Reasoning About Functional Programs

Christian Sternagel

January 29, 2011

Perhaps the most favored property of today's programs is that they are correct, that is, do not contain errors (in the end it is often enough to have as few errors as possible). Programmers are sometimes already satisfied when they have a close look at their program and do not find anything wrong. Obviously that does not suffice. The best thing that could happen would be if one was able to *prove* that some program is correct. Since functional programming is so close to mathematics, some mathematical proof methods (most notably *induction*) can directly be applied to functional programs giving rise to rigorous correctness proofs. The process of proving the correctness of programs is called *program verification*. In this chapter one method to verify programs is presented: *Structural induction*.

1 Structural Induction

Structural induction is a generalization of induction over natural numbers (aka *mathematical induction*). In mathematical induction the goal is to prove that some property holds for all natural numbers.

Example 1.1. Consider the formula

$$1 + 2 + \cdots + n = \frac{n \cdot (n + 1)}{2}, \tag{1}$$

stating that the sum of the first n natural numbers can be computed by $(n \cdot (n + 1))/2$.

Proof. Using the principle of mathematical induction this can be proved by first considering the *base case*, which happens to be $n = 0$. Clearly the sum of the first 0 natural numbers is 0. Substituting 0 for n in the right-hand side of (1) results in

$$\frac{0 \cdot (0 + 1)}{2} = 0.$$

Hence the statement is true for the base case. Afterwards the *induction step* (or *step case*) is considered. For natural numbers that is, proving—under the assumption that

the desired property holds for n—that the property does hold for $n+1$. The assumption is called *induction hypothesis* (IH). In the current example the IH is

$$1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}.$$

It remains to be shown, that the left-hand side of (1) equals the right-hand side of (1) if $n+1$ is substituted for n. After the substitution the left-hand side becomes $1 + 2 + \cdots + (n+1)$ which can be transformed as follows:

$$\begin{aligned}
&(1 + 2 + \cdots + n) + (n+1) && \\
&= \frac{n \cdot (n+1)}{2} + (n+1) && \text{(by IH on } 1 + 2 + \cdots + n\text{)} \\
&= \frac{n \cdot (n+1) + 2n + 2}{2} && \text{(denominator adaption)} \\
&= \frac{n^2 + n + 2n + 2}{2} && \text{(multiplication)} \\
&= \frac{(n+1) \cdot (n+2)}{2} && \text{(expansion)}
\end{aligned}$$

which is the right-hand side of (1) where $n+1$ is substituted for n and thus concludes the proof. □

The intuition behind this proof method is the following: Suppose you want to convince yourself that (1) does hold for $n = 3$. The available ingredients are a proof that the formula does hold for $n = 0$ and a proof of the implication:

$$\text{If } 1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2} \text{ then } 1 + 2 + \cdots + n + (n+1) = \frac{(n+1) \cdot (n+2)}{2}.$$

Then starting at $0 = (0 \cdot (0+1))/2$ the implication can be used to get the result for 1 namely $1 = (1 \cdot (1+1))/2$. Applying the implication another two times yields the desired result $6 = (3 \cdot (3+1))/2$. In this way, every natural number can be reached and hence the property has to hold for all natural numbers.

Definition 1.1. The principle of *mathematical induction* states that if a property P does hold for 0 (that is, $P(0)$) and $P(n) \longrightarrow P(n+1)$, for all $n \in \mathbb{N}$, then $P(n)$ for all $n \in \mathbb{N}$. Put more formally,

$$(P(0) \wedge \forall n.\ (P(n) \longrightarrow P(n+1))) \longrightarrow \forall n.\ P(n).$$

Now it is obvious that the second bound occurrence of n does not depend on the first and hence it could also be stated, for example,

$$(P(0) \wedge \forall k.\ (P(k) \longrightarrow P(k+1))) \longrightarrow \forall n.\ P(n)$$

since renaming bound variables does not change the meaning.

