CPSC 449:
Notes on proofs and structural induction
(for discussion in labs!)

Robin Cockett

January 27, 2021

1 Introduction to proofs

What is a proof? Well that is a much harder question than one might at first suspect! Indeed it is a subject both of philosophical discussion (i.e. there is probably no answer) and mathematical investigation (logic, type theory, proof theory, etc.).

1.1 Proof as well-structured thought

The first basic strategy behind writing a proof is common sense: if you want to make an argument that something is true you should break the argument down into smaller steps. Once one has broken it into small steps then each step can be individually examined to see whether it is valid – if each step is small enough one should be able to easily determine whether it is correct. When each step is valid then the whole argument will be valid!

If there is an error in the proof then it will become apparent because one of the small steps into which you have broken the argument must be wrong. Just one slip is enough to invalidate the whole proof ... mathematics in this sense is completely unforgiving! Mathematicians read each others proofs to detect these false steps and there are many famous instances when slips in proofs have been made (e.g. the early "proofs" of Fermat's last theorem, of the four color problem, ...). Errors of this nature are a normal occurrence in the process of developing mathematical ideas .... but it is worth remember that we hear of successes: usually we don't hear about it when the ideas were simply wrong!

To a programmer this should make perfect sense: to develop a big program one breaks it down into modules. When all the modules work one can put them together to make the big program. If the modules are small enough and well circumscribed enough they can easily be tested and documented. An error in any one of the modules, on the other hand can make a program behave in an arbitrary bad way! This should make one suspect that there is a close relationship between proofs and developing programs .... and there absolutely is! These connections have been made quite explicit by proof theorists: the Curry-Howard isomorphism is one well-known way in which they are connected.

1.2 Proof systems

The analogy with programming should also alert you to another aspect of proofs. Just as a program is developed in a particular programming system so a proof belongs to a particular proof system. In mathematics there are a number of well-known proof systems: equational proofs, propositional and predicate logic proofs, inductive proofs, coinductive proofs, etc.

So this raises an interesting question: what does a good proof system look like? Clearly a rather crucial aspect is that one should be able to decompose proofs into basic steps and then form the proof by composing these steps. This is a requirement on the proof system and there are definitely systems out there for which this basic requirement fails. A proof system with this property is called *compositional*. It is also important that there are a small (finite) number of basic steps from which every proof can be constructed: there are definitely proof systems which fail to have this property. A proof system with this property is said to be *finitely presented*. Finally, the system should have information content: that is it should have some things which are provable and other things which are not provable. A system with this property is said to be *non-degenerate*.

Constructing a proof theory with all these properties is a non-trivial task. In the beginning of the 20th century there was a significant effort to try to establish the foundations of mathematics. This culminated in various axioms for set theory, the development of proof theory, and indeed the theory computability. The fact that these developments were not straightforward is indicated by the litter of "paradoxes" (Russell's paradox, the liar paradox, ...) which accompanied the development of set theory. These paradoxes were developed as signpost to the boundaries of where set theory and logic becomes degenerate. Famously some of the early attempts at "set theory" turned out to be degenerate but fortunately for mathematics were subsequently corrected ...

1.3 Provability and computability

Even more philosophically dramatic than the foundational problems in the development of set theory was the collapse of Hilbert's program. Hilbert was one of the most influential mathematician's of the beginning of the 20th century. In 1900 he gave a famous lecture to the International Congress of Mathematicians in which he outlined a number of unsolved mathematical problems. These problems came from many different areas of mathematics and subsequently greatly influenced the development of mathematics itself.

Hilbert believed that by steadily breaking any problem down one could with diligence determine whether it was true or false. Undecidability had no role in his world. Brower, who was also a well-known Dutch mathematician, had a philosophically rather different view which became known as "intuitionistic" mathematics: this allowed for propositions which where neither true or false (i.e. from a modern perspective one might say undecidable – although this notion was not developed until much later). Hilbert found Brower's intuitionistic view of mathematics "unscientific", furthermore, he carried most of the mathematical establishment with him. Not only did the view that everything was either true of false make perfect sense but also it was reflected in the set theoretic foundations for mathematics which, of course, had been the major achievement of 19th century mathematics. Hilbert's comment that "no one shall expel us from the paradise that Cantor has created for us" won the day. Brower famously lost a very public argument for his more complex intuitionistic reality to Hilbert's onslaught which extended far beyond mere mathematical niceties.

In the 1930's Godel proved his incompleteness theorem which showed that Hilbert's philosophical view of mathematics was wrong. Significantly even some of the problems Hilbert had described in his 1900 presentation (and the very presentation in which he put forward his philosophy of provability), turned out to be undecidable. Brower had in some sense been right! Godel's results ultimately led to a much better understanding of the link between computability and provability. The set theoretic foundations of mathematics have endured for didactic reasons: set theory is consistent even if is not realistic or the whole story. As a foundation, it is more easily explained – and certainly more widely exposed – than intuitionist or constructive mathematics. In this regard Hilbert's influence on mathematics is still felt: the hubris of results of 20th century mathematics built on these foundations made it even more difficult to "leave the paradise" of set theory.

