Contents

1 Type Theory for CCC + records, co-records, data types

1.1 LogicRules

1.1.1 Structural rules

We first introduce the standard rules of logic, with types.

$$\frac{A \text{ atomic.}}{x :: A \vdash x :: A}\ atom \qquad \frac{\Gamma_1, \Gamma_2 \vdash t :: Y}{\Gamma_1, x :: X, \Gamma_2 \vdash t :: Y}\ weak$$

$$\frac{\Gamma_1, x_1 :: X, x_2 :: X, \Gamma_2 \vdash t :: Y}{\Gamma_1, x' :: X, \Gamma_2 \vdash t[\![x'/x_1, x'/x_2]\!] :: Y}\ contraction$$

$$\frac{\Gamma_1, x_1 :: X, x_2 :: X, \Gamma_2 \vdash t :: Y}{\Gamma_1, x_2 :: X, x_1 :: X, \Gamma_2 \vdash t :: Y}\ exchange \qquad \frac{\Gamma \vdash t :: C \quad \Gamma_1, x :: C, \Gamma_2 \vdash t' :: Y}{\Gamma_1, \Gamma, \Gamma_2 \vdash [\![x \mapsto t']\!]t :: Y}\ cut$$

$$\begin{array}{c} \textit{create context} \\ \boxed{\begin{array}{c} \Gamma_1 \\ \hline \\ \vdots \\ \hline \Gamma_2 \vdash \Delta \end{array}} \\ \Gamma_1, \Gamma_2 \vdash \Delta \end{array} \qquad \begin{array}{c} \textit{introduce context} \\ \boxed{\begin{array}{c} \Gamma_1, x : X, \Gamma_2 \\ \hline \\ \dfrac{}{\Gamma \vdash x : X}\ context \\ \vdots \end{array}} \\ \cdots \end{array}$$

These rules are the typical rules found in logic systems. Informally, we may think of them in this manner:

atom This rule states that whenever you know of a type, you may state that having an element of that type gives you an element of that type.

weak This rule states that whenever you can infer something from a set of statements, you may add another statement to the set and still infer the same thing.

contraction This rule allows you to substitute a single item for items having the same type.

exchange We are allowed to commute the left hand suppositions.

cut Also known as *modus ponens*, this rule states:

- Assume we can derive $t :: C$ from Γ.
- And we can derive $t' :: Y$ from a set including a variable x of type C.
- Then, from Γ and the second set less the variable x we may derive that the function that takes x to t' applied to t is of type Y. In lambda notation, we would have $(\lambda\, x.t')t$ is of type Y.

create context We may remove any set of rules on the left and place them in a *context* box. Within the box, we need to derive the entailment where the context rules have been removed from the left hand side.

introduce context Within a context box, we may introduce any element of the context on the right at any time.

1.1.2 Co-records, sums, co-products

$$\frac{\{\Gamma_1, x_c :: X_c, \Gamma_2 \vdash t_c :: Y\}_{c\in C}}{\Gamma_1, x :: \{x_c : X_c\}_{c\in C}, \Gamma_2 \vdash \{c(x_c) \mapsto t_c\}z :: Y}\ \textit{cotuple} \qquad \frac{\Gamma \vdash t :: Y_k,\ k \in C}{\Gamma \vdash \bar{k} \cdot t :: \{c : Y_c\}_{c\in C}}\ \textit{coprod}$$

$$\frac{\{\Gamma \vdash t_c :: Y_c\}_{c\in C}}{\Gamma \vdash (c : t_c) :: (c : Y_c),\ c \in C}\ \textit{tuple}$$

$$\frac{\Gamma_1, x_1 :: X_1, \ldots, x_n :: X_n, \Gamma_2 \vdash t :: Y}{\Gamma_1, (c_1 : x_1, \ldots, c_n : x_n) :: (c_1 : X_1, \ldots, c_n : X_n), \Gamma_2 \vdash t :: Y}\ \textit{prod.pattern}$$

$$\frac{\Gamma \vdash s :: Y \qquad \Gamma, x :: X \vdash t :: Z}{\Gamma, f :: Y \Rightarrow X \vdash t[\![f@s/x]\!] :: Z}\ \textit{application} \qquad \frac{x :: X, \Gamma \vdash t :: Y}{\Gamma \vdash \lambda x.t :: X \Rightarrow Y}\ \textit{abstraction}$$

In the case of tuples, there is an equivalent notation which we will use in some proofs. In this alternative, the elements are written as a simple unlabelled tuple and the types as a product.

$$(0{:}t_0, 1{:}t_1, \ldots n{:}t_n){::}(0{:}Y_0, 1{:}Y_1, \ldots n{:}Y_n) \equiv (t_0, t_1, \ldots, t_n){::}(Y_0 \times Y_1 \times \cdots \times Y_n)$$

cotuple The requirement for this rule is that we have a series of entailments above the line that have all the unique elements of a coproduct in their left hand sides and the types of the right hand sides are all the same. We may then combine the individual left hand elements of the coproduct below the line into one element having the coproduct as it's type. The resulting right hand side is an application of the coproduct to a new variable of the common right hand type.

coprod When we can derive one term of a type that is part of a coproduct, we may then conclude that we can derive the coproduct type with the appropriate selector applied to the term we had on the right hand side of the entailment.

tuple Similarly to *cotuple*, we require having a series of entailments above the line. The difference is that this rule requires all the left hand sides to be the same and the right hand sides to be all the occurences of a give product type. When this happens, we may then conclude that the left hand side entails the product type with the member terms being the terms we had on the right hand side of the entailments.

prod.pattern When we have all the individual types of a product on the left hand side allowing us to entail a term, we can combine those individual types into their product and still entail the same term.

