Examples of Charity term logic proofs

Robin Cockett

July 2, 1993

1 Introduction

Whenever a datatype – be it inductive or coinductive – is declared in **charity** this declaration delivers three term constructions. In the case of inductive datatypes the fold, case, and map construction are delivered and for coinductive datatypes the unfold, record, and map construction. Each is delivered with certain rewriting rules so that they can be evaluated. However, they are also delivered with the proof rules which determine their formal properties.

For both inductive and coinductive datatype declarations the proof rules fall into three groups. Each group is concerned with a particular term construction. Thus, for an inductive datatype there are rules for the fold, case and map construction. Now in fact all these rules are implied by the group associated with the fold but it is mighty convenient to have all three groups available.

For inductive datatypes the proof rules are those of primitive induction. They are specifically the fold (primitive) induction rule and the fold unrolling rules. The unrolling rules are actually the rewritings used in evaluating case terms and are inverse as proof rules to fold induction. The fold induction rule asserts the uniqueness of the fold construct in a given iteration environment. It is a weaker proof rule than, for example, structural induction although, in certain circumstances, it is equivalent to the latter. Its significant advantage lies in its potential suitability for automation.

Very similar to the fold induction rule is the map induction rule with the map unrolling rules. Again the unrolling rules are used as rewriting rules in map term evaluation and are the proof inverse to the induction rule. The rules for map are easily seen to be consequences of the fold rules: they may be regarded as isolating a very particular sort of fold (the map) and its properties. The map is a functor and so is a very natural construct to isolate and have in the language.

The case rules allow arguments by case analysis. These rules also come in two mutually inverse proof directions: the choice rules which are also the rewritings used in the evaluation and the case induction rule (sometimes called "flat" induction as there is no self-reference).

In a similar (in fact dual) vein the rules describing the formal properties of coinductive datatypes comes in three groups for respectively the unfold, map, and record term construction. They are respectively the unfold (primitive) coinduction and unfold destruction, map coinduction and map destruction, and record (flat) coinduction) and record destruction rules.

The purpose of this document is to provide some examples of these rules in action for some specific datatypes. We shall start with the natural numbers and work up through lists to colists.

As this document does not attempt to give a complete description of the proof system, some of the more intuitive identities such as those for abstraction and substitution are omitted. However, a complete account of the term logic but can be found in chapter 8 of [Spe92].

2 Arithmetic with the natural numbers

To declare the natural numbers, $I\!N$, in **charity** one writes:

```
data nat -> C = zero: 1 -> C
              | succ: C -> C.
```

Clearly we should want to know how to add:

$$\mathrm{add}(x,y) = \left\{\!\!\left|\begin{array}{lll}\mathrm{zero}:() & \mapsto & y\\ \mathrm{succ}:n & \mapsto & \mathrm{succ}(n)\end{array}\right|\!\!\right\}(x).$$

The brackets $\{\!|...|\!\}$ (barbed-wire) contain the phrases of the **fold** operation for natural numbers. Each phrase consists of a constructor name labeling (this is the significance of the colon) an abstract map. The zero phrase, for example, has the abstraction, that is a variable base separated by a $\mapsto$ from a term, which indicates that we should start at the value given by that term. The succ phrase of the fold indicates that one should apply succ the number of times given by the first argment x. This one quickly realizes is precisely what addition is!

Charity, of course, has a fold operation for any datatype you might wish to declare – it is a sequence of phrases labeled by the constructors of the datatype and all enclosed in barbed wire. At the moment, however, we are specifically interested in the natural numbers and the consequence of having them available.

Well with such success under our belt we might consider multiplication and even exponentiation:

$$\begin{array}{rcl}\mathrm{mult}(x,y) & = & \left\{\!\!\left|\begin{array}{lll}\mathrm{zero}:() & \mapsto & \mathrm{zero}\\ \mathrm{succ}:n & \mapsto & \mathrm{add}(x,n)\end{array}\right|\!\!\right\}(y)\\ \exp(n,m) & = & \left\{\!\!\left|\begin{array}{lll}\mathrm{zero}:() & \mapsto & \mathrm{succ}(\mathrm{zero})\\ \mathrm{succ}:x & \mapsto & \mathrm{mult}(n,x)\end{array}\right|\!\!\right\}(m).\end{array}$$

From this sample of primitive recursive functions you may think that one can only program increasing functions. This is not the case as the following programs demonstrate:

$$\begin{array}{rcl}\mathrm{half}(n) & = & \mathrm{p}_0(\left\{\!\!\left|\begin{array}{rll}\mathrm{zero}:() & \mapsto & (\mathrm{zero},\mathrm{zero})\\ \mathrm{succ}:(x,y) & \mapsto & (y,\mathrm{succ}(x))\end{array}\right|\!\!\right\}(n))\\ \mathrm{pred}(n) & = & \mathrm{p}_0(\left\{\!\!\left|\begin{array}{rll}\mathrm{zero}:() & \mapsto & (\mathrm{zero},\mathrm{zero})\\ \mathrm{succ}:(x,y) & \mapsto & (y,\mathrm{succ}(y))\end{array}\right|\!\!\right\}(n)).\end{array}$$

The first program halves the number (truncating fractions) the second program gives the predecessor of the number. The latter program one might reasonably comment is a rather convoluted way of programming the predecessor function. This is a fair comment and **charity** does, of course, have a more appropriate construct for this function: the case construct. This illustrates how one can formulate the case construct using a fold. Here is the predecessor function written sensibly:

$$\mathrm{pred}(n) = \left\{\begin{array}{rll}\mathrm{zero}() & \mapsto & \mathrm{zero}\\ \mathrm{succ}(n') & \mapsto & n'\end{array}\right\}(n).$$

Once one has the predecessor function one can program the monus function: truncated minus. This is often written $x \mathbin{\dot{-}} y$:

$$n \mathbin{\dot{-}} m = \left\{\left|\begin{array}{lcl} \mathrm{zero} : () & \mapsto & n \\ \mathrm{succ} : x & \mapsto & \mathrm{pred}(x) \end{array}\right|\right\}(m).$$

2.1 Inference rules for the natural numbers

The group of rules concerned with the fold over the natural numbers (which could be regarded as an iteration or "for" loop) are as follows. First the fold induction rule:

$$\frac{t_0 =_x t(\mathrm{zero}) \mid t(\mathrm{succ}(n)) =_x \{e \mapsto t_1\}(t(n))}{t(m) =_x \left\{\left|\begin{array}{lcl} \mathrm{zero} : () & \mapsto & t_0 \\ \mathrm{succ} : e & \mapsto & t_1 \end{array}\right|\right\}(m)}$$

there are then two unrolling rules:

$$\left\{\left|\begin{array}{lcl} \mathrm{zero} : () & \mapsto & t_0 \\ \mathrm{succ} : e' & \mapsto & t_1 \end{array}\right|\right\}(\mathrm{zero}) =_e t_0,$$

$$\left\{\left|\begin{array}{lcl} \mathrm{zero} : () & \mapsto & t_0 \\ \mathrm{succ} : e' & \mapsto & t_1 \end{array}\right|\right\}(\mathrm{succ}(n)) =_e \{e' \mapsto t_1\}\left(\left\{\left|\begin{array}{lcl} \mathrm{zero} : () & \mapsto & t_0 \\ \mathrm{succ} : e' & \mapsto & t_1 \end{array}\right|\right\}(n)\right).$$

2.2 Proof Techniques

The fold uniqueness rule is surprisingly powerful, however, it is sometimes hard to see how it can be used. This section develops some supplementary proof techniques. In the first section we discuss the case construct and how we can simulate the presence of a Boolean type. This means that we may add such a type without disturbing the power of the system. Next we discuss a rather useful observation which Grant Malcolm called *promotion* which seems to recur in proofs. Finally, we describe the extent to which mathematical induction is applicable to the setting.

Before we start we state some very basic results:

Lemma 2.1

(i)

$$\left\{\left|\begin{array}{lcl} \mathit{zero} : () & \mapsto & \mathit{zero} \\ \mathit{succ} : n & \mapsto & \mathit{succ}(n) \end{array}\right|\right\}(x) = x,$$

(ii)

$$\left\{\left|\begin{array}{rcl} \mathit{zero} : () & \mapsto & (v, w) \\ \mathit{succ} : (v', w') & \mapsto & (f(v, x), w) \end{array}\right|\right\}(n) = \left(\left\{\left|\begin{array}{lcl} \mathit{zero} : () & \mapsto & v \\ \mathit{succ} : n & \mapsto & f(v, x) \end{array}\right|\right\}(n), w\right)$$

(iii)

$$\{(x, y) \mapsto x\}\left(\left\{\left|\begin{array}{rcl} \mathit{zero} : () & \mapsto & (v, w) \\ \mathit{succ} : (v', w') & \mapsto & (f(v, x), g(v, w, x)) \end{array}\right|\right\}(n)\right) = \left\{\left|\begin{array}{lcl} \mathit{zero} : () & \mapsto & v \\ \mathit{succ} : n & \mapsto & f(v, x) \end{array}\right|\right\}(n),$$

(iv)

$$\left\{\!\!\left|\begin{array}{lcl} \mathit{zero}:() & \mapsto & z \\ \mathit{succ}:x & \mapsto & x \end{array}\right|\!\!\right\}(n) = z.$$

Proof. We prove (ii) and leave the rest as an exercise. We use fold uniqueness over n on the right-hand side.