2 Arguing by induction

The principle of mathematical induction says this: suppose you have a set of natural numbers $S \subseteq \mathbb{N}$ (natural numbers are the numbers 0, 1, 2, 3, 4, . . .), suppose, furthermore, that $0 \in S$, that is 0 is in the set, and, finally, suppose that, whenever n is in the set, $n + 1$ is also in the set, that is $n \in S \Rightarrow n + 1 \in S$, then every natural number is in the set, that is $S = \mathbb{N}$.

To state it more informally: suppose you have been collecting numbers and the number 0 is in your collection, but also you notice that for each number k that you have in your collection, you also have the number $k + 1$. Then you have a very large collection as it consists of all the natural numbers!

Intuitively, the idea is that if you start with the number 0, and keep on adding 1 to it, you will eventually get to every number.

The principle of induction is extremely important because it allows one to prove many results that are much more difficult, or impossible, to prove in other ways. The most common application is when one has a statement one wants to prove about each natural number. It may be quite difficult to prove the statement directly, but easy to derive the truth of the statement about $n + 1$ from the truth of the statement about n. In that case, one appeals to the principle of induction by showing:

- The statement is true when $n = 0$.
- Whenever the statement is true for one number n, then it's also true for the next number $n + 1$.

If you can prove those two things, then the principle of induction says that the statement must be true for all natural numbers. The reasoning is: let S be the set of numbers for which the statement is true. The first item says that 0 is in the set, and the second item says that, whenever one number n is in the set, $n + 1$ is also in the set. Therefore, all numbers are in the set).

Example 2.1 As an example, consider proving that $1+2+3+...+n = n(n+1)/2$. To try to prove that equality for a general, unspecified n just by algebraic manipulations is difficult (though it may be easy to see it must be true). However, it is easy to prove by induction: it's true when $n = 0$ (as $0 = 0(0 + 1)/2$ and the sum of no numbers is 0), and whenever it's true for some number n, thus, $1+2+3+...+n = n(n+1)/2$, so $1+2+3+...+n+(n+1) = n(n+1)/2+(n+1) = (n+1)(n+2)/2$, so it's also true for $n + 1$. These two facts, combined with the principle of induction, mean that it's true for all n.

2.1 Complete induction

A consequence of mathematical induction is course-of-values induction (also known as complete induction or strong induction) which says, given a set of natural numbers, if you know n is in the set whenever each $i < n$ is in the set then every number is in the set.

Here is a proof of this principle: notice that as there are no numbers less than 0, so that 0 must be in the set. Define a subset of the set, namely those number for which all predecessors are in the set. 0 is certainly in this subset. Let k be any number in this set, as all of its predecessors are in the original set, then it is in the original set and so $k + 1$ is in the subset. But this means the subset contains all numbers whence the original set did!

Example 2.2 An example of the use of strong induction is to derive the Fibonacci formula using the "Golden Ratio":

$$\mathsf{fib}(n) = (G^n - (-1/G)^n)/\mathsf{sqrt}(5)$$

where G is the "Golden Ratio", that is $G = (1 + \mathsf{sqrt}(5))/2$ and the Fibonacci sequence is given by

$$\mathsf{fib}(0) = 0, \mathsf{fib}(1) = 1, \mathsf{fib}(n+2) = \mathsf{fib}(n) + \mathsf{fib}(n+1).$$

First we note that $\mathsf{fib}(0)$ and $\mathsf{fib}(1)$ satisfy the formula. Now suppose $\mathsf{fib}(k)$ for every $k < n + 2$ satisfies the formula (we have dealt with 0 and 1) then

$$\begin{aligned}\mathsf{fib}(n+2) = \mathsf{fib}(n) + \mathsf{fib}(n+1) &= (G^n - (-1/G)^n + G^{(n+1)} - (-1/G)^{(n+1)})/\mathsf{sqrt}(5) \\ &= (G^{(n+1)}(1 + 1/G) - (-1/G)^{(n+1)}(1 - G)/\mathsf{sqrt}(5) \\ &= G^{(n+2)} - (-1/G)^{(n+2)}\end{aligned}$$

where the last step uses the two equations:

$$1 + 1/G = G \quad \text{and} \quad 1 - G = -1/G$$

which are really the same equation and simplified give the quadratic

$$G^2 - G - 1 = 0$$

for which G as defined is a root! This is how Fibonacci found it!

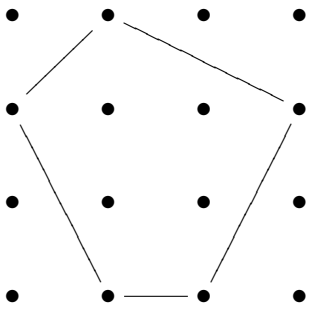
2.2 Least number principle

The least number principle says that any non-empty set of natural numbers has a least element. This can be used to do a proof by minimal counterexample: given a set which does not contain all numbers there must be a least number not in the set. Suppose you can show that there is no minimal counterexample (typically by using the fact that it is a counterexample to construct another smaller counterexample) then it must be that the set of counterexamples is empty. This is actually the contrapositive of complete induction! If you can prove that for any n not in the set there is a smaller m not in the set then, whenever everything smaller than the given element n is in the set, n must be in the set. So one can apply complete induction to conclude the set is all natural numbers.