application If we have two entailments, where in the first, the left hand side entails a term of type Y and in the second, the left hand side is the same as the first with the addition of a term of type X and it entails a term t of type Z, then we can conclude that given the common parts of the left hand sides, a function of type $Y \Rightarrow X$, we can entail that the term t with all occurences of the subterm of type X replaced by the function applied to the term of type Y will also be of type Z. Said still another way, we can replace terms of a particular type with function applications to other terms that result in the same type.

abstraction If we can entail a particular term of type Y from some term of type X and some other set of terms, then we conclude that the other set of terms by themselvs will entail a function of type $X \Rightarrow Y$

1.1.3 Inductive data types

$$
\begin{array}{c}
\textit{Circular rule} \\
\hline
\begin{array}{c|c}
z_i := \mu x.P(x) & z_1, \ldots, z_n, \Gamma_1 \vdash Y \\
\text{(new type vars)} & \text{(Assumption)}
\end{array} \\
\hline
\vdots \\
\hline
x_1' :: P(z_1), \ldots, x_n' :: P(n_1), y' :: \Gamma_1 \vdash t :: Y \\
\hline
x_1 :: \mu x.P_1(x), \ldots x_n :: \mu x.P_n(x), y :: \Gamma_1, z :: \Gamma_2 \vdash \\
\quad \mu f.[(:= x_1', \ldots, x_n', y') \mapsto t](x_1, \ldots, x_n, y) :: Y
\end{array}
$$

$$\frac{\Gamma \vdash t :: P(\mu x.P(x))}{\Gamma \vdash \bar{\mu} \cdot \bar{t} :: \mu x.P(x)} \; \textit{construction}$$

1.1.4 Co-inductive data types

$$
\begin{array}{c}
\textit{Circular rule} \\
\hline
\begin{array}{c|c}
z := \nu x.P(x) & \Gamma \vdash z \\
\text{(new type vars)} & \text{(Assumption)}
\end{array} \\
\hline
\vdots \\
\hline
y' :: \Gamma_1 \vdash t :: P(z) \\
\hline
y :: \Gamma_1, w :: \Gamma_2 \vdash \nu g.[\![y' \mapsto t]\!]y :: \nu x.P(x)
\end{array}
$$

$$\frac{\Gamma_1, s :: P(\nu x.P(x), \Gamma_2 \vdash t :: Y}{\Gamma_1, \nu \cdot s :: \nu x.P(x), \Gamma_2 \vdash t :: Y} \; \textit{destruction}$$

1.2 Examples

1.2.1 *append*

Our first example is that of *append*. This function operates on $\mathbb{L}(A)$ where

$$\mathbb{L}(A) = \mu x.\{\text{nil:}(), \text{cons:}(0{:}A, 1{:}x)\}$$

The proof/definition of *append* is:

$$X := \mathbb{L}(A) \quad \Big| \quad X, \mathbb{L}(A) \vdash_{\text{app}} \mathbb{L}(A)$$

$$\dfrac{\dfrac{\dfrac{}{y'{::}\mathbb{L}(A) \vdash y'{::}\mathbb{L}(A)}\,(3)}{(){::}(), y'{::}\mathbb{L}(A) \vdash y'{::}\mathbb{L}(A)}\,(2) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v{::}A \vdash v{::}A}\,(9)}{v{::}A, vs{::}X, y'{::}\mathbb{L}(A) \vdash v{::}A}\,(8)}{(v,vs){::}(A\times X), y'{::}\mathbb{L}(A) \vdash v{::}A}\,(7) \quad \dfrac{\dfrac{\dfrac{X, \mathbb{L}(A) \vdash_{\text{app}} \mathbb{L}(A)}{vs{::}X, y'{::}\mathbb{L}(A) \vdash \text{app}(vs,y'){::}\mathbb{L}(A)}\,(12)}{v{::}A, vs{::}X, y'{::}\mathbb{L}(A) \vdash \text{app}(vs,y'){::}\mathbb{L}(A)}\,(11)}{(v,vs){::}(A\times X), y'{::}\mathbb{L}(A) \vdash \text{app}(vs,y'){::}\mathbb{L}(A)}\,(10)}{(v,vs){::}(A\times X), y'{::}\mathbb{L}(A) \vdash (v,\text{app}(vs,y')){::}A\times\mathbb{L}(A)}\,(6)}{(v,vs){::}(A\times X), y'{::}\mathbb{L}(A) \vdash \overline{\text{cons}}(v,\text{app}(vs,y')){::}\{nil{:}(), cons{:}A\times\mathbb{L}(A)\}}\,(5)}{(v,vs){::}(A\times X), y'{::}\mathbb{L}(A) \vdash \overleftarrow{\text{cons}}(v,\text{app}(vs,y')){::}\mathbb{L}(A)}\,(4)}{x'{::}\{\text{nil}{:}(), \text{cons}{:}(A\times X)\}, y'{::}\mathbb{L}(A) \vdash \left\{ \begin{array}{l} \text{nil} \mapsto y' \\ \text{cons}(v,vs) \mapsto \bar{\mu}\cdot\overline{\text{cons}}(v,\text{app}(vs,y')) \end{array} \right\} x'{::}\mathbb{L}(A)}\,(1)$$

$$x{::}\mathbb{L}(A), y{::}\mathbb{L}(A) \vdash \mu\text{app.}\left[\; := x', y' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto y' \\ \text{cons}(v,vs) \mapsto \overleftarrow{\text{cons}}(v,\text{app}(vs,y')) \end{array} \right\} (x') \;\right] (x,y)$$

The first step of this proof was obviously the application of the circular inductive rule. Each of the other steps are explained below.

