Dependently Typed Metaprogramming
(in Agda)

Conor McBride

August 26, 2013
Introduction

If you have never met a metaprogram in a dependently typed programming language like Agda [Norell, 2008], then prepare to be underwhelmed. Once we have types which can depend computationally upon first class values, metaprogams just become ordinary programs manipulating and interpreting data which happen to stand for types and operations.

This course, developed in the summer of 2013, explores methods of metaprogramming in the dependently typed setting. I happen to be using Agda to deliver this material, but the ideas transfer to any setting with enough dependent types. It would certainly be worth trying to repeat these experiments in Idris, or in Coq, or in Haskell, or in your own dependently typed language, or maybe one day in mine.
Chapter 1

Vectors and Normal Functors

It might be easy to mistake this chapter for a bland introduction to dependently typed programming based on the yawning-already example of lists indexed by their length, known to their friends as vectors, but in fact, vectors offer us a way to start analysing data structures into 'shape and contents'. Indeed, the typical motivation for introducing vectors is exactly to allow types to express shape invariants.

1.1 Zipping Lists of Compatible Shape

Let us remind ourselves of the situation with ordinary lists, which we may define in Agda as follows:

```
data List (X : Set) : Set where
  ⟨⟩ : List X
  _ _ : X → List X → List X
infixr 4 _ _
```

The classic operation which morally involves a shape invariant is `zip`, taking two lists, one of `Ss`, the other of `Ts`, and yielding a list of pairs in the product `S × T` formed from elements in corresponding positions. The trouble, of course, is ensuring that positions correspond.

```
zip : {S T : Set} → List S → List T → List (S × T)
zip ⟨⟩ ⟨⟩ = ⟨⟩
zip (s, ss) (t, ts) = (s, t), zip ss ts
zip _ _ = ⟨⟩ -- a dummy value, for cases we should not reach
```

Overloading Constructors  Note that I have used ‘,’ both for tuple pairing and as list ‘cons’. Agda permits the overloading of constructors, using type information to disambiguate them. Of course, just because overloading is permitted, that does not make it compulsory, so you may deduce that I have overloaded deliberately. As data structures in the memory of a computer, I think of pairing and consing as the same, and I do not expect data to tell me what they mean. I see types as an external rationalisation imposed upon the raw stuff of computation, to help us check that it makes sense (for multiple possible notions of sense) and indeed to infer details (in accordance with notions of sense). Those of you who have grown used to thinking of type annotations as glorified comments will need to retrain your minds to pay attention to them.
Our \texttt{zip} function imposes a ‘garbage in? garbage out!’ deal, but logically, we might want to ensure the obverse: if we supply meaningful input, we want to be sure of meaningful output. But what is meaningful input? Lists the same length! Locally, we have a \textit{relative} notion of meaningfulness. What is meaningful output? We could say that if the inputs were the same length, we expect output of that length. How shall we express this property? We could externalise it in some suitable program logic, first explaining what ‘length’ is.

\begin{verbatim}
data \texttt{N} : \texttt{Set} where
  \texttt{zero} : \texttt{N}
  \texttt{suc} : \texttt{N} \to \texttt{N}
{--# BUILTIN NATURAL \texttt{Nat} #-}
{--# BUILTIN \texttt{ZERO} \texttt{zero} #-}
{--# BUILTIN \texttt{SUC} \texttt{suc} #-}

|length : \texttt{\{X : Set\} \to List X \to \texttt{N}}
|length \texttt{\{\}} = \texttt{zero}
|length \texttt{\{(x, xs) = suc (length xs)\}}
\end{verbatim}

Informally,\footnote{by which I mean, not to a computer} we might state and prove something like

\[\forall ss, ts. \text{length } ss = \text{length } ts \Rightarrow \text{length } (\text{zip } ss \ ts) = \text{length } ss\]

by structural induction \cite{Burstall:1969} on \texttt{ss}, say. Of course, we could just as well have concluded that \text{length } (\text{zip } ss \ ts) = \text{length } ts, and if we carry on \text{zipping}, we shall accumulate a multitude of expressions known to denote the same number.

Matters get worse if we try to work with matrices as lists of lists (a matrix is a column of rows, say). How do we express rectangularity? Can we define a function to compute the dimensions of a matrix? Do we want to? What happens in degenerate cases? Given \(m, n\), we might at least say that the outer list has length \(m\) and that all the inner lists have length \(n\). Talking about matrices gets easier if we imagine that the dimensions are \textit{prescribed}—to be checked, not measured.

\section{Vectors}

Dependent types allow us to \textit{internalize} length invariants in lists, yielding \textit{vectors}. The index describes the shape of the list, thus offers no real choice of constructors.

\begin{verbatim}
data \texttt{Vec} \texttt{(X : Set)} : \texttt{N} \to \texttt{Set} where
  \texttt{\{\} : Vec X \texttt{zero}}
  \texttt{- - : \{n : \texttt{N}\} \to X \to \texttt{Vec X n} \to \texttt{Vec X (suc n)}}
\end{verbatim}

\textbf{Parameters and indices.} In the above definition, the element type is abstracted uniformly as \(X\) across the whole thing. The definition could be instantiated to any particular set \(X\) and still make sense, so we say that \(X\) is a \textit{parameter} of the definition. Meanwhile, \texttt{Vec}'s second argument varies in each of the three places it is instantiated, so that we are really making a mutually inductive definition of the vectors at every possible length, so we say that the length is an \textit{index}. In an Agda \texttt{data} declaration head, arguments left of \(:\) (\texttt{X} here) scope over all constructor declarations and must be used uniformly in constructor return types, so it is sensible to put parameters left of \(:\). However, as we shall see, such arguments may be
freely instantiated in recursive positions, so we should not presume that they are necessarily parameters.

Let us now develop \( \text{zip} \) for vectors, stating the length invariant in the type.

\[
\text{zip} : \forall \{ n S T \} \to \text{Vec} \ S \ n \to \text{Vec} \ T \ n \to \text{Vec} \ (S \times T) \ n
\]

The length argument and the two element types are marked implicit by default, as indicated by the \{ .. \} after the forall. We write a left-hand-side naming the explicit inputs, which we declare equal to an unknown \(?\). Loading the file with \([C - c C - l]\), we find that Agda checks the unfinished program, turning the \(?\) into labelled braces,

\[
\text{zip} : \forall \{ n S T \} \to \text{Vec} \ S \ n \to \text{Vec} \ T \ n \to \text{Vec} \ (S \times T) \ n
\]

and tells us, in the information window,

\[
?0 : \text{Vec} \ (.S \times .T) \ .n
\]

that the type of the 'hole' corresponds to the return type we wrote. The dots before \( S, T, \) and \( n \) indicate that these variables exist behind the scenes, but have not been brought into scope by anything in the program text: Agda can refer to them, but we cannot.

If we click between the braces to select that hole, and issue keystroke \([C - c C - c]\), we will gain more information about the goal:

\[
\text{Goal} : \text{Vec} \ (\Sigma .S (\lambda _.T)) \ .n
\]

\[
\begin{align*}
\text{ts} & : \text{Vec} \ .T \ .n \\
\text{ss} & : \text{Vec} \ .S \ .n \\
.T & : \text{Set} \\
.S & : \text{Set} \\
.n & : \text{N}
\end{align*}
\]

revealing the definition of \( \times \) used in the goal, about which more shortly, but also telling us about the types and visibility of variables in the context.

Our next move is to split one of the inputs into cases. We can see from the type information \( \text{ss} : \text{Vec} \ .S \ .n \) that we do not know the length of \( \text{ss} \), so it might be given by either constructor. To see if Agda agrees, we type \( \text{ss} \) in the hole and issue the ‘case-split’ command \([C - c C - c]\).

\[
\text{zip} : \forall \{ n S T \} \to \text{Vec} \ S \ n \to \text{Vec} \ T \ n \to \text{Vec} \ (S \times T) \ n
\]

Agda responds by editing our source code, replacing the single line of definition by two more specific cases.

\[
\begin{align*}
\text{zip} : \forall \{ n S T \} \to \text{Vec} \ S \ n \to \text{Vec} \ T \ n \to \text{Vec} \ (S \times T) \ n
\end{align*}
\]

Moreover, we gain the refined type information

\[
\begin{align*}
?0 : \text{Vec} \ (.S \times .T) \ 0 \\
?1 : \text{Vec} \ (.S \times .T) \ (\text{suc} \ .n)
\end{align*}
\]
which goes to show that the type system is now tracking what information is learned about the problem by inspecting $ss$. This capacity for learning by testing is the paradigmatic characteristic of dependently typed programming.

Now, when we split $ts$ in the $\emptyset$ case, we get

$$zip : \forall \{ n S T \} \to Vec S n \to Vec T n \to Vec (S \times T) n$$

$$zip \{ \} \{ \} = \{ \}$$

$$zip (x, ss) ts = \{ \}$$

and in the $\text{suc}$ case,

$$zip : \forall \{ n S T \} \to Vec S n \to Vec T n \to Vec (S \times T) n$$

$$zip \{ \} \{ \} = \emptyset$$

$$zip (x, ss) (x1, ts) = (x, x1) ; zip ss ts$$

It’s not even as clever as Epigram.

As the more specific type now determines the shape. Sadly, Agda is not very clever about choosing names, but let us persevere. We have now made sufficient analysis of the input to determine the output, and shape-indexing has helpfully ruled out shape mismatch. It is now so obvious what must be output that Agda can figure it out for itself. If we issue the keystroke $[C-a]$ in each hole, a type-directed program search robot called ‘Agsy’ tries to find an expression which will fit in the hole, assembling it from the available information without further case analysis. We obtain a complete program.

$$zip : \forall \{ n S T \} \to Vec S n \to Vec T n \to Vec (S \times T) n$$

$$zip \{ \} \{ \} = \emptyset$$

$$zip (x, ss) (x1, ts) = (x, x1) ; zip ss ts$$

I tend to $\alpha$-convert and realign such programs manually, yielding

$$zip : \forall \{ n S T \} \to Vec S n \to Vec T n \to Vec (S \times T) n$$

$$zip \{ \} \{ \} = \emptyset$$

$$zip (s, ss) (t, ts) = (s, t) ; zip ss ts$$

What just happened? We made $Vec$, a version of $List$, indexed by $N$, and suddenly became able to work with ‘elements in corresponding positions’ with some degree of precision. That worked because $N$ describes the shape of lists: indeed $N \cong List \text{ One}$, instantiating the $List$ element type to the type $\text{One}$ with the single element $\{ \}$, so that the only information present is the shape. Once we fix the shape, we acquire a fixed notion of position.

**Exercise 1.1 (vec)** Complete the implementation of

$$vec : \forall \{ n X \} \to X \to Vec X n$$

$$vec \{ n \} x = ?$$

Why is there no specification? using only control codes and arrow keys. (Note the brace notation, making the implicit $n$ explicit. It is not unusual for arguments to be inferrable at usage sites from type information, but none the less computationally relevant.)

**Exercise 1.2 (vector application)** Complete the implementation of

$$vapp : \forall \{ n S T \} \to Vec (S \to T) n \to Vec S n \to Vec T n$$

$$vapp fs ss = ?$$

using only control codes and arrow keys. The function should apply the functions from its first input vector to the arguments in corresponding positions from its second input vector, yielding values in corresponding positions in the output.
1.3. APPLICATIVE AND TRAVERSABLE STRUCTURE

Exercise 1.3 (vmap) Using vec and vapp, define the functorial ‘map’ operator for vectors, applying the given function to each element.

\[
\text{vmap} : \forall \{ n \; S \; T \} \to (S \to T) \to \text{Vec} S n \to \text{Vec} T n
\]

\[
\text{vmap} f \; ss = ?
\]

Note that you can make Agsy notice a defined function by writing its name as a hint in the relevant hole before you \([ C - c \; C - a]\).

Exercise 1.4 (zip) Using vec and vapp, give an alternative definition of zip.

\[
\text{zip} : \forall \{ n \; S \; T \} \to \text{Vec} S n \to \text{Vec} T n \to \text{Vec} (S \times T) n
\]

\[
\text{zip} \; ss \; ts = ?
\]

Exercise 1.5 (Finite sets and projection from vectors) We may define a type of finite sets, suitable for indexing into vectors, as follows:

\[
\text{data Fin} : \mathbb{N} \to \text{Set} \text{ where}
\]

\[
\text{zero} : \{ n : \mathbb{N} \} \to \text{Fin} (\text{suc} \; n)
\]

\[
\text{suc} : \{ n : \mathbb{N} \} \to \text{Fin} n \to \text{Fin} (\text{suc} \; n)
\]

Implement projection:

\[
\text{proj} : \forall \{ n \; X \} \to \text{Vec} X n \to \text{Fin} n \to X
\]

\[
\text{proj} \; xs \; i = ?
\]

Implement, tabulation, the inverse of projection.

\[
\text{tabulate} : \forall \{ n \; X \} \to (\text{Fin} n \to X) \to \text{Vec} X n
\]

\[
\text{tabulate} \; \{ n \} \; f = ?
\]

Hint: think higher order.

1.3 | Applicative and Traversable Structure

The vec and vapp operations from the previous section equip vectors with the structure of an applicative functor. Before we get to Applicative, let us first say what is an EndoFunctor:

\[
\text{record EndoFunctor} \; (F : \text{Set} \to \text{Set}) : \text{Set} \text{ where}
\]

\[
\text{field}
\]

\[
\text{map} : \forall \{ S \; T \} \to (S \to T) \to F S \to F T
\]

\[
\text{open EndoFunctor} \; \{ \{ \ldots \} \} \; \text{public}
\]

The above record declaration creates new types EndoFunctor \( F \) and a new module, EndoFunctor, containing a function, EndoFunctor.map, which projects the map field from a record. The open declaration brings map into top level scope, and the \( \{ \{ \ldots \} \} \) syntax indicates that map’s record argument is an instance argument. Instance arguments are found by searching the context for something of the required type, succeeding if exactly one candidate is found.

Of course, we should ensure that such structures should obey the functor laws, with map preserving identity and composition. Dependent types allow us to state and prove these laws, as we shall see shortly.

First, however, let us refine EndoFunctor to Applicative.

For now, I shall just work in Set, but we should remember to break out and live, categorically, later. Why Set_1?
The `Applicative` structure decomposes \( F \)'s `map` as the ability to make 'constant' \( F \) -structures and closure under application.

Given that instance arguments are collected from the context, let us seed the context with suitable candidates for `Vec`:

\[
\text{applicativeVec} : \text{forall} \{ n \} \rightarrow \text{Applicative} \quad \lambda X \rightarrow \text{Vec} X n
\]

Indeed, the definition of `endoFunctorVec` already makes use of way its `EndoFunctor` searches the context and finds `applicativeVec`.

There are lots of applicative functors about the place. Here's another famous one:

\[
\text{applicativeFun} : \text{forall} \{ S \} \rightarrow \text{Applicative} \quad \lambda X \rightarrow S \rightarrow X
\]

Monadic structure induces applicative structure:

\[
\text{record Monad} \quad \text{forall} \{ F \} \rightarrow \text{Set} \rightarrow \text{Set} : \text{Set} 1 \quad \text{where}
\begin{align*}
\text{field} & \quad \text{return} : \text{forall} \{ X \} \rightarrow X \rightarrow F X \\
& \quad \text{(monadApplicative : Applicative} \ F
\end{align*}
\]

\[
\text{record} \quad \text{forall} \{ F G \} \rightarrow \text{Applicative} \quad \lambda F \rightarrow \text{Applicative} \quad \lambda G \rightarrow \text{Applicative} \quad (F \circ G)
\]

**Exercise 1.6 (Vec monad)** Construct a `Monad` satisfying the `Monad` laws

\[
\text{monadVec} = ?
\]

such that `monadApplicative` agrees extensionally with `applicativeVec`.

**Exercise 1.7 (Applicative identity and composition)** Show by construction that the identity endofunctor is `Applicative`, and that the composition of `Applicatives` is `Applicative`.

\[
\text{record} \quad \text{forall} \{ F G \} \rightarrow \text{Applicative} \quad \lambda F \rightarrow \text{Applicative} \quad \lambda G \rightarrow \text{Applicative} \quad (F \circ G)
\]

\[
\text{applicativeId} = ?
\]

\[
\text{applicativeComp} = ?
\]
1.3. APPLICATIVE AND TRAVERSABLE STRUCTURE

Exercise 1.8 (Monoid makes Applicative) Let us give the signature for a monoid thus:

record Monoid (X : Set) : Set where
  infixr 4 _•_
  field
  ε : X
  _•_ : X → X → X
  monoidApplicative : Applicative λ _ → X
  monoidApplicative = ?
open Monoid {{ ... }} public -- it's not obvious that we'll avoid ambiguity

Complete the Applicative so that it behaves like the Monoid.

Exercise 1.9 (Applicative product) Show by construction that the pointwise product of Applicatives is Applicative.

record Traversable (F : Set → Set) : Set₁ where
  field
  traverse : forall {G S T} {{AG : Applicative G}} →
  (S → G T) → F S → G (F T)
  traversableEndoFunctor : EndoFunctor F
  traversableEndoFunctor = record {map = traverse}
open Traversable {{ ... }} public

The explicit aG became needed after I introduced the applicativeId exercise, making resolution ambiguous.

traversableVec : {n : N} → Traversable λ X → Vec X n
traversableVec = record {traverse = vtr} where
  vtr : forall {n G S T} {{AG : Applicative G}} →
  (S → G T) → Vec S n → G (Vec T n)
  vtr {{aG}} f ⟨⟩ = pure {{aG}} ⟨⟩
  vtr {{aG}} f (s,ss) = pure {{aG}} _•_ ⊛ f s ⊛ vtr f ss

Exercise 1.10 (transpose) Implement matrix transposition in one line.

transpose : forall {m n X} → Vec (Vec X n) m → Vec (Vec X m) n
transpose = ?

We may define the crush operation, accumulating values in a monoid stored in a Traversable structure:

crush : forall {F X Y} {{TF : Traversable F}} {{M : Monoid Y}} →
  (X → Y) → F X → Y

crush {{M = M}} = traverse {T = One} {{AG = monoidApplicative {{M}}} → T arbitrary arguments.

Amusingly, we must tell Agda which T is intended when viewing X → Y as X → (λ _ → Y) T. In a Hindley-Milner language, such uninferred things are unimportant because they are in any case parametric. In the dependently typed setting, we cannot rely on quantification being parametric (although in the absence of typecase, quantification over types cannot help so being).

Exercise 1.11 (Traversable functors) Show that Traversable is closed under identity and composition. What other structure does it preserve?
1.4 \( \Sigma \)-types and Other Equipment

Before we go any further, let us establish that the type \( \Sigma (S : \text{Set}) (T : S \to \text{Set}) \) has elements \((s : S), (t : T s)\), so that the type of the second component depends on the value of the first. From \( p : \Sigma S T \), we may project \( \text{fst} p : S \) and \( \text{snd} p : T (\text{fst} p) \), but I also define \( \upsilon \) to be a low precedence uncurrying operator, so that \( \upsilon \lambda s t \to ... \) gives access to the components.

On the one hand, we may take \( S \times T = \Sigma S \lambda _{-} \to T \) and generalize the binary product to its dependent version. On the other hand, we can see \( \Sigma S T \) as generalising the binary sum to an \( S \)-ary sum, which is why the type is called \( \Sigma \) in the first place.

We can recover the binary sum (coproduct) by defining a two element type:

\[
\text{data Two : Set where } \uparrow \downarrow : \text{Two}
\]

It is useful to define a conditional operator, indulging my penchant for giving infix operators three arguments,

\[
\langle ? \rangle : \text{forall } \{ l \} \{ P : \text{Two } \to \text{Set} \} \to P \uparrow \to P \downarrow \to (b : \text{Two}) \to P b
\]

\[
(t \langle ? \rangle f) \uparrow = t \quad (t \langle ? \rangle f) \downarrow = f
\]

for we may then define:

\[
\Sigma : \text{Set } \to \text{Set } \to \text{Set}
\]

\[
S + T = \Sigma \text{Two } (S \langle ? \rangle T)
\]

Note that \( \langle ? \rangle \) has been defined to work at all levels of the predicative hierarchy, so that we can use it to choose between \( \text{Sets} \), as well as between ordinary values. \( \Sigma \) thus models both choice and pairing in data structures. That is, \( \Sigma \) generalizes binary product to the dependent case, and binary sum to arbitrary arity. I advise calling a \( \Sigma \)-type neither a ‘dependent sum’ nor a ‘dependent product’ (for a dependent function type is a something-adic product), but rather a ‘dependent pair type’.

1.5 Arithmetic

I don’t know about you, but I find I do a lot more arithmetic with types than I do with numbers, which is why I have used \( \times \) and \( + \) for \( \text{Sets} \). However, we shall soon need a little arithmetic for the sizes of things.