Example 2.3 Picks theorem:
Given a simple polygon constructed on a 2-dimensional grid of points with integer coordinates such that all the polygon's vertices are grid points, Pick's theorem provides a simple formula for calculating the area A of this polygon in terms of the number, i, of interior points located in the polygon and the number b of boundary points placed on the polygon's perimeter:

$$A = i + \frac{1}{2}b - 1$$

So for example this simple polygon has area $5\frac{1}{2}$:



Note that the theorem is only valid for *simple* polygons, i.e. ones that consist of a single connected interior and, thus, does not contain "holes".

The result was first described by Georg Alexander Pick in 1899. It can be generalized to three dimensions and higher with Ehrhart polynomials. The formula also generalizes to surfaces of polyhedra.

PROOF: Every polygon must involve at least three grid points. We shall suppose that there is a polygon which is a counterexample which involves a minimum number of grid points. Either the polygon involves only three grid points (in which case it is straight forward to check Pick's formula actually holds) or it is possible to split the polygon into two smaller polygons by drawing a cord across the polygon. The sum of the areas of the two smaller polygons is then the area of the original. If the path across the polygon involves d points (so there are d-2 interior points on the path) then the two polygons must satisfy:

$$i = i_1 + i_2 + d - 2 \quad \text{and} \quad b = b_1 - d + b_2 - d + 2 = b_1 + b_2 - 2d + 2$$

whence

$$\begin{aligned} A = A_1 + A_2 &= i1 + \frac{1}{2}b_1?1 + i_2 + \frac{1}{2}b_2?1 \\ &= i_1 + i_2 + \frac{1}{2}(b_1 + b_2) - 2 \\ &= (i_1 + i_2 + d - 2) + \frac{1}{2}(b_1 + b_2 - 2d + 2) - 1 \\ &= i + \frac{1}{2}b - 1 \end{aligned}$$

so this cannot be a counterexample unless one of the smaller polygons is already a counterexample. So there are no minimal counter-examples! □

A neat application of Pick's theorem is to show:

It is impossible to draw an equilateral triangle with its vertices on grid points.

As the area of an equilateral triangle of base B has area $\mathsf{sqrt}(3)B^2/2$ which, as B^2 is an integer, is an irrational number!

2.3 Well-founded Induction

Let S be a non-empty set, and R be a binary relation on S. Then R is said to be a *well-founded* relation if and only if every nonempty subset $X \subseteq S$ contains an R-minimal element, that is an $x \in X$ such that there is no $y \in X$ with yRx. Such a relation cannot be reflexive, that is we cannot have xRx for any x as then $\{x\}$ would not have a minimal element.

Note that R is by no means required to be a total order. A classical example of a well-founded set that is *not* totally ordered is the set of natural numbers ordered by division, i.e. a div b if and only if a divides[1] b, that is $a > 1$ and there is an $r > 1$ such that $ra = b$.

Let S be a subset of a set X with a well-founded relation R. The principle of *well-founded induction* states that if the following is true :

- For every a, if for every x such that xRa we have $x \in S$ then $a \in S$ then $S = X$.

Notice that every R-minimal element of X must be in S as there are no elements below such an element so all of them are in S!

In fact, a relation for which this principle of induction is holds is, necessarily, a well-founded relation! Thus, well-founded relations and relations for which arguments by (well-founded) induction work are one and the same thing. Mathematical induction is obtained by asserting (as an axiom) that the successor relation is well-founded! Complete induction is obtained by the assertion that the usual order (the transitive closure of the successor relation) is well-ordered.

Thus well-founded induction is the most general inductive scheme. The difficulty is, however, to determine whether a relation is well-founded or not to start with ...

Example 2.4 An example of the use of well-founded induction, when the order is not the successor or the usual ordering on numbers, consider the fundamental theorem of arithmetic: this says that every natural number has a prime factorization.

First note that the division relation (as defined above) on natural numbers is well-founded and division-minimal elements are exactly the prime numbers. We detail the proof by splitting it into considering the minimal elements and the non-minimal elements:

- If n is prime, then n is its own factorization into primes, so the assertion is true for the division-minimal elements.
- If n is not prime, then n has a non-trivial factorization: $n = rs$, where r, s are not one. By our well-founded inductive assumption, r and s have prime factorizations, and therefore n has one too!

[1] Here we have removed the possibility that 1 divides p (which is often allowed) so the primes are minimal in this division relation.

3 Structural induction

In a programming language, such as Haskell, programming is done using datatypes. For doing proofs over (finite) datatypes a form of mathematical induction called structural induction is often used. It is an instance of well-founded induction which relies on the fact that there is a well-founded relation on all (inductive) datatypes.

Given an (inductive) datatype which is built by applying constructors to arguments:

$$\begin{aligned}\mathsf{data}\ \mathsf{D}(a) = \ & \mathsf{Cons}_1\ \ \mathsf{T}_{11}(a, \mathsf{D}(a))\ \ \mathsf{T}_{12}(a, \mathsf{D}(a))\ \ \ldots\ \ \mathsf{T}_{1m_1}(a, \mathsf{D}(a)) \\ & |.... \\ & |\ \mathsf{Cons}_n\ \ \mathsf{T}_{n1}(a, \mathsf{D}(a))\ \ \mathsf{T}_{n2}(a, \mathsf{D}(a))\ \ \ldots\ \ \mathsf{T}_{1m_n}(a, \mathsf{D}(a))\end{aligned}$$

then the terms of these datatypes have a well-founded relation given by $t_2 < t_1$ whenever t_1 is a strict subterm of t_2 and there are no subterms between t_1 and t_2.

