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, metaprograms 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 _,_
```

Agda has a very simple lexer and very few special characters. To a first approximation, (){}; stand alone and everything else must be delimited with whitespace.

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 $S \times T$ 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
```

The braces indicate that S and T are *implicit arguments*. Agda will try to infer them unless we override manually.

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 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 *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.

The number of c's in suc is a long standing area of open warfare.

Agda users tend to use lowercase-vs-uppercase to distinguish things in Sets from things which are or manipulate Sets.

The pragmas let you use Arabic numerals.

```
data ℕ : Set where
  zero :      ℕ
  suc  : ℕ → ℕ
{-# BUILTIN NATURAL Nat #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC suc #-}


length : {X : Set} → List X → ℕ
length ⟨⟩       = zero
length (x , xs) = suc (length xs)
```

Informally,[1] we might state and prove something like

$$\forall ss, ts.\ \mathsf{length}\ ss = \mathsf{length}\ ts \Rightarrow \mathsf{length}\ (\mathsf{zip}\ ss\ ts) = \mathsf{length}\ ss$$

by structural induction [Burstall, 1969] on ss, say. Of course, we could just as well have concluded that length (zip ss ts) = length ts, and if we carry on 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 *prescribed*—to be checked, not measured.

1.2 Vectors

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

```
data Vec (X : Set) : ℕ → Set where
  ⟨⟩    :                          Vec X zero
  _,_ : {n : ℕ} → X → Vec X n → Vec X (suc n)
```

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 *parameter* of the definition. Meanwhile, 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 *index*. In an Agda **data** declaration head, arguments left of : (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

[1] by which I mean, not to a computer

freely instantiated in *recursive* positions, so we should not presume that they are necessarily parameters.

Let us now develop zip for vectors, stating the length invariant in the type.

```
zip : forall {n S T} → Vec S n → Vec T n → Vec (S × T) n
zip ss ts = ?
```

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,

```
zip : forall {n S T} → Vec S n → Vec T n → Vec (S × T) n
zip ss ts = { }0
```

and tells us, in the information window,

```
?0 : Vec (.S × .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−,], we will gain more information about the goal:

```
Goal : Vec (Σ .S (λ _ .T)) .n
————————————————————————————————————————————————————————————
ts  : Vec .T .n
ss  : Vec .S .n
.T  : Set
.S  : Set
.n  : ℕ
```

revealing the definition of × 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 ss : Vec .S .n that we do not know the length of ss, so it might be given by either constructor. To see if Agda agrees, we type ss in the hole and issue the 'case-split' command [C − c C − c].

```
zip : forall {n S T} → Vec S n → Vec T n → Vec (S × T) n
zip ss ts = {ss [C − c C − c]}0
```

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

```
zip : forall {n S T} → Vec S n → Vec T n → Vec (S × T) n
zip ⟨⟩ ts = { }0
zip (x, ss) ts = { }1
```

Moreover, we gain the refined type information

```
?0 : Vec (.S × .T) 0
?1 : Vec (.S × .T) (suc .n)
```

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 0 case, we get

```
zip : forall {n S T} → Vec S n → Vec T n → Vec (S × T) n
zip ⟨⟩ ⟨⟩ = { }0
zip (x, ss) ts = { }1
```

and in the suc case,

```
zip : forall {n S T} → Vec S n → Vec T n → Vec (S × T) n
zip ⟨⟩ ⟨⟩ = { }0
zip (x, ss) (x1 , ts) = { }1
```

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 $[C - c\ C - a]$ in each hole, a type-directed program search robot called 'Agsy' tries to find an expression which will fit in the hole, asssembling it from the available information without further case analysis. We obtain a complete program.

```
zip : forall {n S T} → Vec S n → Vec T n → Vec (S × T) n
zip ⟨⟩ ⟨⟩ = ⟨⟩
zip (x, ss) (x1 , ts) = (x, x1 ), zip ss ts
```

I tend to α-convert and realign such programs manually, yielding

```
zip : forall {n S T} → Vec S n → Vec T n → Vec (S × T) n
zip ⟨⟩      ⟨⟩      = ⟨⟩
zip (s, ss) (t, ts) = (s, t), zip ss ts
```

What just happened? We made Vec, a version of List, indexed by $\mathbb{N}$, and suddenly became able to work with 'elements in corresponding positions' with some degree of precision. That worked because $\mathbb{N}$ describes the *shape* of lists: indeed $\mathbb{N} \cong$ List One, instantiating the List element type to the type One with the single element $\langle\rangle$, 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} → X → 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} → Vec (S → T) n → Vec S n → 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.

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

```
vmap : forall {n S T} → (S → T) → Vec S n → Vec T n
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.*

```
zip : forall {n S T} → Vec S n → Vec T n → Vec (S × T) n
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:*

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} → Fin n → Fin (suc n)
```

Implement projection:

```
proj : forall {n X} → Vec X n → Fin n → X
proj xs i = ?
```

Implement, tabulation, the inverse of projection.

```
tabulate : forall {n X} → (Fin n → X) → Vec X n
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:

For now, I shall just work in Set, but we should remember to break out and live, categorically, later.

Why Set_1?

```
record EndoFunctor (F : Set → Set) : Set₁ where
  field
    map : forall {S T} → (S → T) → F S → F T
open EndoFunctor {{...}} 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 {{...}} 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.

```
record Applicative (F : Set → Set) : Set₁ where
  infixl 2 _⊛_
  field
    pure : forall {X} → X → F X
    _⊛_  : forall {S T} → F (S → T) → F S → F T
  applicativeEndoFunctor : EndoFunctor F
  applicativeEndoFunctor = record {map = _⊛_ ∘ pure}
open Applicative {{...}} public
```

The Applicative F 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:

```
applicativeVec    : forall {n} → Applicative λ X → Vec X n
applicativeVec    = record {pure = vec; _⊛_ = vapp}
endoFunctorVec : forall {n} → EndoFunctor λ X → Vec X n
endoFunctorVec = applicativeEndoFunctor
```

Indeed, the definition of endoFunctorVec already makes use of way *itsEndoFunctor* searches the context and finds applicativeVec.

proj and tabulate turn the vec and vapp applicative into this one.

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

```
applicativeFun : forall {S} → Applicative λ X → S → X
applicativeFun = record
  {pure = λ x s → x              -- also known as K (drop environment)
  ; _⊛_ = λ f a s → f s (a s)    -- also known as S (share environment)
  }
```

Monadic structure induces applicative structure:

```
record Monad (F : Set → Set) : Set₁ where
  field
    return : forall {X} → X → F X
    _>>=_  : forall {S T} → F S → (S → F T) → F T
  monadApplicative : Applicative F
  monadApplicative = record
    {pure = return
    ; _⊛_ = λ ff fs → ff >>= λ f → fs >>= λ s → return (f s)}
open Monad {{...}} public
```

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

```
monadVec : {n : ℕ} → Monad λ X → Vec X n
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.

```
applicativeId : Applicative id
applicativeId = ?
applicativeComp : forall {F G} → Applicative F → Applicative G → Applicative (F ∘ G)
applicativeComp aF aG = ?
```

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
```

```
traversableVec : {n : ℕ} → Traversable λ X → Vec X n
traversableVec = record {traverse = vtr} where
  vtr : forall {n G S T} {{_ : 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
```

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

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
```

I was going to set this as an exercise, but it's mostly instructive in how to override implicit and instance arguments.

Amusingly, we must tell Agda which T is intended when viewing $X \to Y$ as $X \to (\lambda \_ \to Y)\ T$. 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 Σ-types and Other Equipment

Before we go any further, let us establish that the type $\Sigma\ (S : \mathsf{Set})\ (T : S \to \mathsf{Set})$ 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 $\mathsf{fst}\ p : S$ and $\mathsf{snd}\ p : T\ (\mathsf{fst}\ p)$, but I also define V to be a low precedence uncurrying operator, so that $\mathrm{V}\ \lambda\ s\ t \to \ldots$ 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 Σ in the first place.

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

```
data Two : Set where tt ff : Two
```

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

```
_⟨?⟩_ : forall {l} {P : Two → Set l} → P tt → P ff → (b : Two) → P b
(t ⟨?⟩ f) tt = t
(t ⟨?⟩ f) ff = f
```

for we may then define:

```
_+_ : Set → Set → Set
S + T = Σ Two (S ⟨?⟩ 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 Sets, as well as between ordinary values. Σ thus models both choice and pairing in data structures. That is, Σ generalizes binary product to the dependent case, and binary sum to arbitrary arity. I advise calling a Σ-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 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.*

```
_+ℕ_ : ℕ → ℕ → ℕ
x +ℕ y = ?
_×ℕ_ : ℕ → ℕ → ℕ
x ×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.

```
record Normal : Set₁ where
  constructor _◁_
  field
    Shape : Set
    size  : Shape → ℕ
  ⟦_⟧N : Set → Set
  ⟦ S ◁ P ⟧N X = Σ S λ s → Vec X (P 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.

```
VecN : ℕ → Normal
VecN n = One ◁ pure n

ListN : Normal
ListN = ℕ ◁ 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.

```
K₀ : Set → Normal
K₀ A = A ◁ λ _ → 0

IK₀ : Normal
IK₀ = VecN 1
```

Let us construct sums and products of normal functors.

```
_+₀_ : Normal → Normal → Normal
(ShF ◁ szF) +₀ (ShG ◁ szG) = (ShF + ShG) ◁ [ szF , szG ]

_×₀_ : Normal → Normal → Normal
(ShF ◁ szF) ×₀ (ShG ◁ szG) = (ShF × ShG) ◁ λ { (f , g) → szF f +₀ 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.

```
nInj : {X : Set} (F G : Normal) → ⟦ F ⟧N X + ⟦ G ⟧N X → ⟦ F +₀ G ⟧N X
nInj F G (tt , ShF , xs) = (tt , ShF) , xs
nInj F G (ff , ShG , xs) = (ff , ShG) , xs
```

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

```
data _^-1_ {S T : Set} (f : S → T) : T → Set where
  from : (s : S) → f ^-1 f s
```

Let us now show that nInj is surjective.

```
nCase : {X : Set} F G (s : ⟦ F +₀ G ⟧N X) → nInj F G ^-1 s
nCase F G ((tt , ShF) , xs) = from (tt , ShF , xs)
nCase F G ((ff , ShG) , xs) = from (ff , ShG , 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 nInj is injective shortly, once we have suitable equipment to say so.

The inverse of 'nInj' can be computed by nCase thus:

```
nOut : forall {X} (F G : Normal) → ⟦ F +N G ⟧N X → ⟦ F ⟧N X + ⟦ G ⟧N X
nOut F G xs' with nCase F G xs'
nOut F G .(nInj F G xs) | from xs = xs
```

The **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 nCase F G xs' as from xs, we discover that *ipso facto*, xs' is nInj xs. 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 **with** as a syntax for bringing new information to the problem. The usual Burstallian '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.*

```
_++_ : forall {m n X} → Vec X m → Vec X n → Vec X (m +ℕ n)
xs ++ ys = ?
nPair : forall {X} (F G : Normal) → ⟦ F ⟧N X × ⟦ G ⟧N X → ⟦ F ×N G ⟧N X
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.*

```
listNMonoid : {X : Set} → Monoid (⟦ ListN ⟧N X)
listNMonoid = ?
```

We have already seen that the identity functor VecN 1 is Normal, but can we define composition?

```
_∘N_ : Normal → Normal → Normal
F ∘N (ShG / 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:

