$

Induction Step: $t = \texttt{Node(}t_1\texttt{, }t_2\texttt{)}$.

Induction hypothesis: Assume that for any value acc of type $s\,\texttt{list}$,
$\texttt{flatten2(}t1\texttt{, }acc\texttt{)} \cong \texttt{flatten(}t1\texttt{) @ }acc$ and
$\texttt{flatten2(}t2\texttt{, }acc\texttt{)} \cong \texttt{flatten(}t2\texttt{) @ }acc$.

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

We compute the value of both sides, using Lemma 4.

$$\begin{array}{lll}
 & \texttt{flatten2(Node(}t_1\texttt{, }t_2\texttt{), }acc\texttt{)} & \\
\Longrightarrow & \texttt{flatten2(}t_1\texttt{, flatten2(}t_2\texttt{, }acc\texttt{))} & \\
\Longrightarrow & \texttt{flatten2(}t_1\texttt{, }l_2\texttt{)} & \text{for some list } l_2 \\
\Longrightarrow & l_{12} & \text{for some list } l_{12}
\end{array}$$

and

$$\begin{array}{lll}
 & \texttt{flatten(Node(}t_1\texttt{, }t_2\texttt{)) @ }acc & \\
\Longrightarrow & \texttt{(flatten(}t_1\texttt{) @ flatten(}t_2\texttt{)) @ }acc & \\
\cong & \texttt{flatten(}t_1\texttt{) @ (flatten(}t_2\texttt{) @ }acc\texttt{)} & \text{by associativity of } \texttt{@} \text{ (Lemma 4)} \\
\Longrightarrow & \texttt{flatten(}t_1\texttt{) @ }l_2 & \text{by induction hypothesis on } t_2 \\
\Longrightarrow & l_{12} & \text{by induction hypothesis on } t_1
\end{array}$$

□

The theorem now follows directly:

Theorem 6 *For any value t of type s `tree` we have*

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

Proof: We compute directly:

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

Where the last equality holds by a property of `@` 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.