Let

$$h(n) = \left\{\!\!\left|\begin{array}{lcl} \mathrm{zero}:() & \mapsto & v \\ \mathrm{succ}:n & \mapsto & f(v,x) \end{array}\right|\!\!\right\}(n)$$

then

$$\begin{array}{l|l}
(h(\mathrm{zero}), w) & (h(\mathrm{succ}(n')), w) \\
= (v, w) & = (\{n \mapsto f(v,x)\}(h(n')), w) \\
 & = (f(v,x), w) \\
 & = \{(v', w') \mapsto (f(v,x), w)\}(h(n'), w) \\
\hline
\multicolumn{2}{c}{(h(m), w) = \left\{\!\!\left|\begin{array}{ccc} \mathrm{zero}:() & \mapsto & (v,w) \\ \mathrm{succ}:(v',w') & \mapsto & (f(v,x),w) \end{array}\right|\!\!\right\}(m).}
\end{array}$$

□

2.2.1 The case construct

A crucial observation is the following:

Lemma 2.2 *Suppose* $\{x \mapsto t_0\} : X \longrightarrow Z$ *and* $\{(n,x) \mapsto t_1\} : \mathbb{N} \times X \longrightarrow Z$ *then there is a unique map* g *written*

$$g(m,x) = \left\{\begin{array}{ccc} \mathit{zero}() & \mapsto & t_0 \\ \mathit{succ}(n) & \mapsto & t_1 \end{array}\right\}(m)$$

such that $g(\mathit{zero}, x) = t_0$ *and* $g(\mathit{succ}(m), x) = t_1$.

Proof. Set

$$g(m,x) = \mathrm{p}_0\left(\left\{\!\!\left|\begin{array}{ccc} \mathrm{zero}:() & \mapsto & (t_0, \mathrm{zero}) \\ \mathrm{succ}:(\_, n) & \mapsto & (t_1, \mathrm{succ}(n)) \end{array}\right|\!\!\right\}(m)\right)$$

then clearly $g(\mathrm{zero}, x) = t_0$ and

$$\begin{array}{rcl}
g(\mathrm{succ}(n), x) & = & \mathrm{p}_0(\{(\_, n) \mapsto (t_1, \mathrm{succ}(n))\}\left(\left\{\!\!\left|\begin{array}{ccc} \mathrm{zero}:() & \mapsto & (t_0, \mathrm{zero}) \\ \mathrm{succ}:(\_, n) & \mapsto & (t_1, \mathrm{succ}(n)) \end{array}\right|\!\!\right\}(n)\right) \\
 & = & \{n \mapsto t_1\}\left(\mathrm{p}_1\left(\left\{\!\!\left|\begin{array}{ccc} \mathrm{zero}:() & \mapsto & (t_0, \mathrm{zero}) \\ \mathrm{succ}:(\_, n) & \mapsto & (t_1, \mathrm{succ}(n)) \end{array}\right|\!\!\right\}(n)\right)\right) \\
 & = & \{n \mapsto t_1\}\left(\left\{\!\!\left|\begin{array}{ccc} \mathrm{zero}:() & \mapsto & \mathrm{zero} \\ \mathrm{succ}:(\_, n) & \mapsto & \mathrm{succ}(n) \end{array}\right|\!\!\right\}(n)\right) \\
 & = & \{n \mapsto t_1\}(n) = t_1
\end{array}$$

So that this certainly satisfies the property. Suppose now that some other g' also satifies this property then we could replace the fold by $q(m,x) = (g'(m,x), m)$. However, the fold uniqueness can now be applied:

$$\begin{array}{l|l}
q(\text{zero}, x) & q(\text{succ}(n), x) \\
= (g'(\text{zero}, x), \text{zero}) & = (g'(\text{succ}(n), x), \text{succ}(n)) \\
= (t_0, \text{zero}) & = (t_1, \text{succ}(n)) \\
 & = \{(\_, n) \mapsto (t_1, \text{succ}(n))\}(g'(n, x), n) \\
\hline
\multicolumn{2}{c}{q(m, x) = \left\{\begin{array}{|rcl|} \text{zero} : () & \mapsto & (t_0, \text{zero}) \\ \text{succ} : (\_, n) & \mapsto & (t_1, \text{succ}(n)) \end{array}\right\}(m)}
\end{array}$$

Now $g'(v) = \text{p}_0(q(v)) = g(v)$, so g is unique.

□

This lemma allows us to add to our language the case terms as a way of defining functions together with the following inference rules:

$$\left\{\begin{array}{rcl} \text{zero}() & \mapsto & t_0 \\ \text{succ}(n) & \mapsto & t_1 \end{array}\right\}(\text{zero}) \quad =_x \quad t_0$$

$$\left\{\begin{array}{rcl} \text{zero}() & \mapsto & t_0 \\ \text{succ}(n) & \mapsto & t_1 \end{array}\right\}(\text{succ}(n)) \quad =_x \quad t_1$$

$$\frac{g(\text{zero}) =_x t_0 \qquad g(\text{succ}(n)) = t_1}{g(m) =_x \left\{\begin{array}{rcl} \text{zero}() & \mapsto & t_0 \\ \text{succ}(n) & \mapsto & t_1 \end{array}\right\}(m)}$$

The last rule, case induction, is the most significant as it says any two functions which are equal on zero and succ(n) are equal everywhere. This is, of course, what allows one to break down a proof into cases. The two earlier rules are the choice rules.

2.2.2 The promotion theorem

It is not always the case that something we call a theorem need be hard to prove: sometimes it is the strategic importance of the result rather than its difficulty which causes us to draw attention to it. The Grant Malcolm promotion theorem[Mal90] is one of these. It is a result which is almost so simple that one might well fail to draw attention to it. Yet it is used again and again in proofs: for good reason for promotion is the only way we can pull a function out in front of a fold.

Proposition 2.3 (Promotion) *Suppose $f'(g(x)) = g(f(x))$ then*

$$g\left(\left\{\begin{array}{|rcl|} \mathit{zero} : () & \mapsto & t_0 \\ \mathit{succ} : x & \mapsto & f(x) \end{array}\right\}(m)\right) = \left\{\begin{array}{|rcl|} \mathit{zero} : () & \mapsto & g(t_0) \\ \mathit{succ} : x & \mapsto & f'(x) \end{array}\right\}(m).$$

Proof.

$$\frac{\begin{array}{l|l} g\left(\left\{\!\left|\begin{array}{lcl}\text{zero}:() & \mapsto & t_0\\ \text{succ}:x & \mapsto & f(x)\end{array}\right|\!\right\}(\text{zero})\right) & g\left(\left\{\!\left|\begin{array}{lcl}\text{zero}:() & \mapsto & t_0\\ \text{succ}:x & \mapsto & f(x)\end{array}\right|\!\right\}(\text{succ}(n))\right) \\ = g(t_0) & = g\left(f\left(\left\{\!\left|\begin{array}{lcl}\text{zero}:() & \mapsto & t_0\\ \text{succ}:x & \mapsto & f(x)\end{array}\right|\!\right\}(n)\right)\right) \\ & = f'\left(g\left(\left\{\!\left|\begin{array}{lcl}\text{zero}:() & \mapsto & t_0\\ \text{succ}:x & \mapsto & f(x)\end{array}\right|\!\right\}(n)\right)\right) \\ & = \{x \mapsto f'(x)\}\left(g\left(\left\{\!\left|\begin{array}{lcl}\text{zero}:() & \mapsto & t_0\\ \text{succ}:x & \mapsto & f(x)\end{array}\right|\!\right\}(n)\right)\right) \end{array}}{g\left(\left\{\!\left|\begin{array}{lcl}\text{zero}:() & \mapsto & t_0\\ \text{succ}:x & \mapsto & f(x)\end{array}\right|\!\right\}(n)\right) = \left\{\!\left|\begin{array}{lcl}\text{zero}:() & \mapsto & g(t_0)\\ \text{succ}:x & \mapsto & f'(x)\end{array}\right|\!\right\}(n)}$$

$\square$

2.2.3 Mathematical induction

The formal theory of primitive recursion (that is **charity** with $I\!N$) does not support arbitrary mathematical induction but it does for primitive recursive propositions. A proposition is a (primitive recursive) map $P : X \longrightarrow \text{Bool}$.

We shall assume that we have defined all the usual logical functions on the Boolean type.

Proposition 2.4 *Suppose $P : I\!N \times A \longrightarrow Bool$ is a proposition and*

$$P(zero, a) = true \quad and \quad P(n, a) \Rightarrow P(succ(n), a)$$

then $P(n, a) = true$.

Proof. We may define proposition Q as follows

$$Q(n,a) = \text{p}_0(\left\{\!\left|\begin{array}{rcl}\text{zero}:() & \mapsto & (\text{true}, \text{zero})\\ \text{succ}:(v,m) & \mapsto & (v \wedge P(m,a), \text{succ}(m))\end{array}\right|\!\right\}(n)).$$

We may use the fold uniqueness as follows

$$\frac{\begin{array}{l|l} Q(\text{succ}(\text{zero}), a) & Q(\text{succ}(\text{succ}(n)), a) \\ \quad = P(\text{zero}, a) \wedge \text{true} & \quad = P(\text{succ}(n), a) \wedge Q(\text{succ}(n), a) \\ \quad = \text{true} & \quad = P(\text{succ}(n), a) \wedge P(n, a) \wedge Q(n, a) \\ & \quad = P(n, a) \wedge Q(n, a) \\ & \quad = Q(\text{succ}(n), a) \end{array}}{Q(\text{succ}(n), a) = \left\{\!\left|\begin{array}{lcl}\text{zero}:() & \mapsto & \text{true}\\ \text{succ}:x & \mapsto & x\end{array}\right|\!\right\}(m) = \text{true}}$$

where we use the fact that $P(\text{succ}(n), a) \wedge P(n, a) = P(n, a)$. Now as $Q(\text{zero}, a) = \text{true}$ it follows that $Q(n, a) = \text{true}$ by case induction. In that case

$$P(n, a) = P(n, a) \wedge Q(n, a) = Q(\text{succ}(n), a) = \text{true}\,.$$

$\square$

Mathematical induction is a powerful proof technique. Our current problem is that we do not quite know what is a proposition and what is not!

2.3 The basic results of arithmetic

The basic structure of arithmetic may be viewed as coming in two stages. First the structure of addition and multiplication and how these operations interact: we collect this structure together by proving that the natural numbers under these two operations form a *rig*. Second, one must consider the effect of the truncated subtraction on the system. When this is added we have what is called an *arithmetic rig*.

2.3.1 The natural numbers as a rig

A **rig** is a structure with two operations **addition**, written $x+y$, and **multiplication**, written $x \cdot y$, both of which are associative and have units, respectively 0 and 1. The addition is always assumed to be commutative, and the multiplication distributes over the addition. When the multiplication is commutative the rig is said to be **commutative**.