**Exercise 1.12 (unary arithmetic)** Implement addition and multiplication for numbers.

\[
\ nabla N : N \to N \to N
\]

\[
x +_N y = ?
\]

\[
\ nabla N : N \to N \to N
\]

\[
x \times_N y = ?
\]

1.6 Normal Functors

A normal functor is given, up to isomorphism, by a set of shapes and a function which assigns to each shape a size. It is interpreted as the dependent pair of a shape, \( s \), and a vector of elements whose length is the size of \( s \).
1.6. NORMAL FUNCTORS

record Normal : Set₁ where
  constructor _/
  field
    Shape : Set
    size : Shape → ℕ
  N₁ : Set → Set
  N₁ X = ∑ Shape λ s → Vec X (size s)
open Normal public
infixr 0 _/

Let us have two examples. Vectors are the normal functors with a unique shape.
Lists are the normal functors whose shape is their size.

\[ \text{Vec}_N : \mathbb{N} \to \text{Normal} \]
\[ \text{Vec}_N n = \text{One} / \text{pure} n \]
\[ \text{List}_N : \text{Normal} \]
\[ \text{List}_N = \mathbb{N} / \text{id} \]

But let us not get ahead of ourselves. We can build a kit for normal functors corresponding to the type constructors that we often define, then build up composite structures. For example, let us have that constants and the identity are Normal.

\[ \text{K}_N : \text{Set} \to \text{Normal} \]
\[ \text{K}_N A = A / \lambda \_ \to 0 \]
\[ \text{IK}_N : \text{Normal} \]
\[ \text{IK}_N = \text{Vec}_N 1 \]

Let us construct sums and products of normal functors.

\[ _+N : \text{Normal} \to \text{Normal} \to \text{Normal} \]
\[ (\text{Sh} F / szF) +_N (\text{Sh} G / szG) = (\text{Sh} F + \text{Sh} G) / \lor szF \langle?\rangle szG \]

\[ _\times N : \text{Normal} \to \text{Normal} \to \text{Normal} \]
\[ (\text{Sh} F / szF) \times_N (\text{Sh} G / szG) = (\text{Sh} F \times \text{Sh} G) / \lor \lambda f g \to szF f +_N szG g \]

Of course, it is one thing to construct these binary operators on Normal, but quite another to show they are worthy of their names.

\[ \text{nlnj} \ : \ \forall \{ X \} \ (F G : \text{Normal}) \to \llbracket F \rrbracket_N X + \llbracket G \rrbracket_N X \to \llbracket F +_N G \rrbracket_N X \]
\[ \text{nlnj} \ F G (\text{tt}, \text{Sh} F, xs) = (\text{tt}, \text{Sh} F), xs \]
\[ \text{nlnj} \ F G (\text{ff}, \text{Sh} G, xs) = (\text{ff}, \text{Sh} G), xs \]

Now, we could implement the other direction of the isomorphism, but an alternative is to define the inverse image.

\[ \text{data} \ \_^{-1} \ : \ \{ S T : \text{Set} \} \ (f : S \to T) : T \to \text{Set} \ where \]
\[ \text{from} : (s : S) \to f^{-1} f s \]

Let us now show that nlnj is surjective.

\[ \text{nCase} \ : \ \forall \{ X \} \ (F G : \text{Normal})(s : \llbracket F +_N G \rrbracket_N X) \to \text{nlnj} \ F G^{-1} s \]
\[ \text{nCase} \ F G ((\text{tt}, \text{Sh} F), xs) = \text{from} (\text{tt}, \text{Sh} F, xs) \]
\[ \text{nCase} \ F G ((\text{ff}, \text{Sh} G), xs) = \text{from} (\text{ff}, \text{Sh} G, xs) \]

That is, we have written more or less the other direction of the iso, but we have acquired some of the correctness proof for the cost of asking. We shall check that nlnj is injective shortly, once we have suitable equipment to say so.
The inverse of \( \text{nlnj} \) can be computed by \( \text{nCase} \) thus:

\[
\text{nOut} : \forall \{ X \} \ (F \ G : \text{Normal}) \to [F +_{\text{N}} G]_\text{N} X \to [F]_\text{N} X + [G]_\text{N} X \\
\text{nOut} F G x s' \text{ with } \text{nCase} F G x s' \\
\text{nOut} F G . (\text{nlnj} F G x s) \mid \text{from } x s = x s
\]

The \text{with} notation allows us to compute some useful information and add it to the collection of things available for inspection in pattern matching. By matching the result of \( \text{nCase} F G x s' \) as \( \text{from } x s \), we discover that \text{ipso facto}, \( x s' \) is \( \text{nlnj} x s \).

It is in the nature of dependent types that inspecting one piece of data can refine our knowledge of the whole programming problem, hence McKinna and I designed \text{with} as a syntax for bringing new information to the problem. The usual Burstallian \text{‘case expression’} focuses on one scrutinee and shows us its refinements, but hides from us the refinement of the rest of the problem: in simply typed programming there is no such refinement, but here there is. Agda prefixes with a dot those parts of patterns, not necessarily linear constructor forms, which need not be checked dynamically because the corresponding value must be as indicated in any well typed usage.

**Exercise 1.13 (normal pairing)** Implement the constructor for normal functor pairs. It may help to define vector concatenation.

\[
\text{++} : \forall \{ m \ n \ X \} \ \rightarrow \text{Vec} X m \rightarrow \text{Vec} X n \rightarrow \text{Vec} X (m +_{\text{N}} n) \\
xs ++ ys = ? \\
\text{nPair} : \forall \{ X \} \ (F \ G : \text{Normal}) \rightarrow [F]_\text{N} X \times [G]_\text{N} X \rightarrow [F \times_{\text{N}} G]_\text{N} X \\
\text{nPair} F G fxgx = ?
\]

Show that your constructor is surjective.

**Exercise 1.14 (ListN monoid)** While you are in this general area, construct (from readily available components) the usual monoid structure for our normal presentation of lists.

\[
\text{listNMonoid} : \{ X : \text{Set} \} \rightarrow \text{Monoid} ([\text{ListN}]_\text{N} X) \\
\text{listNMonoid} = ?
\]

We have already seen that the identity functor \( \text{VecN} \ I \) is \text{Normal}, but can we define composition?

\[
\circ_{\text{N}} : \text{Normal} \rightarrow \text{Normal} \rightarrow \text{Normal} \\
F \circ_{\text{N}} (\text{ShG} / \text{szG}) = ? / ?
\]

To choose the shape for the composite, we need to know the outer shape, and then the inner shape at each element position. That is:

\[
\circ_{\text{N}} : \text{Normal} \rightarrow \text{Normal} \rightarrow \text{Normal} \\
F \circ_{\text{N}} (\text{ShG} / \text{szG}) = [F]_\text{N} \text{ShG} / \{!!\}
\]

Now, the composite must have a place for each element of each inner structure, so the size of the whole is the sum of the sizes of its parts. That is to say, we must traverse the shape, summing the sizes of each inner shape therein. Indeed, we can use \text{traverse}, given that \( \text{N} \) is a monoid for \( +_{\text{N}} \) and that \text{Normal} functors are traversable because vectors are.

\[
\text{sumMonoid} : \text{Monoid} \text{N} \\
\text{sumMonoid} = \text{record} \{ \varepsilon = 0; \text{•} = \circ_{\text{N}} \}
\]
1.6. NORMAL FUNCTORS

\[ \text{normalTraversable} : (F : \text{Normal}) \to \text{Traversable} \, F \, \|_N \]

\[ \text{normalTraversable} \, F = \text{record} \{
\begin{array}{l}
\text{traverse} = \lambda \{ \{ aG \} \} f \to V \langle \lambda s \, xs \to \text{pure} \{ \{ aG \} \} \langle s, s \rangle \odot \text{traverse} \, f \, xs \rangle \;
\end{array} \]

Armed with this structure, we can implement the composite size operator as a crush.

\[ \text{\( \subseteq \)} : \text{Normal} \to \text{Normal} \to \text{Normal} \]

\[ F \odot (\text{ShG} / \text{szG}) = \| F \|_N \text{ShG} / \text{crush} \{ \{ \text{normalTraversable} \, F \} \} \, \text{szG} \]

The fact that we needed only the Traversable interface to \( F \) is a bit of a clue to a connection between Traversable and Normal functors. Traversable structures have a notion of size induced by the Monoid structure for \( \mathbb{N} \):

\[ \text{sizeT} : \forall \{ F \} \{ \{ TF : \text{Traversable} \, F \} \} \{ X \} \to F \, X \to \mathbb{N} \]

\[ \text{sizeT} = \text{crush} \, (\lambda \_ \to 1) \]

Hence, every Traversable functor has a Normal counterpart

\[ \text{normalT} : \forall \{ F \} \{ \{ TF : \text{Traversable} \, F \} \} \to \text{Normal} \]

\[ \text{normalT} \, F = F \, \text{One} / \text{sizeT} \]

where the shape is an \( F \) with placeholder elements and the size is the number of such places.

Can we put a Traversable structure into its Normal representation? We can certainly extract the shape:

\[ \text{shapeT} : \forall \{ F \} \{ \{ TF : \text{Traversable} \, F \} \} \{ X \} \to F \, X \to F \, \text{One} \]

\[ \text{shapeT} = \text{traverse} \, (\lambda \_ \to \langle \rangle) \]

We can also define the list of elements, which should have the same length as the size

\[ \text{one} : \forall \{ X \} \to X \to \| \text{ListN} \|_N \, X \]

\[ \text{one} \, x = 1, \langle x, \langle \rangle \rangle \]

\[ \text{contentsT} : \forall \{ F \} \{ \{ TF : \text{Traversable} \, F \} \} \{ X \} \to F \, X \to \| \text{ListN} \|_N \, X \]

\[ \text{contentsT} = \text{crush} \, \text{one} \]

and then try

\[ \text{toNormal} : \forall \{ F \} \{ \{ TF : \text{Traversable} \, F \} \} \{ X \} \to F \, X \to \| \text{normalT} \, F \|_N \, X \]

\[ \text{toNormal} \, fx = \text{BAD} \, (\text{shapeT} \, fx, \text{snd} \, (\text{contentsT} \, fx)) \]

but it fails to typecheck because the size of the shape of \( fx \) is not obviously the length of the contents of \( fx \). The trouble is that Traversable \( F \) is underspecified. In due course, we shall discover that it means just that \( F \) is naturally isomorphic to \( \| \text{normalT} \, F \|_N \). To see this, however, we shall need the capacity to reason equationally, which must wait until the next section.

**Exercise 1.15 (normal morphisms)** A normal morphism is given as follows

\[ \text{nMorph} : \forall \{ F \, G \} \to F \to G \to \forall \{ X \} \to \| F \|_N \, X \to \| G \|_N \, X \]

\[ \text{nMorph} \, f \, (s, xs) \text{ with } f \, s \]

\[ ... \]

| \( s' \), \( is = s' \), map (proj \, xs) is |
Show how to compute the normal morphism representing a given natural transformation.

\[
morphN : \forall \{ F \ G \} \to (\forall \{ X \} \to [F]_N X \to [G]_N X) \to F \to_N G
\]

Exercise 1.16 (Hancock’s tensor) Let

\[
\otimes : \text{Normal} \to \text{Normal} \to \text{Normal}
\]

\[(\text{ShF} / \text{szF}) \otimes (\text{ShG} / \text{szG}) = (\text{ShF} \times \text{ShG}) / \forall \lambda f g \to \text{szF} f \times_{\text{N}} \text{szG} g
\]

Construct normal morphisms:

\[
\text{swap} : (F G : \text{Normal}) \to (F \otimes G) \to_N (G \otimes F)
\]
\[
\text{swap} F G x = ?
\]

\[
\text{drop} : (F G : \text{Normal}) \to (F \otimes G) \to_N (F \text{N}_\otimes G)
\]
\[
\text{drop} F G x = ?
\]

Hint: for \text{swap}, you may find you need to build some operations manipulating matrices.
Hint: for \text{drop}, it may help to prove a theorem about multiplication (see next section for details of equality), but you can get away without so doing.

1.7 Proving Equations

The best way to start a fight in a room full of type theorists is to bring up the topic of equality. There’s a huge design space, not least because we often have two notions of equality to work with, so we need to design both and their interaction.

On the one hand, we have judgmental equality. Suppose you have \( s : S \) and you want to put \( s \) where a value of type \( T \) is expected. Can you? You can if \( S \equiv T \).

Different systems specify \( \equiv \) differently. Before dependent types arrived, syntactic equality (perhaps up to \( \alpha \)-conversion) was often enough.

In dependently typed languages, it is quite convenient if \( \text{Vec} X (2 + 2) \) is the same type as \( \text{Vec} X 4 \), so we often consider types up to the \( \alpha \beta \)-conversion of the \( \lambda \)-calculus further extended by the defining equations of total functions. If we’ve been careful enough to keep the open-terms reduction of the language strongly normalizing, then \( \equiv \) is decidable, by normalize-and-compare in theory and by more carefully tuned heuristics in practice.

Agda takes things a little further by supporting \( \eta \)-conversion at some ‘negative’ types—specifically, function types and record types—where a type-directed and terminating \( \eta \)-expansion makes sense. Note that a syntax-directed ‘tit-for-tat’ approach, e.g. testing \( f \equiv \lambda x \to t \) by testing \( x \vdash f x \equiv t \) or \( p \equiv (s,t) \) by \( \text{fst} \ p \equiv s \) and \( \text{snd} \ p \equiv t \), works fine because two non-canonical functions and pairs are equal if and only if their expansions are. But if you want the \( \eta \)-rule for \text{One}, you need a cue to notice that \( u \equiv v \) when both inhabit \text{One} and neither is \( \langle \rangle \).

It is always tempting (hence, dangerous) to try to extract more work from the computer by making judgmental equality admit more equations which we consider morally true, but it is clear that any \text{decidable} judgmental equality will always disappoint—extensional equality of functions is undecidable, for example. Consequently, the equational theory of open terms (conceived as functions from valuations of their variables) will always be to some extent beyond the ken of the computer.

The remedy for our inevitable disappointment with judgmental equality is to define a notion of \text{evidence} for equality. It is standard practice to establish decidable certificate-checking for undecidable problems, and we have a standard mechanism for so doing—checking types. Let us have types \( s \simeq t \) inhabited by proofs...
1.7. PROVING EQUATIONS

that \( s \) and \( t \) are equal. We should ensure that \( t \simeq t \) for all \( t \), and that for all \( P \), \( s \simeq t \rightarrow P \rightarrow P \), in accordance with the philosophy of Leibniz. On this much, we may agree. But after that, the fight starts.

The above story is largely by way of an apology for the following declaration.

\[
\text{data } \simeq \{ l \} \{ X : \text{Set} \} \{ x : X \} : X \rightarrow \text{Set} \text{ where}
\]

- \( \text{refl} : x \simeq x \)
- \( \text{infix } l \simeq \)

We may certainly implement Leibniz’s rule.

\[
\text{subst} : \forall l \{ X : \text{Set} \} \{ s t : X \} \rightarrow s \simeq t \rightarrow (P : X \rightarrow \text{Set} l) \rightarrow P s \rightarrow P t
\]

The only canonical proof of \( s \simeq t \) is \( \text{refl} \), available only if \( s \equiv t \), so we have declared that the equality predicate for closed terms is whatever judgmental equality we happen to have chosen. We have sealed our disappointment in, but we have gained the ability to prove useful equations on open terms. Moreover, the restriction to the judgmental equality is fundamental to the computational behaviour of our \( \text{subst} \) implementation: we take \( p : P s \) and we return it unaltered as \( p : P t \), so we need to ensure that \( P s \equiv P t \), and hence that \( s \equiv t \). If we want to make \( \simeq \) larger than \( \equiv \), we need a more invasive approach to transporting data between provably equal types. For now, let us acknowledge the problem and make do.

We may register equality with Agda, via the following pragmas,

```
{-# BUILTIN EQUALITY _==_ #-}
{-# BUILTIN REFL refl #-}
```

and thus gain access to Agda’s support for equational reasoning.

Now that we have some sort of equality, we can specify laws for our structures, e.g., for \text{Monoid}.

```
record MonoidOK X { { M : Monoid X } } : Set where
  field
  absorbL : (x : X) \rightarrow \varepsilon \bullet x \simeq x
  absorbR : (x : X) \rightarrow x \bullet \varepsilon \simeq x
  assoc : (x y z : X) \rightarrow (x \bullet y) \bullet z \simeq x \bullet (y \bullet z)
```

Let’s check that \( \mathbb{N} \) really gives a monoid.

```
natMonoidOK : MonoidOK N
natMonoidOK = record
  { absorbL = \_ \rightarrow refl
  ; absorbR = \_ \rightarrow zero
  ; assoc = assoc+
  } where -- see below
```

The \( \text{absorbL} \) law follows by computation, but the other two require inductive proof.

```
  _\rightarrow zero : \forall x \rightarrow x \mathbb{N} \text{ zero} \simeq x
  zero \rightarrow zero = refl
  suc n \rightarrow zero rewrite n \rightarrow zero = refl
  assoc+ : \forall x y z \rightarrow (x \mathbb{N} y) \mathbb{N} z \simeq x \mathbb{N} (y \mathbb{N} z)
  assoc+ \rightarrow zero y z = refl
  assoc+ (suc x) y z rewrite assoc+ x y z = refl
```
The usual inductive proofs become structurally recursive functions, pattern matching on the argument in which \( \text{Nat} \) is strict, so that computation unfolds. Sadly, an Agda program, seen as a proof document does not show you the subgoal structure differently from the way in which a Coq script also does not.

Exercise 1.17 (ListN monoid)  
This is a nasty little exercise. By all means warm up by proving that \( \text{List} X \) is a monoid with respect to concatenation, but I want you to have a crack at

\[
\text{listNMonoidOK} : \{ X : \text{Set} \} \to \text{MonoidOK} (\text{ListN} \ N X)
\]

Hint 1: use curried helper functions to ensure structural recursion. The inductive step cases are tricky because the hypotheses equate number-vector pairs, but the components of those pairs are scattered in the goal, so \text{rewrite} will not help. Hint 2: use \text{subst} with a predicate of form \( \lambda \ n \ \text{xs} \to \ldots \), which will allow you to abstract over separated places with \( n \) and \( \text{xs} \).

Exercise 1.18 (a not inconsiderable problem)  
Find out what goes wrong when you try to state associativity of vector \( + + \), let alone prove it. What does it tell you about our \( \equiv \) setup?

A monoid homomorphism is a map between their carrier sets which respects the operations.

\[
\begin{align*}
\text{record MonoidHom} & \{ X \} \{ \{ MX \ : \text{Monoid} X \} \} \{ \{ MY \ : \text{Monoid} Y \} \} (f : X \to Y) : \text{Set} \text{ where} \\
\text{field} & \quad \text{resp} : f \varepsilon \equiv \varepsilon \\
& \quad \text{resp} \cdot : \forall x x' \to f (x \cdot x') \equiv f x \cdot f x'
\end{align*}
\]

For example, taking the length of a list is, in the Normal representation, trivially a homomorphism.

\[
\text{fstHom} : \forall \{ X \} \to \text{MonoidHom} \{\text{ListN} \ N X\} \{\text{N}\} \text{ fst}
\]

\[
\text{fstHom} = \text{record} \{ \text{resp} \varepsilon = \text{refl}; \text{resp} \cdot = \lambda \_ \_ \to \text{refl} \}
\]

Moving along to functorial structures, let us explore laws about the transformation of functions. Equations at higher order mean trouble ahead!

\[
\begin{align*}
\text{record EndoFunctorOK} F & \{ \{ FF \ : \text{EndoFunctor} F \} \} : \text{Set} \text{ where} \\
\text{field} & \quad \text{endoFunctorId} : \forall \{ X \} \to \\
& \quad \text{map} \ \{ \{ FF \} \} \{ X \} \text{ id} \equiv \text{id} \\
\text{endoFunctorCo} : \forall \{ S T \} (f : S \to T) (g : R \to S) \to \\
& \quad \text{map} \ \{ \{ FF \} \} f \circ \text{map} g \equiv \text{map} (f \circ g)
\end{align*}
\]

However, when we try to show,

\[
\text{vecEndoFunctorOK} : \forall \{ n \} \to \text{EndoFunctorOK} \lambda X \to \text{Vec} X n
\]

\[
\text{vecEndoFunctorOK} = \text{record} \\
\{ \text{endoFunctorId} = \{ \} \_ \}; \text{endoFunctorCo} = \lambda f g \to \{ \}_1
\]

we see concrete goals (up to some tidying):
1.8 Laws for Applicative and Traversable

This is a fool’s errand. The pattern matching definition of \( \text{vapp} \) will not allow these equations on functions to hold at the level of \( \equiv \). We could make them a little more concrete by doing induction on \( n \), but we will still not force enough computation. Our \( \simeq \) cannot be extensional for functions because it has canonical proofs for nothing more than \( \equiv \), and \( \equiv \) cannot incorporate extensionality and remain decidable.

We can define pointwise equality,

\[
\overset{\triangle}{=}_\circ : \forall \{ l \} \{ T : \text{Set} \} \{ S : \text{Set} \} \{ X : S \rightarrow T \} \{ f : S \rightarrow T \} \{ g : S \rightarrow T \} \rightarrow S \rightarrow T \\
\]    

\[
f \overset{\triangle}{=} g = \forall x \rightarrow f x \simeq g x \]

infix \( 1 \)

which is reflexive but not substitutive.

Now we can at least require:

\[
\operatorname{record} \operatorname{EndoFunctorOKP} \ F \ {\{ FF : \text{EndoFunctor} \ F \} : \text{Set}_1 \ where} \\
\text{field} \\
\text{endoFunctorId} : \forall \{ X \} \rightarrow \map{\{ FF \}}{X} \id \simeq \id \\
\text{endoFunctorCo} : \forall \{ R S T \} \{ f : S \rightarrow T \} \{ g : R \rightarrow S \} \rightarrow \map{\{ FF \}}{f \circ g} \simeq \map{\{ FF \}}{f \circ g}
\]

Exercise 1.19 (Vec functor laws) Show that vectors are functorial.

\[
\operatorname{vecEndoFunctorOKP} : \forall \{ n \} \rightarrow \operatorname{EndoFunctorOKP} \lambda X \rightarrow \text{Vec} \ X \ n \\
\operatorname{vecEndoFunctorOKP} = \ \equiv
\]

1.8 Laws for Applicative and Traversable

Developing the laws for Applicative and Traversable requires more substantial chains of equational reasoning. Here are some operators which serve that purpose, inspired by work from Lennart Augustsson and Shin-Cheng Mu.

\[
\overset{\square}{=} : \forall \{ l \} \{ X : \text{Set} \} \{ y z \} \rightarrow x \simeq y \rightarrow y \simeq z \rightarrow x \simeq z \\
\overset{\square}{=} = \operatorname{refl} \\
\overset{\square}{=} = \operatorname{refl} \\
\overset{\square}{=} = \operatorname{refl} \\
\overset{\square}{=} = \operatorname{refl}
\]

These three build right-nested chains of equations. Each requires an explicit statement of where to start. The first two step along an equation used left-to-right or right-to-left, respectively, then continue the chain. Then, \( x \square \) marks the end of the chain.