1. *cotuple* In this rule, the antecedent has two parts, corresponding to our choices on the right and the elements in x' on the left.
2. *weak* We weaken by adding $(){::}()$ to the antecedent.
3. *atom* identity on Atoms is immediately derivable.
4. *construction* Whenever a $\bar{\mu}$ appears on the right, we do the construction, which unrolls the type.
5. *coprod* Co-product selection of a particular construction function, in this case cons.
6. *tuple* In this rule, we break down the tuple of $(v, \text{app}(vs, y'))$, giving us two antecedents.
7. *product pattern* We break down the tuple (v, vs) on the left hand side. This gives us a single antecedent with each item of the tuple written separately.
8. *weak* where we add two items, $vs{::}X$ and $y'{::}\mathbb{L}(A)$ to the left hand side.
9. *atom* Atomic rule on A.
10. *tuple* In this rule, we break down the tuple of $(v, \text{app}(vs, y'))$, giving us two antecedents. (Note this is the same as the application in step 7.)
11. *weak* where we add one item, $v{::}A$ to the left hand side.
12. *induction* as we have a match in our inductive hypothesis and the form of the inference.

1.2.2 *append* with a context.

In this section, we repeat the proof of *append*, but with a context. This function operates on $\mathbb{L}(A)$ but will be defined slightly differently.

$$\mathbb{L}(A) = \mu x.\{\text{nil}{:}(), \text{cons}{:}(0{:}A, 1{:}x)\}$$

The proof/definition of *append* is:

$$y{::}\mathbb{L}(A)$$

$$X := \mathbb{L}(A) \quad \Big| \quad X \vdash_{\text{app}} \mathbb{L}(A)$$

$$\dfrac{\dfrac{\dfrac{}{\vdash y{::}\mathbb{L}(A)}\,(3)}{()::() \vdash y{::}\mathbb{L}(A)}\,(2) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v{::}A \vdash v{::}A}\,(9)}{v{::}A, vs{::}X \vdash v{::}A}\,(8)}{(v,vs){::}(A\times X) \vdash v{::}A}\,(7) \quad \dfrac{\dfrac{\dfrac{X \vdash_{\text{app}} \mathbb{L}(A)}{vs{::}X \vdash \text{app } vs{::}\mathbb{L}(A)}\,(12)}{v{::}A, vs{::}X \vdash \text{app } vs{::}\mathbb{L}(A)}\,(11)}{(v,vs){::}(A\times X) \vdash \text{app } vs{::}\mathbb{L}(A)}\,(10)}{(v,vs){::}(A\times X) \vdash (v, \text{app } vs){::}A\times\mathbb{L}(A)}\,(6)}{(v,vs){::}(A\times X) \vdash \overline{\text{cons}}(v, \text{app } vs){::}\{nil{:}(), cons{:}A\times\mathbb{L}(A)\}}\,(5)}{(v,vs){::}(A\times X) \vdash \overleftarrow{\text{cons}}(v, \text{app } vs){::}\mathbb{L}(A)}\,(4)}{x'{::}\{\text{nil}{:}(), \text{cons}{:}(A\times X)\} \vdash \left\{\begin{array}{l}\text{nil} \mapsto y \\ \text{cons}(v,vs) \mapsto \bar{\mu}\cdot\overline{\text{cons}}(v, \text{app } vs)\end{array}\right\} x'{::}\mathbb{L}(A)}\,(1)$$

$$x{::}\mathbb{L}(A) \vdash \mu\text{app.}\left[\ := x' \mapsto \left\{\begin{array}{l}\text{nil} \mapsto y \\ \text{cons}(v,vs) \mapsto \overleftarrow{\text{cons}}(v, \text{app } vs)\end{array}\right\}(x')\ \right] x{::}\mathbb{L}(A)$$

$$x{::}\mathbb{L}(A), y{::}\mathbb{L}(A) \vdash \mu\text{app.}\left[\ := x' \mapsto \left\{\begin{array}{l}\text{nil} \mapsto y \\ \text{cons}(v,vs) \mapsto \overleftarrow{\text{cons}}(v, \text{app } vs)\end{array}\right\}(x')\ \right] x{::}\mathbb{L}(A)$$

The first two steps of this proof were to apply the context introduction rule, followed by an application of the circular inductive rule. Each of the other steps are explained below.

1. *cotuple* In this rule, the antecedent has two parts, corresponding to our choices on the right and the elements in x' on the left.
2. *weak* We weaken by adding $()::()$ to the antecedent.
3. *context* We may introduce $y{::}\mathbb{L}(A)$ on the right at any time.
4. *construction* Whenever a $\bar{\mu}$ appears on the right, we do the construction, which unrolls the type.
5. *coprod* Co-product selection of a particular construction function, in this case cons.
6. *tuple* In this rule, we break down the tuple of $(v, \text{app}(vs, y'))$, giving us two antecedents.
7. *product pattern* We break down the tuple (v, vs) on the left hand side. This gives us a single antecedent with each item of the tuple written separately.
8. *weak* where we add two items, $vs{::}X$ and $y'{::}\mathbb{L}(A)$ to the left hand side.
9. *atom* Atomic rule on A.
10. *tuple* In this rule, we break down the tuple of $(v, \text{app}(vs, y'))$, giving us two antecedents. (Note this is the same as the application in step 7.)
11. *weak* where we add one item, $v{::}A$ to the left hand side.
12. *induction* as we have a match in our inductive hypothesis and the form of the inference.