More formally, a **rig** is a quintuple $(R, +, 0, \cdot, 1)$ where R is an object with operations $+, \cdot : R \times R \longrightarrow R$ and constants $0, 1 : 1 \longrightarrow R$ such that:

[R.1] $(x+y)+z = x+(y+z)$,

[R.2] $x+0 = x = 0+x$,

[R.3] $x+y = y+x$,

[R.4] $(x \cdot y) \cdot z = x \cdot (y \cdot z)$,

[R.5] $x \cdot 1 = x = 1 \cdot x$,

[R.6] $x \cdot (y+z) = x \cdot y + x \cdot z$,

[R.7] $(y+z) \cdot x = y \cdot x + z \cdot x$,

[R.8] $z \cdot 0 = 0 = 0 \cdot z$.

A rig is said to be **commutative** if $x \cdot y = y \cdot x$.

The purpose of this subsection is to establish:

Proposition 2.5 *In the formal theory of primitive recursive arithmetic (**charity** with $I\!N$) or any extension thereof the natural numbers under addition and multiplication form a commutative rig.*

The definition of the operations are as follows:

$$\begin{aligned} x+y &= \left\{\begin{array}{lcl} \text{zero} : () & \mapsto & y \\ \text{succ} : n & \mapsto & \text{succ}(n) \end{array}\right\}(x) \\ x \cdot y &= \left\{\begin{array}{lcl} \text{zero} : () & \mapsto & \text{zero} \\ \text{succ} : n & \mapsto & n+x \end{array}\right\}(y) \end{aligned}$$

We shall prove the proposition in two lemmas:

Lemma 2.6

(i) $zero + y = y$,

(ii) $x + zero = x$,

(iii) $succ(x) + y = succ(x + y)$,

(iv) $x + succ(y) = succ(x + y)$,

(v) $(x + y) + z = x + (y + z)$,

(vi) $x + y = y + x$.

Proof.

(i) By unrolling.

(ii) The following application of fold induction:

$$\frac{\text{zero} + \text{zero} = \text{zero} \;\Big|\; \text{succ}(x) + \text{zero} = \text{succ}(x + \text{zero})}{x + \text{zero} = \left\{\left|\begin{array}{lcl} \text{zero} : () & \mapsto & \text{zero} \\ \text{succ} : m & \mapsto & \text{succ}(m) \end{array}\right|\right\}(x)}$$

where this fold is the identity map.

(iii) By unrolling.

(iv) Promote succ.

(v) We press the x button on the right-hand side:

$$\frac{(\text{zero} + x) + y = x + y \;\Big|\; \begin{array}{l}(\text{succ}(x) + y) + z = \text{succ}(x + y) + z \\ \quad = \text{succ}((x + y) + z)\end{array}}{x + \text{zero} = \left\{\left|\begin{array}{lcl} \text{zero} : () & \mapsto & y + z \\ \text{succ} : m & \mapsto & \text{succ}(m) \end{array}\right|\right\}(x)}$$

Now the fold is $x + (y + z)$.

(vi) We have pressing the y button:

$$\frac{x + \text{zero} = x \;\Big|\; x + \text{succ}(y) = \text{succ}(x + y)}{x + y = \left\{\left|\begin{array}{lcl} \text{zero} : () & \mapsto & x \\ \text{succ} : m & \mapsto & \text{succ}(m) \end{array}\right|\right\}(x) = y + x}$$

$\square$

Now we add the multiplication:

Lemma 2.7

(i) $zero \cdot y = zero$,

(ii) $x \cdot zero = zero$,

(iii) $x \cdot succ(zero) = x$,

(iv) $succ(zero) \cdot y = y$,

(v) $succ(x) \cdot y = y + (x \cdot y)$,

(vi) $x \cdot succ(y) = x + (x \cdot y)$,

(vii) $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$,

(viii) $(x \cdot y) \cdot z = x \cdot (y \cdot z)$,

(ix) $x \cdot y = y \cdot x$.

Proof.

(i) To prove $\mathrm{zero} \cdot y = \mathrm{zero}$ unroll.

(ii) To prove $x \cdot \mathrm{zero} = \mathrm{zero}$ we push the buttons of x:

$$\begin{array}{r|l}
\mathrm{zero} \cdot \mathrm{zero} = \mathrm{zero} & \mathrm{succ}(x) \cdot \mathrm{zero} = (x \cdot \mathrm{zero}) + \mathrm{zero} \\
 & \quad = x \cdot \mathrm{zero} \\
\hline
\multicolumn{2}{c}{x \cdot \mathrm{zero} = \left\{\!\left| \begin{array}{lcl} \mathrm{zero} : () & \mapsto & \mathrm{zero} \\ \mathrm{succ} : m & \mapsto & m \end{array} \right|\!\right\} (x) = \mathrm{zero}}
\end{array}$$

(iii) To show $x \cdot \mathrm{succ}(\mathrm{zero}) = x$ unroll.

(iv) To show $\mathrm{succ}(\mathrm{zero}) \cdot y = y$ we press the y buttons:

$$\begin{array}{r|l}
\mathrm{succ}(\mathrm{zero}) \cdot \mathrm{zero} = \mathrm{zero} & \mathrm{succ}(\mathrm{zero}) \cdot \mathrm{succ}(y) = \mathrm{succ}(\mathrm{zero}) \cdot y + \mathrm{succ}(\mathrm{zero}) \\
 & \quad = \mathrm{succ}(\mathrm{zero}) \cdot y + \mathrm{succ}(\mathrm{zero}) \\
 & \quad = \mathrm{succ}(\mathrm{succ}(\mathrm{zero}) \cdot y) \\
\hline
\multicolumn{2}{c}{\mathrm{succ}(\mathrm{zero}) \cdot y = \left\{\!\left| \begin{array}{lcl} \mathrm{zero} : () & \mapsto & \mathrm{zero} \\ \mathrm{succ} : m & \mapsto & \mathrm{succ}(m) \end{array} \right|\!\right\} (y) = y}
\end{array}$$

(v) To obtain $x \cdot \mathrm{succ}(y) = x + (x \cdot y)$ unroll and use the commutativity of $+$.

(vi) To prove $\mathrm{succ}(x) \cdot y = y + (x \cdot y)$ we press the y buttons:

$$\begin{array}{r|l}
\mathrm{zero} + x \cdot \mathrm{zero} = \mathrm{zero} & \mathrm{succ}(y) + x \cdot \mathrm{succ}(y) \\
 & \quad = \mathrm{succ}(y) + x + x \cdot y \\
 & \quad = \mathrm{succ}(x) + y + x \cdot y \\
\hline
\multicolumn{2}{c}{\mathrm{succ}(x) \cdot y = \left\{\!\left| \begin{array}{lcl} \mathrm{zero} : () & \mapsto & \mathrm{zero} \\ \mathrm{succ} : m & \mapsto & m + \mathrm{succ}(x) \end{array} \right|\!\right\} (y) = \mathrm{succ}(x) \cdot y}
\end{array}$$

(vii) To prove $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ we try pushing the buttons of z to reduce each side in turn to the same fold:

$$\begin{array}{c|l} x\cdot(y+\mathrm{zero}) = x\cdot y & x\cdot(y+\mathrm{succ}(z)) \\ & \quad = x\cdot\mathrm{succ}(y+z) \\ & = x\cdot(y+z)+x \\ \hline \multicolumn{2}{c}{\mathrm{succ}(x)\cdot y = \left\{\!\!\left|\begin{array}{lcl}\mathrm{zero}:() & \mapsto & x\cdot y\\ \mathrm{succ}:m & \mapsto & m+x\end{array}\right|\!\!\right\}(z)}\end{array}$$

Now for the other side we have:

$$\begin{array}{c|l} x\cdot y + x\cdot\mathrm{zero} = x\cdot y & x\cdot y + x\cdot\mathrm{succ}(z) \\ & \quad = x\cdot y + x\cdot z + x \\ \hline \multicolumn{2}{c}{x\cdot y + x\cdot z = \left\{\!\!\left|\begin{array}{lcl}\mathrm{zero}:() & \mapsto & x\cdot y\\ \mathrm{succ}:m & \mapsto & m+x\end{array}\right|\!\!\right\}(z)}\end{array}$$

Thus, as each side can be reduced to the same fold, we may conclude they are equal.

(viii) To prove $(x\cdot y)\cdot z = x\cdot(y\cdot z)$ we shall push the z buttons in the right-hand side:

$$\begin{array}{l|l} x\cdot(y\cdot\mathrm{zero}) = x\cdot\mathrm{zero} & x\cdot(y\cdot\mathrm{succ}(z)) \\ \quad = \mathrm{zero} & \quad = x\cdot(y+y\cdot z) \\ & \quad = x\cdot y + x\cdot(y\cdot z) \\ \hline \multicolumn{2}{c}{x\cdot(y\cdot z) = \left\{\!\!\left|\begin{array}{lcl}\mathrm{zero}:() & \mapsto & \mathrm{zero}\\ \mathrm{succ}:m & \mapsto & m+x\cdot y\end{array}\right|\!\!\right\}(z) = (x\cdot y)\cdot z}\end{array}$$

(ix) To obtain $x\cdot y = y\cdot x$ we push the buttons of y in the right-hand side:

$$\begin{array}{c|c} \mathrm{zero}\cdot x = \mathrm{zero} & \mathrm{succ}(y)\cdot x = y\cdot x + x \\ \hline \multicolumn{2}{c}{y\cdot x = \left\{\!\!\left|\begin{array}{lcl}\mathrm{zero}:() & \mapsto & \mathrm{zero}\\ \mathrm{succ}:m & \mapsto & m+x\end{array}\right|\!\!\right\}(y) = x\cdot y}\end{array}$$

□

2.3.2 The natural numbers as an arithmetic rig

An **arithmetic rig** is a (commutative) rig with an extra **cancellation** operator:

$$\dot{-} : R\times R \longrightarrow R$$

satisfying the following conditions:

[AR.1] $(x \dot{-} y)\cdot z = (x\cdot z) \dot{-} (y\cdot z)$,

[AR.2] $x \dot{-} 0 = x$,

[AR.3] $(x \dot{-} y) \dot{-} z = x \dot{-} (y+z)$,

[AR.4] $(x+y) \dot{-} z = (x \dot{-} (z \dot{-} y)) + (y \dot{-} z)$,

[AR.5] $(1 \dot{-} x)\cdot x = 0$,

[AR.6] $(x \dot{-} y) \cdot (y \dot{-} x) = 0$.