Let's have a second look at the proof of (1). This time concentrating on the different ingredients of the principle of mathematical induction that occur in it. The property P is identified to be

$$P(x) = \left(\sum_{i=1}^{x} i = \frac{x \cdot (x+1)}{2} \right),$$

that is, the whole equation from (1) (which should be indicated by the surrounding parentheses). From a functional programming point of view, P can be seen as a function of type `Int -> Bool`, returning `True` if the given number satisfies (1) and `False` otherwise. A different reading of the formula

$$(P(0) \wedge \forall k.\, (P(k) \longrightarrow P(k+1))) \longrightarrow \forall n.\, P(n)$$

would be: "In order to prove $\forall n.\, P(n)$, it suffices to show that $P(0)$ is true and $\forall k.\, (P(k) \longrightarrow P(k+1))$ is true." Hence there are two things to show. Firstly, $P(0)$ which is called the *base case* and secondly $\forall k.\, (P(k) \longrightarrow P(k+1))$ which is called the *step case*. A different reading of the step case would be: "Assuming that $P(k)$ for arbitrary k ($\in \mathbb{N}$) show that also $P(k+1)$." Hence to prove the step case, $P(k)$ is used as a fact (called *induction hypothesis*) and using this fact, $P(k+1)$ has to be shown. Consider the following proof of (1):

Base Case The property P has to be shown for 0. By substituting x by 0 in $P(x)$ this translates to

$$\sum_{i=1}^{0} i = \frac{0 \cdot (0+1)}{2}.$$

This is obviously true, hence $P(0)$ has been shown.

Step Case Assume $P(k)$ holds for an arbitrary k, that is, $\sum_{i=1}^{k} = \frac{k \cdot (k+1)}{2}$. Using this try to show $P(k+1)$, that is,

$$\sum_{i=1}^{k+1} i = \frac{(k+1) \cdot ((k+1)+1)}{2}.$$

This can be shown in a similar way as in the previous proof of (1).

Reconsidering what has been shown yields $P(0)$ (from the base case) and $\forall k.\, (P(k) \longrightarrow P(k+1))$ (from the step case, since k has been arbitrary). Combining these two formulas yields

$$P(0) \wedge \forall k.\, (P(k) \longrightarrow P(k+1)).$$

This is exactly the premise of

$$(P(0) \wedge \forall k.\, (P(k) \longrightarrow P(k+1))) \longrightarrow \forall n.\, P(n)$$

and hence it follows that $\forall n.\, P(n)$ denoting that P is true for all natural numbers.

In the case of structural induction a proof is very similar, the only difference being that the number of base cases and step cases depends on the exact structure induction is applied upon. In the following this *structure* is always an algebraic data type, where the base cases correspond to constructors that do not refer recursively to the defined type and the step cases correspond to those that do.

1.1 Structural Induction Over Lists

Recall the type of lists that could be defined by

```
data [a] = [] | (:) a [a]
```

With respect to induction, lists are very similar to natural numbers. The base case being '`[]`' (that is, a list of length 0) and the step case $P(xs) \longrightarrow P(x : xs)$ (that is, assuming that the property holds for lists of length n it does also hold for lists of length $n + 1$). Indeed structural induction over lists is exactly the same as mathematical induction on the length of lists.

Example 1.2. As an example it is proved that the sum of the lengths of two lists is the same as the length of the combined list, that is, for all lists xs and ys it holds that

$$\texttt{length}\ xs + \texttt{length}\ ys = \texttt{length}\ (xs\ \texttt{++}\ ys)$$

Where `length` is defined by

```
length []     = 0
length (x:xs) = 1 + length xs
```

and '`++`' by

```
  []   ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Proof. Since `(++)` is defined via recursion over its first argument, it is obvious to perform induction over xs instead of ys (otherwise we would not be able to use the definition of `(++)` when proving the step case).

Base Case ($xs = \texttt{[]}$). By the definition of `length` the length of an empty list is 0. Hence the left-hand side equals `length` ys. By the definition of '`++`' the right-hand side also yields `length` ys.

Step Case ($xs = z : zs$). The IH is the equation

$$\texttt{length}\ zs + \texttt{length}\ ys = \texttt{length}\ (zs\ \texttt{++}\ ys)$$

Let's try to transform the left-hand side for $z : zs$ to the corresponding right-hand side:

$$\begin{aligned}\texttt{length}(z : zs) + \texttt{length}\ ys &= 1 + \texttt{length}\ zs + \texttt{length}\ ys \\ &\overset{\text{IH}}{=} 1 + \texttt{length}\ (zs\ \texttt{++}\ ys) \\ &= \texttt{length}\ (z : (zs\ \texttt{++}\ ys)) \\ &= \texttt{length}\ ((z : zs)\ \texttt{++}\ ys)\end{aligned}$$

□

Often, induction over lists is used to prove equality between two expressions. If one of the expressions is intuitively easy to understand but inefficient and the other is very complex but fast, then induction is a nice way to make sure that replacing the easy expressions by the complex ones will not alter the result of a program (but maybe the program will be much faster afterwards). However, to prove such equalities there is still something missing. Consider for example the function `head` that is defined by