Meanwhile, we may need to rewrite in a context whilst building these proofs. In the expression syntax, we have nothing like \texttt{rewrite}.

\[
\operatorname{cong} : \forall \{ k l \} \{ X : \text{Set} \} \{ Y : \text{Set} \} \{ f : X \rightarrow Y \} \{ x y \} \rightarrow x \simeq y \rightarrow f x \simeq f y \\
\operatorname{cong} f \operatorname{refl} = \operatorname{refl}
\]
Thus armed, let us specify what makes an Applicative acceptable, then show that such a thing is certainly a Functor.

```
record ApplicativeOKP F {{ AF : Applicative F }} : Set₁ where
  field
  lawId :  forall { X } (x : F X) →
           pure {{ AF }} id ⊙ x ≈ x
  lawCo :  forall { R S T } (f : F (S → T)) (g : F (R → S)) (r : F R) →
           pure {{ AF }} (λ f g → f ∘ g) ⊙ f ⊙ g ⊙ r ≈ f ⊙ (g ⊙ r)
  lawHom :  forall { S T } (f : S → T) (s : S) →
            pure {{ AF }} (λ f → f s) ⊙ f
  lawCom :  forall { S T } (f : F (S → T)) (s : S) →
            f ⊙ pure s ≈ pure {{ AF }} (λ f → f s) ⊙ f

applicativeEndoFunctorOKP : EndoFunctorOKP F {{ applicativeEndoFunctor }}
applicativeEndoFunctorOKP = record
  { endoFunctorId = lawId ; endoFunctorCo = λ f g r →
     pure {{ AF }} f ⊙ (pure {{ AF }} g ⊙ r) ;
     (lawCo (pure f) (pure g) r) =
     pure {{ AF }} (λ f g → f ∘ g) ⊙ pure f ⊙ pure g ⊙ r
               ≈ cong (λ x → x ⊙ pure g ⊙ r) (lawHom (λ f g → f ∘ g) f)
     pure {{ AF }} (λ x → x ⊙ pure r) (lawHom (λ x → f ⊙ x) g)
       pure {{ AF }} (f ⊙ g) ⊙ r
     □ }
```

**Exercise 1.20 (ApplicativeOKP for Vec)** Check that vectors are properly applicative. You can get away with rewrite for these proofs, but you might like to try the new tools.

```
vecApplicativeOKP : { n : N } → ApplicativeOKP λ X → Vec X n
vecApplicativeOKP = ?
```

Given that traverse is parametric in an Applicative, we should expect to observe the corresponding naturality. We thus need a notion of applicative homomorphism, being a natural transformation which respects pure and ⊙. That is,

```
φ : forall (F G : Set → Set) → Set₁
F → G = forall { X } → F X → G X
record AppHom { F } {{ AF : Applicative F }} { G } {{ AG : Applicative G }}
  (k : F → G) : Set₁ where
  field
  respPure :  forall { X } (x : X) → k (pure x) ≈ pure x
  resp⊙ :  forall { S T } (f : F (S → T)) (s : F S) → k (f ⊙ s) ≈ k f ⊙ k s
```

We may readily check that monoid homomorphisms lift to applicative homomorphisms.

```
monoidApplicativeHom :
  forall { X } {{ MX : Monoid X }} { Y } {{ MY : Monoid Y }}
  (f : X → Y) {{ hf : MonoidHom f }} →
  AppHom { { monoidApplicative {{ MX }} } } {{ monoidApplicative {{ MY }} } } f
monoidApplicativeHom f {{ hf }} = record
```
1.8. LAWS FOR APPLICATIVE AND TRAVERSABLE

\{\text{respPure} = \lambda x \rightarrow \text{MonoidHom}.\text{resp}\rhd hf\}
; \text{resp} \odot = \text{MonoidHom}.\text{resp} \bullet hf

**Exercise 1.21 (homomorphism begets applicative)** Show that a homomorphism from \(F\) to \(G\) induces applicative structure on their pointwise sum.

\[
\text{homSum} : \forall \{F \to G\} \{\{AF : \text{Applicative } F\}\} \{\{AG : \text{Applicative } G\}\} \rightarrow \text{Applicative } \lambda X \rightarrow F X + G X
\]

Check that your solution obeys the laws.

\[
\text{homSumOKP} : \forall \{F \to G\} \{\{AF : \text{Applicative } F\}\} \{\{AG : \text{Applicative } G\}\} \rightarrow \text{Applicative OKP} F \rightarrow \text{Applicative OKP} G \rightarrow \text{AppHom} f \rightarrow \text{Applicative OKP} f \text{ homf } = ?
\]

Laws for Traversable functors are given thus:

\[
\text{record TraversableOKP } F \{\{TF : \text{Traversable } F\}\} : \text{Set}_1 \text{ where}
\]
\[
\begin{align*}
\text{lawId} : & \forall \{X\} \{xs : F X\} \rightarrow \text{traverse \text{id} } \text{xs } \simeq \text{xs} \\
\text{lawCo} : & \forall \{G\} \{\{AG : \text{Applicative } G\}\} \{H\} \{\{AH : \text{Applicative } H\}\} \{RS T\} \{g : S \rightarrow G T\} \{h : R \rightarrow H S\} \{rs : F R\} \rightarrow \\
& \text{let } EH : \text{EndoFunctor } H; EH = \text{applicativeEndoFunctor} \in \\
& \text{map } \{H\} (\text{traverse } g) (\text{traverse } h rs) \\
& \simeq \\
& \text{traverse } \{\{TF\}\} \{\{\text{applicativeComp }AH AG\}\} (\text{map } \{H\} (g \circ h) rs) \\
\text{lawHom} : & \forall \{G\} \{\{AG : \text{Applicative } G\}\} \{H\} \{\{AH : \text{Applicative } H\}\} \{hs : F S\} \rightarrow \\
& \text{traverse } (h \circ g) \text{ ss } \simeq h (\text{traverse } g \text{ ss})
\end{align*}
\]

Let us now check the coherence property we needed earlier.

\[
\text{toNormal} : \forall \{F\} \{\{TF : \text{Traversable } F\}\} \rightarrow \text{TraversableOKP } F \rightarrow \\
\forall \{X\} \rightarrow F X \rightarrow \| \text{normalT } F \|_N X
\]

We may now construct
toNormal \( tokf \ fx \)
\[
= \text{shapeT } fx \\
, \quad \text{subst} \left( \text{lengthContentsSizeShape } tokf \ fx \right) \left( \text{Vec}_\_ \right) \left( \text{snd} \left( \text{contentsT } fx \right) \right)
\]

**Exercise 1.22** Define fromNormal, reversing the direction of toNormal. One way to do it is to define what it means to be able to build something from a batch of contents.

**Batch**
\[
: \text{Set} \to \text{Set} \to \text{Set} \\
X \ Y = \Sigma \ N \lambda n \to \text{Vec} \ X \ n \to Y
\]

Show Batch \( X \) is applicative. You can then use traverse on a shape to build a Batch job which reinserts the contents. As above, you will need to prove a coherence property to show that the contents vector in your hand has the required length. Warning: you may encounter a consequence of defining sizeT via crush with ignored target type One, and need to prove that you get the same answer if you ignore something else. Agda's 'Toggle display of hidden arguments' menu option may help you detect that scenario.

Showing that toNormal and fromNormal are mutually inverse looks like a tall order, given that the programs have been glued together with coherence conditions. At time of writing, it remains undone. When I see a mess like that, I wonder whether replacing indexing by the measure of size might help.

### 1.9 Fixpoints of Normal Functors

The universal first order simple datatype is given by taking the least fixpoint of a normal functor.

**data**
\[
\text{Tree} \ N : \text{Normal} \ni \text{Set} \text{ where} \\
\langle \_ \rangle : \llbracket N \rrbracket_N (\text{Tree} \ N) \to \text{Tree} \ N
\]

We may, for example, define the natural numbers this way:

**NatT**
\[
: \text{Normal} \\
\text{NatT} = \text{Two} / \text{Zero} \langle ? \rangle \ 1 \\
\text{zeroT} : \text{Tree} \ \text{NatT} \\
\text{zeroT} = \langle \text{tt}, \langle \rangle \rangle \\
\text{sucT} : \text{Tree} \ \text{NatT} \to \text{Tree} \ \text{NatT} \\
\text{sucT} \ n = \langle \text{ff}, n, \langle \rangle \rangle
\]

Of course, to prove these are the natural numbers, we need the eliminator as well as the constructors.

**Exercise 1.23** Prove the principle of induction for these numbers.

\[
\text{NatInd} : \text{forall} \ \{ l \} \left( P : \text{Tree} \ \text{NatT} \to \text{Set} \ l \right) \to \\
\left( P \ \text{zeroT} \to \\
\left( (n : \text{Tree} \ \text{NatT}) \to P \ n \to P \ (\text{sucT} \ n) \right) \to \\
\left( n : \text{Tree} \ \text{NatT} \right) \to P \ n \right) \rightsquigarrow \ P z s n
\]

Indeed, there’s a generic induction principle for the whole lot of these types. First, we need predicate transformer to generate the induction hypothesis.
1.9. FIXPOINTS OF NORMAL FUNCTORS

\textbf{All}: \forall l \{ X : X \to \text{Set} \} \{ n \} \to \text{Vec} X n \to \text{Set} l

All \ P \langle \rangle = \text{One}

All \ P (x, xs) = P x \times \text{All} \ P xs

We then acquire

\textbf{induction}: \forall (N : \text{Normal}) \{ l \} \{ P : \text{Tree} N \to \text{Set} l \} \to
\langle (s : \text{Shape} N) (ts : \text{Vec} (\text{Tree} N) (\text{size} N s)) \to \text{All} P ts \to P \langle s, ts \rangle \rangle \to
\langle t : \text{Tree} N \to P t \rangle

\textbf{induction} \ N P p (s, ts) = p s ts (\text{hyps} ts) \text{ where}

\text{hyps}: \forall \{ n \} (ts : \text{Vec} (\text{Tree} N) n) \to \text{All} P ts

\text{hyps} \langle \rangle = \langle \rangle

\text{hyps} (t, ts) = \text{induction} \ N P p t, \text{hyps} ts

\textbf{Exercise 1.24 (decidable equality)} We say a property is decided if we know whether it is true or false, where falsity is indicated by function to \text{Zero}, an empty type.

\text{Dec} : \text{Set} \to \text{Set}

\text{Dec} \ X = X + (X \to \text{Zero})

Show that if a normal functor has decidable equality for its shapes, then its fixpoint also has decidable equality.

\text{eq?} : (N : \text{Normal}) (\text{sheq}?) : (s, s' : \text{Shape} N) \to \text{Dec} (s \simeq s') \to
\langle t, t' : \text{Tree} N \to \text{Dec} (t \simeq t') \rangle

eq? \ N \text{ sheq?} t t' = ?
Chapter 2

Simply Typed λ-Calculus

This chapter contains some standard techniques for the representation of typed syntax and its semantics. The joy of typed syntax is the avoidance of junk in its interpretation. Everything fits, just so.

2.1 Syntax

Last century, I learned the following recipe for well typed terms of the simply typed λ-calculus from Altenkirch and Reus.

First, give a syntax for types. I shall start with a base type and close under function spaces.

```
data $\star : \text{Set}$ where
  $\text{t : } \star$
  $\Rightarrow \star : \star \rightarrow \star \rightarrow \star$
  $\textit{infixr } \Rightarrow \star$
```

Next, build contexts as snoc-lists.

```
data Cx (X : \text{Set}) : \text{Set}$ where
  $E : Cx X$
  $\textit{'} : Cx X \rightarrow X \rightarrow Cx X$
  $\textit{infixl } \text{'}$
```

Now, define typed de Bruijn indices to be context membership evidence.

```
data $\in (\tau : \star) : Cx \star \rightarrow \text{Set}$ where
  $\text{zero : forall } \{ \Gamma \} \rightarrow \tau \in \Gamma, \tau$
  $\text{succ : forall } \{ \Gamma \sigma \} \rightarrow \tau \in \Gamma \rightarrow \tau \in \Gamma, \sigma$
  $\textit{infix } \text{'}$
```

That done, we can build well typed terms by writing syntax-directed rules for the typing judgment.

```
data $\vdash (\Gamma : Cx \star) : \star \rightarrow \text{Set}$ where
  $\text{var : forall } \{ \tau \}$
  $\rightarrow \tau \in \Gamma$
  $\rightarrow \Gamma \vdash \tau$
  $\text{lam : forall } \{ \sigma \tau \}$
```
CHAPTER 2. SIMPLY TYPED λ-CALCULUS

\[ \rightarrow \Gamma, \sigma \vdash \tau \]
\[ \rightarrow \Gamma \vdash \sigma \triangleright \tau \]

**app:**
\[ \text{forall} \{ \sigma \tau \} \]
\[ \rightarrow \Gamma \vdash \sigma \triangleright \tau \rightarrow \Gamma \vdash \sigma \]
\[ \rightarrow \Gamma \vdash \tau \]

**infix \# \_**

### 2.2 Semantics

Writing an interpreter for such a calculus is an exercise also from last century, for which we should thank Augustsson and Carlsson. Start by defining the semantics of each type.

\[
\begin{align*}
[\cdot]_* & : \ast \rightarrow \text{Set} \\
[i]_* & = \mathbb{N} \quad \text{by way of being nontrivial} \\
[\sigma \triangleright \tau]_* & = [\sigma]_* \rightarrow [\tau]_*
\end{align*}
\]

Next, define *environments* for contexts, with projection. We can reuse these definitions in the rest of the section if we abstract over the notion of value.

\[
\begin{align*}
[\cdot]_{C_x} & : C_x \ast \rightarrow (\ast \rightarrow \text{Set}) \rightarrow \text{Set} \\
[E]_{C_x} V & = \text{One} \\
[\Gamma, \sigma]_{C_x} V & = [\Gamma]_{C_x} V \times V \sigma \\
[\cdot]_e : \text{forall} \{ \Gamma \tau V \} \rightarrow \tau \in \Gamma \rightarrow [\Gamma]_{C_x} V \rightarrow V \tau \\
[\text{zero}]_e (\gamma, t) & = t \\
[\text{suc} i]_e (\gamma, s) & = [i]_e \in \gamma
\end{align*}
\]

Finally, define the meaning of terms.

\[
\begin{align*}
[\cdot]_e : \text{forall} \{ \Gamma \tau \} \rightarrow \Gamma \vdash \tau \rightarrow [\Gamma]_{C_x} [\cdot]_* \rightarrow [\tau]_* \\
[\var i]_e \gamma & = [i]_e \in \gamma \\
[\text{lam} t]_e \gamma & = \lambda s \rightarrow [t]_e (\gamma, s) \\
[\text{app} f s]_e \gamma & = [f]_e \gamma ([s]_e \gamma)
\end{align*}
\]

### 2.3 Substitution with a Friendly Fish

We may define the types of simultaneous renamings and substitutions as type-preserving maps from variables:

\[
\begin{align*}
\text{Ren Sub} : & \ C_x \ast \rightarrow C_x \ast \rightarrow \text{Set} \\
\text{Ren} \Gamma \Delta & = \text{forall} \{ \tau \} \rightarrow \tau \in \Gamma \rightarrow \tau \in \Delta \\
\text{Sub} \Gamma \Delta & = \text{forall} \{ \tau \} \rightarrow \tau \in \Gamma \rightarrow \Delta \vdash \tau
\end{align*}
\]

The trouble with defining the action of substitution for a de Bruijn representation is the need to shift indices when the context grows. Here is one way to address that situation. First, let me define context extension as concatenation with a cons-list, using the \(\langle \rangle\) operator.

\[
\begin{align*}
\langle \rangle & : \forall X \rightarrow C_x X \rightarrow \text{List} X \rightarrow C_x X \\
x \# < \# & = x \#
\end{align*}
\]

\(\langle \rangle\) is pronounce ‘fish’, for historical reasons.
2.4. A MODERN CONVENIENCE

We may then define the shiftable simultaneous substitutions from $\Gamma$ to $\Delta$ as type-preserving mappings from the variables in any extension of $\Gamma$ to terms in the same extension of $\Delta$.

\[
\text{Shub} : \text{Cx} \times \rightarrow \text{Cx} \times \rightarrow \text{Set} \\
\text{Shub} \Gamma \Delta = \forall \Xi . \text{Sub} (\Gamma \ll \Xi) (\Delta \ll \Xi)
\]

By the computational behaviour of $\ll$, a $\text{Shub} \Gamma \Delta$ can be used as a $\text{Shub} (\Gamma', \sigma) (\Delta', \sigma)$, so we can push substitutions under binders very easily.

\[
\text{wkr} : \forall \{ \Gamma \Delta \sigma \} . \text{Ren} \Gamma \Delta \rightarrow \text{Ren} (\Gamma', \sigma) (\Delta', \sigma) \\
\text{wkr} r \text{zero} = \text{zero} \\
\text{wkr} r \text{suc} i = \text{suc} (r i) \\
\text{ren} : \forall \{ \Gamma \Delta \} . \text{Ren} \Gamma \Delta \rightarrow \text{Shub} \Gamma \Delta \\
\text{ren} r \text{} = \var i \\
\text{ren} r (\ll \Xi) = \text{ren} (\text{wkr} r \Xi)
\]

With renaming available, we can play the same game for substitutions.

\[
\text{wks} : \forall \{ \Gamma \Delta \sigma \} . \text{Sub} \Gamma \Delta \rightarrow \text{Sub} (\Gamma', \sigma) (\Delta', \sigma) \\
\text{wks} s \text{zero} = \text{var} \text{zero} \\
\text{wks} s \text{suc} i = \text{ren} \text{suc} \ll s i \\
\text{sub} : \forall \{ \Gamma \Delta \} . \text{Sub} \Gamma \Delta \rightarrow \text{Shub} \Gamma \Delta \\
\text{sub} s \text{} = s \\
\text{sub} s (\ll \Xi) = \text{sub} (\text{wks} s) \Xi
\]

2.4 A Modern Convenience

Bob Atkey once remarked that ability to cope with de Bruijn indices was a good reverse Turing Test, suitable for detecting humaniform robotic infiltrators. Correspondingly, we might like to write terms which use real names. I had an idea about how to do that.

We can build the renaming which shifts past any context extension.

\[
\text{weak} : \forall \{ \Gamma \} . \Xi \rightarrow \text{Ren} (\Gamma \ll \Xi) \\
\text{weak} (\ll i) = i \\
\text{weak} (\ll \Xi) i = \text{weak} \Xi \text{ (suc } i) 
\]

Then, we can observe that to build the body of a binder, it is enough to supply a function which will deliver the term representing the variable in any suitably extended context. The context extension is given implicitly, to be inferred from the usage site, and then the correct weakening is applied to the bound variable.
**Chapter 2. Simply Typed λ-Calculus**

\[ \lambda : \text{forall } \{ \Gamma \sigma \tau \} \rightarrow \]
\[ ((\text{forall } \{ \Xi \} \rightarrow \Gamma \, \sigma \iff \Xi \vdash \sigma) \rightarrow \Gamma \vdash \sigma \triangleright \tau) \rightarrow \]
\[ \Gamma \vdash \sigma \triangleright \tau \]
\[ \lambda f = \text{lam} (f : \lambda \{ \Xi \} \rightarrow \text{var} (\text{weak } \Xi \text{ zero})) \]

But sadly, the following does not typecheck

\[ \text{myTest } : \mathcal{E} \vdash \iota \triangleright \iota \]
\[ \text{myTest } = \lambda \lambda \lambda x \rightarrow x \]

because the following constraint is not solved:

\[ (\mathcal{E}, \iota \ll \_ \Xi_{232} x) = (\mathcal{E}, \iota) : \text{Cx } \star \]

That is, constructor-based unification is insufficient to solve for the prefix of a context, given a common suffix.

By contrast, solving for a suffix is easy when the prefix is just a value: it requires only the stripping off of matching constructors. So, we can cajole Agda into solving the problem by working with its reversal, via the ‘chips’ operator:

\[ \lllll : \text{forall } \{ X \} \rightarrow \text{Cx } X \rightarrow \text{List } X \rightarrow \text{List } X \]
\[ \mathcal{E} \lllll y s = y s \]
\[ (x z, x) \lllll y s = x z \lllll (x, y s) \]

Of course, one must prove that solving the reverse problem is good for solving the original.

**Exercise 2.1 (reversing lemma)**

Show

\[ \text{lem } : \text{forall } \{ X \} \ (\Delta \Gamma : \text{Cx } X \Xi) \rightarrow \]
\[ \Delta \lllll () \simeq \Gamma \lllll \Xi \rightarrow \Gamma \lllll \Xi \simeq \Delta \]
\[ \text{lem } \Delta \Gamma \Xi q = ? \]

Now we can frame the constraint solve as an instance argument supplying a proof of the relevant equation on cons-lists: Agda will try to use \textit{refl} to solve the instance argument, triggering the tractable version of the unification problem.

\[ \lambda : \text{forall } \{ \Gamma \sigma \tau \} \rightarrow \]
\[ ((\text{forall } \{ \Delta \Xi \} \{ \{ \_ : \Delta \lllll () \simeq \Delta \lllll (\sigma, \Xi) \} \} \rightarrow \Delta \vdash \sigma) \rightarrow \]
\[ \Gamma \vdash \sigma \triangleright \tau \]
\[ \lambda f = \text{lam} (f : \lambda \{ \Delta \Xi \} \{ \{ q \} \} \rightarrow \]
\[ \text{subt} (\text{lem } \Delta \Gamma (-, \Xi) q) (\lambda \Gamma \rightarrow \Gamma \vdash -) (\text{var} (\text{weak } \Xi \text{ zero}))) \]
\[ \text{myTest } : \mathcal{E} \vdash (\iota \triangleright \iota) \triangleright (\iota \triangleright \iota) \]
\[ \text{myTest } = \lambda \lambda \lambda x \rightarrow \text{app } f (\text{app } f x) \]

### 2.5 Hereditary Substitution

This section is a structured series of exercises, delivering a \(\beta\eta\)-long normalization algorithm for our \(\lambda\)-calculus by the method of \textit{hereditary substitution}.

The target type for the algorithm is the following right-nested spine representation of \(\beta\)-normal \(\eta\)-long forms.
2.5. HEREDITARY SUBSTITUTION

mutual
data ⊨_ (Γ : Cx ⋆) : ⋆ → Set where
lam : forall {σ τ} → Γ, σ ⊨ τ → Γ ⊨ σ → τ
ₗₜ : forall {τ} → τ ∈ Γ → Γ ⊨ τ → Γ ⊨ t

data ⊩*_ (Γ : Cx ⋆) : ⋆ → Set where
⟨⟩ : Γ ⊨ ⋆
ₗₜ : forall {τ} → Γ ⊨ τ → Γ ⊨* τ → Γ ⊨* σ → τ

infix 3 ⊨_ ⋆
infix 3 $:

That is Γ ⊨ τ is the type of normal forms in τ, and Γ ⊩* τ is the type of spines for a τ, delivering ι.

The operation of hereditary substitution replaces one variable with a normal form and immediately performs all the resulting computation (i.e., more substitution), returning a normal form. You will need some equipment for talking about individual variables.

Exercise 2.2 (thinning) Define the function → which removes a designated entry from a context, then implement the thinning operator, being the renaming which maps the embed the smaller context back into the larger.

$\downarrow$ : forall (Γ : Cx ⋆) {τ} (x : τ ∈ Γ) → Cx ⋆
Γ $\downarrow$ x = ?
infix 4 $\downarrow$
$\uparrow$ : forall {Γ σ} (x : σ ∈ Γ) → Ren (Γ $\downarrow$ x) Γ
x $\neq$ y = ?

This much will let us frame the problem. We have a candidate value for x which does not depend on x, so we should be able to eliminate x from any term by substituting out. If we try, we find this situation:

(\downarrow) : forall {Γ σ τ} → (x : σ ∈ Γ) → Γ $\downarrow$ x ⊨ σ →
Γ $\vDash$ τ → Γ $\vDash$ x $\vDash$ τ
(x $\mapsto$ s) lam t = lam ((\ suc x $\mapsto$ ?) t)
(x $\mapsto$ s) y $\uparrow$ ts = ?
infix 3 (\downarrow)\

Let us now address the challenges we face.

In the application case, we shall need to test whether or not y is the x for which we must substitute, so we need some sort of equality test. A Boolean equality test does not generate enough useful information—if y is x, we need to know that ts is a suitable spine for s; if y is not x, we need to know its representation in Γ $\vDash$ x. Hence, let us rather prove that any variable is either the one we are looking for or another. We may express this discriminability property as a predicate on variables.

data Veq? {Γ σ} (x : σ ∈ Γ) : forall {τ} → τ ∈ Γ → Set where
same : Veq? x x
diff : forall {τ} (y : τ ∈ Γ $\Rightarrow$ x) → Veq? x (x $\neq$ y)

Exercise 2.3 (variable equality testing) Show that every y is discriminable with respect to a given x.

veq? : forall {Γ σ τ} (x : σ ∈ Γ) (y : τ ∈ Γ) → Veq? x y
veq? x y = ?