A **weak arithmetic rig** satisfies all the axioms except **[AR.6]**. Notice the first three axioms are valid in any ring, however, the last three are peculiar to arithmetic rigs.

The purpose of this subsection is to establish that the natural numbers with the "cancellation structure" given by monus, see below, forms an arithmetic rig:

Proposition 2.8 *In the formal theory of primitive recursive arithmetic the natural numbers under addition, multiplication, and monus form an arithmetic rig.*

The definition of monus is as follows:

$$\begin{aligned} \mathrm{pred}(x) &= \left\{ \begin{array}{lcl} \mathrm{zero}() & \mapsto & \mathrm{zero}() \\ \mathrm{succ}(n) & \mapsto & n \end{array} \right\}(x) \\ x \dot{-} y &= \left\{\!\!\left| \begin{array}{lcl} \mathrm{zero} : () & \mapsto & x \\ \mathrm{succ} : n & \mapsto & \mathrm{pred}(n) \end{array} \right|\!\!\right\}(x) \end{aligned}$$

The proofs are somewhat more difficult than those for the previous section as the definition of monus involves a case construct. This means that the proofs will often center on a case analysis.

Three lemmas are useful (actually the second is crucial!):

Lemma 2.9

(i) $zero \dot{-} x = zero$,

(ii) $x \dot{-} x = zero$,

(iii) $pred(x) \dot{-} y = pred(x \dot{-} y)$,

(iv) $succ(x) \dot{-} succ(y) = x \dot{-} y$,

(v) $(x + y) \dot{-} y = x$.

These are straightforward to prove. It helps to notice that *(iii)* is obtained by applying the promotion theorem.

Lemma 2.10 *Let*

$$h(x, y, v_0, v_1, v_2) = \left\{ \begin{array}{lcl} zero() & \mapsto & \left\{ \begin{array}{lcl} zero() & \mapsto & v_1 \\ succ(\_) & \mapsto & v_0 \end{array} \right\}(x \dot{-} y) \\ succ(\_) & \mapsto & \left\{ \begin{array}{lcl} zero() & \mapsto & v_0 \\ succ(\_) & \mapsto & v_2 \end{array} \right\}(x \dot{-} y) \end{array} \right\}(succ(y) \dot{-} x)$$

(i) $h(succ(x), succ(y), v_0, v_1, v_2) = h(x, y, v_0, v_1, v_2)$,

(ii) $h(zero, y, v_0, v_1, v_2) = v_0$,

(iii) $h(succ(x), y, v_0, v_1, v_2) = h(x, pred(y), v_0, v_1, v_2)$,

(iv) $h(x, y, v_0, v_1, v_2) = h(pred(x), pred(y), v_0, v_1, v_2)$,

(v) $h(x, y, v_0, v_1, v_2) = h(x \dot{-} z, y \dot{-} z, v_0, v_1, v_2)$,

(vi) $h(x, y, v_0, v_1, v_2) = v_0$.

Proof.

(i) The values of the conditions are unchanged.

(ii) The only applicable pattern is the third.

(iii) We may work by cases on y:

Case $y =$ zero: $h(\text{succ}(x), \text{zero}, v_0, v_1, v_2) = v_0$ and $h(x, \text{pred}(y), v_0, v_1, v_2) = v_0$.
Case $y =$ succ(y'): Then $h(\text{succ}(x), \text{succ}(y'), v_0, v_1, v_2) = h(x, y', v_0, v_1, v_2) = h(x, \text{pred}(y), v_0, v_1, v_2)$.

To be formal, case induction over y would be used. Applying case induction to the left-hand side of the equation:

$$\frac{\begin{array}{l} h(\text{succ}(x), \text{zero}, v_0, v_1, v_2) \\ \quad = \left\{\begin{array}{lcl} \text{zero}() & \mapsto & v_1 \\ \text{succ}(\_) & \mapsto & v_0 \end{array}\right\}(succ(x)) \\ \quad = v_0 \end{array} \;\Bigg|\; \begin{array}{l} h(\text{succ}(x), \text{succ}(y'), v_0, v_1, v_2) \\ \quad = h(x, y', v_0, v_1, v_2) \end{array}}{h(\text{succ}(x), y, v_0, v_1, v_2) = \left\{\begin{array}{lcl} \text{zero}() & \mapsto & v_0 \\ \text{succ}(m) & \mapsto & h(x, m, v_0, v_1, v_2) \end{array}\right\}(y)}$$

For the right-hand side, a lemma is required. Let

$$h'(x) = \left\{\begin{array}{lcl} \text{zero}() & \mapsto & \left\{\begin{array}{lcl} \text{zero}() & \mapsto & v_1 \\ \text{succ}(\_) & \mapsto & v_0 \end{array}\right\}(x) \\ \text{succ}(\_) & \mapsto & \left\{\begin{array}{lcl} \text{zero}() & \mapsto & v_0 \\ \text{succ}(\_) & \mapsto & v_2 \end{array}\right\}(x) \end{array}\right\}(\text{succ}(\text{zero}) \dot{-} x).$$

Lemma 2.11 $\quad h'(x) = v_0$.

Proof. We proceed by case induction over x.

$$\frac{h'(\text{zero}) = v_0 \;\big|\; h'(\text{succ}(\text{zero})) = v_0}{h'(x) = \left\{\begin{array}{lcl} \text{zero}() & \mapsto & v_0 \\ \text{succ}(m) & \mapsto & v_0 \end{array}\right\}(x) = v_0}$$

□

$$\frac{\begin{array}{l} h(x, \text{pred}(\text{zero}), v_0, v_1, v_2) \\ = \left\{\begin{array}{lcl} \text{zero}() & \mapsto & \left\{\begin{array}{lcl} \text{zero}() & \mapsto & v_1 \\ \text{succ}(\_) & \mapsto & v_0 \end{array}\right\}(x \dot{-} \text{zero}) \\ \text{succ}(\_) & \mapsto & \left\{\begin{array}{lcl} \text{zero}() & \mapsto & v_0 \\ \text{succ}(\_) & \mapsto & v_2 \end{array}\right\}(x \dot{-} \text{zero}) \end{array}\right\}(\text{succ}(\text{zero}) \dot{-} x) \\ = h'(x) = v_0 \end{array} \;\Bigg|\; \begin{array}{l} h(x, \text{pred}(\text{succ}(y')), v_0, v_1, v_2) \\ \quad = h(x, y', v_0, v_1, v_2) \end{array}}{h(x, \text{pred}(y), v_0, v_1, v_2) = \left\{\begin{array}{lcl} \text{zero}() & \mapsto & v_0 \\ \text{succ}(m) & \mapsto & h(x, m, v_0, v_1, v_2) \end{array}\right\}(y)}$$

(iv) We may work by cases on x:

Case $x =$ zero: $h(\text{zero}, y, v_0, v_1, v_2) = v_0 = h(\text{pred}(\text{zero}), \text{pred}(y), v_0, v_1, v_2)$.

Case $x =$ succ(x'): Then $h(\text{succ}(x'), y, v_0, v_1, v_2) = h(x', \text{pred}(y), v_0, v_1, v_2)$
$= h(\text{pred}(x), \text{pred}(y), v_0, v_1, v2)$.

(v) Let us hit the z button:

$$\frac{\begin{array}{l|l} h(x \dot{-} \text{zero}, y \dot{-} \text{zero}, v_0, v_1, v_2) & h(x \dot{-} \text{succ}(z'), y \dot{-} \text{succ}(z'), v_0, v_1, v_2) \\ = h(x, y, v_0, v_1, v_2) & = h(\text{pred}(x \dot{-} z'), \text{pred}(y \dot{-} z'), v_0, v_1, v_2) \\ & = h(x \dot{-} z', y \dot{-} z', v_0, v_1, v_2) \end{array}}{h(x \dot{-} z, y \dot{-} z, v_0, v_1, v_2) = \left\{\begin{array}{|lll|} \text{zero} : () & \mapsto & h(x, y, v_0, v_1, v_2) \\ \text{succ} : n & \mapsto & n \end{array}\right\}(z) = h(x, y, v_0, v_1, v_2)}$$

(vi)

$$\begin{aligned} h(x, y, v_0, v_1, v_2) &= h(x \dot{-} x, y \dot{-} x, v_0, v_1, v_2) \\ &= h(\text{zero}, y \dot{-} x, v_0, v_1, v_2) \\ &= v_0 \end{aligned}$$

□

The importance of lemma 2.10 is it tells us that to test for equality it suffices to test only the cases

$$\text{succ}(y) \dot{-} x = \text{zero} \quad \text{and} \quad x \dot{-} y = \text{succ}(\_)$$

$$\text{succ}(y) \dot{-} x = \text{succ}(\_) \quad \text{and} \quad x \dot{-} y = \text{zero}\,.$$

Lemma 2.12

$$\mathit{succ}(z) \dot{-} y = \left\{\begin{array}{lll} \mathit{zero}() & \mapsto & \mathit{zero} \\ \mathit{succ}(\_) & \mapsto & \mathit{succ}(z \dot{-} y) \end{array}\right\}(\mathit{succ}(z) \dot{-} y)$$

Proof. Perform a case analysis using y

Case $y =$ zero: The LHS reduces to succ(z) while the RHS takes the second phrase and reduces to succ$(z \dot{-} \text{zero}) = \text{succ}(z)$.