1.2.3 Proof of the associativity of *append*.

In section 1.2.2 we proved that

$$append(x,y) = \mu\text{app}.\left[\ := x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto y \\ \text{cons}(v,vs) \mapsto \overleftarrow{\text{cons}}(v, \text{app}[\![y]\!](vs)) \end{array} \right\} (x')\ \right] x{::}\mathbb{L}(A)$$

This has demand on the first argument of the interior app. Thefore, our first step is to unroll this definiton once, giving us:

$$append(x,y) = \text{app}[\![y]\!](x) = \mu\_.\left[\ := x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto y \\ \text{cons}(v,vs) \mapsto \overleftarrow{\text{cons}}(v, \text{app}[\![y]\!](vs)) \end{array} \right\} (x')\ \right] (x)$$

where the demand is now exposed. From here, we can examine $\text{app}[\![y]\!](vs)$ in further investigation. To determine associativity,

$$append(append(x,y),z)$$

$$= \text{app}[\![z]\!](\text{app}[\![y]\!](x))$$

$$= \text{app}[\![z]\!]\left(\mu\_.\left[\ := x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto y \\ \text{cons}(v,vs) \mapsto \bar{\mu} \cdot \overline{\text{cons}}(v, \text{app}[\![y]\!](vs)) \end{array} \right\} (x')\ \right] (x)\right) \tag{1.1}$$

$$= \mu\_.\left[\ := x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto \text{app}[\![z]\!]y \\ \text{cons}(v,vs) \mapsto \text{app}[\![z]\!]\,(\bar{\mu} \cdot \overline{\text{cons}}(v, \text{app}[\![y]\!](vs))) \end{array} \right\} (x')\ \right] (x) \tag{1.2}$$

$$= \mu\_.\left[\ := x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto \text{app}[\![z]\!]y \\ \text{cons}(v,vs) \mapsto \bar{\mu} \cdot \overline{\text{cons}}(v, \text{app}[\![z]\!](\text{app}[\![y]\!](vs))) \end{array} \right\} (x')\ \right] (x) \tag{1.3}$$

Given that we can see our original pattern $(\text{app}[\![z]\!](\text{app}[\![y]\!](x)))$ appears once again in Equation (1.3), we can apply knot tying to get the following:

$$\text{app}[\![z]\!](\text{app}[\![y]\!](x)) \equiv \mu\,\text{f}.\left[\ := x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto \text{app}[\![z]\!](y) \\ \text{cons}(v,vs) \mapsto \bar{\mu} \cdot \overline{\text{cons}}(v, f(vs)) \end{array} \right\} (x')\ \right] (x) \tag{1.4}$$

$$\equiv \text{app}[\![\text{app}[\![z]\!](y)]\!](x) \tag{1.5}$$

$$= append(x, append(y,z))$$

In this proof, we used a number of identities that will be applied in a variety of proofs. The ones used in this proof were:

1. $\mu\text{f}.[y \mapsto t](\bar{\mu} \cdot x) = [y \mapsto t][\![\mu\text{f}.y \mapsto t/f]\!]x$. This just says to unwrap a μ function one time. This was used in Equations (1.1) and (1.3).
2. $[x \mapsto t]t' = t[\![t'/x]\!]$. Applying a function that takes x to t to the term t' is the same as taking the term t and replacing occurrences of x with t' in it. This was used in Equation (1.1).
3. $f\{c_i(x) \mapsto t_i\}v = \{c_i(x) \mapsto f\,t_i\}v$. We may distribute a function into the results of a coproduct mapping. This was used in Equation (1.2).
4. $\{c_i(x_i) \mapsto t_i\}\,\overline{c_k} \cdot v = [x_k \mapsto t_k]v$. Constructor selectors may be applied directly. This was used in Equation (1.4).
5. We have:
$$\frac{t(\bar{\mu} \cdot x') = c[\![[z \mapsto t(z)]/f]\!](x')}{t(x) = \mu\text{f}.c[\![f/f]\!](x)}$$
This identity is called *Knot Tying*. This identity is used in the last step to give us the results in Equations (1.5).

1.2.4 Derivation of *freverse*.

Fast reverse is defined by the following:

$$x :: \mathbb{L}(A) \mapsto \mu\mathrm{frev}.\left[\ := x', y' \mapsto \left\{ \begin{array}{l} \mathrm{nil} \mapsto y' \\ \mathrm{cons}(v, vs) \mapsto \mathrm{frev}(vs, \bar{\mu} \cdot \overline{\mathrm{cons}}(v, y')) \end{array} \right\} (x') \right] (x, \bar{\mu} \cdot \overline{\mathrm{nil}})$$

The derivation of this is as follows:

$$\begin{array}{c|c} Z := \mathbb{L}(A) & Z,\ \mathbb{L}(A) \vdash_{\mathrm{frev}} \mathbb{L}(A) \end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v{::}A \vdash v{::}A}\,(13) \quad \dfrac{}{y'{::}\mathbb{L}(A) \vdash\ y'{::}\mathbb{L}(A)}\,(14)}{v{::}A,\ y'{::}\mathbb{L}(A) \vdash (v, y'){::}(A \times \mathbb{L}(A))}\,(12)}{v{::}A,\ y'{::}\mathbb{L}(A) \vdash \overline{\mathrm{cons}}(v, y'){::}\{\mathrm{nil}{:}(), \mathrm{cons}{:}A \times \mathbb{L}(A)\}}\,(11)}{v{::}A,\ y'{::}\mathbb{L}(A) \vdash \bar{\mu}\overline{\mathrm{cons}}(v, y'){::}\mathbb{L}(A)}\,(10) \quad \dfrac{}{vs{::}Z,\ w{::}\mathbb{L}(A) \vdash \mathrm{frev}(vs, w){::}\mathbb{L}(A)}\,(15)}{vs{::}Z,\ v{::}A,\ y'{::}\mathbb{L}(A) \vdash \mathrm{frev}(vs, \bar{\mu} \cdot \overline{\mathrm{cons}}(v, y'){::}\mathbb{L}(A)}\,(9)$$