Hint: it will help to use with in the recursive case.
Meanwhile, in the \texttt{lam} case, we may easily shift $x$ to account for the new variable in $t$, but we shall also need to shift $s$.

**Exercise 2.4 (closure under renaming)** Show how to propagate a renaming through a normal form.

\begin{verbatim}
mutual
renNm : forall \{ \Gamma \Delta \tau \} \rightarrow \text{Ren} \quad \Gamma \Delta \rightarrow \Gamma \Downarrow \tau \rightarrow \Delta \Downarrow \tau
renNm r t = ?
renSp : forall \{ \Gamma \Delta \tau \} \rightarrow \text{Ren} \quad \Gamma \Delta \rightarrow \Gamma \Downarrow^* \tau \rightarrow \Delta \Downarrow^* \tau
renSp r ss = ?
\end{verbatim}

Now we have everything we need to implement hereditary substitution.

**Exercise 2.5 (hereditary substitution)** Implement hereditary substitution for normal forms and spines, defined mutually with application of a normal form to a spine, performing $\beta$-reduction.

\begin{verbatim}
mutual
\langle \_ \mapsto \_ \rangle : forall \{ \Gamma \sigma \tau \} \rightarrow (x : \sigma \in \Gamma) \rightarrow \Gamma - x \Downarrow \sigma \rightarrow \Gamma \Downarrow \tau
\langle x \mapsto s \rangle t = ?
\langle \_ \mapsto \_ \rangle^* : forall \{ \Gamma \sigma \tau \} \rightarrow (x : \sigma \in \Gamma) \rightarrow \Gamma - x \Downarrow \sigma \rightarrow \Gamma \Downarrow^* \tau
\langle x \mapsto s \rangle^* ts = ?
\text{\_e} : forall \{ \Gamma \tau \} \rightarrow 
\quad \Gamma \Downarrow \tau \rightarrow \Gamma \Downarrow^* \tau \rightarrow \Gamma \Downarrow t
f \# ss = ?
infix \beta \text{\_e}
infix 2 \langle \_ \mapsto \_ \rangle
\end{verbatim}

Do you think these functions are mutually structurally recursive?

With hereditary substitution, it should be a breeze to implement normalization, but there is one little tricky part remaining.

**Exercise 2.6 ($\eta$-expansion for normalize)** If we start implementing \texttt{normalize}, it is easy to get this far:

\begin{verbatim}
normalize : forall \{ \Gamma \tau \} \rightarrow \Gamma \Downarrow \tau \rightarrow \Gamma \Downarrow \tau
normalize (var x) = ?
normalize (lam t) = lam (normalize t)
normalize (app f s) with normalize f | normalize s
normalize (app f s) | lam t | s' = (zero \mapsto s') t
\end{verbatim}

We can easily push under \texttt{lam} and implement \texttt{app} by hereditary substitution. However, if we encounter a variable, $x$, we must deliver it in $\eta$-long form. You will need to figure out how to expand $x$ in a type-directed manner, which is not a trivial thing to do. Hint: if you need to represent the prefix of a spine, it suffices to consider functions from suffices.

Here are a couple of test examples for you to try. You may need to translate them into de Bruijn terms manually if you have not yet proven the ‘reversing lemma’.
2.6 Normalization by Evaluation

Let’s cook normalization a different way, extracting more leverage from Agda’s computation machinery. The idea is to model values as either ‘going’ (capable of computation if applied) or ‘stopping’ (incapable of computation, but not ‹η›-long). The latter terms look like left-nested applications of a variable.

\[
\begin{align*}
\text{data } \text{Stop} & (\Gamma : \text{Cx} \times) (\tau : 
\text{*}) : \text{Set} \ 	ext{where} \\
\text{var} & : \tau \in \Gamma \rightarrow \text{Stop } \Gamma \tau \\
\text{gov} & : \forall \{ \sigma \} \rightarrow \text{Stop } \Gamma (\sigma \triangleright \tau) \rightarrow \Gamma \models \sigma \rightarrow \text{Stop } \Gamma \tau
\end{align*}
\]

Exercise 2.7 (Stop equipment) Show that Stop terms are closed under renaming, and that you can apply them to a spine to get a normal form.

\[
\begin{align*}
\text{renSt} & : \forall \{ \Gamma \Delta \tau \} \rightarrow \text{Ren } \Gamma \Delta \rightarrow \text{Stop } \Gamma \tau \rightarrow \text{Stop } \Delta \tau \\
\text{renSt } r u & = ? \\
\text{stopSp} & : \forall \{ \Gamma \tau \} \rightarrow \text{Stop } \Gamma \tau \rightarrow \Gamma \models \ast \tau \rightarrow \Gamma \models \iota \\
\text{stopSp } us & ss = ?
\end{align*}
\]

Let us now give a contextualized semantics to each type. Values either Go or Stop. Ground values cannot go: Zero is a datatype with no constructors. Functional values have a Kripke semantics. Wherever their context is meaningful, they take values to values.

\[
\begin{align*}
\text{mutual} & \\
\text{Val} & : \text{Cx} \times \rightarrow \times \rightarrow \text{Set} \\
\text{Val } \Gamma \tau & = \text{Go } \Gamma \tau + \text{Stop } \Gamma \tau \\
\text{Go} & : \text{Cx} \ast \rightarrow \ast \rightarrow \text{Set} \\
\text{Go } \Gamma \iota & = \text{Zero} \\
\text{Go } \Gamma (\sigma \triangleright \tau) & = \forall \{ \Delta \} \rightarrow \text{Ren } \Gamma \Delta \rightarrow \text{Val } \Delta \sigma \rightarrow \text{Val } \Delta \tau
\end{align*}
\]

Exercise 2.8 (renaming values and environments) Show that values admit renaming. Extend renaming to environments storing values. Construct the identity environment, mapping each variable to itself.

\[
\begin{align*}
\text{renVal} & : \forall \{ \Gamma \Delta \tau \} \rightarrow \text{Ren } \Gamma \Delta \rightarrow \text{Val } \Gamma \tau \rightarrow \text{Val } \Delta \tau \\
\text{renVal } r u v & = ? \\
\text{renVals} & : \forall \{ \Gamma \Delta \} \rightarrow \text{Ren } \Gamma \Delta \rightarrow \text{Val } \Gamma \tau \rightarrow \text{Val } \Delta \sigma \rightarrow \text{Val } \Delta \tau
\end{align*}
\]

\[
\begin{align*}
\text{idEnv} & : \forall \{ \Gamma \} \rightarrow \text{Val } \Gamma \\
\text{idEnv } \Gamma & = ?
\end{align*}
\]
Exercise 2.9 (application and quotation) Implement application for values. In order to apply a stopped function, you will need to be able to extract a normal form for the argument, so you will also need to be able to ‘quote’ values as normal forms. It seems `quote` is a reserved symbol in Agda.

mutual

apply : \forall \{ \Gamma \sigma \tau \} \to \text{Val} \Gamma (\sigma \triangleright \tau) \to \text{Val} \Gamma \sigma \to \text{Val} \Gamma \tau
apply f s = ?

quo : \forall \{ \Gamma \} \tau \to \text{Val} \Gamma \tau \to \Gamma \models \tau
quo \tau v = ?

For the last step, we need to compute values from terms.

Exercise 2.10 (evaluation) Show that every well typed term can be given a value in any context where its free variables have values.

eval : \forall \{ \Gamma \Delta \tau \} \to \Gamma \vdash \tau \to [[ \Gamma ] \text{C}_x (\text{Val} \Delta) \to \text{Val} \Delta \tau

eval t \gamma = ?

With all the pieces in place, we get

\text{normByEval} : \forall \{ \Gamma \tau \} \to \Gamma \vdash \tau \to \Gamma \models \tau
\text{normByEval} \{ \Gamma \} \{ \tau \} t = \text{quo} \tau (\text{eval} t (\text{idEnv} \Gamma))

Exercise 2.11 (numbers and primitive recursion) Consider extending the term language with constructors for numbers and a primitive recursion operator.

\text{zero} : \Gamma \vdash t
\text{suc} : \Gamma \vdash t \to \Gamma \vdash t
\text{rec} : \forall \{ \tau \} \to \Gamma \vdash \tau \to \Gamma \vdash (t \triangleright \tau \triangleright \tau) \to \Gamma \vdash t \to \Gamma \vdash \tau

How should the normal forms change? How should the values change? Can you extend the implementation of normalization?

Exercise 2.12 (adding adding) Consider making the further extension with a hardwired addition operator.

\text{suc} : \Gamma \vdash t \to \Gamma \vdash t \to \Gamma \vdash t

Can you engineer the notion of value and the evaluator so that \text{normByEval} identifies

\begin{align*}
\text{add zero} \ t & \quad \text{with} \quad t \\
\text{add} \ s \ \text{zero} & \quad \text{with} \quad s \\
\text{add} \ (\text{suc} \ s) \ t & \quad \text{with} \quad \text{suc} \ (\text{add} \ s \ t) \\
\text{add} \ s \ (\text{suc} \ t) & \quad \text{with} \quad \text{suc} \ (\text{add} \ s \ t) \\
\text{add} \ (\text{add} \ r \ s) \ t & \quad \text{with} \quad \text{add} \ r \ (\text{add} \ s \ t) \\
\text{add} \ s \ t & \quad \text{with} \quad \text{add} \ t \ s
\end{align*}

and thus yields a stronger decision procedure for equality of expressions involving adding? (This is not an easy exercise, especially if you want the last equation to hold. I must confess I have not worked out the details.)
Chapter 3

Containers and W-types

Containers are the infinitary generalization of normal functors.

record Con : Set₁ where
  constructor _≤_
  field
    Sh : Set  -- a set of shapes
    Po : Sh → Set  -- a family of positions
  [ ] where : Set → Set
  [ ] where X = Σ Sh λ s → Po s → X
open Con public
infixr 1 _≤_

Instead of having a size and a vector of contents, we represent the positions for each shape as a set, and the contents as a function from positions.

3.1 Closure Properties

We may readily check that the polynomials are all containers.

Kₐ : Set → Con
Kₐ A = A ‘λ _ → Zero
Iₐ : Con
Iₐ = One ‘λ _ → One
_+₁₋ₐ : Con → Con → Con
(S ‘a P) ‘₁₋₋ₐ (S’ ‘a P’) = (S ‘λ s → P ‘₁₇₇₅₇{s}) ‘₊₁₋₋ₐ P’
ₓₓ₁₋₋ₐ : Con → Con → Con
(S ‘a P) ‘ₓₓ₁₋₋ₐ (S’ ‘a P’) = (S ‘λ s s’ → P ‘₁₇₇₅₇{s}) ‘₊₁₋₋ₐ P’

Moreover, we may readily close containers under dependent pairs and functions, a fact which immediately tells us how to compose containers.

Σₐ : (A : Set) (C : A → Con) → Con
Σₐ θ C = (Σ A λ a → Sh (C a)) ‘₊₁₋₋ₐ λ a s → Po (C a) s
Πₐ : (A : Set) (C : A → Con) → Con
Πₐ θ C = ((a : A) → Sh (C a)) ‘₊₁₋₋ₐ λ f → Σ A λ a → Po (C a) (f a)
ₒₒ₋₋ₐ : Con → Con → Con
(S ‘a P) ‘ₒₒ₋₋ₐ C = Σₐ S ‘λ s → Πₐ (P ‘₁₇₇₅₇{s}) ‘ₒₒ₋₋ₐ P → C
Exercise 3.1 (containers are endofunctors) Check that containers yield endofunctors which obey the laws.

\[
\text{conEndoFunctor} : \{ C : \text{Con} \} \to \text{EndoFunctor} \llbracket C \rrbracket_d
\]
\[
\text{conEndoFunctor} \{ S \triangleleft P \} = ?
\]
\[
\text{conEndoFunctorOKP} : \{ C : \text{Con} \} \to \text{EndoFunctorOKP} \llbracket C \rrbracket_d
\]
\[
\text{conEndoFunctorOKP} \{ S \triangleleft P \} = ?
\]

Exercise 3.2 (closure properties) Check that the meanings of the operations on containers are justified by their interpretations as functors.

3.2 Container Morphisms

A container morphism describes a natural transformation between the functors given by containers. As the element type is abstract, there is nowhere that the elements of the output can come from except somewhere in the input. Correspondingly, a container morphism is given by a pair of functions, the first mapping input shapes to output shapes, and the second mapping output positions back to the input positions from which they fetch elements.

\[
\to_d \triangleleft : \text{Con} \to \text{Con} \to \text{Set}
\]
\[
(S \triangleleft P) \to_d (S' \triangleleft P') = \Sigma (S \to S') \lambda f \rightarrow (s \in S) \to P' (f s) \to P s
\]

The action of a container morphism is thus

\[
\text{morph}_d : \forall \{ C, C' \} \to C \to_d C' \to \forall \{ X \} \to \llbracket C \rrbracket_d X \to \llbracket C' \rrbracket_d X
\]

\[
\text{morph}_d f = ?
\]

Interactive Interpretation Peter Hancock encourages us to think of \( S \triangleleft P \) as the description of a command-response protocol, where \( S \) is a set of commands we may invoke and \( P \) tells us which responses may be returned for each command. The type \( \llbracket S \triangleleft P \rrbracket_d X \) is thus a strategy for obtaining an \( X \) by one run of the protocol. Meanwhile, a container morphism is thus a kind of ‘device driver’, translating commands one way, then responses the other.

Exercise 3.3 (representing natural transformations) Check that you can represent any natural transformation between containers as a container morphism.

\[
\text{id}_d \to_d : \forall \{ C \} \to C \to_d C
\]
\[
\text{id}_d \to_d = ?
\]

Exercise 3.4 (identity and composition) Check that you can define identity and composition for container morphisms.

\[
\text{id}_d \to_d : \forall \{ C \} \to C \to_d C
\]
\[
\text{id}_d \to_d = ?
\]

\[
\text{morph}_d : \forall \{ C, C' \} \to (\forall \{ X \} \to \llbracket C \rrbracket_d X \to \llbracket C' \rrbracket_d X) \to C \to_d C'
\]
\[
\text{morph}_d f = ?
\]

\[
\text{id}_d \to_d \circ_d \text{id}_d \to_d = ?
\]

Container-of-positions presentation The above exercise might suggest an equivalent presentation of container morphisms, namely

\[
(S \triangleleft P) \to_d C = (s : S) \to \llbracket C \rrbracket_d (P s)
\]

but the to-and-fro presentation is usually slightly easier to work with. You win some, you lose some.

Exercise 3.5 (identity and composition) Check that you can define identity and composition for container morphisms.
3.3 W-TYPES

The least fixpoint of a container is a \( W \)-type—\( W \) for ‘well founded’.

\[
\text{data } W \ (C \ : \ Con) \ : \ Set \ \text{where} \\
\langle \_ \rangle \ : \ [\ [ \ C \ ] \ a \ (W \ C) \rightarrow W \ C
\]

In an extensional setting, \( W \) can be used to represent a great many datatypes, but intensional systems have some difficulties achieving faithful representations of first order data via \( W \)-types.

**Exercise 3.5 (natural numbers)** Define natural numbers as a \( W \)-type. Implement the constructors. Hint: \textit{magic}: \( \text{Zero} \rightarrow \{ A : \text{Set} \} \rightarrow A \). Implement primitive recursion and use it to implement addition.

\[
\begin{align*}
\text{NatW} & : \text{Set} \\
\text{NatW} & = W ? \\
\text{zeroW} & : \text{NatW} \\
\text{zeroW} & = \langle ? \rangle \\
\text{sucW} & : \text{NatW} \rightarrow \text{NatW} \\
\text{sucW} \ n & = \langle ? \rangle \\
\text{precW} & : \forall \{ l \} \{ T : \text{Set} \} \rightarrow T \rightarrow (\text{NatW} \rightarrow T \rightarrow T) \rightarrow \text{NatW} \rightarrow T \\
\text{precW} \ z \ s \ n & = ? \\
\text{addW} & : \text{NatW} \rightarrow \text{NatW} \rightarrow \text{NatW} \\
\text{addW} \ x \ y & = \text{precW} \ ? \ ? \ x
\end{align*}
\]

How many different implementations of \( \text{zeroW} \) can you find? Meanwhile, discover for yourself why an attempt to establish the induction principle is a fool’s errand.

\[
\begin{align*}
\text{indW} & : \forall \{ l \} \{ P : \text{NatW} \rightarrow \text{Set} \ l \} \rightarrow \\
& \phantom{=} \ P \ \text{zeroW} \rightarrow \\
& \phantom{=} \ ((n : \text{NatW}) \rightarrow P \ n \rightarrow P \ (\text{sucW} \ n)) \rightarrow \\
& \phantom{=} \ (n : \text{NatW}) \rightarrow P \ n \\
\text{indW} \ P \ z \ s \ n & = ?
\end{align*}
\]

A useful deployment of the \( W \)-type is to define the free monad for a container.

\[
\_ \ : \text{Con} \rightarrow \text{Set} \rightarrow \text{Set} \\
C \_ X = W (K_a X + a \ C)
\]

**Exercise 3.6 (free monad)** Construct the components for

\[
\text{freeMonad} : (C : \text{Con}) \rightarrow \text{Monad} (\_ \ C)
\]

\[
\text{freeMonad} \ C = ?
\]

**Exercise 3.7 (free monad closure)** Define an operator

\[
\_ \ : \text{Con} \rightarrow \text{Con}
\]

\[
\_ \ C = ?
\]

and exhibit an isomorphism

\[
C \_ X \cong [ [ C \_ ] a \ X
\]
Exercise 3.8 (general recursion) Define the monadic computation which performs one command-response interaction:

\[
\text{call} : \forall \{C\} \to (s : \text{Sh} C) \to C \to \text{Po} C s
\]

We can model, the general recursive function space as the means to perform finite, on too much detail demand expansion of call trees.

\[
\Pi \bot : (S : \text{Set}) (T : S \to \text{Set}) \to \text{Set}
\]

\[
\Pi_L S T = (s : S) \to (S \triangleleft T) \to T s
\]

Give the ‘gasoline-driven’ interpreter for this function space, delivering a result provided the call tree does not expand more times than a given number.

\[
gas : \forall \{S \cdot T\} \to \mathbb{N} \to \Pi L S T \to (s : S) \to T s + \text{One}
\]

Feel free to implement reduction for the untyped \(\lambda\)-calculus, or some other model of computation, as a recursive function in this way.

Turing completeness To say that Agda fails to be Turing complete is manifest nonsense. It does not stop you writing general recursive programs. It does not stop you feeding them to a client who is willing to risk running them. It does stop you giving a general recursive program a type which claims it is guaranteed to terminate, nor can you persuade Agda to execute such a program unboundedly in the course of checking a type. It is not unusual for typecheckers to refuse to run general recursive type-level programs. So the situation is not that we give up power for totality. Totality buys us a degree of honesty which partial languages just discard.

3.4 Derivatives of Containers

We have

\[
\llbracket S \triangleleft P \rrbracket_a X = \Sigma S \lambda s \to P s \to X
\]

but we could translate the right-hand side into a more mathematical notation and observe that a container is something a bit like a power series:

\[
\llbracket S \triangleleft P \rrbracket_a X = \sum_{s : S} X^{(P s)}
\]

We might imagine computing a formal derivative of such a series, ‘multiplying down by each index, then subtracting one’, but we are not merely counting data—they have individual existences. Let us define a kind of ‘dependent decrement’, subtracting a particular element from a type.

\[
\_ \_ : (X : \text{Set}) (x : X) \to \text{Set}
\]

\[
X - x = \Sigma X \lambda x' \to x' \simeq x \to \text{Zero}
\]

That is, an element of \(X - x\) is some element for \(X\) which is known to be other than \(x\).

We may now define the formal derivative of a container.

\[
\partial : \text{Con} \to \text{Con}
\]

\[
\partial (S \triangleleft P) = \Sigma S \lambda p \to \text{Po} P s - p
\]

The shape of the derivative is the pair of a shape with one position, which we call the ‘hole’, and the positions in the derivative are ‘everywhere but the hole’.
3.5. DENORMALIZED CONTAINERS

Exercise 3.9 (plug) Exhibit a container morphism which witnesses the ability to fill the hole, provided equality on positions is decidable.

\[
\text{plug} : \forall \{ C \} \rightarrow ((s : \text{Sh } C) \ (p \ p' : \text{Po } C \ s) \rightarrow \text{Dec } (p \simeq p')) \rightarrow \\
(\partial C \times \text{a } I) \rightarrow_a C \\
\text{plug } \{ C \} \ poeq? = ?
\]

Exercise 3.10 (laws of calculus) Check that the following laws hold at the level of mutually inverse container morphisms.

\[
\partial (K_a \ A) \equiv K_a \ \text{Zero} \\
\partial I \equiv K_a \ \text{One} \\
\partial (C \vdash_a D) \equiv \partial C \vdash_a \partial D \\
\partial (C \times_a D) \equiv (\partial C \times_a D) \vdash_a (C \times_a \partial D) \\
\partial (C \circ_a D) \equiv (\partial C \circ_a D) \times_a \partial D
\]

What is \( \partial (C \ast) \)?

3.5 Denormalized Containers

These may appear later.
Chapter 4
Indexed Containers (Levitated)

There are lots of ways to present indexed containers, giving ample opportunities for exercises, but I shall use the Hancock presentation, as it has become my preferred version, too.

The idea is to describe functors between indexed families of sets.

```agda
record ⊲ (I J : Set) : Set₁ where
constructor ⊲-ς-ς-
  field
    ShIx : J → Set
    Polx : (j : J) → ShIx j → Set
    riIx : (j : J) (s : ShIx j) (p : Polx j s) → I

open ⊲ public
```

An $I \triangleright J$ describes a $J$-indexed thing with places for $I$-indexed elements. Correspondingly, some $j : J$ tells us which sort of thing we’re making, determining a shape set $Sh j$ and a position family $Po j$, just as with plain containers. The $ri$ function then determines which $I$-index is demanded in each element position.

**Interaction structures** Hancock calls these indexed containers *interaction structures*. Consider $J$ to be the set of possible ‘states of the world’ before an interaction, and $I$ the possible states afterward. The ‘before’ states will determine a choice of commands we can issue, each of which has a set of possible responses which will then determine the state ‘after’. An interaction structure thus describes the predicate transformer which describes the precondition for achieving a postcondition by one step of interaction. We are just using proof-relevant Hoare logic as the type system!