Case $y =$ succ(y'): The LHS reduces to $z \dot{-} y'$ while the RHS reduces as follows:

$$\begin{aligned} &\left\{\begin{array}{lll} \text{zero}() & \mapsto & \text{zero} \\ \text{succ}(\_) & \mapsto & \text{succ}(z \dot{-} \text{succ}(y')) \end{array}\right\}(\text{succ}(z) \dot{-} \text{succ}(y')) \\ &= \left\{\begin{array}{lll} \text{zero}() & \mapsto & z \dot{-} y' \\ \text{succ}(\_) & \mapsto & \text{succ}(\text{pred}(z \dot{-} y')) \end{array}\right\}(z \dot{-} y') \\ &= \left\{\begin{array}{lll} \text{zero}() & \mapsto & z \dot{-} y' \\ \text{succ}(v) & \mapsto & \text{succ}(\text{pred}(\text{succ}(v))) \end{array}\right\}(z \dot{-} y') \end{aligned}$$

$$\begin{aligned}
&= \left\{\begin{array}{lcl} \text{zero}() & \mapsto & z \dot{-} y' \\ \text{succ}(v) & \mapsto & \text{succ}(v) \end{array}\right\} (z \dot{-} y') \\
&= \left\{\begin{array}{lcl} \text{zero}() & \mapsto & z \dot{-} y' \\ \text{succ}(v) & \mapsto & z \dot{-} y' \end{array}\right\} (z \dot{-} y') \\
&= z \dot{-} y'
\end{aligned}$$

□

We shall leave as an exercise all the above results except [**AR. 4**] we prove below:

Lemma 2.13 $(x + y) \dot{-} z = (x \dot{-} (z \dot{-} y)) + (y \dot{-} z)$.

Proof. We shall push the z buttons in the RHS:

$$\begin{array}{r|ll}
x \dot{-} (\text{zero} \dot{-} y) & (x \dot{-} (\text{succ}(z') \dot{-} y)) + (y \dot{-} \text{succ}(z')) & \\
+(y \dot{-} \text{zero}) & \textbf{Case } \text{succ}(z') \dot{-} y = \text{zero} & \textbf{Case } \text{succ}(z') \dot{-} y = \text{succ}(z'') \\
= x + y & \textbf{and } \ y \dot{-} z' = \text{succ}(y''): & \textbf{and } y \dot{-} z' = \text{zero} \\
 & = (x \dot{-} \text{zero}) + (y \dot{-} \text{succ}(z')) & = (x \dot{-} \text{succ}(z \dot{-} y)) + (y \dot{-} \text{succ}(z')) \\
 & = x + \text{pred}(y \dot{-} z') & = \text{pred}(x \dot{-} (z' \dot{-} y)) + (y \dot{-} \text{succ}(z')) \\
 & = x + \text{pred}(\text{succ}(y'')) & = \text{pred}(x \dot{-} (z' \dot{-} y)) + \text{pred}(y \dot{-} z') \\
 & = x + y'' & = \text{pred}(x \dot{-} (z' \dot{-} y)) + \text{zero} \\
 & = \text{pred}(x + \text{succ}(y'')) & = \text{pred}((x \dot{-} (z' \dot{-} y)) + \text{zero}) \\
 & = \text{pred}((x \dot{-} \text{zero}) + (y \dot{-} z')) & = \text{pred}((x \dot{-} (z' \dot{-} y)) + (y \dot{-} z')) \\
 & = \text{pred}((x \dot{-} \text{pred}(\text{zero})) + (y \dot{-} z')) & \\
 & = \text{pred}((x \dot{-} \text{pred}(\text{succ}(z') \dot{-} y)) + (y \dot{-} z')) & \\
 & = \text{pred}((x \dot{-} (\text{pred}(\text{succ}(z')) \dot{-} y)) + (y \dot{-} z')) & \\
 & = \text{pred}((x \dot{-} (z' \dot{-} y)) + (y \dot{-} z')) & \\
\hline
\end{array}$$

$$x \dot{-} (z \dot{-} y) + (y \dot{-} z) = \left\{\begin{array}{|lcl|} \text{zero} : () & \mapsto & x + y \\ \text{succ} : n & \mapsto & \text{pred}(n) \end{array}\right\} (z) = (x + y) \dot{-} z$$

□

3 Manipulations of lists

In **charity** lists are defined as follows

```
data list(A) -> C = nil: 1 -> C
                  | cons: A * C -> C.
```

Three common functions associated with lists are:

$$\mathrm{append}(x,y) = \left\{\!\!\left|\begin{array}{rcl} \mathrm{nil}:() & \mapsto & y \\ \mathrm{cons}:(a,y') & \mapsto & \mathrm{cons}(a,y') \end{array}\right|\!\!\right\}(x)$$

$$\mathrm{flatten}(l) = \left\{\!\!\left|\begin{array}{rcl} \mathrm{nil}:() & \mapsto & [] \\ \mathrm{cons}:(a,z) & \mapsto & \mathrm{append}(a,z) \end{array}\right|\!\!\right\}(x)$$

$$\mathrm{reverse}(x) = \left\{\!\!\left|\begin{array}{rcl} \mathrm{nil}:() & \mapsto & [] \\ \mathrm{cons}:(a,r) & \mapsto & \mathrm{append}(r,[a]) \end{array}\right|\!\!\right\}(x).$$

We shall explore their basic properties using the fold induction, unrolling, choice, case, and map rules for lists.

3.1 Inference rules for list

The fold induction rule for lists takes the following form:

$$\frac{h([]) =_x t_0 \;\Big|\; h(\mathrm{cons}(y,ys)) =_{x,y,ys} \{(y,z)\mapsto t_1(y,z)\}(y,h(ys))}{h(ys) =_{x,ys} \left\{\!\!\left|\begin{array}{rcl} \mathrm{nil}:() & \mapsto & t_0 \\ \mathrm{cons}:(y,z) & \mapsto & t_1(y,z) \end{array}\right|\!\!\right\}(ys).}$$

It asserts the uniqueness of the fold expression as that map which satisfies the list iteration. This rule together with the unrolling rules for lists determine all the properties of a list. The unrolling rules are:

$$\begin{array}{rcl} \left\{\!\!\left|\begin{array}{rcl} \mathrm{nil}:() & \mapsto & t_0 \\ \mathrm{cons}:(y,z) & \mapsto & t_1(y,z) \end{array}\right|\!\!\right\}([]) & = & t_0 \\ \left\{\!\!\left|\begin{array}{rcl} \mathrm{nil}:() & \mapsto & t_0 \\ \mathrm{cons}:(y,z) & \mapsto & t_1(y,z) \end{array}\right|\!\!\right\}(\mathrm{cons}(y',ys)) & = & t_1(y, \left\{\!\!\left|\begin{array}{rcl} \mathrm{nil}:() & \mapsto & t_0 \\ \mathrm{cons}:(y,z) & \mapsto & t_1(y,z) \end{array}\right|\!\!\right\}(ys)). \end{array}$$

While all that follows is a consequence of the above rules it is useful to have independently and explicitly the remaining rules. The case (flat) induction rule for lists takes the following form:

$$\frac{h([]) =_x t_0 \;\Big|\; h(\mathrm{cons}(y,ys)) =_{x,y,ys} t_1(y,ys)}{h(y') =_{x,y'} \left\{\begin{array}{rcl} \mathrm{nil}() & \mapsto & t_0 \\ \mathrm{cons}(y,ys) & \mapsto & t_1(y,ys) \end{array}\right\}(y').}$$

This asserts that a map from a list is uniquely determined by its effect on empty and non-empty lists. The choice rules for lists are:

$$\begin{aligned}
\left\{\begin{array}{rcl} \mathrm{nil}() & \mapsto & t_0 \\ \mathrm{cons}(y, ys) & \mapsto & t_1(y, ys) \end{array}\right\}([]) &=_x t_0 \\
\left\{\begin{array}{rcl} \mathrm{nil}() & \mapsto & t_0 \\ \mathrm{cons}(y, ys) & \mapsto & t_1(y, ys) \end{array}\right\}(\mathrm{cons}(y', ys')) &= t_1(y', ys')
\end{aligned}$$

Finally, we have the map induction rule:

$$\frac{h([]) =_x [] \;\big|\; h(\mathrm{cons}(y, ys)) =_{x,y,ys} \mathrm{cons}(f(y), h(ys))}{h(ys) =_{x,ys} \mathrm{list}\{y \mapsto f(y)\}(ys).}$$

with its unrolling rules:

$$\begin{aligned}
\mathrm{list}\{y' \mapsto f(y')\}([]) &= [] \\
\mathrm{list}\{y' \mapsto f(y')\}(\mathrm{cons}(y, ys)) &= \mathrm{cons}(f(y), \mathrm{list}\{y \mapsto f(y)\}(ys)).
\end{aligned}$$

An important consequence of the fold induction rule is the Grant Malcolm "promotion" theorem:

Lemma 3.1

$$\frac{g_0(x) = h(f_0(x)) \;\big|\; g_1(a, h(z), x) = h(f_1(a, z, x))}{\left\{\!\left|\begin{array}{rcl} nil: () & \mapsto & g_0(x) \\ cons: (y, z) & \mapsto & g_1(y, z, x) \end{array}\right|\!\right\}(ys) = h(\left\{\!\left|\begin{array}{rcl} nil: () & \mapsto & f_0(x) \\ cons: (y, z) & \mapsto & f_1(y, z, x) \end{array}\right|\!\right\}(ys)).}$$

Proof.

$$\frac{\begin{array}{l|l}
\begin{array}{l} h(\left\{\!\left|\begin{array}{rcl} \mathrm{nil}: () & \mapsto & f_0(x) \\ \mathrm{cons}: (y, z) & \mapsto & f_1(y, z, x) \end{array}\right|\!\right\}([])) \\ \quad = h(f_0(x)) = g_0(x) \end{array}
&
\begin{array}{l} h(\left\{\!\left|\begin{array}{rcl} \mathrm{nil}: () & \mapsto & f_0(x) \\ \mathrm{cons}: (y, z) & \mapsto & f_1(y, z, x) \end{array}\right|\!\right\}(\mathrm{cons}(v, vs))) \\
\quad = h(f_1(v, \left\{\!\left|\begin{array}{rcl} \mathrm{nil}: () & \mapsto & f_0(x) \\ \mathrm{cons}: (y, z) & \mapsto & f_1(y, z, x) \end{array}\right|\!\right\}(vs), x)) \\
\quad = g_1(v, h(\left\{\!\left|\begin{array}{rcl} \mathrm{nil}: () & \mapsto & f_0(x) \\ \mathrm{cons}: (y, z) & \mapsto & f_1(y, z, x) \end{array}\right|\!\right\}(vs)), x) \end{array}
\end{array}}{h(\left\{\!\left|\begin{array}{rcl} \mathrm{nil}: () & \mapsto & f_0(x) \\ \mathrm{cons}: (y, z) & \mapsto & f_1(y, z, x) \end{array}\right|\!\right\}(vs)) = \left\{\!\left|\begin{array}{rcl} \mathrm{nil}: () & \mapsto & g_0(x) \\ \mathrm{cons}: (y, z) & \mapsto & g_1(y, z, x) \end{array}\right|\!\right\}(vs).}$$

□

This result is often used to "promote" function into the body of a fold.