There are two wrinkle that one must be aware of when proving properties of programs:

(a) A program may not terminate! This means that structural inductions should really include an argument either that the functions always terminate or for the behavior when they do not terminate. In much of what follows we shall assume that the functions do terminate in order to showcase the proofs by structural induction.

(b) Datatypes in Haskell can be infinite: Haskell's lazy evaluation allows one to "see" parts of the infinite values of datatypes and, indeed, these infinite values can provide powerful programming techniques. An example is given by forming the infinite "list" of all prime numbers. These infinite structures make the sets of values of datatypes non-well founded and so makes proofs by structural induction invalid. However, one can separate the proof into cases: when the datatype values are finite and when they are infinite. Throughout this section we shall be considering only the finite elements in the datatypes.

3.1 Natural numbers

For the natural numbers defined as a datatype by:

$$\begin{aligned}\mathsf{data}\ \ \mathsf{Nat} = \ & \mathsf{Zero} \\ & |\ \mathsf{Succ}\ \ \mathsf{Nat}\end{aligned}$$

we have the usual well-founded relation:

$$\mathsf{Zero} < \mathsf{Succ}(\mathsf{Zero}), \quad \mathsf{Succ}(\mathsf{Zero}) < \mathsf{Succ}(\mathsf{Succ}(\mathsf{Zero})), \quad \ldots$$

Structural induction, in this case, is mathematical induction.

Example 3.1 To add two natural numbers in unary we have:

$$\begin{aligned}\mathsf{add}\ \mathsf{Zero}\ m &= m \\ \mathsf{add}\ (\mathsf{Succ}\ n)\ m &= \mathsf{Succ}\ (\mathsf{add}\ n\ m)\end{aligned}$$

Suppose we want to prove that add is associative:

$$\mathsf{add}\ (\mathsf{add}\ x\ y)\ z = \mathsf{add}\ x\ (\mathsf{add}\ y\ z)$$

PROOF: We may use structural induction on x:

$$\begin{aligned}
\mathsf{add}\ (\mathsf{add}\ \mathsf{Zero}\ y)\ z &= \mathsf{add}\ y\ z \\
&= \mathsf{add}\ \mathsf{Zero}\ (\mathsf{add}\ y\ z) \\
\mathsf{add}\ (\mathsf{add}\ (\mathsf{Succ}\ x)\ y)\ z &= \mathsf{Succ}\ (\mathsf{add}\ (\mathsf{add}\ x\ y))\ z) \\
&= \mathsf{Succ}\ (\mathsf{add}\ x\ (\mathsf{add}\ y\ z)) && \text{(induction hypothesis)} \\
&= \mathsf{add}\ (\mathsf{Succ}\ x)\ (\mathsf{add}\ y\ z)
\end{aligned}$$

□

Example 3.2 Another important property of addition is that it is commutative:

$$\mathsf{add}\ x\ y = \mathsf{add}\ y\ x$$

This is much harder to prove! It is necessary to prove two additional properties of addition which are needed to complete the basic structural induction:
PROOF: Let us try an induction on x then we have:

$$\begin{aligned}
\mathsf{add}\ \mathsf{Zero}\ y &= y \\
&= \mathsf{add}\ y\ \mathsf{Zero} && \text{(see proof (1) below!)} \\
\mathsf{add}\ (\mathsf{Succ}\ x)\ y &= \mathsf{Succ}(\mathsf{add}\ x\ y) \\
&= \mathsf{Succ}(\mathsf{add}\ y\ x) && \text{(induction hypothesis)} \\
&= \mathsf{add}\ y\ (\mathsf{Succ}\ x) && \text{(see proof (2) below!)}
\end{aligned}$$

So this attempt at induction has introduced two proof obligations:

(1) We need

$$\mathsf{add}\ y\ \mathsf{Zero} = y$$

PROOF: We do a structural induction on y:

$$\begin{aligned}
\mathsf{add}\ \mathsf{Zero}\ \mathsf{Zero} &= \mathsf{Zero} \\
\mathsf{add}\ (\mathsf{Succ}\ y)\ \mathsf{Zero} &= \mathsf{Succ}(\mathsf{add}\ y\ \mathsf{Zero}) \\
&= \mathsf{Succ}\ y && \text{(induction hypothesis)}
\end{aligned}$$

□

(2) We also need:

$$\mathsf{Succ}\ (\mathsf{add}\ x\ y) = \mathsf{add}\ x\ (\mathsf{Succ}\ y)$$

PROOF: By structural induction on x:

$$\begin{aligned}
\mathsf{Succ}\ (\mathsf{add}\ \mathsf{Zero}\ y) &= \mathsf{Succ}\ y \\
&= \mathsf{add}\ \mathsf{Zero}\ (\mathsf{Succ}\ y) \\
\mathsf{Succ}\ (\mathsf{add}\ (\mathsf{Succ}\ x)\ y)) &= \mathsf{Succ}\ (\mathsf{Succ}\ (\mathsf{add}\ x\ y)) \\
&= \mathsf{Succ}\ (\mathsf{add}\ x\ (\mathsf{Succ}\ y)) && \text{(induction hypothesis)} \\
&= \mathsf{add}\ (\mathsf{Succ}\ x)\ (\mathsf{Succ}\ y))
\end{aligned}$$

3.2 Lists

For lists:

$$\begin{aligned}\mathsf{data\ List}\ a = \ & \mathsf{Nil} \\ & \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a)\end{aligned}$$

the well founded relation has $t < \mathsf{Cons}\ a\ t$ for every $t \in \mathsf{List}\ a$. This allows us to do structural inductive proofs using the principle of structural induction for lists. The structural induction says:

- if a property holds for Nil (the minimal element)
- and whenever the property holds for t then the property holds for $\mathsf{Cons}\ a\ t$ (using well-founded relation $t < \mathsf{Cons}\ a\ t$)

THEN the property holds for all elements of the list.