**Exercise 4.1 (functoriality)** We have given the interpretation of indexed containers as operations on indexed families of sets. Equip them with their functorial action for the following notion of morphism

$$\triangleleft_\llcorner : \forall \{ k l \} \{ I : Set k \} \to (I \to Set l) \to (I \to Set l) \to Set (\| l k)$$

$$X \to Y = \forall i \to X i \to Y i$$

$$\text{ixMap} : \forall \{ I J \} \{ C : I \triangleright J \} \{ X Y \} \to (X \to Y) \to \| C \|, X \to \| C \|, Y$$

$$\text{ixMap} f j xs = ?$$
4.1 Petersson-Synek Trees

Kent Petersson and Dan Synek proposed a universal inductive family, amounting to the fixpoint of an indexed container

```haskell
data ITree { J : Set } { ( C : J ⊬ J ) } ( j : J ) : Set where
  ( _ ) : [ C ] ( ITree C ) j → ITree C j
```

The natural numbers are a friendly, if degenerate example.

```haskell
NatC : One ⊬ One
NatC = ( λ _ → Two ) ◁ ( λ _ → Zero ⟨ ? ⟩ One ) $ _
zeroC : ITree NatC ⟨ ⟩
zeroC = ⟨ ⟨ t t , magic ⟩ ⟩
sucC : ITree NatC ⟨ ⟩ → ITree NatC ⟨ ⟩
sucC n = ⟨ ⟨ f f , pure n ⟩ ⟩
```

This is just the indexed version of the \( W \)-type, so the same issue with extensionality arises.

We may also define the node structure for vectors as an instance.

```haskell
VecC : Set → N ⊬ N
VecC X = VS ◁ VP $ Vr where -- depending on the length
  VS : N → Set
  VS zero = One -- nil is unlabelled
  VS ( suc n ) = X -- cons carried an element
  VP : ( n : N ) → VS n → Set
  VP zero _ = Zero -- nil has no children
  VP ( suc n ) _ = One -- cons has one child
  Vr : ( n : N ) ( s : VS n ) ( p : VP n s ) → N
  Vr zero ⟨ ⟩ = ⟨ ⟩ -- nil has no children to index
  Vr ( suc n ) x ⟨ ⟩ = n -- the tail of a cons has the length one less
```

Let us at least confirm that we can rebuild the constructors.

```haskell
vnil : forall { X } → ITree ( VecC X ) zero
vnil = ( ⟨ ⟩ , ( λ ⟨ ⟩ ) )
vcons : forall { X n } → X → ITree ( VecC X ) n → ITree ( VecC X ) ( suc n )
vcons x xs = ( ⟨ ( x , ( λ _ → xs ) ) ⟩ )
```

Why don’t you have a go at rebuilding an inductive family in this manner?

**Exercise 4.2 (simply typed \( \lambda \)-calculus)** Define the simply typed \( \lambda \)-terms as Petersson-Synek trees.

```haskell
STLC : ( C x ⇝ x ) ⊬ ( C x ⇝ x )
STLC = ?
```

4.2 Closure Properties

It is not difficult to show that indexed containers have an identity composition which is compatible up to isomorphism with those of their interpretations.
**Exercise 4.3 (identity and composition)** Construct

\[ \text{Idx} : \forall \{ I \} \to I \triangleright I \]

such that \[ \llbracket \text{Idx} \rrbracket : X i \cong X i \]

Similarly, construct the composition \[ \text{CoIx} : \forall \{ I J K \} \to J \triangleright K \to I \triangleright J \to I \triangleright K \]

such that \[ \llbracket \text{CoIx} \rrbracket : X k \cong \llbracket \{ C \} ; \{ C' \} ; X \rrbracket k \]

It may be useful to consider constructing binary products and coproducts, but let us chase after richer structure, exploiting dependent types to a greater extent.

We may describe a class of indexed functors, as follows.

**data** \( \text{Desc} \{ l \} (I : \text{Set} l) : \text{Set} (\text{lsuc} l) \) where

\[ \begin{align*}
\text{var} &: I \to \text{Desc} I \\
\sigma \pi &: (A : \text{Set} l) (D : A \to \text{Desc} I) \to \text{Desc} I \\
\times_{D_l} &: \text{Desc} I \to \text{Desc} I \to \text{Desc} I \\
\kappa &: \text{Set} l \to \text{Desc} I \\
\text{infixr} 4 \times_{D_l}
\end{align*} \]

which admit a direct interpretation as follows

\[ \begin{align*}
\llbracket \text{var} \rrbracket : \forall \{ l \} \{ I : \text{Set} l \} \to \text{Desc} I \to (I \to \text{Set} l) \to \text{Set} l \\
\llbracket \sigma A D \rrbracket : X = \Sigma A \lambda a \to \llbracket D a \rrbracket X \\
\llbracket \pi A D \rrbracket : X = (a : A) \to \llbracket D a \rrbracket X \\
\llbracket D \times_{D_l} E \rrbracket : X = \llbracket D \rrbracket X \times \llbracket E \rrbracket X \\
\llbracket \kappa A \rrbracket : X = A
\end{align*} \]

A family of such descriptions in \( J \to \text{Desc} I \) thus determines, pointwise, a functor from \( I \to \text{Set} \) to \( J \to \text{Set} \). It is easy to see that every indexed container has a description.

\[ \text{ixConDesc} : \forall \{ I J \} \{ I : \text{Set} l \} \to \text{Desc} I \to (I \to \text{Set} l) \to \text{Set} l \]

\[ \text{ixConDesc} (S \triangleleft P \triangleright r) j = \sigma (S j) \lambda s \to \pi (P j s) \lambda p \to \text{var} (r j s p) \]

Meanwhile, up to isomorphism at least, we can go the other way around.

**Exercise 4.4 (from \( J \to \text{Desc} I \) to \( I \triangleright J \)) Construct functions**

\[ \begin{align*}
\text{DSh} &: \{ I : \text{Set} \} \to \text{Desc} I \to \text{Set} \\
\text{DSh} D &= ? \\
\text{DPo} &: \forall \{ I \} (D : \text{Desc} I) \to \text{DSh} D \to \text{Set} \\
\text{DPo} D s &= ? \\
\text{Dri} &: \forall \{ I \} (D : \text{Desc} I) (s : \text{DSh} D) \to \text{DPo} D s \to I \\
\text{Dri} D s p &= ?
\end{align*} \]

in order to compute the indexed container form of a family of descriptions.

\[ \text{descIxCon} : \forall \{ I J \} \to (J \to \text{Desc} I) \to I \triangleright J \]

\[ \text{descIxCon} F = (\text{DSh} \circ F) \triangleleft (\text{DPo} \circ F) \triangleright (\text{Dri} \circ F) \]

Exhibit the isomorphism

\[ \llbracket \text{descIxCon} F \rrbracket : X j \cong \llbracket F j \rrbracket X \]

We shall find further closure properties of indexed containers later, but let us explore description awhile.
4.3 Describing Datatypes

Descriptions are quite a lot like inductive family declarations. The traditional `Vec` declaration corresponds to

\[
\text{vecD} : \text{Set} \to \text{N} \to \text{Desc} \text{N} \\
\text{vecD} X n = \\
\sigma \text{Two} (\kappa (n \simeq \text{zero})) \\
(\sigma \text{Set} \lambda k \to \kappa X \times_D \text{var} k \times_D \kappa (n \simeq \text{suc} k))
\]

The choice of constructors becomes a $\sigma$-description. Indices specialized in the return types of constructors become explicit equational constraints. However, in defining a family of descriptions, we are free to use the full computational power of the function space, inspecting the index, e.g.

\[
\text{vecD} : \text{Set} \to \text{N} \to \text{Desc} \text{N} \\
\text{vecD} X \text{zero} = \kappa \text{One} \\
\text{vecD} X (\text{suc} n) = \kappa X \times_D \text{var} n
\]

To obtain a datatype from a description, we can turn it into a container and use the Petersson-Synek tree, or we can preserve the first orderness of first order things and use the direct interpretation.

\[
\text{data Data} \{ l \} \{ J : \text{Set} \} (F : J \to \text{Desc} J) (j : J) : \text{Set} \text{l where} \\
\langle_\_\rangle : [F j \_]_D (\text{Data} F) \to \text{Data} F \_j
\]

For example, let us once again construct vectors.

\[
\text{vnil} : \forall \{ X \} \to \text{Data} (\text{vecD} X) \text{zero} \\
\text{vnil} = \langle \_ \_ \_ \rangle \\
\text{vcons} : \forall \{ X n \} \to X \to \text{Data} (\text{vecD} X) n \to \text{Data} (\text{vecD} X) (\text{suc} n) \\
\text{vcons} x \_xs = \langle x, xs \rangle
\]

Exercise 4.5 (something like ‘levitation’)  Construct a family of descriptions which describes a type like $\text{Desc}$. As Agda is not natively cumulative, you will need to shunt types up through the $\text{Set} \_l$ hierarchy by hand, with this gadget:

\[
\text{record} \uparrow \{ l \} (X : \text{Set} \_l) : \text{Set} (\text{lsuc} l) \_l : \text{Set} \_l \text{ where} \\
\text{constructor} \uparrow \\
\text{field} \\
\text{open} \uparrow \text{public}
\]

Now implement

\[
\text{DescD} : \forall \{ l \} (I : \text{Set} \_l) \to \text{One} \{ \text{lsuc} l \} \to \text{Desc} (\text{One} \{ \text{lsuc} l \}) \\
\text{DescD} \{ l \} I \_ = \_?
\]

Check that you can map your described descriptions back to descriptions.

\[
\text{desc} : \forall \{ l \} \{ I : \text{Set} \_l \to \text{Data} (\text{DescD} I) \_ \} \to \text{Desc} I \\
\text{desc} \_D = \_?
\]

We could, if we choose, work entirely with described datatypes. Perhaps, in some future programming language, the external $\text{Desc} \_l$ type will be identified with the internal $\text{Data} (\text{DescD} I) \_\_$ so that $\text{Data}$ is the only datatype.
4.4 Some Useful Predicate Transformers

A container stores a bunch of data. If we have a predicate $P$ on data, it might be useful to formulate the predicates on bunches of data asserting that $P$ holds everywhere or somewhere. But an indexed container is a predicate transformer! We can thus close indexed containers under the formation of ‘everywhere’.

**Everywhere**: $\forall\{I,J\} (C : I \triangleright J) (X : I \to \text{Set}) \to \Sigma I X \triangleright \Sigma J (\llbracket C \rrbracket, X)$

Everywhere $(S \triangleleft P \triangleleft r) X$

$\triangleleft (\lambda \{j, s, k\} \rightarrow P j s)$

$\triangleleft (\lambda \{j, s, k\} \rightarrow P j s)$

The witnesses to the property of the elements of the original container become the elements of the derived container. The trivial predicate holds everywhere.

**allTrivial**: $\forall\{I,J\} (C : I \triangleright J) (X : I \to \text{Set}) jc \to \llbracket \text{Everywhere} C X \rrbracket, (\lambda \_ \to \text{One}) jc$

allTrivial $C X \_ = (\_ , \lambda p \rightarrow \_ )$

If you think of simply typed $\lambda$-calculus contexts as containers of types, then an environment is given by supplying values Everywhere.

Meanwhile, the finger now points at you, pointing a finger at an element.

**Exercise 4.6 (Somewhere)** Construct the transformer which takes $C$ to the container for witnesses that a property holds for some element of a $C$-structure

**Somewhere**: $\forall\{I,J\} (C : I \triangleright J) (X : I \to \text{Set}) \to \Sigma I X \triangleright \Sigma J (\llbracket C \rrbracket, X)$

Somewhere $(S \triangleleft P \triangleleft r) X$

$\triangleleft ?$

$\triangleleft ?$

$\triangleleft ?$

Check that the impossible predicate cannot hold somewhere.

**noMagic**: $\forall\{I,J\} (C : I \triangleright J) (X : I \to \text{Set}) jc \to \llbracket \text{Somewhere} C X \rrbracket, (\lambda \_ \to \text{Zero}) jc \to \text{Zero}$

noMagic $C X \_ (p, m) = ?$

For simply typed $\lambda$-calculus contexts, a variable of type $T$ is just the evidence that a type equal to $T$ is somewhere. Environment lookup is just the obvious property that if $Q$ holds everywhere and $R$ holds somewhere, then their conjunction holds somewhere, too.

**Exercise 4.7 (lookup)** Implement generalized environment lookup.

**lookup**: $\forall\{I,J\} (C : I \triangleright J) (X : I \to \text{Set}) jc \{ Q R \} \to$

$\llbracket \text{Everywhere} C X \rrbracket, Q jc \rightarrow \llbracket \text{Somewhere} C X \rrbracket, R jc \rightarrow$

$\llbracket \text{Somewhere} C X \rrbracket, (\lambda ix \rightarrow Q ix \times R ix) jc$

lookup $C X \_ jq qs r = ?$

A key use of the Everywhere transformer is in the formulation of induction principles. The induction hypotheses amount to asserting that the induction predicate holds at every substructure.
The step method of the above looks a bit like an algebra, modulo plumbing.

Exercise 4.8 (induction as a fold) Petersson-Synek trees come with a `fold’ operator, making \( \texttt{ITree C} \) (weakly) initial for \([\_\_\_]\). We can compute any \( P \) from a \( \texttt{ITree C} \), given a \( C \)-algebra for \( P \).

\[
\text{treeFold} : \forall \{ I \} \ (C : \_ \triangleright I) \ (P : \Sigma I \ (\texttt{ITree C}) \rightarrow \text{Set}) \rightarrow \\
\text{treeFold} C P m i \ (s, k) = m \ (i, s, k) \ (\langle \rangle, \lambda p \rightarrow \text{treeInd} C P m \ (k \ p))
\]

However, \( \text{treeFold} \) does not give us dependent induction on \( \text{ITree C} \). If all you have is a hammer, everything looks like a nail. If we want to compute why some \( P : \Sigma I \ (\texttt{ITree C}) \rightarrow \text{Set} \) always holds, we’ll need an indexed container storing \( P \) in positions corresponding to the children of a given tree. The \( \texttt{Everywhere C} \) construct does most of the work, but you need a little adaptor to unwrap the \( C \) container inside the \( \text{ITree C} \).

\[
\text{Children} : \forall \{ I \} \ (C : \_ \triangleright I) \rightarrow \Sigma I \ (\texttt{ITree C}) \rightarrow \Sigma I \ (\texttt{ITree C})
\]

Children \( C = \text{Colx} \ ? \ (\texttt{Everywhere C} (\texttt{ITree C})) \)

Now, you can extract a general induction principle for \( \texttt{ITree C} \) from \( \text{treeFold} \) (\( \text{Children C} \)), but you will need a little construction. Finish the job.

\[
\text{treeFoldInd} : \forall \{ I \} \ (C : \_ \triangleright I) \ P \rightarrow \\
\text{treeFoldInd} C P m \ (i, t) = \text{treeFold} \ (\text{Children C}) P m \ (\ i, t\ )
\]

Of course, you need to do what is effectively an inductive proof to fill in the hole. Induction really does amount to more than weak initiality. But one last induction will serve for all.

What goes for containers goes for descriptions. We can build all the equipment of this section for \( \texttt{Desc} \) and \( \texttt{Data} \), too.

Exercise 4.9 (\texttt{Everywhere} and \texttt{Somewhere} for \texttt{Desc}) Define suitable description transformers, capturing what it means for a predicate to hold in every or some element position within a given described structure.

\[
\text{EverywhereD SomewhereD} : \{ I : \texttt{Set} \} \ (D : \texttt{Desc I}) \ (X : I \rightarrow \text{Set}) \rightarrow \\
\text{EverywhereD} D X x s = ? \text{ SomewhereD} D X x s = ?
\]

Now construct

\[
\text{datalnd} : \forall \{ I : \texttt{Set} \} \ (F : I \rightarrow \texttt{Desc I}) \ (P : \Sigma I \ (\texttt{Data F}) \rightarrow \text{Set}) \rightarrow \\
\text{datalnd} F P m i d = ?
\]
4.5 Indexed Containers are Closed Under Fixpoints

So far, we have used indexed containers to describe the node structures of recursive data, but we have not considered recursive data structures to be containers themselves. Consider, e.g., the humble vector: might we not consider the vector’s elements to be a kind of contained thing, just as much as its subvectors? We can just throw in an extra kind of element!

\[
\text{vecNodeIx} : (\text{One} + \mathbb{N}) \triangleright \mathbb{N}
\]

\[
\text{vecNodeIx} = \text{descIxCon} \{ J = \mathbb{N} \} \lambda
\]

\[
\begin{align*}
\text{zero} & \to \kappa \text{One} \\
; \text{(suc n)} & \to \text{var} (\text{tt}, \langle \rangle) \times \text{var} (\text{ff}, n)
\end{align*}
\]

That is enough to see vector nodes as containers of elements or subnodes, but it still does not give vectors as containers:

\[
\text{vecIx} : \text{One} \triangleright \mathbb{N}
\]

\[
\text{vecIx} = ?
\]

We should be able to solve this goal by taking \(\text{vecNodeIx}\) and tying a recursive knot at positions labelled \((\text{ff}, n)\), retaining positions labelled \((\text{tt}, \langle \rangle)\). Let us try the general case.

\[
\mu \text{Ix} : \forall \{ I J \} \to (I + J) \triangleright J \to I \triangleright J
\]

\[
\mu \text{Ix} \{ I \} \{ J \} F = (\text{ITree } F' \circ \_ \_ \_ \_ \text{ff}) \triangleleft (P' \circ \_ \_ \_ \_ \text{ff}) s (r' \circ \_ \_ \_ \_ \text{ff}) \text{ where }
\]

The shapes of the recursive structures are themselves trees, with unlabelled leaves at \(I\)-indexed places and \(F\)-nodes in \(J\)-indexed places. We could try to work in \(J \triangleright J\), cutting out the non-recursive positions. However, it is easier to shift to \((I + J) \triangleright (I + J)\), introducing ‘unlabelled leaf’ as the dull node structure whenever an \(I\) shape is requested. We may construct

\[
F : (I + J) \triangleright (I + J)
\]

\[
F = (V (\lambda i \to \text{One}) ?\text{ ShIx } F) \triangleleft (V (\lambda _ _ \to \text{Zero}) ?\text{ PolIx } F) s (V (\lambda t s (\_)) ?\text{ rlixF } F)
\]

and then choose to start with \((\text{ff}, j)\) for the given top level \(j\) index. A position is then a path to a leaf: either we are at a leaf already, or we must descend further.

\[
P : (x : I + J) \to \text{ITree } F x \to \text{Set}
\]

\[
P (\text{tt}, \_ i) = \text{One}
\]

\[
P (\text{ff}, j) \langle \_ s, \_ k \rangle = \sum (\text{PolIx } F j s) \lambda p \to P (\text{rlixF } F j s p) (\_ k p)
\]

Finally, we may follow each path to its indicated leaf and return the index which sent us there.

\[
r : (x : I + J) (t : \text{ITree } F x) \to P x t \to I
\]

\[
r (\text{tt}, i) _ _ _ = i
\]

\[
r (\text{ff}, j) \langle \_ s, \_ k \rangle (p, \_ ps) = r _ _ (k p) ps
\]

Let us check that this recipe cooks the vectors.

\[
\text{vecIx} : \text{One} \triangleright \mathbb{N}
\]

\[
\text{vecIx} = \mu \text{Ix vecNodeIx}
\]
4.6 Adding fixpoints to Desc

We can extend descriptions to include a fixpoint operator:

\[
\text{data } \text{Desc} (I : \text{Set}) : \text{Set} \text{ where }
\]

\[
\var : I \rightarrow \text{Desc } I
\]

\[
\sigma \pi : (A : \text{Set}) \ (D : A \rightarrow \text{Desc } I) \rightarrow \text{Desc } I
\]

\[
\times D : \text{Desc } I \rightarrow \text{Desc } I \rightarrow \text{Desc } I
\]

\[
\kappa : \text{Set} \rightarrow \text{Desc } I
\]

\[
\mu : (J : \text{Set}) \rightarrow (J \rightarrow \text{Desc } (I + J)) \rightarrow J \rightarrow \text{Desc } I
\]

The interpretation must now be defined mutually with the universal inductive type.

\[
\text{mutual }
\]

\[
\llbracket \_ \rrbracket_o : \ \forall \ (I) \rightarrow \text{Desc } I \rightarrow (I \rightarrow \text{Set}) \rightarrow \text{Set}
\]

\[
\llbracket \var \rrbracket_o \ X = X \ i
\]

\[
\llbracket \sigma A D \rrbracket_o \ X = \Sigma A \lambda \ a \rightarrow \llbracket \ D \ a \rrbracket_o \ X
\]

\[
\llbracket \pi A D \rrbracket_o \ X = (a : A) \rightarrow \llbracket \ D \ a \rrbracket_o \ X
\]

\[
\llbracket D \times D \rrbracket_o \ X = \llbracket D \rrbracket_o \ X \times \llbracket D \rrbracket_o \ X
\]

\[
\llbracket \kappa A \rrbracket_o \ X = A
\]

\[
\llbracket \mu \ J \times D \rrbracket_o \ X = \text{Data } F X j
\]

\[
\text{data } \text{Data } (I \ J) \ (F : J \rightarrow \text{Desc } (I + J)) \ (X : I \rightarrow \text{Set}) \ (j : J) : \text{Set where }
\]

\[
\llbracket \_ \rrbracket : \ \llbracket F j \rrbracket_o \ (V X (?)) \text{Data } F X j \rightarrow \text{Data } F X j
\]

Indeed, Desc Zero now does quite a good job of reflecting Set, except that the domains of \(\sigma\) and \(\pi\) are not concretely represented, an issue we shall attend to in the next chapter.

Exercise 4.10 (induction) State and prove the induction principle for Desc. (This is not an easy exercise.)

4.7 Jacobians

I am always amused when computing people complain about being made to learn mathematics choose calculus as their favourite example of something that is of no use to them. I, for one, am profoundly grateful to have learned vector calculus: it is exactly what you need to develop notions of ‘context’ for dependent datatypes.

An indexed container in \(I \triangleright J\) explains \(J\) sorts of structure in terms of \(I\) sorts of elements, and as such, we acquire a Jacobian matrix of partial derivatives, in \(I \triangleright (J \times I)\). A \((j, i)\) derivative is a structure of index \(j\) with a hole of index \(i\). Here’s how we build it.
\[ J : \text{forall} \{ I J \} \rightarrow I \triangleright J \rightarrow I \triangleright (J \times I) \]
\[ J (S \in P \; \triangleright \; r) \]
\[ = (\lambda \{ (j, i) \rightarrow \Sigma (S j) \lambda s \rightarrow r j s^{-1} i \}) \]
\[ \triangleright (\lambda \{ (j, (r j s p)) (s, \text{from} \; p) \rightarrow P j s - p \}) \]
\[ \triangleright (\lambda \{ (j, (r j s p)) (s, \text{from} \; p) (p', \ldots) \rightarrow r j s p' \}) \]

The shape of an \((i, j)\)-derivative must select a \(j\)-indexed shape for the structure, together with a position (the hole) whose index is \(i\). As in the simple case, a position in the derivative is any position other than the hole, and its index is calculated as before.