3.2 Basic list results

Lists naturally form monoids and furthermore give rise to a monad. The monoid and monad properties of lists are very basic to the manipulation of lists.

3.2.1 Lists as a monoid

The object list(A) forms a monoid precisely when there is a unit $[]$ and a multiplication append(x, y) satifying the following identities:

Proposition 3.2

(i) append$([], y) = y$*,*

(ii) append$(x, []) = x$*,*

(iii) append$(x,$ *append*$(y, z)) =$ *append*$($*append*$(x, y), z))$*.*

Proof. (i) append$([], y) = y$ is immediate from unrolling.

(ii)

$$\frac{\text{append}([],[]) = [] \;\Big|\; \text{append}(\text{cons}(x, xs), []) = \text{cons}(x, \text{append}(xs, []))}{\text{append}(x, []) = x}$$

(iii)

$$\frac{\text{append}(\text{append}([], y), z) = \text{append}(y, z) \;\left|\; \begin{array}{l} \text{append}(\text{append}(\text{cons}(x, xs), y), z) \\ \quad = \text{append}(\text{cons}(x, \text{append}(xs, y)), z) \\ \quad = \text{cons}(x, \text{append}(\text{append}(xs, y), z)) \end{array}\right.}{\text{append}(\text{append}(x, y), z) = \left\{\begin{array}{rcl} \text{nil} : () & \mapsto & \text{append}(y, z) \\ \text{cons} : (x, z) & \mapsto & \text{cons}(x, z) \end{array}\right\}(x) = \text{append}(x, \text{append}(y, z)).}$$

$\square$

3.2.2 Lists as a monad

We start with the observation that flatten : list(list$(A)) \longrightarrow$ list(A) is a homomorphism of list monoids:

Lemma 3.3

(i) flatten$([]) = []$*,*

(ii) flatten$($*append*$(x, y)) =$ *append*$($*flatten*$(x),$ *flatten*$(y))$*.*

Proof. The only difficulty is the second part. For this we start by working with the left-hand side:

$$\frac{\text{flatten}(\text{append}([], y)) = \text{flatten}(y) \;\left|\; \begin{array}{l} \text{flatten}(\text{append}(\text{cons}(x, xs), y)) \\ \quad = \text{flatten}(\text{cons}(x, \text{append}(xs, y))) \\ \quad = \text{append}(x, \text{flatten}(\text{append}(xs, y))) \end{array}\right.}{\text{flatten}(\text{append}(x, y)) = \left\{\begin{array}{rcl} \text{nil} : () & \mapsto & \text{flatten}(y) \\ \text{cons} : (x', z) & \mapsto & \text{append}(a, z) \end{array}\right\}(x)}$$

Next with the right-hand side:

$$\frac{\text{append}(\text{flatten}([]), \text{flatten}(y)) = \text{flatten}(y) \quad \Bigg| \quad \begin{array}{l}\text{append}(\text{flatten}(\text{cons}(x, xs)), \text{flatten}(y)) \\ \quad = \text{append}(\text{append}(x, \text{flatten}(xs)), \text{flatten}(y)) \\ \quad = \text{append}(x, \text{append}(\text{flatten}(xs), \text{flatten}(y)))\end{array}}{\text{append}(\text{flatten}(x), \text{flatten}(y)) = \left\{\!\!\left|\begin{array}{rcl}\text{nil} : () & \mapsto & \text{flatten}(y) \\ \text{cons} : (x', z) & \mapsto & \text{append}(a, z)\end{array}\right|\!\!\right\}(x)}$$

Whence the result is immediate.

□

In order to show that the triple (list, $\{a \mapsto [a]\}$, flatten) is a monad we need to prove:

Proposition 3.4

(i) flatten$([as]) = as$,

(ii) flatten(list$\{a \mapsto [a]\}(as)) = as$,

(iii) flatten(list$\{xs \mapsto$ *flatten*$(xs)\}(xss)) =$ *flatten(flatten*$(xss))$.

Proof.

(i) This is a straightforward unrolling:

$$\begin{aligned}\text{flatten}([as]) &= \left\{\!\!\left|\begin{array}{rcl}\text{nil} : () & \mapsto & [] \\ \text{cons} : (a, z) & \mapsto & \text{append}(a, z)\end{array}\right|\!\!\right\}([as]) \\ &= \text{append}(as, []) \\ &= as.\end{aligned}$$

(ii)

$$\frac{\begin{array}{l}\text{flatten}(\text{list}\{a \mapsto [a]\}([])) \\ \quad = \text{flatten}([]) = []\end{array} \quad \Bigg| \quad \begin{array}{l}\text{flatten}(\text{list}\{a \mapsto [a]\}(\text{cons}(a, as))) \\ \quad = \text{flatten}(\text{cons}([a], \text{list}\{a \mapsto [a]\}(as))) \\ \quad = \left\{\!\!\left|\begin{array}{rcl}\text{nil} : () & \mapsto & [] \\ \text{cons} : (a, z) & \mapsto & \text{append}(a, z)\end{array}\right|\!\!\right\} \\ \qquad (\text{cons}([a], \text{list}\{a \mapsto [a]\}(as))) \\ \quad = \text{append}([a], \left\{\!\!\left|\begin{array}{rcl}\text{nil} : () & \mapsto & [] \\ \text{cons} : (a, z) & \mapsto & \text{append}(a, z)\end{array}\right|\!\!\right\}(\text{list}\{a \mapsto [a]\}(as))) \\ \quad = \text{cons}(a, \left\{\!\!\left|\begin{array}{rcl}\text{nil} : () & \mapsto & [] \\ \text{cons} : (a, z) & \mapsto & \text{append}(a, z)\end{array}\right|\!\!\right\}(\text{list}\{a \mapsto [a]\}(as)))\end{array}}{\text{flatten}(\text{list}\{a \mapsto [a]\}(as)) = as}$$

(iii)

$$\begin{array}{r|l} \mathrm{flatten}(\mathrm{flatten}([])) = [] & \mathrm{flatten}(\mathrm{flatten}(\mathrm{cons}(xs, xss))) \\ & \quad = \mathrm{flatten}(\mathrm{append}(xs, \mathrm{flatten}(xss))) \\ & \quad = \mathrm{append}(\mathrm{flatten}(xs), \mathrm{flatten}(\mathrm{flatten}(xss))) \\ \hline \multicolumn{2}{c}{\mathrm{flatten}(\mathrm{flatten}(xss)) = \mathrm{flatten}(\mathrm{list}\{as \mapsto \mathrm{flatten}(as)\}(xss))} \end{array}$$

where we used the generalization given by the lemma above.

□

If one examines this proof one will see that naturally from it arises the need for a lemma. The lemma, however, is more specific than the one given above. In the process of automating the trick is to spot when a lemma is required, however, to generalize the particular form which arises from the proof before attempting to prove it. This often simplifies the search for a proof considerably.

3.3 Basic results for reverse

3.3.1 Reverse as an involution

The map reverse is an involution of the list monoid and the monad. It is an involution of a monoid precisely as the following equations holding:

Proposition 3.5

(i) $\mathit{reverse}([]) = []$,

(ii) $\mathit{reverse}(\mathit{append}(x, y)) = \mathit{append}(\mathit{reverse}(y), \mathit{reverse}(x))$,

(iii) $\mathit{reverse}(\mathit{reverse}(x)) = x$.

Proof.

(i) Immediate from unrolling.

(ii) Working with the left-hand side of the equation we obtain:

$$\begin{array}{l|l} \mathrm{reverse}(\mathrm{append}([], y)) & \mathrm{reverse}(\mathrm{append}(\mathrm{cons}(x, xs), y)) \\ \quad = \mathrm{reverse}(y) & \quad = \mathrm{reverse}(\mathrm{cons}(x, \mathrm{append}(xs, y))) \\ & \quad = \mathrm{append}(\mathrm{reverse}(\mathrm{append}(xs, y)), [x]) \\ \hline \multicolumn{2}{c}{\mathrm{reverse}(\mathrm{append}(x, y)) = \left\{\!\!\left|\begin{array}{rcl} \mathrm{nil} : () & \mapsto & \mathrm{reverse}(y) \\ \mathrm{cons} : (x, z) & \mapsto & \mathrm{append}(z, [x]) \end{array}\right|\!\!\right\}(x).} \end{array}$$

Now working with the right-hand side we have:

$$\begin{array}{l|l} \mathrm{append}(\mathrm{reverse}(y), \mathrm{reverse}([])) & \mathrm{append}(\mathrm{reverse}(y), \mathrm{reverse}(\mathrm{cons}(x, xs))) \\ \quad = \mathrm{reverse}(y) & \quad = \mathrm{append}(\mathrm{reverse}(y), \mathrm{append}(\mathrm{reverse}(xs), [x])) \\ & \quad = \mathrm{append}(\mathrm{append}(\mathrm{reverse}(y), \mathrm{reverse}(xs)), [x]) \\ \hline \multicolumn{2}{c}{\mathrm{append}(\mathrm{reverse}(y), \mathrm{reverse}(x)) = \left\{\!\!\left|\begin{array}{rcl} \mathrm{nil} : () & \mapsto & \mathrm{reverse}(y) \\ \mathrm{cons} : (x, z) & \mapsto & \mathrm{append}(z, [x]) \end{array}\right|\!\!\right\}(x).} \end{array}$$

Thus $\text{reverse}(\text{append}(x, y)) = \text{append}(\text{reverse}(y), \text{reverse}(x))$ as desired.