```
_∘N_ : Normal → Normal → Normal
F ∘N (ShG / szG) = ⟦ F ⟧N 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 traverse, given that $\mathbb{N}$ is a monoid for $+_{\mathsf{N}}$ and that Normal functors are traversable because vectors are.

```
sumMonoid : Monoid ℕ
sumMonoid = record {ε = 0; _•_ = _+ℕ_}
```

```
normalTraversable : (F : Normal) → Traversable ⟦ F ⟧N
normalTraversable F = record
  {traverse = λ {{aG}} f → ^V λ s xs → pure {{aG}} (_ , _ s) ⊛ traverse f xs}
```

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

```
_∘N_ : Normal → Normal → Normal
F ∘N (ShG / szG) = ⟦ F ⟧N ShG / crush {{normalTraversable F}} 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 $\mathbb{N}$:

```
sizeT : forall {F} {{TF : Traversable F}} {X} → F X → ℕ
sizeT = crush (λ _ → 1)
```

Hence, every Traversable functor has a Normal counterpart

```
normalT : forall F {{TF : Traversable F}} → Normal
normalT F = F One / 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:

```
shapeT : forall {F} {{TF : Traversable F}} {X} → F X → F One
shapeT = traverse (λ _ → ⟨⟩)
```

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

```
one : forall {X} → X → ⟦ ListN ⟧N X
one x = 1 , (x , ⟨⟩)
contentsT : forall {F} {{TF : Traversable F}} {X} → F X → ⟦ ListN ⟧N X
contentsT = crush one
```

and then try

```
toNormal : forall {F} {{TF : Traversable F}} {X} → F X → ⟦ normalT F ⟧N X
toNormal fx = BAD (shapeT fx , snd (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 ⟦ normalT F ⟧N. To see this, however, we shall need the capacity to reason equationally, which must wait until the next section. Check this.

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

```
_→N_ : Normal → Normal → Set
F →N G = (s : Shape F) → ⟦ G ⟧N (Fin (size F s))
```

where any such thing determines a natural transformation *from F to G.*

```
nMorph : forall {F G} → F →N G → forall {X} → ⟦ F ⟧N X → ⟦ G ⟧N X
nMorph f (s , xs) 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} → (forall {X} → ⟦ F ⟧N X → ⟦ G ⟧N X) → F →N G
morphN f s = ?
```

Exercise 1.16 (Hancock's tensor) *Let*

```
_⊗_ : Normal → Normal → Normal
(ShF / szF) ⊗ (ShG / szG) = (ShF × ShG) / ^ λ f g → szF f ×ℕ szG g
```

Construct normal morphisms:

```
swap : (F G : Normal) → (F ⊗ G) →N (G ⊗ F)
swap F G x = ?
drop : (F G : Normal) → (F ⊗ G) →N (F ∘N G)
drop F G x = ?
```

Hint: for swap*, you may find you need to build some operations manipulating matrices. Hint: for* 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

Never trust a type theorist who has not changed their mind about equality at least once.

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 α-conversion) was often enough.

In dependently typed languages, it is quite convenient if Vec X $(2 + 2)$ is the same type as Vec X 4, so we often consider types up to the $\alpha\beta$-conversion of the λ-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 η-conversion at some 'negative' types—specifically, function types and record types—where a type-directed and terminating η-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 fst $p \equiv s$ and snd $p = t$, works fine because two non-canonical functions and pairs are equal if and only if their expansions are. But if you want the η-rule for One, you need a cue to notice that $u \equiv v$ when both inhabit 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 *decidable* judgmental equality will always disappont—extensional equality of functions is undecidable, for example. Correspondingly, 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 *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

that s and t are equal. We should ensure that $t \simeq t$ for all t, and that for all P, $s \simeq t \to P\ s \to P\ t$, 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.

The size of equality types is also moot. Agda would allow us to put $s \simeq t$ in Set, however large s and t may be...

```
data _≃_ {l} {X : Set l} (x : X) : X → Set l where
  refl : x ≃ x
infix 1 _≃_
```

We may certainly implement Leibniz's rule.

```
subst : forall {k l} {X : Set k} {s t : X} →
        s ≃ t → (P : X → Set l) → P s → P t
subst refl P p = p
```

The only canonical proof of $s \simeq t$ is 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 abilty to prove useful equations on *open* terms. Moreover, the restriction to the judgmental equality is fundamental to the computational behaviour of our 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,

...but for this pragma, we need $\_\simeq\_\ \{l\}\ \{X\}\ s\ t$: Set l

```
{-# 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 Monoid.

```
record MonoidOK X {{M : Monoid X}} : Set where
  field
    absorbL : (x : X) →     ε • x ≃ x
    absorbR : (x : X) →     x • ε ≃ x
    assoc   : (x y z : X) → (x • y) • z ≃ x • (y • z)
```

Let's check that $+_{\mathbb{N}}$ really gives a monoid.

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

The absorbL law follows by computation, but the other two require inductive proof.

```
_+zero : forall x → x +ℕ zero ≃ x
zero  +zero                 = refl
suc n +zero rewrite n +zero = refl
assoc+ : forall x y z → (x +ℕ y) +ℕ z ≃ x +ℕ (y +ℕ z)
assoc+ 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 $+_{\mathbb{N}}$ is strict, so that computation unfolds. Sadly, an Agda program, seen as a proof document does not show you the subgoal structure. However, we can see that the base case holds computationally and the step case becomes trivial once we have rewritten the goal by the inductive hypothesis (being the type of the structurally recursive call).

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* List X *is a monoid with respect to concatenation, but I want you to have a crack at*

```
listNMonoidOK : {X : Set} → MonoidOK (⟦ ListN ⟧N X)
listNMonoidOK {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* **rewrite** *will not help. Hint 2: use* subst *with a predicate of form* V $\lambda\ n\ xs \to \ldots$*, which will allow you to abstract over separated places with* n *and* 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* $\simeq$ *setup?*

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

```
record MonoidHom {X} {{MX : Monoid X}} {Y} {{MY : Monoid Y}} (f : X → Y) : Set where
  field
    respε  :              f ε ≃ ε
    resp•  : forall x x' → f (x • x') ≃ f x • f x'
```

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

```
fstHom : forall {X} → MonoidHom {⟦ ListN ⟧N X} {ℕ} fst
fstHom = record {respε = refl; resp• = λ _ _ → refl}
```

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

```
record EndoFunctorOK F {{FF : EndoFunctor F}} : Set₁ where
  field
    endoFunctorId : forall {X} →
      map {{FF}} {X} id ≃ id
    endoFunctorCo : forall {R S T} (f : S → T) (g : R → S) →
      map {{FF}} f ∘ map g ≃ map (f ∘ g)
```

However, when we try to show,

```
vecEndoFunctorOK : forall {n} → EndoFunctorOK λ X → Vec X n
vecEndoFunctorOK = record
  {endoFunctorId = { }0
  ; endoFunctorCo = λ f g → { }1
  }
```

we see concrete goals (up to some tidying):

```
?0 : vapp (vec id) ≃ id
?1 : vapp (vec f) ∘ vapp (vec g) ≃ vapp (vec (f ∘ g))
```

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

Some see this as reason enough to abandon decidability of ≡, thence of type-checking.

We can define *pointwise* equality,

```
_≐_ : forall {l} {S : Set l} {T : S → Set l}
      (f g : (x : S) → T x) → Set l
f ≐ g = forall x → f x ≃ g x
infix 1 _≐_
```

which is reflexive but not substitutive.

Now we can at least require:

```
record EndoFunctorOKP F {{FF : EndoFunctor F}} : Set₁ where
  field
    endoFunctorId : forall {X} →
      map {{FF}} {X} id ≐ id
    endoFunctorCo : forall {R S T} (f : S → T) (g : R → S) →
      map {{FF}} f ∘ map g ≐ map (f ∘ g)
```

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

```
vecEndoFunctorOKP : forall {n} → EndoFunctorOKP λ X → Vec X n
vecEndoFunctorOKP = ?
```

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.

```
_=[_⟩_ : forall {l} {X : Set l} (x : X) {y z} → x ≃ y → y ≃ z → x ≃ z
_ =[ refl ⟩ q = q
⟨_]=_ : forall {l} {X : Set l} (x : X) {y z} → y ≃ x → y ≃ z → x ≃ z
_ ⟨ refl ]= q = q
_□ : forall {l} {X : Set l} (x : X) → x ≃ x
x □ = refl
infixr 1 _=[_⟩_ ⟨_]=_ _□
```

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 □ 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 **rewrite**.

```
cong : forall {k l} {X : Set k} {Y : Set l} (f : X → Y) {x y} → x ≃ y → f x ≃ f y
cong f refl = refl
```

Thus armed, let us specify what makes an Applicative acceptable, then show that such a thing is certainly a *Functor*.

I had to η-expand $\circ$ in lieu of subtyping.

```
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}} f ⊛ pure s ≃ pure (f s)
    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}} (_∘_ f) ⊛ pure g ⊛ r
          =⟨ cong (λ x → x ⊛ r) (lawHom (_∘_ f) 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,

```
_⇢_ : 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
```

```
  { respPure = λ x → MonoidHom.respε hf
  ; resp⊛    = MonoidHom.resp • hf
  }
```

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

```
homSum : forall { F G } {{ AF : Applicative F }} {{ AG : Applicative G }} →
         (f : F -:> G) →
         Applicative λ X → F X + G X
homSum {{ AF }} {{ AG }} f = ?
```

Check that your solution obeys the laws.

```
homSumOKP : forall { F G } {{ AF : Applicative F }} {{ AG : Applicative G }} →
            ApplicativeOKP F → ApplicativeOKP G →
            (f : F -:> G) → AppHom f →
            ApplicativeOKP _ {{ homSum f }}
homSumOKP {{ AF }} {{ AG }} FOK GOK f homf = ?
```

Laws for Traversable functors are given thus:

```
record TraversableOKP F {{ TF : Traversable F }} : Set₁ where
  field
    lawId  : forall { X } (xs : F X) → traverse id xs ≃ xs
    lawCo  : forall { G } {{ AG : Applicative G }} { H } {{ AH : Applicative H }}
                    { R S T } (g : S → G T) (h : R → H S) (rs : F R) →
             let EH : EndoFunctor H; EH = applicativeEndoFunctor
             in  map { H } (traverse g) (traverse h rs)
                   ≃
                 traverse {{ TF }} {{ applicativeComp AH AG }} (map { H } g ∘ h) rs
    lawHom : forall { G } {{ AG : Applicative G }} { H } {{ AH : Applicative H }}
                    (h : G -:> H) { S T } (g : S → G T) → AppHom h →
                    (ss : F S) →
                    traverse (h ∘ g) ss ≃ h (traverse g ss)
```

Let us now check the coherence property we needed earlier.

```
lengthContentsSizeShape :
  forall { F } {{ TF : Traversable F }} → TraversableOKP F →
  forall { X } (fx : F X) →
  fst (contentsT fx) ≃ sizeT (shapeT fx)
lengthContentsSizeShape tokF fx =
  fst (contentsT fx)
    ⟨ TraversableOKP.lawHom tokF {{ monoidApplicative }} {{ monoidApplicative }}
        fst one (monoidApplicativeHom fst) fx ⊨
  sizeT fx
    ⟨ TraversableOKP.lawCo tokF {{ monoidApplicative }} {{ applicativeId }}
        (λ _ → 1) (λ _ → ⟨⟩) fx ⊨
  sizeT (shapeT fx) □
```

We may now construct

```
toNormal : forall { F } {{ TF : Traversable F }} → TraversableOKP F →
           forall { X } → F X → ⟦ normalT F ⟧N X
```

```
toNormal tokf fx
  = shapeT fx
  , subst (lengthContentsSizeShape tokf fx) (Vec _) (snd (contentsT fx))
```

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 : Set → Set → Set
Batch X Y = Σ ℕ λ n → Vec X n → 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 Tree (N : Normal) : Set where
  ⟨_⟩ : ⟦ N ⟧N (Tree N) → Tree N
```

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

```
NatT : Normal
NatT = Two / 0 ⟨?⟩ 1

zeroT : Tree NatT
zeroT = ⟨ tt, ⟨⟩ ⟩

sucT : Tree NatT → Tree NatT
sucT n = ⟨ ff, n, ⟨⟩ ⟩
```

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.*

```
NatInd : forall {l} (P : Tree NatT → Set l) →
         P zeroT →
         ((n : Tree NatT) → P n → P (sucT n)) →
         (n : Tree NatT) → P n
NatInd 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.