$$\dfrac{\dfrac{\dfrac{}{y'{::}\mathbb{L}(A) \vdash y'{::}\mathbb{L}(A)}\,(16)}{(){::}(),\ y'{::}\mathbb{L}(A) \vdash y'{::}\mathbb{L}(A)}\,(6) \quad \dfrac{\dfrac{vs{::}Z,\ v{::}A,\ y'{::}\mathbb{L}(A) \vdash \mathrm{frev}(vs, \bar{\mu} \cdot \overline{\mathrm{cons}}(v, y'){::}\mathbb{L}(A)}{v{::}A,\ vs{::}Z,\ y'{::}\mathbb{L}(A) \vdash \mathrm{frev}(vs, \bar{\mu} \cdot \overline{\mathrm{cons}}(v, y')){::}\mathbb{L}(A)}\,(8)}{(v, vs){::}(A \times Z),\ y'{::}\mathbb{L}(A) \vdash \mathrm{frev}(vs, \bar{\mu} \cdot \overline{\mathrm{cons}}(v, y')){::}\mathbb{L}(A)}\,(7)}{x'{::}\{\mathrm{nil}{:}1, \mathrm{cons}{:}A \times Z\},\ y'{::}\mathbb{L}(A) \vdash \left\{ \begin{array}{l} \mathrm{nil} \mapsto y' \\ \mathrm{cons}(v, vs) \mapsto \mathrm{frev}(vs, \bar{\mu} \cdot \overline{\mathrm{cons}}(v, y')) \end{array} \right\} (x'){::}\mathbb{L}(A)}\,(5)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash (){::}()}\,(4)}{\vdash \overline{\mathrm{nil}}{::}\{\mathrm{nil}{:}(), \mathrm{cons}{:}X \times \mathbb{L}(A)\}}\,(3)}{\vdash \bar{\mu} \cdot \overline{\mathrm{nil}}{::}\mathbb{L}(A)}\,(2) \quad x{::}\mathbb{L}(A)\ \ y{::}\mathbb{L}(A) \vdash \mu\mathrm{frev}.\left[\ := x', y' \mapsto \left\{ \begin{array}{l} \mathrm{nil} \mapsto y' \\ \mathrm{cons}(v, vs) \mapsto \mathrm{frev}(vs, \bar{\mu} \cdot \overline{\mathrm{cons}}(v, y')) \end{array} \right\} (x') \right] (x, y)}{x{::}\mathbb{L}(A) \vdash \mu\mathrm{frev}.\left[\ := x', y' \mapsto \left\{ \begin{array}{l} \mathrm{nil} \mapsto y' \\ \mathrm{cons}(v, vs) \mapsto \mathrm{frev}(vs, \bar{\mu} \cdot \overline{\mathrm{cons}}(v, y')) \end{array} \right\} (x') \right] (x, \bar{\mu} \cdot \overline{\mathrm{nil}})}\,(1)$$

We now explain each of the numbered steps of the proof above.

1. *cut* In this rule, the antecedent has two parts. The $\bar{\mu} \cdot \overline{\mathrm{nil}}$ on the left is matched to the y on the right.
2. *construction* We unroll the type of $\mathbb{L}(A)$
3. *coprod* We select the "nil" item from the coproduct.
4. *tuple* We can always derive an empty tuple from nothing.
5. *cotuple* After we have applied the inductive circular rule, we then apply the cotuple rule to x' giving us two antecedants.
6. *weak* We added the $() :: ()$ to the right hand side.
7. *product pattern* We have a pattern of $(v, vs) :: (A \times Z)$ which is broken down.
8. *exchange* We may switch the order of v and vs.
9. *cut* The term $\bar{\mu}\overline{\mathrm{cons}}(v, y')$ on the left is matched to the w on the right.
10. *construction* Once again, we unroll the list type.
11. *coprod* We know we have the "cons" type and therefore get the respective types of v and y'.
12. *tuple,weak* We have a tuple type on the right hand side and therefore break it down. In addition, we remove the extra item from the respective left hand sides.
13. *atom* Atomic type of A.

14. *atom* Atomic type of $\mathbb{L}(A)$.
15. *induction* The premise of the circular rule is used.
16. *atom* Atomic type of $\mathbb{L}(A)$.

1.2.5 Equality of *reverse* and *freverse*

For this example, we will show the equivalence of a naive reverse and our previously encountered fast reverse.

Naive reverse can be considered in the following way.

$$\text{rev } x = \mu\text{rev}. \left[:= x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto \bar{\mu} \cdot \overline{\text{nil}} \\ \text{cons}(v, vs) \mapsto \text{app}(\text{rev } vs,\ \bar{\mu} \cdot \overline{\text{cons}}(v, \bar{\mu} \cdot \overline{\text{nil}})) \end{array} \right\} (x') \right] x \tag{1.6}$$

As we look for demand in this definition, we recall that app creates demand for it's first arguement. When we have an embedded demand like this, our first thought must be about how to bring the demand out into the open. We have two choices on how to do this. Either we "bang" the argument or we unwind the recursion.

Choosing to unwind the recursion should be our first choice as the transformation is an equality, rather than an isomorphism, as in banging. Unwinding will bring us to

$$\text{rev } x = \mu\_. \left[:= x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto \bar{\mu} \cdot \overline{\text{nil}} \\ \text{cons}(v, vs) \mapsto \text{app}(\text{rev } vs,\ \bar{\mu} \cdot \overline{\text{cons}}(v, \bar{\mu} \cdot \overline{\text{nil}})) \end{array} \right\} (x') \right] x \tag{1.7}$$

where Equation 1.6 = Equation 1.7.