```
head (x:_) = x
head _     = error "empty list"
```

What is the result of this function if it is applied to an empty list? In terms of program execution some exception is raised and the program is aborted. But for the purpose of induction proofs it is assumed that the result is *undefined*. Therefore, the value $\bot$ (speak 'bottom') is introduced, representing undefined results of computations. Then

$$\texttt{head}\ [] = \bot.$$

1.1.1 Properties of List Functions

Consider again the append function on lists `++`. It can be shown that nil is a left identity with respect to list concatenation.

Lemma 1.1. *'`[]`' is a left identity of '`++`', that is,*

$$[]\ \texttt{++}\ xs = xs$$

for all lists xs.

Proof. This follows immediately from the definition of `++`. □

It can also be shown that nil is a right identity for list concatenation.

Lemma 1.2. *'`[]`' is a right identity of '`++`', that is,*

$$xs \texttt{ ++ } \texttt{[]} = xs$$

for all lists xs.

Proof. By induction over the list xs.

Base Case ($xs = \texttt{[]}$). By the definition of `++` it follows immediately that `[] ++ [] = []`.

Step Case ($xs = y : ys$). The IH is $ys \texttt{ ++ } \texttt{[]} = ys$.

$$\begin{aligned}(y : ys) \texttt{ ++ } \texttt{[]} &= y : (ys \texttt{ ++ } \texttt{[]}) && \text{(definition of ++)}\\ &\overset{\text{IH}}{=} y : ys.\end{aligned}$$

□

Then by induction it can be proved that the evaluation order of '`++`' is irrelevant.

Lemma 1.3. *Concatenation of lists is associative, that is,*

$$(xs \texttt{ ++ } ys) \texttt{ ++ } zs = xs \texttt{ ++ } (ys \texttt{ ++ } zs).$$

Proof.

Base Case ($xs = \texttt{[]}$). Starting at the left-hand side, the following derivation can be done

$$(\texttt{[]} \texttt{ ++ } ys) \texttt{ ++ } zs = ys \texttt{ ++ } zs \qquad \text{(by Lemma 1.1)}.$$

The same result can be obtained starting at the right-hand side:

$$\texttt{[]} \texttt{ ++ } (ys \texttt{ ++ } zs) = ys \texttt{ ++ } zs \qquad \text{(by Lemma 1.1)}.$$

Step Case ($xs = w : ws$). The IH is $(ws \texttt{ ++ } ys) \texttt{ ++ } zs = ws \texttt{ ++ } (ys \texttt{ ++ } zs)$. For the left-hand side one gets:

$$\begin{aligned}((w : ws) \texttt{ ++ } ys) \texttt{ ++ } zs &= (w : (ws \texttt{ ++ } ys)) \texttt{ ++ } zs && \text{(definition of ++)}\\ &= w : ((ws \texttt{ ++ } ys) \texttt{ ++ } zs) && \text{(definition of ++)}\\ &\overset{\text{IH}}{=} w : (ws \texttt{ ++ } (ys \texttt{ ++ } zs)).\end{aligned}$$

And for the right-hand side the same result is obtained by the derivation step:

$$(w : ws) \texttt{ ++ } (ys \texttt{ ++ } zs) = w : (ws \texttt{ ++ } (ys \texttt{ ++ } zs)) \qquad \text{(definition of ++)}.$$

□

In mathematics, a structure consisting of a set (here the set of lists) and a binary operation on it (here list concatenation) such that the binary operation is associative and has an identity element (here the empty list) is called a *monoid*. Hence lists together with list concatenation build a monoid.

Another application of induction is to prove that `++` can alternatively be implemented in terms of `foldr`, where `foldr` is defined by

```
foldr f b   []   = b
foldr f b (x:xs) = f x (foldr f b xs)
```

This amounts to proving the following lemma.

Lemma 1.4.

$$xs \texttt{ ++ } ys = \texttt{foldr } (\texttt{:})\ ys\ xs$$

Proof.

Base Case ($xs = \texttt{[]}$). The base case follows immediately from the definitions of `++` and `foldr`.

Step Case ($xs = w \texttt{ : } ws$). The IH is $ws \texttt{ ++ } ys = \texttt{foldr } (\texttt{:})\ ys\ ws$. Then by transforming the left-hand side one gets:

$$\begin{aligned}(w \texttt{ : } ws) \texttt{ ++ } ys &= w \texttt{ : } (ws \texttt{ ++ } ys) \\ &\overset{\text{IH}}{=} w \texttt{ : } (\texttt{foldr } (\texttt{:})\ ys\ ws) \\ &= \texttt{foldr } (\texttt{:})\ ys\ (w \texttt{ : } ws)\end{aligned}$$

□

1.2 General Structures

Not only lists are usable for structural induction. In principle every algebraic data type gives rise to possible structural induction proofs over that type.

1.2.1 Binary Trees

As an example binary trees are used to show the more general case of structural induction with several step cases. Recall the definition of the type `BTree a` given by