```
All : forall {l X} (P : X → Set l) {n} → Vec X n → Set l
All P ⟨⟩       = One
All P (x, xs)  = P x × All P xs
```

We then acquire

```
induction : forall (N : Normal) {l} (P : Tree N → Set l) →
  ((s : Shape N) (ts : Vec (Tree N) (size N s)) → All P ts → P ⟨ s, ts ⟩) →
  (t : Tree N) → P t
induction N P p ⟨ s, ts ⟩ = p s ts (hyps ts) where
  hyps : forall {n} (ts : Vec (Tree N) n) → All P ts
  hyps ⟨⟩       = ⟨⟩
  hyps (t, ts)  = induction N P p t, hyps ts
```

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* Zero*, an empty type.*

```
Dec : Set → Set
Dec X = X + (X → Zero)
```

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

```
eq? : (N : Normal) (sheq? : (s s' : Shape N) → Dec (s ≃ s')) →
  (t t' : Tree N) → Dec (t ≃ t')
eq? N 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 ⋆ : Set where
  ι   :          ⋆
  _▷_ : ⋆ → ⋆ → ⋆
infixr 5 _▷_
```

Next, build contexts as snoc-lists.

```
data Cx (X : Set) : Set where
  ε   : Cx X
  _,_ : Cx X → X → Cx X
infixl 4 _,_
```

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

```
data _∈_ (τ : ⋆) : Cx ⋆ → Set where
  zero : forall {Γ}         → τ ∈ Γ , τ
  suc  : forall {Γ σ} → τ ∈ Γ → τ ∈ Γ , σ
infix 3 _∈_
```

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

```
data _⊢_ (Γ : Cx ⋆) : ⋆ → Set where
  var : forall {τ}
        → τ ∈ Γ
        ---------
        → Γ ⊢ τ
  lam : forall {σ τ}
```

```
        → Γ , σ ⊢ τ
        ———————————
        → Γ ⊢ σ ▷ τ
  app : forall {σ τ}
        → Γ ⊢ σ ▷ τ → Γ ⊢ σ
        ———————————————————
        → Γ ⊢ τ
infix 3 ⊢_
```

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.

```
⟦_⟧⋆ : ⋆ → Set
⟦ ι ⟧⋆       = ℕ   -- by way of being nontrivial
⟦ σ ▷ τ ⟧⋆ = ⟦ σ ⟧⋆ → ⟦ τ ⟧⋆
```

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.

```
⟦_⟧Cx : Cx ⋆ → (⋆ → Set) → Set
⟦ ε ⟧Cx     V = One
⟦ Γ , σ ⟧Cx V = ⟦ Γ ⟧Cx V × V σ
⟦_⟧∈ : forall {Γ τ V} → τ ∈ Γ → ⟦ Γ ⟧Cx V → V τ
⟦ zero ⟧∈ (γ , t) = t
⟦ suc i ⟧∈ (γ , s) = ⟦ i ⟧∈ γ
```

Finally, define the meaning of terms.

```
⟦_⟧⊢ : forall {Γ τ} → Γ ⊢ τ → ⟦ Γ ⟧Cx ⟦_⟧⋆ → ⟦ τ ⟧⋆
⟦ var i ⟧⊢   γ = ⟦ i ⟧∈ γ
⟦ lam t ⟧⊢   γ = λ s → ⟦ t ⟧⊢ (γ , s)
⟦ app f s ⟧⊢ γ = ⟦ f ⟧⊢ γ (⟦ s ⟧⊢ γ)
```

2.3 Substitution with a Friendly Fish

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

```
Ren Sub : Cx ⋆ → Cx ⋆ → Set
Ren Γ Δ = forall {τ} → τ ∈ Γ → τ ∈ Δ
Sub Γ Δ = forall {τ} → τ ∈ Γ → Δ ⊢ τ
```

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 ◃▻ operator.

◃▻ is pronounce 'fish', for historical reasons.

```
_◃▻_ : forall {X} → Cx X → List X → Cx X
xz ◃▻ ⟨⟩       = xz
```

```
xz <>< (x , xs) = xz , x <>< xs
infixl 4 _<><_
```

We may then define the *shiftable* simultaneous substitutions from Γ to Δ as type-preserving mappings from the variables in any extension of Γ to terms in the same extension of Δ.

```
Shub : Cx * → Cx * → Set
Shub Γ Δ = forall Ξ → Sub (Γ <>< Ξ) (Δ <>< Ξ)
```

By the computational behaviour of $<><$, a Shub Γ Δ can be used as a Shub (Γ , σ) (Δ , σ), so we can push substitutions under binders very easily.

```
_//_ : forall {Γ Δ} (θ : Shub Γ Δ) {τ} → Γ ⊢ τ → Δ ⊢ τ
θ // var i    = θ <> i
θ // lam t    = lam ((θ o' _ , _ _) // t)
θ // app f s  = app (θ // f) (θ // s)
```

Of course, we shall need to construct some of these joyous shubstitutions. Let us first show that any simultaneous renaming can be made shiftable by iterative weakening.

```
wkr : forall {Γ Δ σ} → Ren Γ Δ → Ren (Γ , σ) (Δ , σ)
wkr r zero     = zero
wkr r (suc i)  = suc (r i)
ren : forall {Γ Δ} → Ren Γ Δ → Shub Γ Δ
ren r <>       = var o' r
ren r (_ , Ξ)  = ren (wkr r) Ξ
```

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

```
wks : forall {Γ Δ σ} → Sub Γ Δ → Sub (Γ , σ) (Δ , σ)
wks s zero     = var zero
wks s (suc i)  = ren suc // s i
sub : forall {Γ Δ} → Sub Γ Δ → Shub Γ Δ
sub s <>       = s
sub s (_ , Ξ)  = sub (wks s) Ξ
```

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.

```
weak : forall {Γ} Ξ → Ren Γ (Γ <>< Ξ)
weak <>       i  = i
weak (_ , Ξ)  i  = weak Ξ (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.

```
lambda : forall {Γ σ τ} →
         ((forall {Ξ} → Γ ‚ σ ≪< Ξ ⊢ σ) → Γ ‚ σ ⊢ τ) →
         Γ ⊢ σ ▷ τ
lambda f = lam (f λ {Ξ} → var (weak Ξ zero))
```

But sadly, the followinf does not typecheck

```
myTest : ℰ ⊢ ι ▷ ι
myTest = lambda λ x → x
```

because the following constraint is not solved:

$$(\mathcal{E} \text{ ‚ } \iota \ll\!< \_Xi\_232\ x) \;=\; (\mathcal{E} \text{ ‚ } \iota) \;:\; \mathsf{Cx}\ \star$$

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:

```
_<>>_ : forall {X} → Cx X → List X → List X
ℰ       <>> ys = ys
(xz ‚ x) <>> ys = xz <>> (x , ys)
```

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

I have discovered a truly appalling proof of this lemma. Fortunately, this margin is too narrow to contain it. See if you can do better.

Exercise 2.1 (reversing lemma) *Show*

```
lem : forall {X} (Δ Γ : Cx X) Ξ →
      Δ <>> ⟨⟩ ≃ Γ <>> Ξ → Γ ≪< Ξ ≃ Δ
lem Δ Γ Ξ 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 refl to solve the instance argument, triggering the tractable version of the unification problem.

```
lambda : forall {Γ σ τ} →
         ((forall {Δ Ξ} {{_ : Δ <>> ⟨⟩ ≃ Γ <>> (σ , Ξ)}} → Δ ⊢ σ) →
           Γ ‚ σ ⊢ τ) →
         Γ ⊢ σ ▷ τ
lambda {Γ} f =
  lam (f λ {Δ Ξ} {{q}} →
       subst (lem Δ Γ (_ , Ξ) q) (λ Γ → Γ ⊢ _) (var (weak Ξ zero)))
myTest : ℰ ⊢ (ι ▷ ι) ▷ (ι ▷ ι)
myTest = lambda λ f → lambda λ x → app f (app f x)
```

2.5 Hereditary Substitution

This section is a structured series of exercises, delivering a $\beta\eta$-long normalization algorithm for our λ-calculus by the method of *hereditary substitution*.

The target type for the algorithm is the following right-nested spine representation of β-normal η-long forms.

```
mutual
  data _⊨_ (Γ : Cx ⋆) : ⋆ → Set where
    lam : forall {σ τ} → Γ , σ ⊨ τ → Γ ⊨ σ ▷ τ
    $_  : forall {τ} → τ ∈ Γ → Γ ⊨* τ → Γ ⊨ ι
  data _⊨*_ (Γ : Cx ⋆) : ⋆ → Set where
    ⟨⟩  : Γ ⊨* ι
    _,_ : forall {σ τ} → Γ ⊨ σ → Γ ⊨* τ → Γ ⊨* σ ▷ τ
infix 3 _⊨_ _⊨*_
infix 3 $_
```

That is $\Gamma \vDash \tau$ is the type of normal forms *in* τ, and $\Gamma \vDash^* \tau$ 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* $\twoheadrightarrow$ *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.*

```
_-_ : forall (Γ : Cx ⋆) {τ} (x : τ ∈ Γ) → Cx ⋆
Γ -↠ x = ?
infixl 4 _-_
_≠_ : forall {Γ σ} (x : σ ∈ Γ) → Ren (Γ -↠ x) Γ
x ≠ 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:

```
⟨_↦_⟩_ : forall {Γ σ τ} → (x : σ ∈ Γ) → Γ -↠ x ⊨ σ →
                          Γ ⊨ τ → Γ -↠ x ⊨ τ
⟨ x ↦ s ⟩ lam t   = lam (⟨ suc x ↦ ? ⟩ t)
⟨ x ↦ s ⟩ y $ ts  = ?
infix 2 ⟨_↦_⟩_
```

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 $\Gamma \twoheadrightarrow x$. 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 -↠ x) → Veq? x (x ≠ 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 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.*

```
mutual
  renNm : forall { Γ Δ τ } → Ren Γ Δ → Γ ⊨ τ → Δ ⊨ τ
  renNm r t = ?
  renSp : forall { Γ Δ τ } → Ren Γ Δ → Γ ⊨* τ → Δ ⊨* τ
  renSp r ss = ?
```

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 β-reduction.*

```
mutual
  ⟨_↦_⟩_ : forall { Γ σ τ } → (x : σ ∈ Γ) → Γ - x ⊨ σ →
                Γ ⊨ τ → Γ - x ⊨ τ
  ⟨ x ↦ s ⟩ t = ?
  ⟨_↦_⟩*_ : forall { Γ σ τ } → (x : σ ∈ Γ) → Γ - x ⊨ σ →
                 Γ ⊨* τ → Γ - x ⊨* τ
  ⟨ x ↦ s ⟩* ts = ?
  $$ : forall { Γ τ } →
           Γ ⊨ τ → Γ ⊨* τ → Γ ⊨ ι
  f $$ ss = ?
infix 3 $$
infix 2 ⟨_↦_⟩_
```