At this point, we are ready to begin our transformation of the inner part of the coproduct which has been exposed by unwinding. As append creates no demand on it's second argument, this is a clue about what to abstract. In fact, we will abstract over the second argument. Our transformation proceeds as follows:

$$\begin{aligned}
&\text{app}[\![z]\!](\text{rev } vs) \\
&\qquad = \text{app}[\![z]\!] \left(\mu_. \left[:= x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto \bar{\mu} \cdot \overline{\text{nil}} \\ \text{cons}(w, ws) \mapsto \text{app}(\text{rev } ws,\ \bar{\mu} \cdot \overline{\text{cons}}(w, \bar{\mu} \cdot \overline{\text{nil}})) \end{array} \right\} (x') \right] vs \right)
\end{aligned} \tag{1.8}$$

$$= \mu\_. \left[:= x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto \text{app}[\![z]\!]\bar{\mu} \cdot \overline{\text{nil}} \\ \text{cons}(w, ws) \mapsto \text{app}[\![z]\!] \left(\text{app}[\![\bar{\mu} \cdot \overline{\text{cons}}(w, \bar{\mu} \cdot \overline{\text{nil}})]\!](\text{rev } ws) \right) \end{array} \right\} (x') \right] vs \tag{1.9}$$

$$= \mu\_. \left[:= x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto \text{app}[\![z]\!]\bar{\mu} \cdot \overline{\text{nil}} \\ \text{cons}(w, ws) \mapsto \text{app}[\![\text{app}[\![z]\!] \left(\bar{\mu} \cdot \overline{\text{cons}}(w, \bar{\mu} \cdot \overline{\text{nil}}) \right)]\!](\text{rev } ws) \end{array} \right\} (x') \right] vs \tag{1.10}$$

$$= \mu\_. \left[:= x' \mapsto \left\{ \begin{array}{l} \text{nil} \mapsto z \\ \text{cons}(w, ws) \mapsto \text{app}[\![\bar{\mu} \cdot \overline{\text{cons}}(w, z)]\!](\text{rev } ws) \end{array} \right\} (x') \right] vs \tag{1.11}$$

where we see we can apply knot-tying as the interior of Equation (1.11) matches the left hand size of Equation (1.8), once we allow for the abstraction.

1.2.6 *accumulate* on infinite lists.

In this section we start to work with our first co-inductive type, Infinte lists. The type definition is:

$$\mathbb{IL}(A) = \nu\, x.(\text{hd} : A,\ \text{tl} : x)$$

Categorically, this corresponds to the following diagram which defines infinite lists:

$$\begin{tikzcd}
A \arrow[d, equal] & C \arrow[l, "h"'] \arrow[r, "t"] \arrow[d, dotted, "f"] & C \arrow[d, dotted, "f"] \\
A & \mathbb{IL}(A) \arrow[l, "\text{hd}"] \arrow[r, "\text{tl}"'] & \mathbb{IL}(A)
\end{tikzcd}$$

where there exists a unique f such that the diagram is commutative. We will examine the function *accumulate* which operates on infinite lists. Informally, it can be though of as follows:

$$a_0, a_1, a_2, \ldots \xrightarrow{\quad acc \quad} [],\ [a_0],\ [a_0, a_1],\ [a_0, a_1, a_2], \ldots$$

and the type of accum is $\mathbb{IL}(A) \to \mathbb{IL}(\mathrm{List}(A))$. The definition is:

$$\mathrm{accum}(L) = \nu\,\mathrm{acc.}\left[\ := L', xs \mapsto \begin{pmatrix} \mathrm{hd} : xs \\ \mathrm{tl} : \mathrm{acc}(\overline{\mathrm{tl}} \cdot \bar{\nu} \cdot L', \mathrm{app}(xs, \bar{\mu} \cdot \overline{\mathrm{cons}}(\overline{\mathrm{hd}} \cdot \bar{\nu} \cdot L', \bar{\mu} \cdot \overline{\mathrm{nil}}))) \end{pmatrix}\ \right](L, \bar{\mu} \cdot \overline{\mathrm{nil}}) \tag{1.12}$$

In Equation (1.12), the variable xs is what is being accumulated to. This is quite inefficient as we walk through the entire list each time to append one element to the end of it.

The demand in Equation (1.12) is created by app and is at the first parameter to it, xs. We note that it is internal to the cocircular definition and would therefore like to resolve it within that definition. The issue here is that the xs comes from the previous invocation of the loob. We therefore need a way to go back 1 step in time. We achieve this by adding an argument to the accumulate function.

$$\mathrm{acc}(L, z) \equiv \mathrm{acc}'(L, z, \lambda x.\mathrm{app}(z, x)) \tag{1.13}$$

$$= \nu\mathrm{acc}'.\left[\ := L', xs \mapsto \begin{pmatrix} \mathrm{hd} : xs & \\ \mathrm{tl} : \mathrm{acc}'(& \overline{\mathrm{tl}} \cdot \bar{\nu} \cdot L', \\ & \mathrm{app}(xs, \bar{\mu} \cdot \overline{\mathrm{cons}}(\overline{\mathrm{hd}} \cdot \bar{\nu} \cdot L', \bar{\mu} \cdot \overline{\mathrm{nil}})), \\ & \lambda x.\mathrm{app}(\mathrm{app}(xs, \bar{\mu} \cdot \overline{\mathrm{cons}}(\overline{\mathrm{hd}} \cdot \bar{\nu} \cdot L', \bar{\mu} \cdot \overline{\mathrm{nil}})), x)) \end{pmatrix}\ \right](L, z) \tag{1.14}$$