Example 3.3 The length of a list is defined by

$$\begin{aligned}\mathsf{length\ Nil} &= 0 \\ \mathsf{length}\ (\mathsf{Cons}\ a\ as) &= 1 + \mathsf{length}\ as\end{aligned}$$

To append two lists we have:

$$\begin{aligned}\mathsf{append\ Nil}\ bs &= bs \\ \mathsf{append}\ (\mathsf{Cons}\ a\ as)\ bs &= \mathsf{Cons}\ a(\mathsf{append}\ as\ bs\end{aligned}$$

We shall prove:

$$\mathsf{length}\ (\mathsf{append}\ x\ y) = (\mathsf{length}\ x) + (\mathsf{length}\ y)$$

PROOF: We shall use structural induction on x:

$$\begin{aligned}\mathsf{length}\ (\mathsf{append\ Nil}\ y) &= \mathsf{length}\ y) \\ &= 0 + \mathsf{length}\ y) \\ &= (\mathsf{length\ Nil}) + \mathsf{length}\ y) \\ \mathsf{length}\ (\mathsf{append}\ (\mathsf{Cons}\ a\ x)\ y) &= \mathsf{length}\ (\mathsf{Cons}\ a\ (\mathsf{append}\ x\ y)) \\ &= 1 + \mathsf{length}\ (\mathsf{append}\ x\ y) \\ &= 1 + (\mathsf{length}\ x) + (\mathsf{length}\ y) \\ &= \mathsf{length}\ (\mathsf{Cons}\ a\ x) + (\mathsf{length}\ y)\end{aligned}$$

□

We also can prove:

$$\mathsf{append}\ x\ (\mathsf{append}\ y\ z) = \mathsf{append}(\mathsf{append}\ x\ y)\ z$$

This is the associative law for append – a useful identity!

PROOF: We shall use structural induction on x:

$$\begin{aligned}
\mathsf{append}\ \mathsf{Nil}\ (\mathsf{append}\ y\ z) &= \mathsf{append}\ y\ z \\
&= \mathsf{append}\ (\mathsf{append}\ \mathsf{Nil}\ y)\ z \\
\mathsf{append}\ (\mathsf{Cons}\ a\ x)\ (\mathsf{append}\ y\ z) &= \mathsf{Cons}\ a\ (\mathsf{append}\ x(\mathsf{append}\ y\ z)) \\
&= \mathsf{Cons}\ a\ (\mathsf{append}(\mathsf{append}\ x\ y)\ z)) \\
&= \mathsf{append}\ (\mathsf{Cons}\ a\ (\mathsf{append}\ x\ y))\ z)) \\
&= \mathsf{append}\ (\mathsf{append}\ (\mathsf{Cons}\ a\ x)\ y)\ z
\end{aligned}$$

□

Example 3.4 To define "naive reverse" we have:

$$\begin{aligned}
\mathsf{reverse}\ \mathsf{Nil} &= \mathsf{Nil} \\
\mathsf{reverse}(\mathsf{Cons}\ a\ as) &= \mathsf{append}\ (\mathsf{reverse}\ as)\ (\mathsf{Cons}\ a\ \mathsf{Nil})
\end{aligned}$$

We want to prove by structural induction:

$$\mathsf{reverse}(\mathsf{append}\ x\ y) = \mathsf{append}\ (\mathsf{reverse}\ x)\ (\mathsf{reverse}\ y)$$

PROOF: We shall do structural induction on x:

$$\begin{aligned}
\mathsf{reverse}(\mathsf{append}\ \mathsf{Nil}\ y) &= \mathsf{reverse}\ y \\
&= \mathsf{append}\ (\mathsf{reverse}\ y)\ \mathsf{Nil} \qquad \text{(use append } y\ \mathsf{Nil} = y \text{ – see below)} \\
&= \mathsf{append}\ (\mathsf{reverse}\ y)\ (\mathsf{reverse}\ \mathsf{Nil}) \\
\mathsf{reverse}(\mathsf{append}\ (\mathsf{Cons}\ a\ x)\ y) &= \mathsf{reverseCons}\ a\ (\mathsf{append}\ x\ y) \\
&= \mathsf{append}\ (\mathsf{reverse}\ (\mathsf{append}\ x\ y))\ (\mathsf{Cons}\ a\ \mathsf{Nil}) \\
&= \mathsf{append}\ (\mathsf{append}\ (\mathsf{reverse}\ y)\ (\mathsf{reverse}\ x))\ (\mathsf{Cons}\ a\ \mathsf{Nil}) \\
&\qquad\qquad \text{(induction hypothesis)} \\
&= \mathsf{append}\ (\mathsf{reverse}\ y)\ (\mathsf{append}\ (\mathsf{reverse}\ x)\ (\mathsf{Cons}\ a\ \mathsf{Nil})) \\
&\qquad\qquad \text{(append is associative)} \\
&= \mathsf{append}\ (\mathsf{reverse}\ y)\ (\mathsf{reverse}\ (\mathsf{Cons}\ a\ x))
\end{aligned}$$