**Exercise 4.11 (plugging)** Check that a decidable equality for positions is enough to define the ‘plugging in’ function.

\[
\text{plug}_I : \text{forall} \{ I J \} (C : I \triangleright J) \rightarrow (\lambda (j : J) (s : \text{Shlx} \; C 
\; j) (p p' : \text{Polx} \; C \; j s) \rightarrow \text{Dec} \; (p \simeq p')) \rightarrow
\text{forall} \{ i j X \} \rightarrow [\; J \; C \; ] \; X \; (j, i) \rightarrow X i \rightarrow [\; C \; ] \; X \; j
\]

**Exercise 4.12 (the Zipper)** For a given \(C : I \triangleright I\), construct the indexed container \(\text{Zipper} \; C : (I \times I) \triangleright (I \times I)\) such that \(\text{ITree} \; (\text{Zipper} \; C) \; (i r, i h)\) represents a one \(i h\)-hole context in a \(\text{ITree} \; C \; i r\), represented as a sequence of hole-to-root layers.

\[ \text{Zipper} : \text{forall} \{ I \} \rightarrow I \triangleright I \rightarrow (I \times I) \triangleright (I \times I) \]

Check that you can zipper all the way out to the root.

\[ \text{zipOut} : \text{forall} \{ I \} \; (C : I \triangleright I) \{ i r \; i h \} \rightarrow
\text{forall} \{ i \} \; (s : \text{Shlx} \; C \; i) (p p' : \text{Polx} \; C \; i s) \rightarrow \text{Dec} \; (p \simeq p')) \rightarrow
\text{ITree} \; (\text{Zipper} \; C) \; (i r, i h) \rightarrow \text{ITree} \; C \; i h \rightarrow \text{ITree} \; C \; i r \]

**Exercise 4.13 (differentiating Desc)** The notion corresponding to \(J\) for descriptions is \(\nabla\), computing a ‘vector’ of partial derivatives. Define it symbolically.

\[ \nabla : \{ I : \text{Set} \} \rightarrow \text{Desc} \; I \rightarrow I \rightarrow \text{Desc} \; I \]

\[ \nabla D \; h = ? \]

Hence construct suitable zippering equipment for \(\text{Data}\).

It is amusing to note that the mathematical notion of divergence, \(\nabla \cdot D\), corresponds exactly to the choice of decompositions of a \(D\)-structure into any element-in-context:

\[ \sigma \; I \; \lambda i \rightarrow \nabla D \; i \; \times_D \; \text{var} \; i \]

I have not yet found a meaning for \(\text{curl}, \nabla \times \; D\), nor am I expecting Maxwell’s equations to pop up anytime soon. But I live in hope for light.

### 4.8 Apocrypha

#### 4.8.1 Roman Containers

A Roman container is given as follows
record Roman (I J : Set) : Set₁ where
  constructor SPqr
  field
    S : Set
    P : S → Set
    q : S → J
    r : (s : S) → P s → I
  Plain : Con
  Plain = S ≪ P
  \[ R_R \] : (I → Set) → (J → Set)
  \[ R_R \] X j = Σ (Σ S λ s → q s ≃ j) (\forall λ s → (p : P s) → X (r s p))
  Plain = RomanPlain
  \[ R_R \] = Roman R}_R

It’s just a plain container, decorated by functions which attach input indices to
positions and an output index to the shape. We can turn Roman containers into
indexed containers whose meanings match on the nose.

\[
\text{FromRoman} : \forall \{ I J \} → \text{Roman} I J → I \triangleright J
\]
\[
\text{FromRoman} (\text{SPqr } S P q r) = (λ j → Σ S λ s → q s ≃ j) \triangleright ((λ j → P \circ \text{fst}) \triangleright (λ f → r \circ \text{fst}))
\]
\[
\text{onTheNose} : \forall \{ I J \} (C : \text{Roman} I J) → \text{ToRoman} C \triangleright R \text{R}_X j
\]
\[
\text{onTheNose} C = \text{refl}
\]

Sadly, the other direction is a little more involved.

**Exercise 4.14 (ToRoman)** Show how to construct the Roman container isomorphic to a
given indexed container and exhibit the isomorphism.

\[
\text{ToRoman} : \forall \{ I J \} → I \triangleright J → \text{Roman} I J
\]
\[
\text{ToRoman} \{ I \} \{ J \} (S ≪ P \triangleright R) = ?
\]
\[
\text{toRoman} : \forall \{ I J \} (C : I \triangleright J) → \text{forall} \{ X j \} → \text{ToRoman} C \triangleright R X j
\]
\[
\text{toRoman} C xs = ?
\]
\[
\text{fromRoman} : \forall \{ I J \} (C : I \triangleright J) → \text{forall} \{ X j \} → \text{ToRoman} C \triangleright R X j → \text{ToRoman} C \triangleright R X j
\]
\[
\text{fromRoman} C xs = ?
\]
\[
\text{toAndFromRoman} : \forall \{ I J \} \{ C : I \triangleright J \} \{ X j \}
\]
\[
→ (\text{forall} xs → \text{toRoman} C \{ X \} \{ j \} (\text{fromRoman} C \{ X \} \{ j \} xs) \simeq xs)
\]
\[
× (\text{forall} xs → \text{fromRoman} C \{ X \} \{ j \} (\text{toRoman} C \{ X \} \{ j \} xs) \simeq xs)
\]
\[
\text{toAndFromRoman} C = ?
\]

The general purpose tree type for Roman containers looks a lot like the
inductive families you find in Agda or the GADTs of Haskell.

\[
\text{data RomanData} \{ I \} \{ C : \text{Roman} I I \} : I → Set where
\]
\[
\_ : \_ : (s : \text{Roman}.S C) →
\]
\[
(\text{p : Roman}.P C s) → \text{RomanData} C \{ \text{Roman}.r C s p \}) → \text{RomanData} C \{ \text{Roman}.q C s \}
\]
I could have just taken the fixpoint of the interpretation, but I wanted to emphasize that the role of \texttt{Roman.q} is to specialize the return type of the constructor, creating the constraint which shows up as an explicit equation in the interpretation. The reason Roman containers are so called is that they invoke equality and its mysterious capacity for transubstantiation.

The \texttt{RomanData} type looks a lot like a \texttt{W}-type, albeit festooned with equations. Let us show that it is exactly that.

\textbf{Exercise 4.15 (Roman containers are \textit{W}-types)} Construct a function which takes plain \textit{W}-type data for a Roman container and marks up each node with the index required of it, using \texttt{Roman.r}.

\begin{align*}
\texttt{ideology} & : \texttt{forall } \{ I \} \ (C : \texttt{Roman }I \ I) \rightarrow \\
& \quad I \rightarrow \texttt{W} \ (\texttt{Plain }C) \rightarrow \texttt{W} \ (\texttt{Plain }C \times \triangleq \texttt{K}_a \ I) \\
\texttt{ideology } C \ i \ t & = \ ?
\end{align*}

Construct a function which takes plain \textit{W}-type data for a Roman container and marks up each node with the index delivered by it, using \texttt{Roman.q}.

\begin{align*}
\texttt{phenomenology} & : \texttt{forall } \{ I \} \ (C : \texttt{Roman }I \ I) \rightarrow \\
& \quad \texttt{W} \ (\texttt{Plain }C) \rightarrow \texttt{W} \ (\texttt{Plain }C \times \triangleq \texttt{K}_a \ I) \\
\texttt{phenomenology } C \ t & = \ ?
\end{align*}

Take the \textit{W}-type interpretation of a Roman container to be the plain data for which the required indices are delivered.

\begin{align*}
\texttt{RomanW} & : \texttt{forall } \{ I \} \rightarrow \texttt{Roman }I \ I \rightarrow I \rightarrow \texttt{Set} \\
\texttt{RomanW } C \ i & = \Sigma \ (\texttt{W} \ (\texttt{Plain }C)) \ \lambda \ t \rightarrow \texttt{phenomenology } C \ t \simeq \texttt{ideology } C \ i \ t
\end{align*}

Now, check that you can extract \texttt{RomanData} from \texttt{RomanW}.

\begin{align*}
\texttt{fromRomanW} & : \texttt{forall } \{ I \} \ (C : \texttt{Roman }I \ I) \ \{ i \} \rightarrow \texttt{RomanW } C \ i \rightarrow \texttt{RomanData } C \ i \\
\texttt{fromRomanW } C \ (t, \texttt{good}) & = \ ?
\end{align*}

To go the other way, it is easy to construct the plain tree, but to prove the constraint, you will need to establish equality of functions. Using

\begin{align*}
\texttt{postulate} \\
\texttt{extensionality} & : \texttt{forall } \{ S : \texttt{Set} \} \ {\{ T : S \rightarrow \texttt{Set} \}} \ (f \ g : (s : S) \rightarrow T \ s) \rightarrow \\
& \quad ((s : S) \rightarrow f \ s \simeq g \ s) \rightarrow f \simeq g
\end{align*}

\texttt{construct}

\begin{align*}
\texttt{toRomanW} & : \texttt{forall } \{ I \} \ (C : \texttt{Roman }I \ I) \ \{ i \} \rightarrow \texttt{RomanData } C \ i \rightarrow \texttt{RomanW } C \ i \\
\texttt{toRomanW } C \ t & = \ ?
\end{align*}

\subsection{4.8.2 Reflexive-Transitive closure}

This does not really belong here, but it is quite fun, and something to do with indexed somethings. Consider the reflexive transitive closure of a relation, also known as the ‘paths in a graph’.

\begin{verbatim}
\texttt{data}_"=" \{ I : \texttt{Set} \} \ (R : I \times I \rightarrow \texttt{Set}) : I \times I \rightarrow \texttt{Set} \ where \\
\langle \rangle & : \{ i : I \} \rightarrow (R"=") \ {i, i} \\
\_"=" & : \{ i \ j \ k : I \} \rightarrow R \ {i,j} \rightarrow (R"=") \ {j,k} \rightarrow (R"=") \ {i,k}
\end{verbatim}

\texttt{infix } 1 _"="
You can construct the natural numbers as an instance.

\[
\text{NAT} : \text{Set} = (\text{Loop \(\ast\ast\)}) \quad \text{where} \quad \text{Loop} : \text{One} \times \text{One} \rightarrow \text{Set}; \text{Loop} \_ = \text{One}
\]

**Exercise 4.16 (further constructions with \(\ast\ast\))** Using no recursive types other than \(\ast\ast\), construct the following

- ordinary lists
- the \(\geq\) relation
- lists of numbers in decreasing order
- vectors
- finite sets
- a set of size \(n!\) for a given \(n\)
- ‘everywhere’ and ‘somewhere’ for edges in paths

**Exercise 4.17 (monadic operations)** Implement

\[
\begin{align*}
\text{one}^{\ast\ast} & : \text{forall } \{I\} \{R : I \times I \rightarrow \text{Set}\} \rightarrow R \rightarrow (R^{\ast\ast}) \\
\text{one}^{\ast\ast} \_ & = ? \\
\text{join}^{\ast\ast} & : \text{forall } \{I\} \{R : I \times I \rightarrow \text{Set}\} \rightarrow ((R^{\ast\ast})^{\ast\ast}) \rightarrow (R^{\ast\ast}) \\
\text{join}^{\ast\ast} \_ \_ & = ?
\end{align*}
\]

such that the monad laws hold.

### 4.8.3 Pow and Fam

We have two ways to formulate a notion of ‘subset’ in type theory. We can define a subset of \(X\) as a predicate in

\[
X \rightarrow \text{Set}
\]

giving a proof-relevant notion of evidence that a given \(X : X\) belongs, or we can pick out some elements of \(X\) as the image of a function

\[
\Sigma \text{Set} \lambda I \rightarrow I \rightarrow X
\]

so we have a family of \(X\)’s indexed by some set.

Are these notions the same? That turns out to be a subtle question. A lot turns on the size of \(X\), so we had best be formal about it. In general, \(X\) is large.

\[
\begin{align*}
\text{Pow} & : \text{Set}_1 \rightarrow \text{Set}_1 \\
\text{Pow} \_ X & = X \rightarrow \text{Set} \\
\text{Fam} & : \text{Set}_1 \rightarrow \text{Set}_1 \\
\text{Fam} \_ X & = \Sigma \text{Set} \lambda I \rightarrow I \rightarrow X
\end{align*}
\]

**Exercise 4.18 (small Pow and Fam)** Show that, given a suitable notion of propositional equality, \(\text{Pow} \circ \uplus\) and \(\text{Fam} \circ \uplus\) capture essentially the same notion of subset.

\[
\begin{align*}
p2f & : (\text{Pow} \circ \uplus) \rightarrow (\text{Fam} \circ \uplus) \\
p2f \_ X \_ P & = ? \\
f2p & : (\text{Fam} \circ \uplus) \rightarrow (\text{Pow} \circ \uplus) \\
f2p \_ X \_ F & = ?
\end{align*}
\]
Exercise 4.19 (functoriality of Pow and Fam) Equip Powell with a contravariant functorial action and Fam with a covariant functorial action.

\[ \forall I J \rightarrow (J \rightarrow ) \rightarrow \text{Pow} I \rightarrow \text{Pow} J \]
\[ \forall I J \rightarrow (I \rightarrow J) \rightarrow \text{Fam} I \rightarrow \text{Fam} J \]

Fam \( \text{Set} \) is Martin-Löf’s notion of a universe, naming a bunch of sets by the elements of some indexing set. Meanwhile, the ‘representation type’ method of describing types concretely in Haskell is just using \( \text{Pow} \text{Set} \) in place of \( \text{Fam} \text{Set} \). It is good to get used to recognizing when concepts are related just by exchanging Fam and Pow.

Modulo currying and \( \lambda \)-lifting of parameters, the distinction between Roman \( I J \) and our Hancock-style \( I \triangleq J \) is just that the former represents indexed shapes by a Fam (so Roman.q reads off the shape) whilst the latter uses a Pow (so the shapes pertain to a given index). Both use Fams for positions.

\begin{align*}
\text{ROMAN} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}_1 \\
\text{ROMAN} I J & = \Sigma (\text{Fam} (\uparrow J)) \lambda ((S, q) \rightarrow S \rightarrow \text{Fam} (\uparrow I)) \\
\text{HANCOCK} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}_1 \\
\text{HANCOCK} I J & = \Sigma (\text{Pow} (\uparrow J)) \lambda S \rightarrow \Sigma J (S \circ \uparrow) \rightarrow \text{Fam} (\uparrow I)
\end{align*}

A ‘Nottingham’ indexed container switches the positions to a Pow (see Altenkirch and Morris).

\begin{align*}
\text{NOTTINGHAM} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}_1 \\
\text{NOTTINGHAM} I J & = \Sigma (\text{Pow} (\uparrow J)) \lambda S \rightarrow \Sigma J (S \circ \uparrow) \rightarrow \text{Pow} (\uparrow I)
\end{align*}

which amounts to a presentation of shapes and positions as predicates:

\begin{align*}
\text{NSh} : J \rightarrow \text{Set} \\
\text{NPo} : (j : J) \rightarrow \text{NSh} j \rightarrow I \rightarrow \text{Set}
\end{align*}

For Hancock and Nottingham, we can abstract the whole construction over \( J \), obtaining:

\begin{align*}
\text{HANCOCK} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}_1 \\
\text{HANCOCK} I J & = J \rightarrow \text{Fam} (\text{Fam} (\uparrow I)) \\
\text{NOTTINGHAM} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}_1 \\
\text{NOTTINGHAM} I J & = J \rightarrow \text{Fam} (\text{Pow} (\uparrow I))
\end{align*}

Exercise 4.20 (HANCOCK to ROMAN) We have, modulo plumbing,

\begin{align*}
\text{HANCOCK} I J & = J \rightarrow \text{Fam} (\text{Fam} (\uparrow I)) \\
\text{ROMAN} I J & = \text{Fam} (\uparrow J \times \text{Fam} (\uparrow I))
\end{align*}

Using Fam-Pow flips and currying, find a path from one to the other. However, see below...

But just when we’re getting casual about Fam-Pow flipping, think about what happens when the argument is a large.
Exercise 4.21 (fool’s errand) Construct the large version of the Fam-Pow exchange

\[
p2f : \text{Pow} \to \text{Fam} \\
p2f \ X \ P = ? \\
f2p : \text{Fam} \to \text{Pow} \\
f2p \ X \ F = ?
\]

In our study of datatypes so far, we have been constructing inductively defined inhabitants of \(\text{Pow}(\uparrow \ I)\). Let us now perform our own flip and consider inductive definition in \(\text{Fam} \ I\). What should we expect? Nothing much different for small \(I\), of course. But for a large \(I\), all Heaven breaks loose.
Chapter 5

Induction-Recursion

Recall that \( \text{Fin} \ n \) is an enumeration of \( n \) elements. We might consider how to take these enumerations as the atomic components of a dependent type system, closed under \( \Sigma \)- and \( \Pi \)-types. Finite sums and products of finite things are finite, so we can compute their sizes.

\[
\begin{align*}
\text{sum prod} : (n : \mathbb{N}) \rightarrow (\text{Fin} \ n \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \\
\text{sum zero} &= 0 \\
\text{sum (suc n) f} &= f \text{ zero} + \text{sum} n (f \circ \text{suc}) \\
\text{prod zero} &= 1 \\
\text{prod (suc n) f} &= f \text{ zero} \times \text{sum} n (f \circ \text{suc})
\end{align*}
\]

But can we write down a precise datatype of the type expressions in our finitary system?

\[
\begin{align*}
data \text{FTy} : \text{Set} \ &\text{where} \\
\text{fin} & : \mathbb{N} \rightarrow \text{FTy} \\
\sigma \pi & : (S : \text{FTy}) (T : \text{Fin} \ ? \rightarrow \text{FTy}) \rightarrow \text{FTy}
\end{align*}
\]

I was not quite able to finish the definition, because I could not give the domain of \( T \). Intuitively, when we take sums or products over a domain, we should have one summand or factor for each element of that domain. But we have only \( S \), the expressions which stands for the domain. We know that it is bound to be finite, so I have filled in \( \text{Fin} \ ? \), but to make further progress, we need to know the size of \( S \). Intuitively, it is easy to compute the size of an \( \text{FTy} \): the base case is direct; the structural cases are captured by \text{sum} and \text{prod}. The trouble is that we cannot wait until after declaring \( \text{FTy} \) to define the size, because we need size information, right there at that \(? \). What can we do?

One thing that Agda lets us do is just the thing we need. We can define \( \text{FTy} \) and its size function, \#, simultaneously.

\[
\begin{align*}
data \text{FTy} & : \text{Set} \ &\text{where} \\
\text{fin} & : \mathbb{N} \rightarrow \text{FTy} \\
\sigma \pi & : (S : \text{FTy}) (T : \text{Fin} \ (# S) \rightarrow \text{FTy}) \rightarrow \text{FTy}
\end{align*}
\]

\[
\begin{align*}
\# & : \text{FTy} \rightarrow \mathbb{N} \\
\# (\text{fin} \ n) &= n \\
\# (\sigma S T) &= \text{sum} (# S) \lambda s \rightarrow \# (T s) \\
\# (\pi S T) &= \text{prod} (# S) \lambda s \rightarrow \# (T s)
\end{align*}
\]

For example, if we define the forgetful map from \( \text{Fin} \) back to \( \mathbb{N} \),
We have just seen our first example of induction-recursion. Where an inductive definition tells us how to perform construction of data incrementally, induction-recursion tells us how to perform construction-with-interpretation incrementally.

Together, $(\mathit{FTy}, \#) : \mathsf{Fam} \mathbb{N}$, with the interpretation just telling us sizes, so that $\mathsf{Fin} \circ \#$ gives an unstructured representation of a given $\mathit{FTy}$ type. If we wanted a structured representation, we could just as well have interpreted $\mathit{FTy}$ in $\mathsf{Set}$.

Now, what has happened? We have $(\mathit{FTy}, \mathit{FEI}) : \mathsf{Fam} \mathsf{Set}$, picking out a subset of $\mathsf{Set}$ by choosing names for them in $\mathit{FTy}$. But $\mathit{FTy}$ is small enough to be a $\mathsf{Set}$ itself! IR is the Incredible Ray that shrinks large sets to small encodings of subsets of them.

Here is a standard example of induction recursion for you to try.

**Exercise 5.1 (FreshList)** By means of a suitable choice of recursive interpretation, fill the $\,?$ with a condition which ensures that FreshLists have distinct elements. Try to make sure that, for any concrete FreshList, $\mathit{ok}$ can be inferred trivially.

```haskell
module FRESHLIST (X : Set) (Xeq? : (x x' : X) → Dec (x ≃ x')) where
  mutual
data FreshList : Set where
    [] : FreshList
    _-- : (x : X) (xs : FreshList) {ok : ?} → FreshList
```

5.1 Records

Randy Pollack identified the task of modelling record types as a key early use of induction-recursion, motivated to organise libraries for mathematical structure.

It doesn’t take IR to have a go at modelling records, just something a bit like $\mathsf{Desc}$, but just describing the right-nested $\Sigma$-types.

```haskell
data RecR : Set₁ where
  ⟨⟩ : RecR
  _-_ : (A : Set) (R : A → RecR) → RecR

[[]]RR : RecR → Set
[[]]RR = One
[⟨A, R⟩]RR = Σ A λ a → [] R a]RR
```
5.1. RECORDS

That gives us a very flexible, variant notion of record, where the values of earlier fields can determine the entire structure of the rest of the record. Sometimes, however, it may be too flexible: you cannot tell from a \texttt{RecR} description how many fields a record has—indeed, this quantity may vary from record to record. You can, of course, count the fields in an actual record, then define projection. You do it.

**Exercise 5.2 (projection from RecR)** Show how to compute the size of a record, then define the projections, first of types, then of values.

\[
\begin{align*}
\text{sizeRR} & : (R : \text{RecR}) \rightarrow \llbracket R \rrbracket_{RR} \rightarrow \mathbb{N} \\
n & \text{sizeRR} R r = ? \\
\text{TyRR} & : (R : \text{RecR}) (r : \llbracket R \rrbracket_{RR}) \rightarrow \text{Fin} (\text{sizeRR} R r) \rightarrow \text{Set} \\
n & \text{TyRR} R r i = ? \\
\text{vaRR} & : (R : \text{RecR}) (r : \llbracket R \rrbracket_{RR}) (i : \text{Fin} (\text{sizeRR} R r)) \rightarrow \text{TyRR} R r i \\
n & \text{vaRR} R r i = ?
\end{align*}
\]

Of course, we could enforce uniformity of length by indexing. But a bigger problem with \texttt{RecR} is that, being right-nested, our access to it is left-anchored. Extending a record with more fields whose types depend on existing fields (e.g., adding laws to a record of operations) is a difficult right-end access, as is suffix-truncation.