(iii)

$$\begin{array}{r|l} \text{reverse}(\text{reverse}([])) = [] & \text{reverse}(\text{reverse}(\text{cons}(x, xs))) \\ & \quad = \text{reverse}(\text{append}(\text{reverse}(xs), [x])) \\ & \quad = \text{append}(\text{reverse}([x]), \text{reverse}(\text{reverse}(xs))) \\ & \quad = \text{append}([x]\ \text{reverse}(\text{reverse}(xs))) \\ & \quad = \text{cons}(x, \text{reverse}(\text{reverse}(xs)))) \\ \hline \multicolumn{2}{c}{\text{reverse}(\text{reverse}(x)) = x.} \end{array}$$

□

We now show that the natural transformation $\text{reverse} : \text{list}(A) \longrightarrow \text{list}(A)$ is an endomorphism of the list monad. This is provided by:

Proposition 3.6

(i) $\mathit{reverse}([x]) = [x]$,

(ii) $\mathit{flatten}(\mathit{reverse}(\mathit{list}\{as \mapsto \mathit{reverse}(as)\}(aas)) = \mathit{reverse}(\mathit{flatten}(aas))$.

Proof. The first part is provided by unrolling. The second part is given by getting a direct fold form for each side:

$$\begin{array}{r|l} \text{reverse}(\text{flatten}([])) = [] & \text{reverse}(\text{flatten}(\text{cons}(as, aas))) \\ & \quad = \text{reverse}(\text{append}(as, \text{flatten}(aas))) \\ & \quad = \text{append}(\text{reverse}(\text{flatten}(aas)), \text{reverse}(as)) \\ \hline \text{reverse}(\text{flatten}(aas)) = & \left\{\begin{array}{rcl} \text{nil} : () & \mapsto & [] \\ \text{cons} : (as, z) & \mapsto & \text{append}(z, \text{reverse}(as)) \end{array}\right\}(aas). \end{array}$$

$$\begin{array}{l|l} \text{flatten}(\text{reverse} & \text{flatten}(\text{reverse}(\text{list}\{as' \mapsto \text{reverse}(as')\}(\text{cons}(as, aas))) \\ (\text{list}\{as \mapsto \text{reverse}(as)\}([])) & \quad = \text{flatten}(\text{reverse}(\text{cons}(\text{reverse}(as), \text{list}\{as' \mapsto \text{reverse}(as')\}(aas))) \\ \quad = \text{flatten}(\text{reverse}([])) & \quad = \text{flatten}(\text{append}(\text{reverse}(\text{list}\{as' \mapsto \text{reverse}(as')\}(aas), \text{cons}(\text{reverse}(as), [])) \\ \quad = [] & \quad = \text{append}(\text{flatten}(\text{reverse}(\text{list}\{as' \mapsto \text{reverse}(as')\}(aas)), \\ & \qquad\quad \text{flatten}(\text{cons}(\text{reverse}(as), []))) \\ & \quad = \text{append}(\text{flatten}(\text{reverse}(\text{list}\{as' \mapsto \text{reverse}(as')\}(aas)), \text{reverse}(as))) \\ \hline \text{flatten}(\text{reverse}(\text{list}\{as \mapsto \text{reverse}(as)\}(aas)) = & \left\{\begin{array}{rcl} \text{nil} : () & \mapsto & [] \\ \text{cons} : (as, z) & \mapsto & \text{append}(\text{reverse}(z), \text{reverse}(as)) \end{array}\right\}(aas). \end{array}$$

□

3.3.2 Fast reverse is equivalent to its specification: naive reverse

The way of defining reverse, described above, is often called the "naive reverse" as it is highly inefficient having a quadratic complexity. To obtain a linear time version we need to transform this into the following form:

$$\text{rev}(x) = \text{p}_0(\text{shuntf}([], x)).$$

where

$$\text{shunt}(x,y) = \left\{\begin{array}{rcl} \text{nil}() & \mapsto & (x, []) \\ \text{cons}(a, y') & \mapsto & (\text{cons}(a,x), y') \end{array}\right\}(y).$$

and

$$\text{shuntf}(x,y) = \left\{\!\!\left\{\begin{array}{rcl} \text{nil} : () & \mapsto & (x, y) \\ \text{cons} : (\_, z) & \mapsto & \text{shunt}(z) \end{array}\right\}\!\!\right\}(y).$$

The naive reverse should be viewed as a specification and the fast reverse the transformation of this specification which can be used as the actual implementation. We need therefore to show how we may transform one version of reverse into the other. For this we need a preliminary lemma:

Lemma 3.7 $\{(v,w) \mapsto (\mathit{append}(v,r), w)\}(\mathit{shunt}(z)) = \mathit{shunt}(\{(v,w) \mapsto (\mathit{append}(v,r),w)\}(z))$.

Proof. The proof is by cases:

$$z = (x, []) \quad \text{or} \quad z = (x, \text{cons}(y, ys))$$

$z = (x, [])$:

$$\begin{array}{ll} \multicolumn{2}{l}{\{(v,w) \mapsto (\text{append}(v,r), w)\}(\text{shunt}(x, []))} \\ = & \{(v,w) \mapsto (\text{append}(v,r), w)\}(x, []) \\ = & (\text{append}(x,r), []) \\ = & \text{shunt}(\text{append}(x,r), []) \\ = & \text{shunt}(\{(v,w) \mapsto (\text{append}(v,r), w)\}(x, [])) \end{array}$$

$z = (x, \text{cons}(y, ys))$:

$$\begin{array}{ll} \multicolumn{2}{l}{\{(v,w) \mapsto (\text{append}(v,r), w)\}(\text{shunt}(x, \text{cons}(y, ys)))} \\ = & \{(v,w) \mapsto (\text{append}(v,r), w)\}(\text{cons}(y,x), ys) \\ = & (\text{append}(\text{cons}(y,x), r), ys) \\ = & (\text{cons}(y, (\text{append}(x,r)), ys) \\ = & \text{shunt}(\{(v,w) \mapsto (\text{append}(v,r), w)\}(x, \text{cons}(y, ys))) \end{array}$$

□

We now can prove:

Proposition 3.8 *reverse*$(x) =$ *rev*(x).

Proof. We shall do this by pushing buttons

$$\frac{\text{rev}([]) = [] \quad \left| \quad \begin{array}{l} \text{rev}(\text{cons}(x, xs)) = \text{p}_0(\text{shuntf}([], \text{cons}(x, xs))) \\ \quad = \text{p}_0(\left\{ \begin{array}{rcl} \text{nil} : () & \mapsto & ([], \text{cons}(x, xs)) \\ \text{cons} : (\_, z) & \mapsto & \text{shunt}(z) \end{array} \right\} (\text{cons}(x, xs))) \\ \quad = \text{p}_0(\{(\_, z) \mapsto \text{shunt}(\left\{ \begin{array}{rcl} \text{nil} : () & \mapsto & ([], \text{cons}(x, xs)) \\ \text{cons} : (\_, z) & \mapsto & \text{shunt}(z) \end{array} \right\} (z))\}(x, xs)) \\ \quad = \text{p}_0(\left\{ \begin{array}{rcl} \text{nil} : () & \mapsto & ([x], xs) \\ \text{cons} : (\_, z) & \mapsto & \text{shunt}(z) \end{array} \right\} (xs)) \\ \quad = \text{p}_0(\left\{ \begin{array}{rcl} \text{nil} : () & \mapsto & \{(v, w) \mapsto (\text{append}(v, [x]), w)\}([], xs) \\ \text{cons} : (\_, z) & \mapsto & \text{shunt}(z) \end{array} \right\} (xs)) \\ \quad = \text{p}_0(\{(v, w) \mapsto (\text{append}(v, r), w)\}(\left\{ \begin{array}{rcl} \text{nil} : () & \mapsto & ([], xs) \\ \text{cons} : (\_, z) & \mapsto & \text{shunt}(z) \end{array} \right\} (xs)) \\ \quad = \text{append}(\text{p}_0(\text{shuntf}([], xs)), [x])) \\ \quad = \text{append}(\text{rev}(xs), [x]) \end{array} \right.}{\text{rev}(x) = \text{reverse}(x)}$$

Notice that we have used the promotion theorem twice: once to get the shunt inside and one to get the $\{(v, w) \mapsto (\text{append}(v, r), w)\}$ phrase outside.

$\square$

The pattern to this proof is given by reading it from its ends inward! That is one knows what one wants and where one starts. One wants the append outside: working backwards we see that we need a promotion lemma for the append over shunt. Working the forwards one simply promotes the shunt. Thus, this is a theorem that we might expect a theorem prover to find relatively easily.

4 Manipulations of colists

In **charity** colists are defined as follows

```
data C -> colist(A) = nxt: C -> 1 + A * C.
```

We may then define an append for colists as follows:

$$\mathrm{app}(x,y) = (\!|(x,y) \mapsto \mathrm{nxt} : \left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \mathrm{b}_0() \\ \mathrm{b}_1(a,y') & \mapsto & \mathrm{b}_1(a,(\epsilon,y')) \end{array} \right\} (\mathrm{nxt}(y)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,(x',y)) \end{array} \right\} (\mathrm{nxt}(x))|\!)(x,y)$$

where $\epsilon = (\mathrm{nxt} : \mathrm{b}_0())$.

We wish to prove that that $(\mathrm{colist}(A), \epsilon, \mathrm{app})$ is a monoid. To do this we shall need to use the inference rules for colists.