To complete the proof we have the proof obligation

$$\mathsf{append}\ x\ \mathsf{Nil} = x$$

for which we have the following proof by structural induction on x:

$$\begin{aligned}
\mathsf{append}\ \mathsf{Nil}\ \mathsf{Nil} &= \mathsf{Nil} \\
\mathsf{append}\ (\mathsf{Cons}\ a\ x)\ \mathsf{Nil} &= \mathsf{Cons}\ a\ (\mathsf{append}\ x\ \mathsf{Nil}) \qquad \text{(induction hypothesis)} \\
&= \mathsf{Cons}\ a\ x
\end{aligned}$$

□

Example 3.5 Consider the "fast" reverse function:

$$\begin{aligned} \mathsf{rev}\ xs &= \mathsf{shunt}\ xs\ [] \\ \mathsf{shunt}\ []\ ys &= ys \\ \mathsf{shunt}\ (x : xs)\ ys &= \mathsf{shunt}\ xs\ (x : ys) \end{aligned}$$

We would like to show that (for finite lists):

$$\mathsf{reverse}\ xs = \mathsf{rev}\ xs$$

PROOF: First we show that for [] the functions agree:

$$\mathsf{reverse}\ [] = [] = \mathsf{shunt}\ []\ []$$

Next we wish to show, supposing that $\mathsf{reverse}\ xs = \mathsf{rev}\ xs$ that $\mathsf{reverse}\ (x : xs) = \mathsf{rev}\ (x : xs)$ this is more complicated:

$$\begin{aligned} \mathsf{rev}\ (x : xs) &= \mathsf{shunt}\ (x : xs)\ [] \\ &= \mathsf{shunt}\ xs\ [x] \\ &= \mathsf{append}\ (\mathsf{shunt}\ xs\ [])\ [x] && \text{(proved this below)} \\ &= \mathsf{append}\ (\mathsf{reverse}\ xs)\ [x] && \text{(induction hypothesis)} \\ &= \mathsf{reverse}\ (x : xs) \end{aligned}$$

This leaves the proof that the third step in the proof is true. To prove this we establish something more general:

$$\mathsf{shunt}\ xs\ ys = \mathsf{append}(\mathsf{shunt}\ xs\ [])\ ys$$

We will do this by structural induction on the variable xs:

$$\begin{aligned} \mathsf{shunt}\ []\ ys &= ys \\ &= \mathsf{append}\ []\ ys \\ &= \mathsf{append}\ (\mathsf{shunt}\ []\ [])\ ys \\ \mathsf{shunt}\ (x : xs)\ ys &= \mathsf{shunt}\ xs\ (x : ys) \\ &= \mathsf{append}\ (\mathsf{shunt}\ xs\ [])\ (x : ys) \\ &= \mathsf{append}\ (\mathsf{append}\ (\mathsf{shunt}\ xs\ [])\ [x])\ ys && \text{(properties of append!)} \\ &= \mathsf{append}\ (\mathsf{shunt}\ xs\ [x])\ ys && \text{(induction hypothesis)} \\ &= \mathsf{append}\ (\mathsf{shunt}\ (x : xs)\ [])\ ys \end{aligned}$$

□

3.3 Trees

For binary trees defined as a datatype by:

$$\begin{aligned}\mathsf{data}\ \mathsf{Tree}\ a = \ & \mathsf{Tip} \\ & \mid \mathsf{Node}\ a\ (\mathsf{Tree}\ a)\ (\mathsf{Tree} a)\end{aligned}$$

The well founded relation is then $t_1 < \mathsf{Node}\ a\ t_1\ t_2$ and $t_2 < \mathsf{Node}\ a\ t_1\ t_2$ so that we get the following structural induction scheme for trees:

- If a property holds for Tip
- and whenever the property holds for t_1 and t_2 (and any a) it holds for $\mathsf{Node}\ a\ t_1\ t_2$

THEN the property holds for all trees.

Example 3.6 The number of "leaves" of a binary tree is defined by

$$\begin{aligned}\mathsf{leaves}\ \mathsf{Tip} &= 1 \\ \mathsf{leaves}(\mathsf{Node}\ a\ t_1\ t_2) &= (\mathsf{leaves}\ t_1) + (\mathsf{leaves}\ t_2)\end{aligned}$$

The number of "nodes" of a binary tree is defined by:

$$\begin{aligned}\mathsf{nodes}\ \mathsf{Tip} &= 0 \\ \mathsf{nodes}\ (\mathsf{Node}\ a\ t_1\ t_2) &= 1 + (\mathsf{nodes}\ t_1) + (\mathsf{nodes}\ t_2)\end{aligned}$$

We shall prove by structural induction:

$$\mathsf{leaves}\ t = 1 + \mathsf{nodes}\ t$$

PROOF: By structural induction:

$$\begin{aligned}\mathsf{leaves}\ \mathsf{Tip} &= 1 = 1 + 0 = 1 + (\mathsf{nodes}\ \mathsf{Tip}) \\ \mathsf{leaves}\ (\mathsf{Node}\ a\ t_1\ t_2) &= \mathsf{leaves}\ t_1 + \mathsf{leaves}\ t_2 \\ &= 1 + \mathsf{nodes}\ t_1 + 1 + \mathsf{nodes}\ t_2 \\ &= 1 + \mathsf{nodes}(\mathsf{Node}\ a\ t_1\ t_2).\end{aligned}$$

