Chapter 11: Proof by Induction

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
http://www.idt.mdh.se/~blr/

August 17, 2007
Induction

Have you ever performed proofs by induction? (You should have . . .)

Then you know induction proofs are to prove properties that hold for all non-negative integers.

For instance, \( \forall n. \sum_{i=1}^{n} i = n(n + 1)/2 \)

Exercise: prove this property by induction!
Proofs by Induction for Properties of Natural Numbers

Goal: show that the property $P$ is true for all natural numbers (whole numbers $\geq 0$)

Proof by induction goes like this:

1. Show that $P$ holds for $0$
2. Show, for all natural numbers $n$, that if $P$ holds for $n$ then $P$ holds also for $n + 1$
3. Conclude that $P$ holds for all $n$

Formulated in formal logic:

$$[P(0) \land \forall n. P(n) \implies P(n + 1)] \implies \forall n. P(n)$$
Why does Induction over the Natural Numbers Work?

The set of natural numbers \( \mathbb{N} \) is an *inductively defined set*

(A variation of) Peano’s axiom:

- \( 0 \in \mathbb{N} \)
- \( \forall x.x \in \mathbb{N} \implies s(x) \in \mathbb{N} \)
- \( \forall x.0 \neq s(x) \)
- \( \forall x, y.x \neq y \implies s(x) \neq s(y) \)

\( s(x) \) “successor” to \( x \), or \( x + 1 \)

\[
\begin{align*}
0 & \rightarrow s(0) \rightarrow s(s(0)) \rightarrow s(s(s(0))) \rightarrow \cdots \\
0 & \quad 1 \quad 2 \quad 3 \quad \cdots
\end{align*}
\]

Proofs by induction follow the structure of the inductively defined set!
The Inductively Defined Set of Lists

Inductively defined sets are typically sets of \textit{infinitely} many \textit{finite} objects

The set \([a]\) of (finite) lists with elements of type \(a\):

- \([\ ] \in [a]\)

- \(x \in a \land xs \in [a] \implies x:xs \in [a]\)

Note similarity with the set of natural numbers!
An Induction Principle for Lists

Proof by induction for finite lists goes like this:

1. Show that $P$ holds for $[]$

2. Show, for all finite lists $xs \in [a]$ and all possible list elements $x \in a$, that if $P$ holds for $xs$ then $P$ holds also for $x:xs$

3. Conclude that $P$ holds for all finite lists in $[a]$

Formulated in formal logic:

$$[P([],) \land \forall x \in a, xs \in [a].P(xs) \rightarrow P(x:xs)] \rightarrow \forall xs \in [a].P(xs)$$
Note: a proof by induction holds *only* for finite lists

_Not* for infinite lists, or the divergent list (⊥)

But very often this is good enough!

At least, it is better than not knowing anything . . . :-)
A Simple Example of Induction Over Lists

Prove that $\text{length} \ (xs \ ++ \ ys) = \text{length} \ xs + \text{length} \ ys$ for all finite lists $xs, ys$

What induction hypothesis?

General rule: look at the function definitions and try to formulate the induction hypothesis so it matches the recursive structure!
Definitions of `length` and `++`:

\[
\begin{align*}
\text{length } \text{[]} &= 0 \\
\text{length } (x:xs) &= 1 + \text{length } xs
\end{align*}
\]

\[
\begin{align*}
\text{[]} ++ ys &= ys \\
(x:xs) ++ ys &= x:(xs ++ ys)
\end{align*}
\]

Now formulate induction hypotheses and prove the result!

Can we extend the proof to infinite lists?
A Number of Interesting Properties

Some properties of \texttt{map}:

\texttt{map id = id, where id = \lambda x \rightarrow x}

\texttt{map (f \cdot g) = map f \cdot map g}

\texttt{map f \cdot tail = tail \cdot map f}

\texttt{map f \cdot reverse = reverse \cdot map f}

\texttt{map f (xs ++ ys) = map f xs ++ map f ys}

(More properties in book, p. 138)
Another Proof by Induction

Let us prove $\text{map} \ (f \ . \ g) = \text{map} \ f \ . \ \text{map} \ g$!