Sometimes we want to know that we are writing down a signature with a fixed set of fields, and we want easy extensibility at the dependent right end. That means left-nested record types (also known as contexts). And that’s where we need IR.

**mutual**

\[
\begin{align*}
\text{data RecL} & : \text{Set}_1 \text{ where} \\
n & \text{E} : \text{RecL} \\
n & \text{E} : (R : \text{RecL}) (A : \llbracket R \rrbracket_{RL} \rightarrow \text{Set}) \rightarrow \text{RecL} \\
n & \llbracket \text{E} \rrbracket_{RL} : \text{RecL} \rightarrow \text{Set} \\
n & \llbracket \text{E} \rrbracket_{RL} = \text{One} \\
n & \llbracket R, A \rrbracket_{RL} = \Sigma \llbracket R \rrbracket_{RL} A
\end{align*}
\]

**Exercise 5.3 (projection from RecL)** Show how to compute the size of a \texttt{RecL} without knowing the individual record. Show how to interpret a projection as a function from a record, first for types, then values.

\[
\begin{align*}
\text{sizeRL} & : \text{RecL} \rightarrow \mathbb{N} \\
n & \text{sizeRL} R = ? \\
\text{TyRL} & : (R : \text{RecL}) \rightarrow \text{Fin} (\text{sizeRL} R) \rightarrow \llbracket R \rrbracket_{RL} \rightarrow \text{Set} \\
n & \text{TyRL} R i = ? \\
\text{vaRL} & : (R : \text{RecL}) (i : \text{Fin} (\text{sizeRL} R)) (r : \llbracket R \rrbracket_{RL}) \rightarrow \text{TyRL} R i r \\
n & \text{vaRL} R i = ?
\end{align*}
\]

**Exercise 5.4 (truncation)** Show how to truncate a record signature from a given field and compute the corresponding projection on structures.

\[
\begin{align*}
\text{TruncRL} & : (R : \text{RecL}) \rightarrow \text{Fin} (\text{sizeRL} R) \rightarrow \text{RecL} \\
n & \text{TruncRL} R i = ? \\
\text{truncRL} & : (R : \text{RecL}) (i : \text{Fin} (\text{sizeRL} R)) \rightarrow \llbracket R \rrbracket_{RL} \rightarrow \llbracket \text{TruncRL} R i \rrbracket_{RL} \\
n & \text{truncRL} R i = ?
\end{align*}
\]
5.1.1 Manifest Fields

Pollack extends his notion of record with manifest fields, i.e., fields whose values are computed from earlier fields. It is rather like allowing definitions in contexts.

First, I define the type of data with a manifest value (sometimes also known as singletons). I deliberately keep the index right of the colon to force Agda to store the singleton value in the data structure.

\[
\text{data Manifest } \{ A : \text{Set}\} : A \rightarrow \text{Set} \quad \text{where}
\]
\[
\langle n \rangle : (a : A) \rightarrow \text{Manifest } a
\]

Now, I extend the notion of record signature with a constructor for manifest fields. I could have chosen simply to omit these fields from the record structure, but instead I make them Manifest so that projection need not involve recomputation. I also index by size, to save on measuring.

\[
\text{mutual data RecM } : \mathbb{N} \rightarrow \text{Set}_1 \quad \text{where}
\]
\[
\begin{align*}
\mathcal{E} &: \text{RecM } \text{zero} \\
\langle \_ \rangle &: \{ n : \mathbb{N}\} \, (R : \text{RecM } n) \, (A : \llbracket R \rrbracket_{\text{RM}} \rightarrow \text{Set}) \rightarrow \text{RecM } (\text{suc } n) \\
\langle \_ \rangle \exists &: \{ n : \mathbb{N}\} \, (R : \text{RecM } n) \, (A : \llbracket R \rrbracket_{\text{RM}} \rightarrow \text{Set}) \\
&\quad \quad (a : (r : \llbracket R \rrbracket_{\text{RM}}) \rightarrow A \, r) \rightarrow \text{RecM } (\text{suc } n)
\end{align*}
\]

Exercise 5.5 (projection from RecM) Implement projection for RecM.

\[
\text{TyRM } : \{ n : \mathbb{N}\} \, (R : \text{RecM } n) \rightarrow \text{Fin } n \rightarrow \llbracket R \rrbracket_{\text{RM}} \rightarrow \text{Set}
\]
\[
\text{TyRM } R \, i \quad = \ ?
\]
\[
\text{vaRM } : \{ n : \mathbb{N}\} \, (R : \text{RecM } n) \, (i : \text{Fin } n) \, (r : \llbracket R \rrbracket_{\text{RM}}) \rightarrow \text{TyRM } R \, i \, r
\]
\[
\text{vaRM } R \, i \quad = \ ?
\]

Be careful not to recompute the value of a manifest field.

Exercise 5.6 (record extension) When building libraries of structures, we are often concerned with the idea of one record signature being the extension of another. The following

\[
\text{mutual data REx } : \{ n \, m : \mathbb{N}\} \rightarrow \text{RecM } n \rightarrow \text{RecM } m \rightarrow \text{Set}_1 \quad \text{where}
\]
\[
\begin{align*}
\mathcal{E} &: \text{REx } \mathcal{E} \, \mathcal{E} \\
\text{rfog} &: \forall \{ n \, m\} \, \{ R : \text{RecM } n\} \, \{ R' : \text{RecM } m\} \, \{ X : \text{REx } R \, R'\} \rightarrow \\
&\quad \llbracket R' \rrbracket_{\text{RM}} \rightarrow \llbracket R \rrbracket_{\text{RM}}
\end{align*}
\]

describes evidence $\text{REx } R \, R'$ that $R'$ is an extension of $R$, interpreted by $\text{rfog}$ as a map from $\llbracket R' \rrbracket_{\text{RM}}$ back to $\llbracket R \rrbracket_{\text{RM}}$. Unfortunately, it captures only the fact that the empty record extends itself. Extend $\text{REx}$ to allow retention of every field, insertion of new fields, and conversion of abstract to manifest fields. (For my solution, I attempted to show that I could always construct the identity extension. Thus far, I have been defeated by equational reasoning in an overly intensional setting.)
5.2 A Universe

We’ve already seen that we can use IR to build a little internal universe. I have a favourite such universe, with a scattering of base types, dependent pairs and functions, and Petersson-Synek trees. That’s quite a lot of Set, right there!

mutual
data TU : Set where
  Zero’ One’ Two’ : TU
  Σ’ Π’ : (S : TU) (T : [S]TU → TU) → TU
  Tree’ : (I : TU)
    (F : [I]TU → Σ TU λ S →
     [S]TU → Σ TU λ P →
     [P]TU → [I]TU)
    (i : [I]TU) → TU

The TU universe is not closed under a principle of inductive-recursive definition, so the shrinking ray has not shrunk the shrinking raygun.

Exercise 5.7 (TU examples) Check that you can encode natural numbers, lists and vectors in TU. For an encore, try the simply typed λ-calculus.

5.3 Universe Upon Universe

Not only can you build one small universe inside Set using induction-recursion, you can build a predicative hierarchy of them. The key is to define the ‘next universe’ operator, and then iterate it. The following construction takes a universe X and builds another, NU X, on top.

mutual
data NU (X : Fam Set) : Set where
  U’ : NU X
  El’ : fst X → NU X
  Nat’ : NU X
  Σ’ Π’ : (S : NU X) (T : [S]NU → NU X) → NU X

As you can see, NU X has names El’ T for the types in X and a name U’ for X itself. Now we can jack up universes as far as we like.
CHAPTER 5. INDUCTION-RECURSION

\text{EMPTY} : \text{Fam} \text{Set} \\
\text{EMPTY} \ = \ \text{Zero}, \lambda () \\
\text{LEVEL} : \mathbb{N} \rightarrow \text{Fam} \text{Set} \\
\text{LEVEL} \ zero \ = \ \text{NU} \ \text{EMPTY} \ \llbracket \cdot \rrbracket_{\text{NU}} \\
\text{LEVEL} \ (\text{suc} \ n) \ = \ \text{NU} \ (\text{LEVEL} \ n), \llbracket \cdot \rrbracket_{\text{NU}}

This hierarchy is explicitly cumulative: \text{El}^t \ embeds \ types \ upward \ without \ changing \ their \ meaning. \ One \ consequence \ is \ that \ we \ have \ a \ redundancy \ of \ representation:

\textbf{Exercise 5.8} \ (\mathbb{N} \rightarrow \mathbb{N}) \ Find \ five \ names \ for \ \mathbb{N} \rightarrow \mathbb{N} \ in \ \text{fst} \ (\text{LEVEL} \ 1).

5.3.1 A Redundancy-Free Hierarchy

We can try to eliminate the redundancy by including only the names for lower universes at each level: we do not need to embed \(\mathbb{N} \rightarrow \mathbb{N}\) from \text{LEVEL} \ 0, because \text{LEVEL} \ 1 has a perfectly good version. This time, we parametrize the universe by a de Bruijn indexed collection of the previous universes.

\text{mutual}

\text{data} \ \text{HU} \ \{n\} \ (U : \text{Fin} \ n \rightarrow \text{Set}) : \text{Set} \ \text{where}
\begin{align*}
\text{U}' & : \text{Fin} \ n \rightarrow \text{HU} \ U \\
\text{Nat}' & : \text{HU} \ U \\
\Pi' & : (S : \text{HU} \ U) \ (T : \llbracket S \rrbracket_{\text{HU}} \rightarrow \text{HU} \ U) \rightarrow \text{HU} \ U \\
\llbracket \cdot \rrbracket_{\text{HU}} & : \text{forall} \ \{n\} \ \{U : \text{Fin} \ n \rightarrow \text{Set}\} \rightarrow \text{HU} \ U \rightarrow \text{Set} \\
\llbracket \cdot \rrbracket_{\text{HU}} \{U = U'\} \ (U' \ i) & = U \ i \\
\llbracket \text{Nat}' \rrbracket_{\text{HU}} & = \mathbb{N} \\
\llbracket \Pi' \ S \ T \rrbracket_{\text{HU}} & = (s : \llbracket S \rrbracket_{\text{HU}}) \rightarrow \llbracket T \ s \rrbracket_{\text{HU}}
\end{align*}

To finish the job, we must build the collections of levels to hand to \text{HU}. At each step, level zero is the new top level, built with a fresh appeal to \text{HU}, but lower levels can be projected from the previous collection.

\text{HPREDs} : (n : \mathbb{N}) \rightarrow \text{Fin} \ n \rightarrow \text{Set} \\
\text{HPREDs zero} \ () \\
\text{HPREDs} \ (\text{suc} \ n) \ \text{zero} \ = \ \text{HU} \ (\text{HPREDs} \ n) \\
\text{HPREDs} \ (\text{suc} \ n) \ (\text{suc} \ i) \ = \ \text{HPREDs} \ n \ i \\
\text{HSET} : \mathbb{N} \rightarrow \text{Set} \\
\text{HSET} \ n \ = \ \text{HU} \ (\text{HPREDs} \ n)

Note that \text{HSET} \ n is indeed \(\llbracket \text{U}' \ \text{zero} \rrbracket_{\text{HU}}\) at level \text{suc} \ n.

The trouble with this representation, however, is that it is not cumulative for free. Intuitively, every type at each level has a counterpart at all higher levels, but how can we get our hands on it?

\textbf{Exercise 5.9 (fool's errand)} \ Find out what breaks when you try to implement cumulativity. What equation do you need to hold? Can you prove it?

\text{Cumu} : (n : \mathbb{N}) \ (T : \text{HSET} \ n) \rightarrow \text{HSET} \ (\text{suc} \ n)
\text{Cumu n T} = ?
5.4 Encoding Induction-Recursion

So far, we have been making \texttt{mutual} declarations of inductive types and recursive functions to which Agda has said 'yes'. Clearly, however, we could write down some rather paradoxical definitions if we were not careful. Fortunately, the following is not permitted,

\begin{verbatim}
mutual -- rejected
\begin{verbatim}
data VV : Set where
  V′ : VV
  \Pi′ : (S : VV) (T : [S]VV \rightarrow VV) \rightarrow VV
\end{verbatim}
\end{verbatim}
[\Pi′]VV : VV \rightarrow Set
[\Pi′]VV = VV
[\Pi′ S T]VV = (s : [S]VV) \rightarrow [T s]VV
\end{verbatim}

but it was not always so.

It would perhaps help to make sense of what is possible, as well as to provide some sort of metaprogramming facility, to give an encoding of the permitted inductive-recursive definitions. Such a thing was given by Peter Dybjer and Anton Setzer in 1999. Their encoding is (morally) as follows, describing one node of an inductive recursive type rather in the manner of a right-nested record, but one from which we expect to read off a \( J \)-value, and whose children allow us to read off \( I \)-values.

\begin{verbatim}
data DS (I J : Set) : Set where
  ι : J \rightarrow DS I J -- no more fields
  σ : (S : Set) (T : S \rightarrow DS I J) \rightarrow DS I J -- ordinary field
  δ : (H : Set) (T : (H \rightarrow I) \rightarrow DS I J) \rightarrow DS I J -- child field
\end{verbatim}

We interpret a \( DS I J \) as a functor from \( \text{Fam} I \) to \( \text{Fam} J \): I build the components separately for readability.

\begin{verbatim}
\begin{verbatim}
  [ι]DS : forall \{I J\} \rightarrow DS I J \rightarrow \text{Fam} I \rightarrow \text{Fam} J
  [ι j]DS Xxi
    = One
    , λ \{() \rightarrow j\}
  [σ S T]DS Xxi
    = (Σ S λ s \rightarrow \text{fst} ([T s]DS Xxi))
    , λ \{(s, t) \rightarrow \text{snd} ([T s]DS Xxi) t\}
  [δ H T]DS (X, xi)
    = (Σ (H \rightarrow X) λ hX \rightarrow \text{fst} ([T (xi o hX)]DS (X, xi)))
    , λ \{(hX, t) \rightarrow \text{snd} ([T (xi o hX)]DS (X, xi)) t\}
\end{verbatim}
\end{verbatim}

In each case, we must say which set is being encoded and how to read off a \( J \) from a value in that set. The \( ι \) constructor carries exactly the \( j \) required. The other two specify a field in the node structure, from which the computation of the \( J \) gains some information. The \( σ \) specifies a field of type \( S \), and the rest of the structure may depend on a value of type \( S \).

The \( δ \) case is the clever bit. It specifies a place for an \( H \)-indexed bunch of children, and even though we do not fix what set represents the children, we do know that they allow us to read off an \( I \). Correspondingly, the rest of the structure can at least depend on knowing a function in \( H \rightarrow I \) which gives access to the interpretation of the children. Once we plug in a specific \((X, xi) : \text{Fam} I\), we represent the field by the \textit{small} function space \( hX : H \rightarrow X \), then the composition \( xi \circ hX \) tells us how to compute the \textit{large} meaning of each child.
Exercise 5.10 (idDS)  A morphism from \((X, xi)\) to \((Y, yi)\) in Fam \(I\) is a function \(f : X \to Y\) such that \(xi = yi \circ f\). Construct a code for the identity functor on Fam \(I\), being

\[
\text{idDS} : \{ I : \text{Set}_1 \} \to \text{DS } I I
\]

\(\text{idDS} = ?\)

such that

\[
\llbracket \text{idDS} \rrbracket_{\text{DS}} \cong \text{id}
\]

in the sense that both take isomorphic inputs to isomorphic outputs.

With this apparatus in place, we could now tie the recursive knot . . .

\[
\text{mutual} \quad \text{-- fails positivity check and termination check}
\]

\[
\text{data DataDS } \{ I \} \ (D : \text{DS } I I) : \text{Set where}
\]

\[
\llbracket \_ \rrbracket_{\text{DS}} : \{ I : \text{Set}_1 \} \{ D : \text{DS } I I \} \to \text{DataDS } D
\]

\[
\llbracket \_ \rrbracket_{\text{DS}} \{ D = D \} \{ ds \} = \text{snd} \ (\llbracket D \rrbracket_{\text{DS}} (\text{DataDS } D, \llbracket \_ \rrbracket_{\text{DS}}) ) \ ds
\]

. . . if only the positivity checker could trace the construction of the node set through the tupled presentation of \(\llbracket \_ \rrbracket_{\text{DS}}\) and the termination checker could accept that the recursive invocation of \(\llbracket \_ \rrbracket_{\text{DS}}\) is used only for the children packed up inside the node record. Not for the first or the last time, we can only get out of the jam by inlining the interpretation:

\[
\text{mutual}
\]

\[
\text{data DataDS } \{ I \} \ (D : \text{DS } I I) : \text{Set where}
\]

\[
\llbracket \_ \rrbracket_{\text{DS}} : \{ I : \text{Set}_1 \} \{ D : \text{DS } I I \} \to \text{DataDS } D \to I
\]

\[
\llbracket \_ \rrbracket_{\text{DS}} \{ D = D \} \{ ds \} = \text{DeDS } D \ D \ ds
\]

\[
\text{NoDS } : \{ I : \text{Set}_1 \} \ (D D' : \text{DS } I I) \to \text{Set}
\]

\[
\text{NoDS } D \ (i \ i) = \text{One}
\]

\[
\text{NoDS } D \ (\circ S T) = \Sigma \ S \ \lambda \ s \to \text{NoDS } D \ (T \ s)
\]

\[
\text{DeDS } : \{ I : \text{Set}_1 \} \ (D D' : \text{DS } I I) \to \text{NoDS } D \ D' \to I
\]

\[
\text{DeDS } D \ (t \ i) \ () = i
\]

\[
\text{DeDS } D \ (\circ S T) \ (s, t) = \text{DeDS } D \ (T \ s) \ t
\]

\[
\text{DeDS } D \ (\circ H T) \ (hd, t) = \text{DeDS } D \ (T \ (\lambda \ h \to \ [\ hd \ h]_{\text{DS}})) \ t
\]

Exercise 5.11 (encode TU)  Construct an encoding of TU in DS Set Set.

If you have an eye for this sort of thing, you may have noticed that DS \(I\) is a monad, with \(\iota\) as its ‘return’.

Exercise 5.12 (bindDS and its meaning)  Implement the appropriate bindDS operator, corresponding to substitution at \(\iota\).

\[
\text{bindDS } : \text{forall } I \ J K \to \text{DS } I J \to (J \to \text{DS } I K) \to \text{DS } I K
\]

\(\text{bindDS } T U = ?\)

Show that bindDS corresponds to a kind of \(\Sigma\) by implementing pairing and projections:

\[
\text{pairDS } : \text{forall } I \ J K \ (T : \text{DS } I J) \ (U : J \to \text{DS } I K) \{ X : \text{Fam } I \} \to (t : \text{fst} \ ([\ T ]_{\text{DS}} X)) \ (u : \text{fst} \ ([\ U \ (\text{snd} ([\ T ]_{\text{DS}} X) \ t]_{\text{DS}} X)))
\]

\[
\to \text{fst} \ ([\ \text{bindDS } T U ]_{\text{DS}} X)
\]
5.5 Irish Induction-Recursion

So I went to this meeting with some friends who like containers, induction-recursion, and other interesting animals in the zoo of datatypes. I presented what I thought was just a boring Desc-like rearrangement of Dybjer and Setzer’s encoding of induction-recursion. ‘That’s not IR!’ said the audience, and it remains an open problem whether or not they were correct: it is certainly IR-ish, but we do not yet know whether it captures just the same class of functors as Dybjer and Setzer’s encoding, or strictly more. (If the latter, we shall need a new model construction, to ensure the system’s consistency.)

I give an inductive-recursive definition of IR. The type Irish I describes node structures where children can be interpreted in I. Deferring the task of interpreting such a node, let us rather compute the type of information we can learn from it.

Note that Info {I} T is large because I is, but fear not, for it is not the type of the nodes themselves.

mutual

data Irish (I : Set_1) : Set_1 where
  i : Irish I
  κ : Set → Irish I
  π : (S : Set) (T : S → Irish I) → Irish I
  σ : (S : Irish I) (T : Info S → Irish I) → Irish I

Info : forall {I} → Irish I → Set_1
Info {I} i = I
Info (κ A) = ↑ A
Info (π S T) = (s : S) → Info (T s)
Info (σ S T) = Σ (Info S) λ s → Info (T s)

To interpret π and σ, we shall need to equip Fam with pointwise lifting and dependent pairs, respectively.