4.1 Inference rules for colists

In some ways the inference rules for the coinductive datatypes are more complex. The main rule is that of unfold coinduction, it asserts the uniqueness of the unfold to colists:

$$\frac{\mathrm{nxt}(t(x)) = \left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \mathrm{b}_0() \\ \mathrm{b}_1(a,v) & \mapsto & \mathrm{b}_1(a,t(v)) \end{array} \right\} (t_1(x))}{t(x) = (\!|v \mapsto \mathrm{nxt} : t_1(v)|\!)(x)}.$$

Coupled with this rule is the destruction rule for a colist:

$$\begin{array}{l} \mathrm{nxt}((\!|v \mapsto \mathrm{nxt} : t_1|\!)(x)) \\ \quad = \left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \mathrm{b}_0() \\ \mathrm{b}_1(a,z) & \mapsto & \mathrm{b}_1(a,(\!|v \mapsto \mathrm{nxt} : t_1|\!)(z)) \end{array} \right\} (\{v \mapsto t_1\}(x)). \end{array}$$

Specializing this to the append function we obtain:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathrm{nxt}(\mathrm{app}(x,y))} \\ = & \mathrm{nxt}((\!|(x,y) \mapsto \mathrm{nxt} : \left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \mathrm{b}_0() \\ \mathrm{b}_1(a,y') & \mapsto & \mathrm{b}_1(a,(\epsilon,y')) \end{array} \right\} (\mathrm{nxt}(y)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,(x',y)) \end{array} \right\} (\mathrm{nxt}(x))|\!)(x,y)) \\ = & \left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \mathrm{b}_0() \\ \mathrm{b}_1(a,z) & \mapsto & \mathrm{b}_1(a,\mathrm{app}(z)) \end{array} \right\} (\left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \mathrm{b}_0() \\ \mathrm{b}_1(a,y') & \mapsto & \mathrm{b}_1(a,(\epsilon,y')) \end{array} \right\} (\mathrm{nxt}(y)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,(x',y)) \end{array} \right\} (\mathrm{nxt}(x))) \\ = & \left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \left\{ \begin{array}{ccc} \mathrm{b}_0() & \mapsto & \mathrm{b}_0() \\ \mathrm{b}_1(a,y') & \mapsto & \mathrm{b}_1(a,\mathrm{app}(\epsilon,y')) \end{array} \right\} (\mathrm{nxt}(y)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,\mathrm{app}(x',y)) \end{array} \right\} (\mathrm{nxt}(x)) \end{array}$$

These are large expressions to carry about – alright for computers but unwieldy for us! For the time being we shall simply work with the unexpiated expressions.

There are four remaining groups of inference rules: two for the records and two for the map. The record destruction rule and the record formation rule in the case of colists are very trivial (as the definition only has one phrase). However, for completeness, they are respectively:

$$\mathrm{nxt}(\mathrm{nxt} : t) =_x t$$

and

$$\frac{\mathrm{nxt}(\mathrm{nxt} : t) =_x h}{t =_x h}.$$

Finally, the two rules for the map, map destruction and map coinduction, are respectively:

$$\mathrm{nxt}(\mathrm{colist}\{a \mapsto f(a)\}(x)) = \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a, xr) & \mapsto & \mathrm{b}_1(f(a), \mathrm{colist}\{a \mapsto f(a)\}(xr)) \end{array}\right\}(\mathrm{nxt}(x))$$

$$\frac{\mathrm{nxt}(t(x)) = \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0() \\ \mathrm{b}_1(a, v) & \mapsto & (f(a), t(v)) \end{array}\right\}(\mathrm{nxt}(x))}{t(x) = \mathrm{colist}\{a \mapsto f(a)\}(x)}.$$

4.2 Colists as a monoid

We shall now prove that that $(\mathrm{colist}(A), \epsilon, \mathrm{app})$ is a monoid. This is given by:

Proposition 4.1

(i) $app(\epsilon, y) = y$,

(ii) $app(x, \epsilon) = x$,

(iii) $app(app(x, y), z) = app(x, app(y, z))$.

Proof.

(i)

$$\frac{\begin{array}{l} \mathrm{nxt}(\mathrm{app}(\epsilon, y)) \\ = \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a, y') & \mapsto & \mathrm{b}_1(a, \mathrm{app}(\epsilon, y')) \end{array}\right\}(\mathrm{nxt}(y)) \\ \mathrm{b}_1(a, x') & \mapsto & \mathrm{b}_1(a, \mathrm{app}(x', y)) \end{array}\right\}(\mathrm{b}_0) \\ = \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a, y') & \mapsto & \mathrm{b}_1(a, \mathrm{app}(\epsilon, y')) \end{array}\right\}(\mathrm{nxt}(y)) \end{array}}{\mathrm{app}(\epsilon, y) = (\!|v \mapsto \mathrm{nxt} : \mathrm{nxt}(v)|\!)(y) = y}$$

(ii)

$$\frac{\begin{array}{l} \mathrm{nxt}(\mathrm{app}(x, \epsilon)) \\ = \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a, y') & \mapsto & \mathrm{b}_1(a, \mathrm{app}(\epsilon, y')) \end{array}\right\}(\mathrm{b}_0()) \\ \mathrm{b}_1(a, x') & \mapsto & \mathrm{b}_1(a, \mathrm{app}(x', \epsilon)) \end{array}\right\}(\mathrm{nxt}(x)) \\ = \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a, x') & \mapsto & \mathrm{b}_1(a, \mathrm{app}(x', \epsilon)) \end{array}\right\}(\mathrm{nxt}(x)) \end{array}}{\mathrm{app}(x, \epsilon) = (\!|v \mapsto \mathrm{nxt} : \mathrm{nxt}(v)|\!)(x) = x}$$

(iii)

$$
\begin{array}{l}
\mathrm{nxt}(\mathrm{app}(\mathrm{app}(x,y),z)) \\
= \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,z') & \mapsto & \mathrm{b}_1(a,z') \end{array}\right\}(\mathrm{nxt}(z)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,\mathrm{app}(x',z)) \end{array}\right\}(\mathrm{nxt}(\mathrm{app}(x,y))) \\
= \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,z') & \mapsto & \mathrm{b}_1(a,z') \end{array}\right\}(\mathrm{nxt}(z)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,\mathrm{app}(x',z)) \end{array}\right\} \\
\quad (\left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,y') & \mapsto & \mathrm{b}_1(a,y') \end{array}\right\}(\mathrm{nxt}(y)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,\mathrm{app}(x',y))) \end{array}\right\}(\mathrm{nxt}(x))) \\
= \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,z') & \mapsto & \mathrm{b}_1(a,z') \end{array}\right\}(\mathrm{nxt}(z)) \\ \mathrm{b}_1(a,y') & \mapsto & \mathrm{b}_1(a,\mathrm{app}(y',z)) \end{array}\right\}(\mathrm{nxt}(y)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,\mathrm{app}(\mathrm{app}(x',y),z)) \end{array}\right\}(\mathrm{nxt}(x)) \\
= \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,(x,y,z)) & \mapsto & \mathrm{b}_1(a,\mathrm{app}(\mathrm{app}(x,y),z)) \end{array}\right\} \\
\quad (\left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,z') & \mapsto & \mathrm{b}_1(a,(\epsilon,\epsilon,z')) \end{array}\right\}(\mathrm{nxt}(z)) \\ \mathrm{b}_1(a,y') & \mapsto & \mathrm{b}_1(a,(\epsilon,y',z)) \end{array}\right\}(\mathrm{nxt}(y)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,(x',y,z)) \end{array}\right\}(\mathrm{nxt}(x))) \\
\hline
\mathrm{app}(\mathrm{app}(x,y),z) = (\!|(x,y,z) \mapsto \mathrm{nxt} : t_1(x,y,z)|\!)(x,y,z)
\end{array}
$$

where

$$
\begin{array}{l}
t_1(x,y,z) \\
= \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,z') & \mapsto & \mathrm{b}_1(a,(\epsilon,\epsilon,z')) \end{array}\right\}(\mathrm{nxt}(z)) \\ \mathrm{b}_1(a,y') & \mapsto & \mathrm{b}_1(a,(\epsilon,y',z)) \end{array}\right\}(\mathrm{nxt}(y)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,(x',y,z)) \end{array}\right\}(\mathrm{nxt}(x))
\end{array}
$$

similarly we have:

$$
\begin{array}{l}
\mathrm{nxt}(\mathrm{app}(x,\mathrm{app}(y,z))) \\
= \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,z') & \mapsto & \mathrm{b}_1(a,z') \end{array}\right\}(\mathrm{nxt}(\mathrm{app}(y,z))) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,\mathrm{app}(x',\mathrm{app}(y,z))) \end{array}\right\}(\mathrm{nxt}(x)) \\
= \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,z') & \mapsto & \mathrm{b}_1(a,z') \end{array}\right\}(\mathrm{nxt}(z)) \\ \mathrm{b}_1(a,y') & \mapsto & \mathrm{b}_1(a,\mathrm{app}(y',z)) \end{array}\right\}(\mathrm{nxt}(y)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,\mathrm{app}(x',\mathrm{app}(y,z))) \end{array}\right\}(\mathrm{nxt}(x)) \\
= \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,(x,y,z)) & \mapsto & \mathrm{b}_1(a,\mathrm{app}(x,\mathrm{app}(y,z))) \end{array}\right\} \\
\quad (\left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \left\{\begin{array}{rcl} \mathrm{b}_0() & \mapsto & \mathrm{b}_0 \\ \mathrm{b}_1(a,z') & \mapsto & \mathrm{b}_1(a,(\epsilon,\epsilon,z')) \end{array}\right\}(\mathrm{nxt}(z)) \\ \mathrm{b}_1(a,y') & \mapsto & \mathrm{b}_1(a,(\epsilon,y',z)) \end{array}\right\}(\mathrm{nxt}(y)) \\ \mathrm{b}_1(a,x') & \mapsto & \mathrm{b}_1(a,(x',y,z)) \end{array}\right\}(\mathrm{nxt}(x))) \\
\hline
\mathrm{app}(\mathrm{app}(x,y),z) = (\!|(x,y,z) \mapsto \mathrm{nxt} : t_1(x,y,z)|\!)(x,y,z)
\end{array}
$$

This completes the proof.

□

References

[CF92] J. R. B. Cockett and T. Fukushima. About charity. Technical Report 92/480/18, The University of Calgary, June 1992.

[Mal90] G. Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, University of Groningen, 1990.

[Spe92] Dwight L. Spencer. *Categorical Programming with Functorial Strength.* PhD thesis, Oregon Graduate Institute of Science & Technology, 1992.