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 (η-expansion for normalize) *If we start implementing normalize, it is easy to get this far:*

```
normalize : forall { Γ τ } → Γ ⊢ τ → Γ ⊨ τ
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 ↦ s′ ⟩ t
```

We can easily push under lam and implement app by hereditary substitution. However, if we encounter a variable, x, we must deliver it in η-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'.

```
try₁ : ℰ ⊨ ((ι ▷ ι) ▷ (ι ▷ ι)) ▷ (ι ▷ ι) ▷ (ι ▷ ι)
try₁ = normalize (lambda λ x → x)
church₂ : forall {τ} → ℰ ⊢ (τ ▷ τ) ▷ τ ▷ τ
church₂ = lambda λ f → lambda λ x → app f (app f x)
try₂ : ℰ ⊨ (ι ▷ ι) ▷ (ι ▷ ι)
try₂ = normalize (app (app church₂ church₂) church₂)
```

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.

```
data Stop (Γ : Cx ⋆) (τ : ⋆) : Set where
  var : τ ∈ Γ → Stop Γ τ
  $ : forall {σ} → Stop Γ (σ ▷ τ) → Γ ⊨ σ → Stop Γ τ
```

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.*

```
renSt : forall {Γ Δ τ} → Ren Γ Δ → Stop Γ τ → Stop Δ τ
renSt r u = ?
stopSp : forall {Γ τ} → Stop Γ τ → Γ ⊨* τ → Γ ⊨ ι
stopSp u ss = ?
```

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.

```
mutual
  Val : Cx ⋆ → ⋆ → Set
  Val Γ τ = Go Γ τ + Stop Γ τ
  Go : Cx ⋆ → ⋆ → Set
  Go Γ ι       = Zero
  Go Γ (σ ▷ τ) = forall {Δ} → Ren Γ Δ → Val Δ σ → Val Δ τ
```

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.*

```
renVal : forall {Γ Δ} τ → Ren Γ Δ → Val Γ τ → Val Δ τ
renVal τ r v = ?
renVals : forall Θ {Γ Δ} → Ren Γ Δ → ⟦ Θ ⟧Cx (Val Γ) → ⟦ Θ ⟧Cx (Val Δ)
renVals Θ r θ = ?
idEnv : forall Γ → ⟦ Γ ⟧Cx (Val Γ)
idEnv Γ = ?
```

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 {Γ σ τ} → Val Γ (σ ▷ τ) → Val Γ σ → Val Γ τ
  apply f s = ?
  quo : forall {Γ} τ → Val Γ τ → Γ ⊨ τ
  quo τ 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 {Γ Δ τ} → Γ ⊢ τ → ⟦ Γ ⟧Cx (Val Δ) → Val Δ τ
eval t γ = ?
```

With all the pieces in place, we get

```
normByEval : forall {Γ τ} → Γ ⊢ τ → Γ ⊨ τ
normByEval {Γ} {τ} t = quo τ (eval t (idEnv Γ))
```

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

```
zero : Γ ⊢ ι
suc  : Γ ⊢ ι → Γ ⊢ ι
rec  : forall {τ} → Γ ⊢ τ → Γ ⊢ (ι ▷ τ ▷ τ)
                  → Γ ⊢ ι → Γ ⊢ τ
```

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.*

```
suc : Γ ⊢ ι → Γ ⊢ ι → Γ ⊢ ι
```

Can you engineer the notion of value and the evaluator so that normByEval *identifies*

```
add zero t       with              t
add s zero       with              s
add (suc s) t    with   suc (add s t)
add s (suc t)    with   suc (add s t)
add (add r s) t  with add r (add s t)
add s t          with         add t s
```

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.

```
K◁ : Set → Con
K◁ A = A ◁ λ _ → Zero

I◁ : Con
I◁ = One ◁ λ _ → One

_+◁_ : Con → Con → Con
(S ◁ P) +◁ (S′ ◁ P′) = (S + S′) ◁ V P ⟨?⟩ P′

_×◁_ : Con → Con → Con
(S ◁ P) ×◁ (S′ ◁ P′) = (S × S′) ◁ V λ s s′ → P s + P′ s′
```

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

```
Σ◁ : (A : Set) (C : A → Con) → Con
Σ◁ A C = (Σ A λ a → Sh (C a)) ◁ V λ a s → Po (C a) s

Π◁ : (A : Set) (C : A → Con) → Con
Π◁ A C = ((a : A) → Sh (C a)) ◁ λ f → Σ A λ a → Po (C a) (f a)

_∘◁_ : Con → Con → Con
(S ◁ P) ∘◁ C = Σ◁ S λ s → Π◁ (P s) λ p → C
```

Exercise 3.1 (containers are endofunctors) *Check that containers yield endofunctors which obey the laws.*

```
conEndoFunctor : {C : Con} → EndoFunctor ⟦ C ⟧◁
conEndoFunctor {S ◁ P} = ?
conEndoFunctorOKP : {C : Con} → EndoFunctorOKP ⟦ C ⟧◁
conEndoFunctorOKP {S ◁ 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.

```
_→◁_ : Con → Con → Set
(S ◁ P) →◁ (S' ◁ P') = Σ (S → S') λ f → (s : S) → P' (f s) → P s
```

The action of a container morphism is thus

```
_/◁_ : forall {C C'} → C →◁ C' → forall {X} → ⟦ C ⟧◁ X → ⟦ C' ⟧◁ X
(to , fro) /◁ (s , k) = to s , k ∘ fro s
```

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 $[\![S \triangleleft P]\!]_\triangleleft X$ 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.*

```
morph◁ : forall {C C'} → (forall {X} → ⟦ C ⟧◁ X → ⟦ C' ⟧◁ X) → C →◁ C'
morph◁ f = ?
```

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

```
(S ◁ P) →◁ C = (s : S) → ⟦ C ⟧◁ (P s)
```

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

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

```
id→◁ : forall {C} → C →◁ C
id→◁ = ?
_∘→◁_ : forall {C D E} → (D →◁ E) → (C →◁ D) → (C →◁ E)
e ∘→◁ d = ?
```

3.3 W-types

The least fixpoint of a container is a $\mathcal{W}$-type—$\mathcal{W}$ for 'well founded'.

```
data W (C : Con) : Set where
  ⟨_⟩ : ⟦ C ⟧◁ (W C) → W C
```

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

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

```
NatW : Set
NatW = W ?
zeroW : NatW
zeroW = ⟨ ? ⟩
sucW : NatW → NatW
sucW n = ⟨ ? ⟩
precW : forall {l} {T : Set l} → T → (NatW → T → T) → NatW → T
precW z s n = ?
addW : NatW → NatW → NatW
addW x y = precW ? ? x
```

How many different implementations of zeroW *can you find? Meanwhile, discover for yourself why an attempt to establish the induction principle is a fool's errand.*

```
indW : forall {l} (P : NatW → Set l) →
       P zeroW →
       ((n : NatW) → P n → P (sucW n)) →
       (n : NatW) → P n
indW P z s n = ?
```

A useful deployment of the $\mathcal{W}$-type is to define the free monad for a container.

```
_*_ : Con → Set → Set
C * X = W (K◁ X +◁ C)
```

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

```
freeMonad : (C : Con) → Monad (_* C)
freeMonad C = ?
```

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

```
_*◁ : Con → Con
_*◁ C = ?
```

and exhibit an isomorphism

$$C * X \cong \llbracket\ C\ {}^{*\triangleleft}\ \rrbracket_{\triangleleft}\ X$$

Exercise 3.8 (general recursion) *Define the monadic computation which performs one command-response interaction:*

```
call : forall {C} → (s : Sh C) → C * Po C s
call s = ?
```

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

```
Π⊥ : (S : Set) (T : S → Set) → Set
Π⊥ S T = (s : S) → (S ◁ T) * 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 T} → ℕ → Π⊥ S T → (s : S) → T s + One
gas n f s = ?
```

Feel free to implement reduction for the untyped λ-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

$$[\![S \lhd P]\!]_\lhd X = \Sigma\ S\ \lambda\ s \to P\ s \to X$$

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:

$$[\![S \lhd P]\!]_\lhd X = \sum_{s:S} X^{(Ps)}$$

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 : Set) (x : X) → Set
X − x = Σ X λ x' → x' ≃ x → 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.

```
∂ : Con → Con
∂ (S ◁ P) = Σ S P ◁ ∨ λ s p → 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'.

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

```
plug : forall { C } → ((s : Sh C) (p p' : Po C s) → Dec (p ≃ p')) →
       (∂ C ×◁ I◁) →◁ C
plug { C } poeq? = ?
```

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

$$
\begin{array}{rcl}
\partial\,(\mathsf{K}_\triangleleft\ A) & \cong & \mathsf{K}_\triangleleft\ \mathsf{Zero} \\
\partial\ I & \cong & \mathsf{K}_\triangleleft\ \mathsf{One} \\
\partial\,(C +_\triangleleft D) & \cong & \partial\ C +_\triangleleft \partial\ D \\
\partial\,(C \times_\triangleleft D) & \cong & (\partial\ C \times_\triangleleft D) +_\triangleleft (C \times_\triangleleft \partial\ D) \\
\partial\,(C \circ_\triangleleft D) & \cong & (\partial\ C \circ_\triangleleft D) \times_\triangleleft \partial\ D
\end{array}
$$

What is $\partial\,(C\ {}^{*_\triangleleft})$ *?*

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.

```
record _▷_ (I J : Set) : Set₁ where
  constructor _◁_$_
  field
    Shx : J       → Set
    Pox : (j : J) → Shx j → Set
    rix : (j : J) (s : Shx j) (p : Pox j s) → I
  ⟦_⟧ᵢ : (I → Set) → (J → Set)
  ⟦_⟧ᵢ X j = Σ (Shx j) λ s → (p : Pox j s) → X (rix j s p)
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 $\mathsf{Sh}\ j$ and a position family $\mathsf{Po}\ j$, 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*

```
_-:→_ : forall {k l} {I : Set k} → (I → Set l) → (I → Set l) → Set (lmax l k)
X -:→ Y = forall i → X i → Y i
```

```
ixMap : forall {I J} {C : I ▷ J} {X Y} → (X -:→ Y) → ⟦ C ⟧ᵢ X -:→ ⟦ C ⟧ᵢ Y
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

```
data ITree {J : Set} (C : J ▷ J) (j : J) : Set where
  ⟨_⟩ : ⟦ C ⟧ᵢ (ITree C) j → ITree C j
```

The natural numbers are a friendly, if degenerate example.

```
NatC : One ▷ One
NatC = (λ _ → Two) ◁ (λ _ → Zero ⟨?⟩ One) $ _

zeroC : ITree NatC ⟨⟩
zeroC = ⟨ tt, magic ⟩

sucC : ITree NatC ⟨⟩ → ITree NatC ⟨⟩
sucC n = ⟨ ff, pure n ⟩
```

This is just the indexed version of the $\mathcal{W}$-type, so the same issue with extensionality arises.

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

```
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.

```
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 λ-calculus) *Define the simply typed λ-terms as Petersson-Synek trees.*

```
STLC : (Cx ⋆ × ⋆) ▷ (Cx ⋆ × ⋆)
STLC = ?
```

Implement the constructors.

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*

```
idIx : forall {I} → I ▷ I
idIx = ?
```

such that

$$[\![\ \mathsf{idIx}\ ]\!]_{\mathsf{i}}\ X\ i \cong X\ i$$

Similarly, construct the composition

```
coIx : forall {I J K} → J ▷ K → I ▷ J → I ▷ K
coIx C C' = ?
```

such that

$$[\![\ \mathsf{coIx}\ C\ C'\ ]\!]_{\mathsf{i}}\ X\ k \cong [\![\ C\ ]\!]_{\mathsf{i}}\ ([\![\ C'\ ]\!]_{\mathsf{i}}\ X)\ k$$

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.

My motivation for level polymorphism will appear in due course.