ΠIF : (S : Set) {J : S → Set_1} (T : (s : S) → Fam (J s)) → Fam ((s : S) → J s)
Now, for any \( T : \text{Irish} \), if someone gives us a \( \text{Fam} \) to represent children, we can compute a \( \text{Fam} \) (Info \( T \)) — a small node structure from which the large Info \( T \) can be extracted.

\[
\begin{align*}
\text{Node} : & \text{forall } \{ I \} ( T : \text{Irish} \ I) \rightarrow \text{Fam} \ I \rightarrow \text{Fam} \ (\text{Info} \ T) \\
\text{Node} \ i & : \ X = X \\
\text{Node} \ (\kappa \, A) & : \ X = A, \uparrow \\
\text{Node} \ (\pi \, S \, T) & : \ X = \text{IF} \ S \, \lambda \, s \rightarrow \text{Node} \ (T \ s) \, X \\
\text{Node} \ (\sigma \, S \, T) & : \ X = \Sigma \ (\text{Node} \ S \ X) \, \lambda \, i \, S \ ightarrow \text{Node} \ (T \ i \ S) \, X
\end{align*}
\]

A functor from \( \text{Fam} \ I \) to \( \text{Fam} \ J \) is then given by a pair

\[
\begin{align*}
\text{IF} & : \text{Set}_1 \rightarrow \text{Set}_1 \rightarrow \text{Set}_1 \\
\text{IF} \ I \ J & = \Sigma \ (\text{Irish} \ I) \, \lambda \, T \rightarrow \text{Info} \ T \rightarrow \text{J} \\
[\_|\_]_F & : \text{forall } \{ I \, J \} \rightarrow \text{IF} \ I \ J \rightarrow \text{Fam} \ I \rightarrow \text{Fam} \ J \\
[\_ | \_]_F X & = d \ s t \, \text{Node} \ T \ X
\end{align*}
\]

With a certain tedious inevitability, we find that Agda rejects the obvious attempt to tie the knot.

\[\text{mutual} \quad \text{-- fails positivity and termination checks}\]

\[
\begin{align*}
\text{data} \ \text{DataIF} \ \{ I \} \ (F : \text{IF} \ I \ I) : \ \text{Set} \text{ where} \\
\langle \_ \rangle & : \ \text{fst} \ (\{ \_ \, | \_\} _F (\text{DataIF} \ F, \{\_ | \_\} _F)) \rightarrow \text{DataIF} \ F \\
[\_ | \_]_F & : \ \text{forall} \ \{ I \} \ \{ F : \text{IF} \ I \ I \} \rightarrow \text{DataIF} \ F \rightarrow \text{I} \\
[\_ | \_]_F \ {\langle F = F \rangle} \ {\langle rs \rangle} & = \ \text{snd} \ (\{ F \} _F (\text{DataIF} \ F, \{\_ | \_\} _F)) \ ds
\end{align*}
\]

Again, specialization of Node fixes the problem

\[\text{mutual} \quad \]

\[
\begin{align*}
\text{data} \ \text{DataIF} \ \{ I \} \ (F : \text{IF} \ I \ I) : \ \text{Set} \text{ where} \\
\langle \_ \rangle & : \ \text{Nat} \ F \ (\text{fst} \ F) \rightarrow \text{DataIF} \ F \\
[\_ | \_]_F & : \ \text{forall} \ \{ I \} \ \{ F : \text{IF} \ I \ I \} \rightarrow \text{DataIF} \ F \rightarrow \text{I} \\
[\_ | \_]_F \ {\langle F = F \rangle} \ {\langle rs \rangle} & = \ \text{snd} \ (\{ F \} _F (\text{DataIF} \ F, \{\_ | \_\} _F)) \ ds \\
\text{NoIF} & : \ \text{forall} \ \{ I \} \ \{ F : \text{IF} \ I \ I \} \ (T : \text{Irish} \ I) \rightarrow \text{Set} \\
\text{NoIF} \ F & = \ \text{DataIF} \ F \\
\text{NoIF} \ (\kappa \, A) & = \ A \\
\text{NoIF} \ F \ (\pi \, S \, T) & = \ (\_ : S) \rightarrow \text{NoIF} \ F \ (T \ s) \\
\text{NoIF} \ F \ (\pi \, S \, T) & = \ \Sigma \ (\text{NoIF} \ F \ S) \, \lambda \, s \rightarrow \text{NoIF} \ F \ (T \ (\text{DelF} \ F \ S \ s)) \\
\text{DelF} & : \ \text{forall} \ \{ I \} \ \{ F : \text{IF} \ I \ I \} \ (T : \text{Irish} \ I) \rightarrow \text{NoIF} \ F \ T \rightarrow \text{Info} \ T \\
\text{DelF} \ F & = \ \text{DataIF} \ F \\
\text{DelF} \ F \ (\kappa \, A) & = \ A \\
\text{DelF} \ F \ (\pi \, S \, T) \ f & = \ \lambda \, s \rightarrow \text{DelF} \ F \ (T \ s) \ (f \ s) \\
\text{DelF} \ F \ (\sigma \, S \, T) \ (s, t) & = \ \text{let} \ s' = \ \text{DelF} \ F \ s \ s \ \text{in} \ s', \ \text{DelF} \ F (T \ s') \ t
\end{align*}
\]

Given that Agda lets us implement Irish IR, one wonders whether it allows even more.

Irish IR is a little closer to the user experience of IR in Agda, in that you give separately a description of your data’s node structure and the ‘algebra’ which decodes it.
Exercise 5.14 (Irish TU) Give a construction for the TU universe as a description-decoder pair in IF Set Set.

We should check that Irish IR allows at least as much as Dybjer-Setzer.

Exercise 5.15 (Irish-to-Swedish) Show how to define

\[ \text{DSIF} : \forall \{ I J \} \to \text{DS} I J \to \text{IF} I J \]

such that

\[ \llbracket \text{DSIF} T \rrbracket_{\text{DS}} \cong \llbracket T \rrbracket_{\text{IF}} \]

We clearly have an identity for Irish IR.

\[ \text{idIF} : \forall \{ I \} \to \text{IF} I I \]

\[ \text{idIF} = \iota, \text{id} \]

Now, DS I J had a substitution-for-\( \iota \) structure which induced a notion of pairing, because \( \iota \) marks ‘end of record’. What makes the Irish encoding conductive to composition is that the \( \iota \)-leaves of an Irish I mark where the children go.

Exercise 5.16 (subIF) Construct a substitution operator for Irish J with a refinement of the following type.

\[ \text{subIF} : \forall \{ I J \} (T : \text{Irish } J) (F : \text{IF } I J) \to \Sigma (\text{Irish } I) \]

\[ \text{subIF} T F = ? \]

Hint: you will find out what you need in the \( \sigma \) case.

Exercise 5.17 (colF) Now define composition for Irish IR functors.

\[ \text{colF} : \forall \{ I J K \} \to \text{IF } J K \to \text{IF } I J \to \text{IF } I K \]

\[ \text{colF} G F = ? \]

Some of us are inclined to suspect that IF does admit more functors than DS, but the exact status of Irish induction-recursion remains the stuff of future work.
Chapter 6

Observational Equality

We cannot have an equality which is both extensional and decidable. We choose to keep judgmental equality decidable, hence it is inevitably disappointing, but we introduce a propositional equality, allowing us to give evidence for equations on open terms which the computer is too stupid to see. Correspondingly, we need a substitution mechanism to transport values from $P_s$ to $P_t$ whenever $s \simeq t$. The way $\text{subst}$ has worked thus far is to wait at least until $s \equiv t$ holds judgmentally, so that $p : P_s$ implies $p : P_t$, allowing $p$ to be transmitted as it stands. (Waiting for the proof of $s \simeq t$ to become $\text{refl}$ means waiting at least until $s \equiv t$.)

The trouble with this way to compute $\text{subst}$ is that we have no way to explain its computation if there are provably equal closed terms which are not judgmentally equal. We can add axioms for extensionality and retain consistency, even extracting working programs which compile $\text{subst}$ to $\text{id}$ and never compute under a binder. However, such axioms impede open computation. If we want a propositional equality to make up for our disappointment with judgmental equality, and a $\text{subst}$ which works, we must figure out how to transport values between provably but not judgmentally equal types.

The situation is particularly galling when you think how a type like $P f$ could possibly depend on a function $f$. If all $P$ ever does with $f$ is to apply it, then of course $P$ respects extensional equality. If types can only depend on values by observing them, then there should be a systematic way to show that transportability between types respects equality-up-to-observation.

But does the hypothesis of the previous sentence hold? Consider

\begin{verbatim}
data Favourite : (N \to N) \to Set where
  favourite : Favourite (\lambda x \to zero +_N x)
\end{verbatim}

We may certainly prove that $\lambda x \to zero +_N x$ and $\lambda x \to x +_N zero$ agree on all inputs. But is there a canonical inhabitant of $\text{Favourite} (\lambda x \to x +_N zero)$? If so, it can only be $\text{favourite}$, for that is the only constructor, but $\text{favourite}$ does not have that type because the two functions are not judgmentally equal. The trouble is that by using the power to ‘focus’ a constructor’s return type on specific indices, $\text{Favourite}$ is an intensional predicate, holding only for a specific implementation of a particular function. We cannot expect a type theory with intensional predicates to admit a sensible notion of extensional equality. Let us do away with them! If, instead, we reformulate $\text{Favourite}$ in the Henry Ford tradition,

\begin{verbatim}
data Favourite (f : N \to N) : Set where
  favourite : Favourite (\lambda x \to zero +_N x) \simeq f \to Favourite f
\end{verbatim}

then our definition of $\text{Favourite}$ becomes just as intensional as our equality. If, somehow, $\simeq$ were to admit extensionality, we could certainly show that $\text{Favourite}$
respects \( \simeq \). If \( q' : f \simeq g \), then we can transport favourite \( q \) from Favourite \( f \) to Favourite \( g \), returning, not the original data but

\[
\text{favourite} ((\lambda x \rightarrow \text{zero} +_{\mathbb{N}} x) \equiv q) f \equiv (q') g \square
\]

with a modified proof.

### 6.1 Observational Equality for Types and Values in \( \text{TU} \)

We have got as far as figuring out that a propositional equality which is more generous than the judgmental equality will require a computation mechanism which might modify the data it transports between provably equal types, but should not change the results of observing the data. To say what that mechanism is, we shall need to inspect the types involved, so let us work with the types of the \( \text{TU} \) universe and develop what equality means for its types and values by metaprogramming.

We shall need to consider when types are equal: I write \( X \leftrightarrow Y \) to indicate that \( X \) and \( Y \) are types whose data are interchangeable. I propose the bold choice to consider only those kinds of interchangeability which can be implemented by the identity function at closed-run time. Enthusiasts for Voevodsky’s univalence axiom are entitled to be disappointed by this choice, but perhaps a simple computational interpretation will prove modest consolation.

Inasmuch as types depend on values, we shall also need to say when values are equal. There is no reason to presume that we shall be interested only to consider the equality of values in types which are judgmentally equal, for we know that judgmental equality is too weak to recognize the sameness of some types whose values are interchangeable. Correspondingly, let us weaken our requirement for the formation of value equalities and have a heterogeneous equality, \( \text{Eq} \ X \ x \ Y \ y \).

We have some options for how to do that:

- We could make add the requirement \( X \leftrightarrow Y \) to the formation rule for Eq.
- We could allow the formation of any \( \text{Eq} \ X \ x \ Y \ y \), but ensure that it holds only if \( X \leftrightarrow Y \).
- We could allow the formation of any \( \text{Eq} \ X \ x \ Y \ y \), but ensure that proofs of such equations are useless information unless \( X \leftrightarrow Y \).

All three are sustainable, but I find the third is the least bureaucratic. The proposition \( \text{Eq} \ X \ x \ Y \ y \) means ‘if \( X \) is \( Y \), then \( x \) is \( y \)’ and should thus be considered ‘true but dull’ if \( X \) is clearly not \( Y \).

We need to define them by recursion on types. It’s convenient to build them together, then project out the type and value components. Note that we work internally to the universe: we already have the types we need to describe the evidence for equality of types and values in this sense.

\[
\begin{align*}
\text{mutual} \\
\text{EQ} : (X \ Y : \text{TU}) \rightarrow \text{TU} \times (\llbracket X \rrbracket_{\text{TU}} \rightarrow \llbracket Y \rrbracket_{\text{TU}} \rightarrow \text{TU}) \\
\downarrow_{\leftrightarrow} : \text{TU} \rightarrow \text{TU} \\
X \leftrightarrow Y & = \text{fst} (\text{EQ} \ X \ Y) \\
\text{Eq} : (X : \text{TU}) (x : \llbracket X \rrbracket_{\text{TU}}) \rightarrow (Y : \text{TU}) (y : \llbracket Y \rrbracket_{\text{TU}}) \rightarrow \text{TU} \\
\text{Eq} \ X \ x \ Y \ y & = \text{snd} (\text{EQ} \ X \ Y) \ x \ y
\end{align*}
\]

We should expect, ultimately, to construct a coercion mechanism which realises equality as transportation.
6.1. OBSERVATIONAL EQUALITY FOR TYPES AND VALUES IN \( \text{TU} \)

\[
\text{coe} : (X Y : \text{TU}) \rightarrow \llbracket X \leftrightarrow Y \rrbracket_{\text{TU}} \rightarrow \llbracket X \rrbracket_{\text{TU}} \rightarrow \llbracket Y \rrbracket_{\text{TU}}
\]

Moreover, we should ensure that coercion does not change the observable properties of values and is thus coherent in the sense that

\[
\text{coh} : (X Y : \text{TU}) \left( Q : \llbracket X \leftrightarrow Y \rrbracket_{\text{TU}} \right) \rightarrow \left[ \text{Eq} X Y (\text{coe} X Y Q x) \right]_{\text{TU}}
\]

Given what we want to use equality for, we should be able to figure out what it needs to be, on a case-by-case basis.

Base types equal only themselves, and we need no help to transport a value from a type to itself. For \( \text{Zero'} \) and \( \text{One'} \), all values are equal as there is at most one value anyway. For \( \text{Two'} \), we must actually test the values.

\[
\begin{align*}
\text{EQ Zero'} \text{Zero'} &= \text{One'}, \lambda \_ \_ \rightarrow \text{One'} \\
\text{EQ One'} \text{One'} &= \text{One'}, \lambda \_ \_ \rightarrow \text{One'} \\
\text{EQ Two'} \text{Two'} &= \text{One'}, \lambda \\
\{ \text{tt tt} \rightarrow \text{One'} \\
; \text{ff ff} \rightarrow \text{One'} \\
; \_ \_ \rightarrow \text{Zero'} \\)
\end{align*}
\]

\( \Sigma' \)-types are interchangeable if their components are, but how are we to express the interchangeability of the dependent second components? It is enough to consider the types of the second components only when the values of the first components agree, a situation we can consider hypothetically by abstracting not over one value, which would need to have both first component types, but rather over a pair of equal values drawn from each.

\[
\begin{align*}
\text{EQ} (\Sigma' S T) (\Sigma' S' T') &= (\Sigma' (S \leftrightarrow S')) \lambda \_ \rightarrow \\
& \Pi' S \lambda s \rightarrow \Pi' S' \lambda s' \rightarrow \Pi' (\text{Eq} S s S' s') \lambda \_ \rightarrow \\
& T s \leftrightarrow T' s') \\
& \lambda \{ (s, t) (s', t') \rightarrow \Sigma' (\text{Eq} S s S' s') \lambda \_ \rightarrow \text{Eq} (T s t) (T' s' t') \}
\end{align*}
\]

Equality of pair values is straightforwardly structural. Notice that if the \( \Sigma' \)-types are equal then their first component types are equal, so it is useful to know that the first component values are equal, which in turn lets us deduce equality of the second component types.

Equality of functions types is similar, save for the contravariant twist I have put in the domain type equation. To coerce a function from left to right, we shall need to coercer its input from right to left.

\[
\begin{align*}
\text{EQ} (\Pi' S T) (\Pi' S' T') &= (\Sigma' (S' \leftrightarrow S)) \lambda \_ \rightarrow \\
& \Pi' S' \lambda s' \rightarrow \Pi' S \lambda s \rightarrow \Pi' (\text{Eq} S' s' S s) \lambda \_ \rightarrow \\
& T s \leftrightarrow T' s') \\
& \lambda \{ f f' \rightarrow \Pi' S \lambda s \rightarrow \Pi' S' \lambda s' \rightarrow \Pi' (\text{Eq} S s S' s') \lambda \_ \rightarrow \\
& \text{Eq} (T s) (f s) (T' s') (f' s') \}
\end{align*}
\]

Function values are considered equal if they take equal inputs to equal outputs. \( \text{Tree'} \)-types are, again, compared structurally, with pointwise equality expressed by abstraction over pairs of equal values.

\[
\begin{align*}
\text{EQ} (\text{Tree'} I F i) (\text{Tree'} I' F' i') &= (\Sigma' (I \leftrightarrow I')) \lambda \_ \rightarrow \\
& \Sigma' (\text{Eq} I i I' i') \lambda \_ \rightarrow
\end{align*}
\]
where do you get stuck?

refl

exercise 6.2 (explore failing to prove)

let \((S, K) = F \ i; S', K' = F \ i'

in \(\Sigma' (S \leftrightarrow S') \lambda \_ \rightarrow

\Pi' S \lambda s \rightarrow \Pi' S' \lambda s' \rightarrow \Pi' (Eq S s S' s') \lambda \_ \rightarrow

let \((P, r) = K \ s; (P, r) = K' s'

in \(\Sigma' (P \leftrightarrow P) \lambda \_ \rightarrow

\Pi' P \lambda p' \rightarrow \Pi' \lambda P \ p \rightarrow \Pi' (Eq P p P p) \lambda \_ \rightarrow

Eq I (r p) I' (r p'))

where
teq i i' = refl

teq : (i : \[ I \]_TU) \rightarrow (i' : \[ I' \]_TU) \rightarrow

\[ Tree' \ I' F i' \]_TU \rightarrow \[ Tree' \ I' F i' \]_TU \rightarrow TU

teq i i' (s, k) (s', k') = let \((S, K) = F \ i; (S', K') = F \ i'

(P, r) = K \ s; (P, r) = K' s'

in \(\Sigma' (Eq S s S' s') \lambda \_ \rightarrow

\Pi' P \lambda p' \rightarrow \Pi' \lambda P \ p \rightarrow \Pi' (Eq P p P p') \lambda \_ \rightarrow

teq (r p) (r p') (k p) (k' p')

Tree' value equality is defined by structural recursion. At each node, we demand equal shapes, then at equal positions, equal subtrees.

Finally, types whose head constructors disagree are considered unequal, hence their values are vacuously equal.

\( EQ \_ \_ = \) Zero', \( \lambda \_ \_ \rightarrow One'\)

exercise 6.1 (define coe, postulate coh)

Implement coercion, assuming coherence.

do coe : (X Y : TU) \rightarrow \[ X \leftrightarrow Y \]_TU \rightarrow \[ X \]_TU \rightarrow \[ Y \]_TU

postulate
do coh : (X Y : TU) (Q : \[ X \leftrightarrow Y \]_TU) (x : \[ X \]_TU) \rightarrow \[ Eq X x Y (coe X Y Q x) \]_TU

do coe X Y Q x = ?

if you look at the definition of EQ quite carefully, you will notice that we did not use all of the types in TU to express equations. There is never any choice about how to be equal, so we need never use Two'; meanwhile, we can avoid expressing tree equality as itself a tree just by using structural recursion. As a result, the only constructor pattern matching coe need ever perform on proofs is on pairs, which is just sugar for the lazy use of projections. Correspondingly, the only way coercion of canonical values between canonical types can get stuck is if those types are conspicuously different. Although we postulated coherence, no computation which relies on it is strict in equality proofs, so it is no source of blockage.

The only way a closed coercion can get stuck is if we can prove a false equation. The machinery works provided the theory is consistent, but we can prove no equations which do not also hold in extensional type theories which are known to be consistent. In general, we are free to assert consistent equations. Let us have

postulate

do refl TU : (X : TU) (x : \[ X \]_TU) \rightarrow \[ Eq X x X x \]_TU

exercise 6.2 (explore failing to prove)

Try proving

do refl TU : (X : TU) (x : \[ X \]_TU) \rightarrow \[ Eq X x X x \]_TU

do refl TU X x = ?

where do you get stuck?
Homogeneous equations between values are made useful just by asserting that predicates respect them. We recover the Leibniz property.

postulate
\[ \text{Resp}_{TU} : (X : TU) \rightarrow (P : [X]_{TU} \rightarrow TU) \]
\[ (x x' : [X]_{TU}) \rightarrow ([\text{Eq} X x X x']_{TU} \rightarrow [P x \leftrightarrow P x']_{TU}) \]
\[ \text{subst}_{TU} : (X : TU) \rightarrow (P : [X]_{TU} \rightarrow TU) \]
\[ (x x' : [X]_{TU}) \rightarrow ([\text{Eq} X x X x']_{TU} \rightarrow [P x \leftrightarrow P x']_{TU}) \]
\[ \text{subst}_{TU} X P x x' q = \text{coe} (P x) (P x') (\text{Resp}_{TU} X P x x' q) \]

It is clearly desirable to construct a model in which these postulated constructs are given computational force, not least because such a model would yield a more direct proof of consistency. However, we have done enough to gain a propositional equality which is extensional for functions, equipped with a mechanism for obtaining canonical forms in 'data' computation.

6.2 A Universe with Propositions

We can express the observation that all of our proofs belong to lazy types by splitting our universe into two Sorts, corresponding to sets and propositions, embedding the latter explicitly into the former with a new set-former, \( \text{Prf}' \).

```
data Sort : Set where set prop : Sort
IsSet : Set \rightarrow Set
IsSet set = One
IsSet prop = Zero
mutual
data Set (u : Sort) : Set where
  Zero' One' : Set u
  Two' : \{ _, IsSet u \} \rightarrow Set u
  Sigma' : (S : Set u) \rightarrow (T : \{ S \}_P \rightarrow Set u) \rightarrow Set u
  Pi' : (S : Set set) \rightarrow (T : \{ S \}_P \rightarrow Set u) \rightarrow Set u
  Tree' : \{ _, IsSet u \} \rightarrow (I : Set set)
    \rightarrow \left( F : \{ I \}_P \rightarrow \Sigma \left( \text{Set set} \right) \lambda S \rightarrow \right.
    \left. \{ S \}_P \rightarrow \Sigma \left( \text{Set set} \right) \lambda P \rightarrow \right)
    \left. \{ P \}_P \rightarrow \{ I \}_P \right) \rightarrow Set u
  Prf' : \{ _, IsSet u \} \rightarrow Set prop \rightarrow Set u

[\_\_P] : \text{forall} \{ u \} \rightarrow Set u \rightarrow Set
[Zero'] \_\_P = Zero
[One'] \_\_P = One
[Two'] \_\_P = Two
[Sigma' ST] \_\_P = Sigma S T \rightarrow \{ S \}_P \rightarrow \{ T s \}_P
[Pi' ST] \_\_P = \Pi S T \rightarrow \{ S \}_P \rightarrow \{ T s \}_P
[Tree' IF i] \_\_P = \Pi Tree F i
  ( (\lambda i \rightarrow \{ \text{fst} (F i) \}_P)
   \triangleleft (\lambda i s \rightarrow \{ \text{fst} (\text{snd} (F i) s) \}_P)
   s \rightarrow (\lambda i s \rightarrow \text{snd} (\text{snd} (F i) s) p)
  ) i
[Prf' P] \_\_P = \{ P \}_P
```

Note that Two' and Tree' are excluded from Set prop and that sort is always preserved in covariant positions and set in contravariant positions. The interpretation
of types is just as before. One could allow the formation of inductive predicates, being Tree′ structures with propositional node shapes, but we should then be careful not to pattern match on proofs when working with data in sets. I have chosen to avoid the risk, allowing only propositions whose eliminators are in any case lazy.

**Exercise 6.3 (observational propositional equality)** Reconstruct the definition of observational equality in this more refined setting. Take equality of propositions to be mutual implication and equality of proofs to be trivial: after all, equality for proofs of the atomic Zero′ and One′ propositions are trivial.

\[ \land : \text{Set prop} \to \text{Set prop} \to \text{Set prop} \]
\[ P \land Q = \Sigma' P \lambda _{-} \to Q \]
\[ \Rightarrow : \text{Set prop} \to \text{Set prop} \to \text{Set prop} \]
\[ P \Rightarrow Q = \Pi' (\text{Prf'} P) \lambda _{-} \to Q \]

**mutual**

PEQ : (X Y : Set set) \to Set prop \times (\mu X \mu Y \nu X_\mu Y_\nu \to Set prop)

PEq : (X : Set set) (x : \mu X \mu Y \nu x) \to (Y : Set set) (y : \mu Y \mu X \nu y) \to Set prop

PEq X x Y y = \text{snd} (PEQ X Y) x y

PEQ (Prf' P) (Prf' Q) = ((P \Rightarrow Q) \land (Q \Rightarrow P)), \lambda _{-} \to One'

\[ \text{PEQ } _{-} _{-} = \text{Zero'}, \lambda _{-} \to One' \]
Chapter 7

Type Theory in Type Theory

A while ago, we defined the simply typed \( \lambda \)-calculus as a syntax of well scoped, well typed terms. Can we do the same for a dependently typed calculus? Yes and no, but not necessarily in that order.
Chapter 8

Reflections and Directions
Bibliography