```
data BTree a = Empty | Node a (BTree a) (BTree a)
```

A binary tree is called *perfect* if all leaf nodes have the same depth (that is, all paths from the root to some leaf node have the same length). By structural induction the following lemma can be shown.

Lemma 1.5. *A perfect binary tree t of height n has exactly $2^n - 1$ nodes.*

Proof.

Base Case ($t = \texttt{Empty}$). By definition of `height`, the height of an empty tree is 0. Substituting 0 for n in the goal results in $2^0 - 1 = 0$ which happens to be the number of nodes in an empty tree.

Step Case ($t = \texttt{Node}\ v\ l\ r$). Since t is a perfect binary tree of height $n + 1$ it follows that l and r are perfect binary trees of respective heights n. Hence by IH it holds that l and r both have $2^n - 1$ nodes. Since t is built by combining l, r, and one additional node, the number of nodes in t equals the number of nodes in l plus the number of nodes in r plus one, that is, $2 \cdot (2^n - 1) + 1$. The proof concludes by the following derivation:

$$\begin{aligned} 2 \cdot (2^n - 1) + 1 &= 2 \cdot 2^n - 2 + 1 && \text{(multiplication)} \\ &= 2 \cdot 2^n - 1 && \text{(addition)} \\ &= 2^{n+1} - 1. \end{aligned}$$

□

1.2.2 λ-Terms

Another example are λ-terms. Recall that a λ-term t is of the form

$$t \stackrel{\text{def}}{=} x \mid (\lambda x.\, t) \mid (t\ t)$$

with $x \in \mathcal{V}$, and that the (Haskell) type of λ-terms is defined by

```
data Term = Var String
          | Lab String Term
          | App Term Term
```

The base case for induction proofs is the case without a recursive reference to the definition of terms itself (that is, x for λ-terms and `Var` for the Haskell type). For the step cases abstractions ($(\lambda x.\, t)$ and `Lab`, respectively) and applications ($(t\ t)$ and `App`, respectively) have to be considered for λ-terms and the corresponding type. First let's prove that under the assumption that there is a unique mapping between variables $x \in \mathcal{V}$ and Haskell values of type `String`, there is exactly one instance of the type `Term` for every λ-term t. This is done via structural induction over t.

Base Case ($t = x$). For the case of a variable `Var x` can be taken for a uniquely determined value `x` of type `String`.

Step Case ($t = (\lambda x.\, s)$). The IH is that there is a unique instance of `Term` that corresponds to the term s. Let's call this instance `s`. Then by taking a uniquely determined identifier `x`, the instance `Lab x s` can be built.

Step Case ($t = (u\ v)$). By IH there are unique representations for u and v respectively (since they are both structurally smaller than t). Let's call these values `u` and `v`. Then the instance `App u v` can be built.

It is left as an exercise to show that (under the above assumption) there is a unique λ-term t for every value of type `Term`. Both proofs together establish that λ-terms and values of type `Term` are equivalent, that is, it does not matter whether to use induction over a λ-term t or its `Term t`.

Example 1.3. It can be shown that for every λ-term t the application $(t\ t)$, that is, t applied to itself, has an odd number of opening parentheses.

Proof.

Base Case ($t = x$). The term $(x\ x)$ has one opening parenthesis. Since 1 is an odd number that concludes the base case.

Step Case ($t = (\lambda x.\, u)$). The IH is that $(u\ u)$ has an odd number of opening parentheses. In the application $((\lambda x.\, u)\ (\lambda x.\, u))$ two more opening parentheses are added to those of $(u\ u)$. This results in an odd number.

Step Case ($t = (u\ v)$). By IH $(u\ u)$ and $(v\ v)$ both have an odd number of opening parentheses. In the application $((u\ v)\ (u\ v))$ one more opening parenthesis in addition to those of $(u\ u)$ and $(v\ v)$ is added. This results in an odd number.

□