```
data Desc {l} (I : Set l) : Set (lsuc l) where
  var   : I → Desc I
  σ π   : (A : Set l) (D : A → Desc I) → Desc I
  _×D_  : Desc I → Desc I → Desc I
  κ     : Set l → Desc I
infixr 4 _×D_
```

which admit a direct interpretation as follows

```
[[_]]D : forall {l} {I : Set l} → Desc I → (I → Set l) → Set l
[[ var i ]]D      X = X i
[[ σ A D ]]D      X = Σ A λ a → [[ D a ]]D X
[[ π A D ]]D      X = (a : A) → [[ D a ]]D X
[[ D ×D E ]]D X = [[ D ]]D X × [[ E ]]D X
[[ κ A ]]D        X = A
```

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

```
ixConDesc : forall {I J} → I ▷ J → J → Desc I
ixConDesc (S ◁ P $ r) j = σ (S j) λ s → π (P j s) λ p → var (r j s p)
```

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

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

```
DSh : {I : Set} → Desc I → Set
DSh D = ?
DPo : forall {I} (D : Desc I) → DSh D → Set
DPo D s = ?
Dri : forall {I} (D : Desc I) (s : DSh D) → DPo D s → I
Dri D s p = ?
```

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

```
descIxCon : forall {I J} → (J → Desc I) → I ▷ J
descIxCon F = (DSh ∘ F) ◁ (DPo ∘ F) $ (Dri ∘ F)
```

Exhibit the isomorphism

$$[\![\ \mathsf{descIxCon}\ F\ ]\!]_{\mathsf{i}}\ X\ j \cong [\![\ F\ j\ ]\!]_{\mathsf{D}}\ X$$

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

```
vecD : Set → ℕ → Desc ℕ
vecD X n =
  σ Two (    κ (n ≃ zero)
        ⟨?⟩ σ ℕ λ k → κ X ×ᴰ var k ×ᴰ κ (n ≃ suc k)
        )
```

The choice of constructors becomes a σ-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.

```
vecD : Set → ℕ → Desc ℕ
vecD X zero     = κ One
vecD X (suc n) = κ X ×ᴰ 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.

```
data Data {l} {J : Set l} (F : J → Desc J) (j : J) : Set l where
  ⟨_⟩ : ⟦ F j ⟧ᴰ (Data F) → Data F j
```

For example, let us once again construct vectors.

```
vnil : forall {X} → Data (vecD X) zero
vnil = ⟨ ⟨⟩ ⟩
vcons : forall {X n} → X → Data (vecD X) n → Data (vecD X) (suc n)
vcons x xs = ⟨ x , xs ⟩
```

Exercise 4.5 (something like 'levitation') *Construct a family of descriptions which describes a type like* Desc*. As Agda is not natively cumulative, you will need to shunt types up through the* Set l *hierarchy by hand, with this gadget:*

```
record ⇑ {l} (X : Set l) : Set (lsuc l) where
  constructor ↑
  field
    ↓ : X
open ⇑ public
```

Now implement

```
DescD : forall {l} (I : Set l) → One {lsuc l} → Desc (One {lsuc l})
DescD {l} I _ = ?
```

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

```
desc : forall {l} {I : Set l} → Data (DescD I) ⟨⟩ → Desc I
desc D = ?
```

We could, if we choose, work entirely with described datatypes. Perhaps, in some future programming language, the external Desc I type will be identified with the internal Data (DescD I) $\langle\rangle$ so that 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 ▷ J) (X : I → Set) → Σ I X ▷ Σ J (⟦ C ⟧ᵢ X)
Everywhere (S ◁ P $ r) X
  = (λ _ → One)
  ◁ (λ {(j , s , k) _ → P j s})
  $ (λ {(j , s , k) _ p → r j s p , k p})
```

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 ▷ J) (X : I → Set) jc →
  ⟦ Everywhere C X ⟧ᵢ (λ _ → One) jc
allTrivial C X _ = ⟨⟩ , λ p → ⟨⟩
```

If you think of simply typed λ-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 ▷ J) (X : I → Set) → Σ I X ▷ Σ J (⟦ C ⟧ᵢ X)
Somewhere (S ◁ P $ r) X
  = ?
  ◁ ?
  $ ?
```

Check that the impossible predicate cannot hold somewhere.

```
noMagic : forall {I J} (C : I ▷ J) (X : I → Set) jc →
  ⟦ Somewhere C X ⟧ᵢ (λ _ → Zero) jc → Zero
noMagic C X _ (p , m) = ?
```

For simply typed λ-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 ▷ J) (X : I → Set) jc {Q R} →
         ⟦ Everywhere C X ⟧ᵢ Q jc → ⟦ Somewhere C X ⟧ᵢ R jc →
         ⟦ Somewhere C X ⟧ᵢ (λ ix → Q ix × R ix) jc
lookup C X jc 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..

```
treeInd : forall {I} (C : I ▷ I) (P : Σ I (ITree C) → Set) →
          ([[ Everywhere C (ITree C) ]]ᵢ P ⇢
           (V λ i ts → P (i , ⟨ ts ⟩))) →
          (i : I) (t : ITree C i) → P (i , t)
treeInd C P m i ⟨ s , k ⟩ = m (i , s , k) (⟨⟩ , λ p → treeInd C P m _ (k p))
```

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* ITree C *(weakly) initial for* $[\![\ C\ ]\!]_{\mathsf{i}}$. *We can compute any* P *from a* ITree C, *given a* C*-algebra for* P.

```
treeFold : forall {I} (C : I ▷ I) (P : I → Set) →
           ([[ C ]]ᵢ P ⇢ P) →
           (ITree C ⇢ P)
treeFold C P m i ⟨ s , k ⟩ = m i (s , λ p → treeFold C P m _ (k p))
```

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

```
Children : forall {I} (C : I ▷ I) → Σ I (ITree C) ▷ Σ I (ITree C)
Children C = Colx ? (Everywhere C (ITree C))
```

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

```
treeFoldInd : forall {I} (C : I ▷ I) P →
              ([[ Children C ]]ᵢ P ⇢ P) →
              forall it → P it
treeFoldInd C P m (i , t) = treeFold (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 Desc and Data, too.

Exercise 4.9 (Everywhere **and** Somewhere **for** 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.*

```
EverywhereD SomewhereD : {I : Set} (D : Desc I) (X : I → Set) →
                         [[ D ]]ᴅ X → Desc (Σ I X)
EverywhereD D X xs = ?
SomewhereD D X xs = ?
```

Now construct

```
dataInd : forall {I : Set} (F : I → Desc I) (P : Σ I (Data F) → Set) →
  ((i : I) (ds : [[ F i ]]ᴅ (Data F)) →
   [[ EverywhereD (F i) (Data F) ds ]]ᴅ P → P (i , ⟨ ds ⟩)) →
  forall i d → P (i , d)
dataInd 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!

```
vecNodeIx : (One + ℕ) ▷ ℕ
vecNodeIx = descIxCon {J = ℕ} λ
  {zero     → κ One
  ; (suc n) → var (tt, ⟨⟩) ×D var (ff, n)
  }
```

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

```
vecIx : One ▷ ℕ
vecIx = ?
```

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

```
μIx : forall {I J} → (I + J) ▷ J → I ▷ J
μIx {I} {J} F = (ITree F' ∘ _, _ ff) ◁ (P' ∘ _, _ ff) $ (r' ∘ _, _ ff) 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) ▷ (I + J)
F = (V (λ i → One)    ⟨?⟩ ShIx F)
  ◁ (V (λ _ _ → Zero) ⟨?⟩ PoIx F)
  $ (V (λ t s ())     ⟨?⟩ riIx F)
```

and then choose to start with (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) → ITree F x → Set
P (tt, i) _       = One
P (ff, j) ⟨ s, k ⟩ = Σ (PoIx F j s) λ p → P (riIx 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 : ITree F x) → P x t → I
r (tt, i) _       _       = i
r (ff, j) ⟨ s, k ⟩ (p, ps) = r _ (k p) ps
```

Let us check that this recipe cooks the vectors.

```
vecIx : One ▷ ℕ
vecIx = μIx vecNodeIx
```

```
Vec : Set → ℕ → Set
Vec X = ⟦ vecIx ⟧ᵢ (λ _ → X)
vnil : forall {X} → Vec X zero
vnil = ⟨ (⟨⟩, λ ()) ⟩, (ᵛ λ ())
vcons : forall {X n} → X → Vec X n → Vec X (suc n)
vcons x (s, k)
  = ⟨ _, (λ {(tt, _) → ⟨ (_, λ ()) ⟩; (ff, _) → s}) ⟩
  ,       (λ {((tt, _), _) → x       ; ((ff, _), p) → k p})
```

4.6 Adding fixpoints to Desc

We can extend descriptions to include a fixpoint operator:

```
data Desc (I : Set) : Set₁ where
  var   : I → Desc I
  σ π   : (A : Set) (D : A → Desc I) → Desc I
  _×ᴰ_  : Desc I → Desc I → Desc I
  κ     : Set → Desc I
  μ     : (J : Set) → (J → Desc (I + J)) → J → Desc I
```

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

```
mutual
  ⟦_⟧ᴰ : forall {I} → Desc I → (I → Set) → Set
  ⟦ var i ⟧ᴰ      X = X i
  ⟦ σ A D ⟧ᴰ      X = Σ A λ a → ⟦ D a ⟧ᴰ X
  ⟦ π A D ⟧ᴰ      X = (a : A) → ⟦ D a ⟧ᴰ X
  ⟦ D ×ᴰ E ⟧ᴰ     X = ⟦ D ⟧ᴰ X × ⟦ E ⟧ᴰ X
  ⟦ κ A ⟧ᴰ        X = A
  ⟦ μ J F j ⟧ᴰ    X = Data F X j
  data Data {I J} (F : J → Desc (I + J)) (X : I → Set) (j : J) : Set where
    ⟨_⟩ : ⟦ F j ⟧ᴰ (ᵛ X ⟨?⟩ Data F X) → Data F X j
```

Indeed, Desc Zero now does quite a good job of reflecting Set, except that the domains of σ and π 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 : forall {I J} → I ▷ J → I ▷ (J × I)
𝒥 (S ◁ P $ r)
  = (λ {(j , i) → Σ (S j) λ s → r j s ⁻¹ i})
  ◁ (λ {(j , . (r j s p)) (s , from p) → P j s − p})
  $ (λ {(j , . (r j s p)) (s , from p) (p′ , _) → 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.*

Einstein's summation convention might be useful to infer the choice and placement of quantifiers.

```
plugIx : forall {I J} (C : I ▷ J) →
  ((j : J) (s : ShIx C j) (p p′ : PoIx C j s) → Dec (p ≃ p′)) →
  forall {i j X} → ⟦ 𝒥 C ⟧ᵢ X (j , i) → X i → ⟦ C ⟧ᵢ X j
plugIx C eq? jx x = ?
```