□

3.4 Higher-order functions

Example 3.7 Consider the fold over lists:

$$\begin{aligned}\mathsf{fold}\ f\ g\ [] &= g \\ \mathsf{fold}\ f\ g\ (x : xs) &= f\ x\ (\mathsf{fold}\ f\ g\ xs)\end{aligned}$$

An important property of the fold function is:

If g and f are terminating and xs is a finite terminating list then $\mathsf{fold}\ f\ g\ xs$ *is terminating.*

Now what exactly is meant by terminating needs be discussed as the notion is a little subtle. For example, does the program which generates the list of all primes "terminate"? Does a function which is not supplied all its arguments terminate?

It is tempting to take the intuitive approach which says that a program terminates provided it does not run forever. However, a function which is run without supplying all its arguments will usually terminate even if it would not have if it had been supplied its arguments! To get round this problem we may use the notion of "values": these are expressions which are in a fully evaluated form so that, for example, entered into Haskell they would be returned verbatim (so there are no higher-order aspects). A function then terminates if, when all its arguments are supplied with values, it will without fail return a value in a finite amount of time. So conceptually you test the function on all possible values for its arguments: only if the function always returns a value in a finite amount of time does the function terminate.

PROOF: We can prove this by a structural induction:

$$\begin{aligned} \mathsf{fold}\ f\ g\ [] &= g && \text{(which terminates!)} \\ \mathsf{fold}\ f\ g\ (x : xs) &= f\ x\ (\mathsf{fold}\ f\ g\ xs) && \text{(which terminates!)} \end{aligned}$$

Where, for the second line, because we know g terminates, by assumption x terminates, and $\mathsf{fold}\ xs\ y$ terminates by induction hypothesis, we know the whole expression terminates. □

Example 3.8 An alternative way of writing the append function is as follows;

$$\mathsf{app}\ x\ y = \mathsf{fold}\ (:)\ y\ x$$

Let us prove that

$$\mathsf{append}\ x\ y = \mathsf{app}\ x\ y$$

PROOF: We shall do a structural induction on x:

$$\begin{aligned} \mathsf{append}\ []\ y &= y \\ &= \mathsf{fold}\ (:)\ y\ [] \\ \mathsf{append}\ (x : xs)\ y &= x : (\mathsf{append}\ xs\ y) \\ &= x : (\mathsf{fold}\ (:)\ y\ xs) && \text{(induction hypothesis)} \\ &= \mathsf{fold}\ (:)\ y\ (x : xs) \end{aligned}$$

□

A small advantage of defining append like this is that one knows the function terminates ...

Example 3.9 A more sophisticated example is given by collecting the values of a tree. To collect the values of a tree into a list the naive algorithm is:

```
collect Tip = []
collect (Branch a t1 t2) = (collect t1) ++ [a]++ (collect t2)
```

However, this is rather inefficient as append traverses the lefthand argument repeatedly. A more efficient $\mathcal{O}(n)$ algorithm is given by the following higher-order collect.

```
coll t = hcoll t []

hcoll:: (Tree a) -> [a] -> [a]
hcoll Tip = id
hcoll (Branch a t1 t2) = (hcoll t1).(\z -> a:z).(hcoll t_2)

(.)::(b -> c) -> (a -> b) -> (a -> c)
g . f = \x ->  g (f x))

id = \x -> x
```

We wish to prove that:

$$\mathsf{coll}\ t = \mathsf{collect}\ t$$

To do so we shall need the observation

$$\mathsf{hcoll}\ t\ xs = (\mathsf{hcoll}\ t\ [])\ {+\!\!+} xs$$

which is used in the following structural induction on t:

$$\begin{aligned}
\mathsf{collect}\ \mathsf{Tip} &= [] \\
&= \mathsf{id}\ [] \\
&= (\mathsf{hcoll}\ \mathsf{Tip})\ [] \\
&= \mathsf{coll}\ \mathsf{Tip} \\
\mathsf{collect}\ (\mathsf{Branch}\ a\ t_1\ t_2) &= (\mathsf{collect}\ t_1)\ {+\!\!+}[a]\ {+\!\!+}(\mathsf{collect}\ t_2) \\
&=: (\mathsf{coll}\ t_1)\ {+\!\!+}[a]\ {+\!\!+}(\mathsf{coll}\ t_2) \qquad \text{(induction hypothesis)} \\
&=: (\mathsf{hcoll}\ t_1\ [])\ {+\!\!+}[a]\ {+\!\!+}(\mathsf{hcoll}\ t_2\ []) \\
&= \mathsf{hcoll}\ t_1([a]\ {+\!\!+}(\mathsf{hcoll}\ t_2\ []) \qquad (\text{use } \mathsf{hcoll}\ t\ xs = (\mathsf{hcoll}\ t\ [])\ {+\!\!+} xs) \\
&= \mathsf{hcoll}\ t_1\ (a : (\mathsf{hcoll}\ t_2[]) \\
&= \mathsf{hcoll}\ t_1\ ((\backslash z \to a : z)\ (\mathsf{hcoll}\ t_2\ []) \\
&= \mathsf{hcoll}\ t_1(((\backslash z \to a : z).(\mathsf{hcoll}\ t_2))\ []) \\
&= ((\mathsf{hcoll}\ t_1).(\backslash z \to a : z).(\mathsf{hcoll}\ t_2))\ [] \\
&= \mathsf{hcoll}\ (\mathsf{Branch}\ a\ t_1\ t_2)[] \\
&= \mathsf{coll}(\mathsf{Branch}\ a\ t_1\ t_2)
\end{aligned}$$