(Proof on wyteboard)
A property of \texttt{fold}:

if \texttt{op} is associative and if \texttt{e} is left and right unit element for \texttt{op}, then, for all finite \texttt{xs}:

\[
\texttt{foldr \ op \ e \ xs} = \texttt{foldl \ op \ e \ xs}
\]

One can use properties of this kind to develop programs by \textit{program transformations}

There is something called the \textit{Bird-Meertens formalism}, which is a theory for functions over lists with many theorems like this
Let us prove a slightly simpler property:

That \( \text{sum \ } xs = \text{sum1 \ } xs \) for all finite lists \( xs \), where:

\[
\begin{align*}
\text{sum \ } [] &= 0 \\
\text{sum \ } (x:xs) &= x + \text{sum \ } xs \\
\text{sum1 \ } xs &= \text{sum2 \ } 0 \ \text{xs} \\
\text{sum2 \ } a \ [{}] &= a \\
\text{sum2 \ } a \ (x:xs) &= \text{sum2 \ } (a+x) \ \text{xs}
\end{align*}
\]
Do you see how to generalize the proof to prove the property of $\text{foldl}$ and $\text{foldr}$ on the previous page?
Induction over Trees

Trees are also inductively defined, e.g., the tree data type in Ch. 7:

\[
\text{data Tree } a = \text{Leaf } a \mid \text{Branch } (\text{Tree } a) \ (\text{Tree } a)
\]

Corresponding, inductively defined set of finite trees:

• for any \( x \in a \), \( \text{Leaf } x \in \text{Tree } a \)

• \( t_1, t_2 \in \text{Tree } a \implies \text{Branch } t_1 \ t_2 \in \text{Tree } a \)
Induction Principle for Trees

1. Show, for any $x \in a$, that $P$ holds for Leaf $x$

2. Show, for any two finite trees $t_1, t_2 \in [a]$, that if $P$ holds for $t_1$ and $t_2$, then $P$ holds also for Branch $t_1 \ t_2$

3. Conclude that $P$ holds for all finite trees in Tree $a$
A Proof by Induction over Trees

Show that $\text{length } (\text{fringe } t) = \text{treeSize } t$ for all finite trees $t$, where

\begin{align*}
\text{fringe } (\text{Leaf } x) &= [x] \\
\text{fringe } (\text{Branch } t1 \ t2) &= \text{fringe } t1 \ +\ + \ \text{fringe } t2 \\
\text{treeSize } (\text{Leaf } x) &= 1 \\
\text{treeSize } (\text{Branch } t1 \ t2) &= \text{treeSize } t1 \ + \ \text{treeSize } t2
\end{align*}
Strictness

A function $f$ is \textit{strict} if $f \perp = \perp$

Let

\begin{align*}
  f \, x &= 17 \\
  g \, x &= x + 1
\end{align*}

In Haskell, is $f$ strict? $g$?

\textbf{Theorem:} \textit{in a language with call-by-value, all user-defined functions are strict}

In a language with lazy evaluation, some user-defined functions can be non-strict
Strictness depends on whether the argument is needed or not

Example: consider definition of $\&\&$ from Standard Prelude:

$$\text{True} \&\& x = x$$
$$\text{False} \&\& \_ = \text{False}$$

The first argument *must* be evaluated to find out whether it is True, thus $\&\&$ is strict in its first argument.

But there are cases where the second argument is not needed, thus $\&\&$ is not strict in its second argument.
Some properties hold only for strict functions:

**Theorem:** If $f$ is strict, then

$$f \ (\text{if } b \ \text{then } x \ \text{else } y) = \text{if } b \ \text{then } f \ x \ \text{else } f \ y$$

Can you prove the theorem?
A strict function in a lazy language can be evaluated with call-by-value!

This is interesting, since call-by-value often is more efficient than lazy evaluation

*Strictness analysis* is a program analysis that sometimes can detect if a function is strict

Good compilers for lazy languages have strictness analyzers

Is the following function strict or not?

\[
f(x) = \begin{cases} 0 & \text{if } x = 0 \\ x + f(x-1) & \text{else} \end{cases}
\]