Exercise 4.12 (the Zipper) *For a given* `C : I ▷ I`*, construct the indexed container* `Zipper C : (I × I) ▷ (I × I)` *such that* `ITree (Zipper C) (ir, ih)` *represents a one* `ih`*-hole context in a* `ITree C ir`*, represented as a sequence of hole-to-root layers.*

```
Zipper : forall {I} → I ▷ I → (I × I) ▷ (I × I)
Zipper C = ?
```

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

```
zipOut : forall {I} (C : I ▷ I) {ir ih} →
         ((i : I) (s : ShIx C i) (p p′ : PoIx C i s) → Dec (p ≃ p′)) →
         ITree (Zipper C) (ir , ih) → ITree C ih → ITree C ir
zipOut C eq? cz t = ?
```

Exercise 4.13 (differentiating Desc) *The notion corresponding to* $\mathcal{J}$ *for descriptions is* ∇*, computing a 'vector' of partial derivatives. Define it symbolically.*

"grad"

Symbolic differentiation is the first example of a pattern matching program in my father's thesis (1970).

```
∇ : {I : Set} → Desc I → I → Desc I
∇ D h = ?
```

Hence construct suitable zippering equipment for `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:

```
σ I λ i → ∇ D i ×ᴰ var i
```

I have not yet found a meaning for *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 : (I → Set) → (J → Set)
  ⟦_⟧R X j = Σ (Σ S λ s → q s ≃ j) (∀ λ s _ → (p : P s) → X (r s p))
Plain = Roman.Plain
⟦_⟧R = Roman.⟦_⟧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.

```
FromRoman : forall {I J} → Roman I J → I ▷ J
FromRoman (SPqr S P q r)
  = (λ j → Σ S λ s → q s ≃ j)
  ◁ (λ j → P ∘ fst)
  $ (λ f → r ∘ fst)
onTheNose : forall {I J} (C : Roman I J) → ⟦ C ⟧R ≃ ⟦ FromRoman C ⟧ⁱ
onTheNose C = 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.*

```
ToRoman : forall {I J} → I ▷ J → Roman I J
ToRoman {I} {J} (S ◁ P $ r) = ?
toRoman   : forall {I J} (C : I ▷ J) →
            forall {X j} → ⟦ C ⟧ⁱ X j → ⟦ ToRoman C ⟧R X j
toRoman C xs = ?
fromRoman : forall {I J} (C : I ▷ J) →
            forall {X j} → ⟦ ToRoman C ⟧R X j → ⟦ C ⟧ⁱ X j
fromRoman C xs = ?
toAndFromRoman :
  forall {I J} (C : I ▷ J) {X j}
  → (forall xs →
       toRoman C {X} {j} (fromRoman C {X} {j} xs) ≃ xs)
  × (forall xs → fromRoman C {X} {j} (toRoman C {X} {j} xs) ≃ xs)
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.

```
data RomanData {I} (C : Roman I I) : I → Set where
  _,_ : (s : Roman.S C) →
        ((p : Roman.P C s) → RomanData C (Roman.r C s p)) →
        RomanData C (Roman.q C s)
```

I could have just taken the fixpoint of the interpretation, but I wanted to emphasize that the role of 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 RomanData type looks a lot like a $\mathcal{W}$-type, albeit festooned with equations. Let us show that it is exactly that.

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

```
ideology : forall {I} (C : Roman I I) →
           I → W (Plain C) → W (Plain C ×◁ K◁ I)
ideology C i t = ?
```

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

```
phenomenology : forall {I} (C : Roman I I) →
                W (Plain C) → W (Plain C ×◁ K◁ I)
phenomenology C t = ?
```

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

```
RomanW : forall {I} → Roman I I → I → Set
RomanW C i = Σ (W (Plain C)) λ t → phenomenology C t ≃ ideology C i t
```

Now, check that you can extract RomanData from RomanW.

```
fromRomanW : forall {I} (C : Roman I I) {i} → RomanW C i → RomanData C i
fromRomanW C (t , good) = ?
```

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

```
postulate
  extensionality : forall {S : Set} {T : S → Set} (f g : (s : S) → T s) →
                   ((s : S) → f s ≃ g s) → f ≃ g
```

construct

```
toRomanW : forall {I} (C : Roman I I) {i} → RomanData C i → RomanW C i
toRomanW C t = ?
```

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'.

```
data _** {I : Set} (R : I × I → Set) : I × I → Set where
  ⟨⟩   : {i : I}     → (R **) (i , i)
  _,_ : {i j k : I} → R (i , j) → (R **) (j , k) → (R **) (i , k)
infix 1 _**
```

You can construct the natural numbers as an instance.

```
NAT : Set
NAT = (Loop **) _ where Loop : One × One → Set; Loop _ = One
```

Exercise 4.16 (further constructions with **) *Using no recursive types other than **, 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*

```
one** : forall {I} {R : I × I → Set} → R ⇢ (R **)
one** r = ?
join** : forall {I} {R : I × I → Set} → ((R **) **) ⇢ (R **)
join** rss = ?
```

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

$$X \to \mathsf{Set}$$

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

$$\Sigma\ \mathsf{Set}\ \lambda\ I \to I \to X$$

so we have a family of Xs 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*.

```
Pow : Set₁ → Set₁
Pow X = X → Set
Fam : Set₁ → Set₁
Fam X = Σ Set λ I → I → X
```

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

```
p2f : (Pow ∘ ⇑) ⇢ (Fam ∘ ⇑)
p2f X P = ?
f2p : (Fam ∘ ⇑) ⇢ (Pow ∘ ⇑)
f2p X F = ?
```

Exercise 4.19 (functoriality of Pow and Fam) *Equip Pow with a contravariant functorial action and Fam with a covariant functorial action.*

```
$P_ : forall {I J} → (J → I) → Pow I → Pow J
f $P P = ?
$F_ : forall {I J} → (I → J) → Fam I → Fam J
f $F F = ?
```

Fam 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 Pow Set in place of Fam Set. It is good to get used to recognizing when concepts are related just by exchanging Fam and Pow.

Modulo currying and λ-lifting of parameters, the distinction between Roman $I\ J$ and our Hancock-style $I \triangleright J$ 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.

```
ROMAN : Set → Set → Set₁
ROMAN I J = Σ (Fam (⇑ J)) λ {(S , q) → S → Fam (⇑ I)}
HANCOCK : Set → Set → Set₁
HANCOCK I J = Σ (Pow (⇑ J)) λ S → Σ J (S ∘ ↑) → Fam (⇑ I)
```

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

```
NOTTINGHAM : Set → Set → Set₁
NOTTINGHAM I J = Σ (Pow (⇑ J)) λ S → Σ J (S ∘ ↑) → Pow (⇑ I)
```

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

```
NSh : J → Set
NPo : (j : J) → NSh j → I → Set
```

For HANCOCK and NOTTINGHAM, we can abstract the whole construction over J, obtaining:

```
HANCOCK : Set → Set → Set₁
HANCOCK     I J = J → Fam (Fam (⇑ I))
NOTTINGHAM : Set → Set → Set₁
NOTTINGHAM I J = J → Fam (Pow (⇑ I))
```

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

```
HANCOCK I J = J → Fam (Fam (⇑ I))
ROMAN   I J = Fam (⇑ J × Fam (⇑ I))
```

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 : Pow ⇢ Fam
p2f X P = ?
f2p : Fam ⇢ Pow
f2p X F = ?
```

In our study of datatypes so far, we have been constructing inductively defined inhabitants of Pow $(\Uparrow I)$. Let us now perform our own flip and consider inductive definition in 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 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 Σ- and Π-types. Finite sums and products of finite things are finite, so we can compute their sizes.

```
sum prod : (n : ℕ) → (Fin n → ℕ) → ℕ
sum  zero    _ = 0
sum  (suc n) f = f zero +ℕ sum n (f ∘ suc)
prod zero    _ = 1
prod (suc n) f = f zero ×ℕ sum n (f ∘ suc)
```

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

```
data FTy : Set where
  fin   : ℕ → FTy
  σ π : (S : FTy) (T : Fin ? → FTy) → FTy
```

I was not quite able to finish the definition, because I could not give the domain of T. Intutively, 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 Fin ?, but to make further progress, we need to know the *size* of S. Intuitively, it is easy to compute the size of an FTy: the base case is direct; the structural cases are captured by sum and prod. The trouble is that we cannot wait until after dclaring 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 FTy and its size function, #, *simultaneously*.

```
mutual
  data FTy : Set where
    fin   : ℕ → FTy
    σ π : (S : FTy) (T : Fin (# S) → FTy) → FTy
  # : FTy → ℕ
  # (fin n)   = n
  # (σ S T) = sum  (# S) λ s → # (T s)
  # (π S T) = prod (# S) λ s → # (T s)
```

For example, if we define the forgetful map from Fin back to $\mathbb{N}$,

```
fog : forall {n} → Fin n → ℕ
fog zero    = zero
fog (suc i) = suc (fog i)
```

in honour of Gauss we can check that

$$\#\ (\sigma\ (\mathsf{fin}\ 101)\ \lambda\ s \to \mathsf{fin}\ (\mathsf{fog}\ s))\ =\ 5050$$

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

```
mutual
  data FTy : Set where
    fin   : ℕ → FTy
    σ π   : (S : FTy) (T : FEl S → FTy) → FTy
  FEl : FTy → Set
  FEl (fin n)   = Fin n
  FEl (σ S T)   = Σ (FEl S) λ s → FEl (T s)
  FEl (π S T)   = (s : FEl S) → FEl (T s)
```

Now, what has happened? We have $(\mathsf{FTy}, \mathsf{FEl})$: Fam Set, picking out a subset of Set by choosing names for them in FTy. But FTy is small enough to be a 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, *ok can be inferred trivially.*

```
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 Desc, but just describing the right-nested Σ-types.

```
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
```

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 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.*

```
sizeRR : (R : RecR) → ⟦ R ⟧RR → ℕ
sizeRR R r = ?
TyRR : (R : RecR) (r : ⟦ R ⟧RR) → Fin (sizeRR R r) → Set
TyRR R r i = ?
vaRR : (R : RecR) (r : ⟦ R ⟧RR) (i : Fin (sizeRR R r)) → TyRR R r i
vaRR R r i = ?
```

Of course, we could enforce uniformity of length by indexing. But a bigger problem with 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
  data RecL : Set₁ where
    ℰ   : RecL
    _,_ : (R : RecL) (A : ⟦ R ⟧RL → Set) → RecL
⟦_⟧RL : RecL → Set
⟦ ℰ ⟧RL       = One
⟦ R , A ⟧RL = Σ ⟦ R ⟧RL A
```

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

```
sizeRL : RecL → ℕ
sizeRL R = ?
TyRL : (R : RecL) → Fin (sizeRL R) → ⟦ R ⟧RL → Set
TyRL R i = ?
vaRL : (R : RecL) (i : Fin (sizeRL R)) (r : ⟦ R ⟧RL) → TyRL R i r
vaRL R i = ?
```

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

```
TruncRL : (R : RecL) → Fin (sizeRL R) → RecL
TruncRL R i = ?
truncRL : (R : RecL) (i : Fin (sizeRL R)) → ⟦ R ⟧RL → ⟦ TruncRL R i ⟧RL
truncRL R i = ?
```

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.

Why is Manifest not an Agda record?