$$= \nu\mathrm{acc}'.\left[\ := L', xs \mapsto \begin{pmatrix} \mathrm{hd} : xs & \\ \mathrm{tl} : \mathrm{acc}'(& \overline{\mathrm{tl}} \cdot \bar{\nu} \cdot L', \\ & \mathrm{app}(xs, \bar{\mu} \cdot \overline{\mathrm{cons}}(\overline{\mathrm{hd}} \cdot \bar{\nu} \cdot L', \bar{\mu} \cdot \overline{\mathrm{nil}})), \\ & \lambda x.\mathrm{app}(xs, \bar{\mu} \cdot \overline{\mathrm{cons}}(\overline{\mathrm{hd}} \cdot \bar{\nu} \cdot L', x))) \end{pmatrix}\ \right](L, z) \tag{1.15}$$

$$\tag{1.16}$$

Now, we see that

$$\lambda x.\mathrm{app}(xs, \bar{\mu} \cdot \overline{\mathrm{cons}}(\overline{\mathrm{hd}} \cdot \bar{\nu} \cdot L', x)) \equiv \lambda x.\left(\lambda w.\mathrm{app}(xs, w)@\bar{\mu} \cdot \overline{\mathrm{cons}}(\overline{\mathrm{hd}} \cdot \bar{\nu} \cdot L', x)\right) \tag{1.17}$$

which allows us to knot-tie. This will give us:

$$\mathrm{accum}(L) = \nu\mathrm{acc}'.\left[\ := L', xs, f \mapsto \begin{pmatrix} \mathrm{hd} : xs & \\ \mathrm{tl} : \mathrm{acc}'(& \overline{\mathrm{tl}} \cdot \bar{\nu} \cdot L', \\ & f@\bar{\mu} \cdot \overline{\mathrm{cons}}(\overline{\mathrm{hd}} \cdot \bar{\nu} \cdot L', \bar{\mu} \cdot \overline{\mathrm{nil}})), \\ & \lambda x.(f@\bar{\mu} \cdot \overline{\mathrm{cons}}(\overline{\mathrm{hd}} \cdot \bar{\nu} \cdot L', x))) \end{pmatrix}\ \right](L, z) \tag{1.18}$$

In Equation (1.18), the demand is gone as f is a generalization. Furthermore, we have:

$$\mathrm{accum}(L) = \mathrm{acc}(L, \bar{\mu} \cdot \overline{\mathrm{nil}}) \tag{1.19}$$

$$= \mathrm{acc}'(L, \bar{\mu} \cdot \overline{\mathrm{nil}}, \lambda x.\mathrm{app}(\bar{\mu} \cdot \overline{\mathrm{nil}}, x)) \tag{1.20}$$

$$= \mathrm{acc}'(L, \bar{\mu} \cdot \overline{\mathrm{nil}}, \lambda x.x \tag{1.21}$$

Comparison of run times for accumulate. The following output is from running the programs in Section A.1.2 on page 11. Both statistics are from running the programs on a Dell Inspiron 8000 with a Pentium Pro 800MHz Cpu and 256Mb memory, under Debian Linux with the 2.4.19 kernel. The programs were compiled with default ghc, using no extra optimizations.

The first set of run statistics is generated by the naïve version. Note that producing the first 100000 elements of the infinite list requires approximately 4.5 hours.

```
gilesb@farimir:~/UofC/courses/613/notes\$ time ./accum -b 5000 +RTS -K5000000
4999

real    0m4.446s
user    0m4.370s
sys     0m0.050s
gilesb@farimir:~/UofC/courses/613/notes\$ time ./accum -b 10000 +RTS -K5000000
9999

real    0m32.072s
user    0m31.970s
sys     0m0.010s
gilesb@farimir:~/UofC/courses/613/notes\$ time ./accum -b 100000 +RTS -K5000000
99999

real    272m13.878s
user    267m39.330s
sys     0m10.060s
```

Our next set of statistics is using the transformed version. 100000 items are produced is slightly more than 1 second. That is about 15 000 times as fast. One million are produced in about 30 seconds.

```
gilesb@farimir:~/UofC/courses/613/notes\$ time ./accum -t 5000 +RTS -K5000000
4999

real    0m0.040s
user    0m0.030s
sys     0m0.000s
gilesb@farimir:~/UofC/courses/613/notes\$ time ./accum -t 10000 +RTS -K5000000
9999

real    0m0.074s
user    0m0.070s
sys     0m0.000s
gilesb@farimir:~/UofC/courses/613/notes\$ time ./accum -t 100000 +RTS -K5000000
99999

real    0m1.105s
user    0m1.080s
sys     0m0.010s
```

A Various pieces of Haskell Code

A.1 Accumulate

A.1.1 Driver file for infinite list accumulate.

```
module Main where

import InfList
import System(getArgs, getProgName, exitWith, ExitCode(..))
```

The program takes two arguments on the command line. It will execute the original accumulator when "-b" is passed and the transformed accumulator when "-t" is passed. The second argument is a count of how many iterations to perform.

```
main = do
       arg <- getArgs
       case arg of
  ["-b", num] -> do let nmval=(read num)::Int
           print $ last $ last$ take nmval $ accum [1..]
  ["-t", num] -> do let nmval=(read num)::Int
           print $last $last $ take nmval $ accum' [1..]
  _  -> do  progname <- getProgName
                   putStrLn ("Usage: " ++ progname ++ "{-b|-t} count ")
                   exitWith (ExitFailure 1)
```

A.1.2 Code for infinite list accumulate.

```
module InfList where
```

We first create the naïve version of *acc* which will traverse the list we are accumulating onto each time.

```
acc1::[a]->[a]->[[a]]
acc1 l xs = xs : (acc1 (tail l) (xs ++ [head l]))
```

Then, the transformed version of *acc* which saves the previous invocation via the function *f*.

```
acc2::[a]->[a]->([a]->[a])->[[a]]
acc2 l xs f = xs:(acc2 (tail l) (f [head l]) (\x -> f ((head l):x)))

accum l = acc1 l []
accum' l = acc2 l [] id
```

B Proofs by Banging.