It remains to prove our helper result

$$\mathsf{hcoll}\ t\ xs = (\mathsf{hcoll}\ t\ [])\ {+\!\!+} xs$$

which we do by a structural induction on t.

$$
\begin{aligned}
\mathsf{hcoll}\ \mathsf{Nil}\ xs &= \mathsf{id}\ xs \\
&= xs \\
&= (\mathsf{id}\ []) +\!\!+ xs \\
&= (\mathsf{hcoll}\ \mathsf{Nil}) +\!\!+ xs \\
\mathsf{hcoll}\ (\mathsf{Branch}\ a\ t_1\ t_2)\ xs &= ((\mathsf{hcoll}\ t_1).(\backslash z \to a : z).(\mathsf{hcoll}\ t_2))\ xs \\
&= \mathsf{hcoll}\ t_1 (a : (\mathsf{hcoll}\ t_2\ xs)) \\
&= (\mathsf{hcoll}\ t_1\ []) +\!\!+ (a : ((\mathsf{hcoll}\ t_2\ []) +\!\!+ xs)) \qquad \text{(induction hypothesis)} \\
&= ((\mathsf{hcol}\ t_1\ []) +\!\!+ (a : (\mathsf{hcoll}\ t_2\ []))) +\!\!+ xs \qquad \text{(properties of append)} \\
&= ((\mathsf{hcoll}\ t_1 ((\backslash z \to a : z)\ (\mathsf{hcoll}\ t_2\ []))) +\!\!+ xs \\
&= ((\mathsf{hcoll}\ t_1).(\backslash z \to a : z).(\mathsf{hcoll} t_2))\ [] +\!\!+ xs \\
&= (\mathsf{hcoll}\ (\mathsf{Branch}\ a\ t_1\ t_2)\ []) +\!\!+ xs
\end{aligned}
$$

4 Exercises

In all the exercises you may assume that the datatypes are finite and the functions all terminate. Consider the following functions:

$$
\begin{aligned}
&(.) :: (b \to c) \to (a \to b) \to (a \to c) \\
&\ g.f = x \to g(fx)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{map} :: (a \to b) \to ([a] \to [b]) \\
&\mathsf{map}\ f\ [] = [] \\
&\mathsf{map}\ f\ (x : xs) = (f\ a) : (\mathsf{map}\ f\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{filter} :: (a \to \mathsf{Bool}) \to ([a] \to [a]) \\
&\mathsf{filter}\ p\ [] = [] \\
&\mathsf{filter}\ p\ (x : xs) | p\ x \qquad\quad = x : (\mathsf{filter}\ p\ xs) \\
&\qquad\qquad\qquad\quad | \mathsf{otherwise} = (\mathsf{filter}\ p\ xs)
\end{aligned}
$$

$$
\mathsf{concat} = \mathsf{fold}\ \mathsf{append}\ []
$$

1. Prove that : $\mathsf{fold}\ f\ z\ (\mathsf{append}\ xs\ ys) = \mathsf{fold}\ f\ (\mathsf{fold}\ f\ z\ ys)\ xs$. Which form is more efficient? Which form is more readable?

2. Prove that: $\mathsf{filter}\ p\ (\mathsf{append}\ xs\ ys) = (\mathsf{append}\ (\mathsf{filter}\ p\ xs)\ (\mathsf{filter}\ p\ ys)$. Which form is more efficient? Which form is more readable?

3. Prove that: $\mathsf{map}\ (f.g)\ xs = \mathsf{map}\ f\ (\mathsf{map}\ g\ xs)$.Which form is more efficient? Which form is more readable?

4. Prove that: $(\mathsf{filter}\ p).(\mathsf{map}\ f\ xs) = \mathsf{filter}\ (p.f)\ xs$. Which form is more efficient? Which form is more readable?

5. Prove that: $\mathsf{rev}(\mathsf{map}\ f\ xs) = \mathsf{map}\ f\ (\mathsf{rev}\ xs)$. Is this still true for infinite list? Which form is more efficient? Which form is more readable?

6. Prove that: $\mathsf{reverse}\ (\mathsf{reverse}\ x) = x$. Is this still true for infinite lists? Is this true for lists with elements which do not terminate?

7. Prove that $\mathsf{filter}\ p\ (\mathsf{filter}\ q\ x) = \mathsf{filter}\ (p\ \&\&\ q)\ x$. Which form is more efficient? Which form is more readable?

8. Define multiplication on the natural numbers, mult, and show that it is associative and commutative and distributes over the addition, add.

9. Prove that the higher-order reverse is equal to the first-order reverse.

10. Prove that $\mathsf{concat}(\mathsf{map}\ (\mathsf{map}\ f)\ xss) = \mathsf{map}\ f(\mathsf{concat}\ xss)$.

11. Define an efficient first order collect function and prove it equal to the naive collect.