```
data Manifest {A : Set} : A → Set where
  ⟨_⟩ : (a : A) → 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.

```
mutual
  data RecM : ℕ → Set₁ where
    ℰ      : RecM zero
    _,_    : {n : ℕ} (R : RecM n) (A : ⟦ R ⟧RM → Set) → RecM (suc n)
    _,_∋_  : {n : ℕ} (R : RecM n) (A : ⟦ R ⟧RM → Set)
                     (a : (r : ⟦ R ⟧RM) → A r) → RecM (suc n)
  ⟦_⟧RM : {n : ℕ} → RecM n → Set
  ⟦ ℰ ⟧RM          = One
  ⟦ R , A ⟧RM      = Σ ⟦ R ⟧RM A
  ⟦ R , A ∋ a ⟧RM  = Σ ⟦ R ⟧RM (Manifest ∘ a)
```

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

```
TyRM : {n : ℕ} (R : RecM n) → Fin n → ⟦ R ⟧RM → Set
TyRM R i = ?
vaRM : {n : ℕ} (R : RecM n) (i : Fin n) (r : ⟦ R ⟧RM) → TyRM R i r
vaRM R i = ?
```

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*

```
mutual
  data REx : {n m : ℕ} → RecM n → RecM m → Set₁ where
    ℰ : REx ℰ ℰ
  rfog : forall {n m} {R : RecM n} {R' : RecM m} (X : REx R R') →
         ⟦ R' ⟧RM → ⟦ R ⟧RM
  rfog ℰ ⟨⟩ = ⟨⟩
```

describes evidence REx R R' that R' is an extension of R, interpreted by rfog as a map from $\llbracket R' \rrbracket_{RM}$ back to $\llbracket R \rrbracket_{RM}$. Unfortunately, it captures only the fact that the empty record extends itself. Extend 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

  ⟦_⟧TU : TU → Set
  ⟦ Zero′          ⟧TU = Zero
  ⟦ One′           ⟧TU = One
  ⟦ Two′           ⟧TU = Two
  ⟦ Σ′ S T         ⟧TU = Σ ⟦ S ⟧TU λ s → ⟦ T s ⟧TU
  ⟦ Π′ S T         ⟧TU = (s : ⟦ S ⟧TU) → ⟦ T s ⟧TU
  ⟦ Tree′ I F i    ⟧TU = ITree
    ( (λ i → ⟦ fst (F i) ⟧TU)
    ◁ (λ i s → ⟦ fst (snd (F i) s) ⟧TU)
    $ (λ i s p → snd (snd (F i) s) p)
    ) i
```

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
    Π′ : (S : NU X) (T : ⟦ S ⟧NU → NU X) → NU X

  ⟦_⟧NU : forall {X} → NU X → Set
  ⟦_⟧NU {U , El} U′       = U
  ⟦_⟧NU {U , El} (El′ T)  = El T
  ⟦ Nat′    ⟧NU           = ℕ
  ⟦ Π′ S T  ⟧NU           = (s : ⟦ S ⟧NU) → ⟦ T s ⟧NU
```

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.

```
EMPTY : Fam Set
EMPTY = Zero, λ ()

LEVEL : ℕ → Fam Set
LEVEL zero     = NU EMPTY, ⟦_⟧NU
LEVEL (suc n)  = NU (LEVEL n), ⟦_⟧NU
```

This hierarchy is explicitly cumulative: El′ embeds types upward without changing their meaning. One consequence is that we have a redundancy of representation:

Exercise 5.8 ($\mathbb{N} \to \mathbb{N}$) *Find five names for* $\mathbb{N} \to \mathbb{N}$ *in* fst (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 $\mathbb{N} \to \mathbb{N}$ from LEVEL *0*, because LEVEL *1* has a perfectly good version. This time, we parametrize the universe by a de Bruijn indexed collection of the previous universes.

```
mutual
  data HU {n} (U : Fin n → Set) : Set where
    U′   : Fin n → HU U
    Nat′ : HU U
    Π′   : (S : HU U) (T : ⟦ S ⟧HU → HU U) → HU U

  ⟦_⟧HU : forall {n} {U : Fin n → Set} → HU U → Set
  ⟦_⟧HU {U = U} (U′ i)  = U i
  ⟦ Nat′ ⟧HU            = ℕ
  ⟦ Π′ S T ⟧HU          = (s : ⟦ S ⟧HU) → ⟦ T s ⟧HU
```

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

```
HPREDS : (n : ℕ) → Fin n → Set
HPREDS zero     ()
HPREDS (suc n) zero     = HU (HPREDS n)
HPREDS (suc n) (suc i)  = HPREDS n i

HSET : ℕ → Set
HSET n = HU (HPREDS n)
```

Note that HSET n is indeed ⟦ U′ zero ⟧HU at level 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?

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?*

```
Cumu : (n : ℕ) (T : HSET n) → HSET (suc n)
Cumu n T = ?
```

5.4 Encoding Induction-Recursion

So far, we have been making **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,

```
mutual    -- rejected
  data VV : Set where
    V′ : VV
    Π′ : (S : VV) (T : ⟦ S ⟧VV → VV) → VV
  ⟦_⟧VV : VV → Set
  ⟦ V′ ⟧VV       = VV
  ⟦ Π′ S T ⟧VV = (s : ⟦ S ⟧VV) → ⟦ T s ⟧VV
```

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.

```
data DS (I J : Set₁) : Set₁ where
  ι : J → DS I J                                    -- no more fields
  σ : (S : Set) (T : S       → DS I J) → DS I J     -- ordinary field
  δ : (H : Set) (T : (H → I) → DS I J) → DS I J     -- child field
```

We interpret a DS I J as a functor from Fam I to Fam J: I build the components separately for readability.

```
⟦_⟧DS : forall {I J} → DS I J → Fam I → Fam J
⟦ ι j     ⟧DS Xxi
  = One
  , λ { ⟨⟩      → j }
⟦ σ S T ⟧DS Xxi
  = (Σ S λ s → fst (⟦ T s ⟧DS Xxi))
  , λ { (s , t)  → snd (⟦ T s ⟧DS Xxi) t }
⟦ δ H T ⟧DS (X , xi)
  = (Σ (H → X) λ hx → fst (⟦ T (xi ∘ hx) ⟧DS (X , xi)))
  , λ { (hx , t) → snd (⟦ T (xi ∘ hx) ⟧DS (X , xi)) t }
```

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 $H \to I$ which gives access to the interpretation of the children. Once we plug in a specific (X , xi) : Fam I, we represent the field by the *small* function space hx : $H \to X$, then the composition $xi \circ hx$ tells us how to compute the *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*

```
idDS : {I : Set₁} → DS I I
idDS = ?
```

such that

$$[\![\ \mathsf{idDS}\ ]\!]_{\mathsf{DS}} \cong \mathsf{id}$$

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

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

```
mutual  -- fails positivity check and termination check
  data DataDS {I} (D : DS I I) : Set where
    ⟨_⟩ : fst ([[ D ]]DS (DataDS D, [[_]]ds)) → DataDS D
  [[_]]ds : {I : Set₁} {D : DS I I} → DataDS D → I
  [[_]]ds {D = D} ⟨ ds ⟩ = snd ([[ D ]]DS (DataDS D, [[_]]ds)) ds
```

...if only the positivity checker could trace the construction of the node set through the tupled presentation of $[\![\_]\!]_{\mathsf{DS}}$ and the termination checker could accept that the recursive invocation of $[\![\_]\!]_{\mathsf{DS}}$ 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:

```
mutual
  data DataDS {I} (D : DS I I) : Set where
    ⟨_⟩ : NoDS D D → DataDS D
  [[_]]ds : {I : Set₁} {D : DS I I} → DataDS D → I
  [[_]]ds {D = D} ⟨ ds ⟩ = DeDS D D ds
  NoDS : {I : Set₁} (D D' : DS I I) → Set
  NoDS D (ι i)     = One
  NoDS D (σ S T)   = Σ S λ s →                    NoDS D (T s)
  NoDS D (δ H T)   = Σ (H → DataDS D) λ hd → NoDS D (T (λ h → [[ hd h ]]ds))
  DeDS : {I : Set₁} (D D' : DS I I) → NoDS D D' → I
  DeDS D (ι i)     ⟨⟩      = i
  DeDS D (σ S T)   (s, t)  = DeDS D (T s) t
  DeDS D (δ H T)   (hd, t) = DeDS D (T (λ h → [[ hd h ]]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 ι as its 'return'.

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

```
bindDS : forall {I J K} → DS I J → (J → DS I K) → DS I K
bindDS T U = ?
```

Show that bindDS *corresponds to a kind of* Σ *by implementing pairing and projections:*

```
pairDS : forall {I J K} (T : DS I J) (U : J → DS I K) {X : Fam I} →
         (t : fst ([[ T ]]DS X)) (u : fst ([[ U (snd ([[ T ]]DS X) t) ]]DS X))
         → fst ([[ bindDS T U ]]DS X)
```

```
pairDS T U t u = ?
projDS : forall {I J K} (T : DS I J) (U : J → DS I K) {X : Fam I} →
         fst ([[ bindDS T U ]]DS X) →
         Σ (fst ([[ T ]]DS X)) λ t → fst ([[ U (snd ([[ T ]]DS X) t) ]]DS X)
projDS T U tu = ?
```

Which coherence properties hold?

There is one current snag with the DS I J coding of functors yielding inductive-recursive definitions, as you will discover if you attempt the following exercise.

Exercise 5.13 (composition for DS) *This is an open problem. Construct*

```
coDS : forall {I J K} → DS J K → DS I J → DS I K
coDS E D = ?
```

such that

$$[\![\ \mathsf{coDS}\ E\ D\ ]\!]_{\mathsf{DS}} \cong [\![\ E\ ]\!]_{\mathsf{DS}} \circ [\![\ D\ ]\!]_{\mathsf{DS}}$$

Alternatively, find a counterexample which wallops the very possibility of coDS*.*

In the next section, we can try to do something about the problem.

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 : Set₁) : Set₁ where
    ι : 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 → Set₁
  Info {I} ι       = 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.

```
ΠF : (S : Set) {J : S → Set₁} (T : (s : S) → Fam (J s)) →
     Fam ((s : S) → J s)
```

```
ΠF S T = ((s : S) → fst (T s)), λ f s → snd (T s) (f s)
ΣF : {I : Set₁} (S : Fam I) {J : I → Set₁} (T : (i : I) → Fam (J i)) →
     Fam (Σ I J)
ΣF S T = Σ (fst S) (fst ∘ (T ∘ snd S))
         ,  λ {(s , t) → snd S s , snd (T (snd S s)) t}
```

Now, for any T : Irish I, if someone gives us a Fam I to represent children, we can compute a Fam (Info T) — a *small* node structure from which the large Info T can be extracted.