B.1 Proof of the associativity of *append*.

In section 1.2.2 we proved that

$$append(x, y) = \mu\mathrm{app}.\left[\ := x' \mapsto \left\{ \begin{array}{l} \mathrm{nil} \mapsto y \\ \mathrm{cons}(v, vs) \mapsto \bar{\mu} \cdot \overline{\mathrm{cons}}(v, \mathrm{app}[\![y]\!](vs)) \end{array} \right\} (x')\ \right] x{::}\mathbb{L}(A)$$

where we see that considering context, we have

$$\mathrm{app}[\![y]\!](\_) = \left[\ := x' \mapsto \left\{ \begin{array}{l} \mathrm{nil} \mapsto y \\ \mathrm{cons}(v, vs) \mapsto \bar{\mu} \cdot \overline{\mathrm{cons}}(v, \mathrm{app}[\![y]\!](vs)) \end{array} \right\} (x')\ \right] (\_)$$

where y is the context. Then we have

$$\begin{aligned}
&\mathit{append}(\mathit{append}(x,y),z)\\
&\qquad\qquad\qquad = \mathrm{app}[\![z]\!](\mathrm{app}[\![y]\!](x))\\
&= \mathrm{app}[\![z]\!](\left\{\begin{array}{l}\mathrm{nil} \mapsto y\\ \mathrm{cons}(v,vs) \mapsto \bar{\mu}\cdot\overline{\mathrm{cons}}(v,\mathrm{app}[\![y]\!](vs))\end{array}\right\}(x'')) && \text{(B.1)}\\
&= \left\{\begin{array}{l}\mathrm{nil} \mapsto \mathrm{app}[\![z]\!](y)\\ \mathrm{cons}(v,vs) \mapsto \mathrm{app}[\![z]\!](\bar{\mu}\cdot\overline{\mathrm{cons}}(v,\mathrm{app}[\![y]\!](vs)))\end{array}\right\}(x'') && \text{(B.2)}\\
&= \left\{\begin{array}{l}\mathrm{nil} \mapsto \mathrm{app}[\![z]\!](y)\\ \mathrm{cons}(v,vs) \mapsto \left\{\begin{array}{l}\mathrm{nil}\mapsto z\\ \mathrm{cons}(v',vs') \mapsto \bar{\mu}\cdot\overline{\mathrm{cons}}(v',\mathrm{app}[\![z]\!](vs'))\end{array}\right\}\overline{\mathrm{cons}}(v,\mathrm{app}[\![y]\!](vs)))\end{array}\right\}(x'') && \text{(B.3)}\\
&= \left\{\begin{array}{l}\mathrm{nil} \mapsto \mathrm{app}[\![z]\!](y)\\ \mathrm{cons}(v,vs) \mapsto \bar{\mu}\cdot\overline{\mathrm{cons}}(v,\mathrm{app}[\![z]\!](\mathrm{app}[\![y]\!](vs)))\end{array}\right\}(x'') && \text{(B.4)}
\end{aligned}$$

Given that we can see our original pattern ($\mathrm{app}[\![z]\!](\mathrm{app}[\![y]\!](x))$) appears once again in Equation (B.4), we can apply knot tying to get the following:

$$\begin{aligned}
\mathrm{app}[\![z]\!](\mathrm{app}[\![y]\!](x)) &\equiv \mu\,\mathrm{f}.\left[\left\{\begin{array}{l}\mathrm{nil}\mapsto \mathrm{app}[\![z]\!](y)\\ \mathrm{cons}(v,vs)\mapsto\bar{\mu}\cdot\overline{\mathrm{cons}}(v,f(vs))\end{array}\right\}\right](x) && \text{(B.5)}\\
&\equiv \mathrm{app}[\![\mathrm{app}[\![z]\!](y)]\!](x) && \text{(B.6)}\\
&= \mathit{append}(x,\mathit{append}(y,z))
\end{aligned}$$

In this proof, we used a number of identities that will be applied in a variety of proofs. The ones used in this proof were:

1. $\mu\mathrm{f}.[y\mapsto t](\bar{\mu}\cdot x) = [y\mapsto t][\![\mu\mathrm{f}.y\mapsto t/f]\!]x$. This just says to unwrap a μ function one time. This was used in Equations (B.1) and (B.3).

2. $[x\mapsto t]t' = t[\![t'/x]\!]$. Applying a function that takes x to t to the term t' is the same as taking the term t and replacing occurrences of x with t' in it. This was used in Equation (B.1).

3. $f\{c_i(x)\mapsto t_i\}v = \{c_i(x)\mapsto f\,t_i\}v$. We may distribute a function into the results of a coproduct mapping. This was used in Equation (B.2).

4. $\{c_i(x_i)\mapsto t_i\}\,\overline{c_k}\cdot v = [x_k\mapsto t_k]v$. Constructor selectors may be applied directly. This was used in Equation (B.4).

5. We have:
$$\frac{t(\bar{\mu}\cdot x') = c[\![[z\mapsto t(z)]/f]\!](x')}{t(x) = \mu\mathrm{f}.c[\![f/f]\!](x)}$$

   This identity is called *Knot Tying*. This identity is used in the last step to give us the results in Equations (B.5) and (B.6).