```
Node : forall {I} (T : Irish I) → Fam I → Fam (Info T)
Node ι       X = X
Node (κ A)   X = A , ↑
Node (π S T) X = ΠF S λ s → Node (T s) X
Node (σ S T) X = ΣF (Node S X) λ iS → Node (T iS) X
```

A functor from Fam I to Fam J is then given by a pair

```
IF : Set₁ → Set₁ → Set₁
IF I J = Σ (Irish I) λ T → Info T → J
⟦_⟧IF : forall {I J} → IF I J → Fam I → Fam J
⟦ T , d ⟧IF X = d $F Node T X
```

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

```
mutual   -- fails positivity and termination checks
  data DataIF {I} (F : IF I I) : Set where
    ⟨_⟩ : fst (⟦ F ⟧IF (DataIF F , ⟦_⟧if)) → DataIF F
  ⟦_⟧if : forall {I} {F : IF I I} → DataIF F → I
  ⟦_⟧if {F = F} ⟨ ds ⟩ = snd (⟦ F ⟧IF (DataIF F , ⟦_⟧if)) ds
```

Again, specialization of Node fixes the problem

```
mutual
  data DataIF {I} (F : IF I I) : Set where
    ⟨_⟩ : NoIF F (fst F) → DataIF F
  ⟦_⟧if : forall {I} {F : IF I I} → DataIF F → I
  ⟦_⟧if {F = F} ⟨ rs ⟩ = snd F (DeIF F (fst F) rs)
  NoIF : forall {I} (F : IF I I) (T : Irish I) → Set
  NoIF F ι       = DataIF F
  NoIF F (κ A)   = A
  NoIF F (π S T) = (s : S) → NoIF F (T s)
  NoIF F (σ S T) = Σ (NoIF F S) λ s → NoIF F (T (DeIF F S s))
  DeIF : forall {I} (F : IF I I) (T : Irish I) → NoIF F T → Info T
  DeIF F ι       r     = ⟦ r ⟧if
  DeIF F (κ A)   a     = ↑ a
  DeIF F (π S T) f     = λ s → DeIF F (T s) (f s)
  DeIF F (σ S T) (s , t) = let s' = DeIF F S s in s' , DeIF F (T s') t
```

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*

```
DSIF : forall {I J} → DS I J → IF I J
DSIF T = ?
```

such that

$$[\![\ \mathsf{DSIF}\ T\ ]\!]_{\mathsf{DS}} \cong [\![\ T\ ]\!]_{\mathsf{IF}}$$

We clearly have an identity for Irish IR.

```
idIF : forall {I} → IF I I
idIF = ι, id
```

Now, DS I J had a substitution-for-ι structure which induced a notion of *pairing*, because ι marks 'end of record'. What makes the Irish encoding conductive to composition is that the ι-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.*

```
subIF : forall {I J} (T : Irish J) (F : IF I J) → Σ (Irish I) ?
subIF T F = ?
```

Hint: you will find out what you need in the σ *case.*

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

```
coIF : forall {I J K} → IF J K → IF I J → IF I K
coIF 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 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 refl means waiting at least until $s \equiv t$.)

The trouble with this way to compute 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 subst to 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 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

```
data Favourite : (ℕ → ℕ) → Set where
  favourite : Favourite (λ x → zero +ℕ x)
```

We may certainly prove that $\lambda\ x \to \mathsf{zero} +_{\mathbb{N}} x$ and $\lambda\ x \to x +_{\mathbb{N}} \mathsf{zero}$ agree on all inputs. But is there a canonical inhabitant of Favourite $(\lambda\ x \to x +_{\mathbb{N}} \mathsf{zero})$? If so, it can only be favourite, for that is the only constructor, but 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, 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 Favourite in the Henry Ford tradition,

```
data Favourite (f : ℕ → ℕ) : Set where
  favourite : (λ x → zero +ℕ x) ≃ f → Favourite f
```

then our definition of Favourite becomes just as intensional as our equality. If, somehow, $\simeq$ were to admit extensionality, we could certainly show that 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

$$\mathsf{favourite}\ ((\lambda\, x \to \mathsf{zero} +_{\mathbb{N}} x) =\![\ q\ ]\!= f =\![\ q'\ ]\!= g\ \Box)$$

with a modified proof.

6.1 Observational Equality for Types and Values in 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 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, 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 Eq $X\ x\ Y\ y$, but ensure that it holds only if $X \leftrightarrow Y$.
- We could allow the formation of any 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 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 descrbe the evidence for equality of types and values in this sense.

```
mutual
  EQ : (X Y : TU) → TU × ([[ X ]]TU → [[ Y ]]TU → TU)
  _↔_ : TU → TU → TU
  X ↔ Y = fst (EQ X Y)
  Eq : (X : TU) (x : [[ X ]]TU) → (Y : TU) (y : [[ Y ]]TU) → TU
  Eq X x Y y = snd (EQ X Y) x y
```

We should expect, ultimately, to construct a coercion mechanism which realises equality as transportation.

$$\mathsf{coe} \;:\; (X\ Y \;:\; \mathsf{TU}) \to [\![\ X \leftrightarrow Y\ ]\!]_{\mathsf{TU}} \to [\![\ X\ ]\!]_{\mathsf{TU}} \to [\![\ Y\ ]\!]_{\mathsf{TU}}$$

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

$$\mathsf{coh} \;:\; (X\ Y \;:\; \mathsf{TU})\ (Q \;:\; [\![\ X \leftrightarrow Y\ ]\!]_{\mathsf{TU}})\ (x \;:\; [\![\ X\ ]\!]_{\mathsf{TU}}) \to [\![\ \mathsf{Eq}\ X\ x\ Y\ (\mathsf{coe}\ X\ Y\ Q\ x)\ ]\!]_{\mathsf{TU}}$$

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 Zero′ and One′, all values are equal as there is at most one value anyway. For Two′, we must actually test the values.

$$\begin{array}{l}
\mathsf{EQ}\ \mathsf{Zero}'\ \mathsf{Zero}' \;=\; \mathsf{One}', \lambda\ _\ _ \to \mathsf{One}' \\
\mathsf{EQ}\ \mathsf{One}'\ \mathsf{One}' \;=\; \mathsf{One}', \lambda\ _\ _ \to \mathsf{One}' \\
\mathsf{EQ}\ \mathsf{Two}'\ \mathsf{Two}' \;=\; \mathsf{One}', \lambda \\
\quad \{\mathsf{tt}\ \mathsf{tt} \to \mathsf{One}' \\
\quad ; \mathsf{ff}\ \mathsf{ff} \to \mathsf{One}' \\
\quad ; _\ _ \to \mathsf{Zero}' \\
\quad \}
\end{array}$$

Σ′-types are interchangable 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.

$$\begin{array}{l}
\mathsf{EQ}\ (\Sigma'\ S\ T)\ (\Sigma'\ S'\ T') \\
\quad =\ (\Sigma'\ (S \leftrightarrow S')\ \lambda\ _ \to \\
\qquad \Pi'\ S\ \lambda\ s \to \Pi'\ S'\ \lambda\ s' \to \Pi'\ (\mathsf{Eq}\ S\ s\ S'\ s')\ \lambda\ _ \to \\
\qquad T\ s \leftrightarrow T'\ s') \\
\quad ,\quad \lambda\ \{(s,t)\ (s',t') \to \\
\qquad\quad \Sigma'\ (\mathsf{Eq}\ S\ s\ S'\ s')\ \lambda\ _ \to \mathsf{Eq}\ (T\ s)\ t\ (T'\ s')\ t'\}
\end{array}$$

Equality of pair values is straightforwardly structural. Notice that if the Σ′-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 coerce its input from right to left.

$$\begin{array}{l}
\mathsf{EQ}\ (\Pi'\ S\ T)\ (\Pi'\ S'\ T') \\
\quad =\ (\Sigma'\ (S' \leftrightarrow S)\ \lambda\ _ \to \\
\qquad \Pi'\ S'\ \lambda\ s' \to \Pi'\ S\ \lambda\ s \to \Pi'\ (\mathsf{Eq}\ S'\ s'\ S\ s)\ \lambda\ _ \to \\
\qquad T\ s \leftrightarrow T'\ s') \\
\quad ,\quad \lambda\ \{f\ f' \to \\
\qquad\quad \Pi'\ S\ \lambda\ s \to \Pi'\ S'\ \lambda\ s' \to \Pi'\ (\mathsf{Eq}\ S\ s\ S'\ s')\ \lambda\ _ \to \\
\qquad\quad \mathsf{Eq}\ (T\ s)\ (f\ s)\ (T'\ s')\ (f'\ s')\}
\end{array}$$

Function values are considered equal if they take equal inputs to equal outputs.

Tree′ types are, again, compared structurally, with pointwise equality expressed by abstraction over pairs of equal values.

$$\begin{array}{l}
\mathsf{EQ}\ (\mathsf{Tree}'\ I\ F\ i)\ (\mathsf{Tree}'\ I'\ F\ i') \\
\quad =\ (\Sigma'\ (I \leftrightarrow I')\ \lambda\ _ \to \Sigma'\ (\mathsf{Eq}\ I\ i\ I'\ i')\ \lambda\ _ \to
\end{array}$$

```
    Π′ I λ i → Π′ I′ λ i′ → Π′ (Eq I i I′ i′) λ _ →
    let (S, K) = F i; S′, K′ = F i′
    in Σ′ (S ↔ S′) λ _ →
       Π′ S λ s → Π′ S′ λ s′ → Π′ (Eq S s S′ s′) λ _ →
       let (P, r) = K s; (P, r) = K′ s′
       in Σ′ (P ↔ P) λ _ →
          Π′ P λ p′ → Π′ P λ p → Π′ (Eq P p′ P p) λ _ →
          Eq I (r p) I′ (r p′))
,   teq i i′ where
    teq : (i : ⟦ I ⟧TU) → (i′ : ⟦ I′ ⟧TU) →
      ⟦ Tree′ I F i ⟧TU → ⟦ Tree′ I′ F i′ ⟧TU → 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 Σ′ (Eq S s S′ s′) λ _ →
           Π′ P λ p → Π′ P λ p′ → Π′ (Eq P p P p′) λ _ →
           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′, λ _ _ → One′
```

Exercise 6.1 (define coe, postulate coh) *Implement coercion, assuming coherence.*

```
coe : (X Y : TU) → ⟦ X ↔ Y ⟧TU → ⟦ X ⟧TU → ⟦ Y ⟧TU
postulate
  coh : (X Y : TU) (Q : ⟦ X ↔ Y ⟧TU) (x : ⟦ X ⟧TU) → ⟦ Eq X x Y (coe X Y Q x) ⟧TU
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
  reflTU : (X : TU) (x : ⟦ X ⟧TU) → ⟦ Eq X x X x ⟧TU
```

Exercise 6.2 (explore failing to prove $\mathsf{refl}_{\mathsf{TU}}$) *Try proving*

```
reflTU : (X : TU) (x : ⟦ X ⟧TU) → ⟦ Eq X x X x ⟧TU
reflTU 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
  Resp_TU : (X : TU) (P : ⟦ X ⟧_TU → TU)
            (x x' : ⟦ X ⟧_TU) → ⟦ Eq X x X x' ⟧_TU → ⟦ P x ↔ P x' ⟧_TU
subst_TU : (X : TU) (P : ⟦ X ⟧_TU → TU)
  (x x' : ⟦ X ⟧_TU) → ⟦ Eq X x X x' ⟧_TU → ⟦ P x ⟧_TU → ⟦ P x' ⟧_TU
subst_TU X P x x' q = coe (P x) (P x') (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, Prf′.

```
data Sort : Set where set prop : Sort

IsSet : Sort → Set
IsSet set  = One
IsSet prop = Zero

mutual
  data Set (u : Sort) : Set where
    Zero′ One′ : Set u
    Two′ : {_ : IsSet u} → Set u
    Σ′   : (S : Set u) (T : ⟦ S ⟧_PU → Set u) → Set u
    Π′   : (S : Set set) (T : ⟦ S ⟧_PU → Set u) → Set u
    Tree′ : {_ : IsSet u}
            (I : Set set)
            (F : ⟦ I ⟧_PU → Σ (Set set) λ S →
                 ⟦ S ⟧_PU → Σ (Set set) λ P →
                 ⟦ P ⟧_PU → ⟦ I ⟧_PU)
            (i : ⟦ I ⟧_PU) → Set u
    Prf′ : {_ : IsSet u} → Set prop → Set u

  ⟦_⟧_PU : forall {u} → Set u → Set
  ⟦ Zero′       ⟧_PU = Zero
  ⟦ One′        ⟧_PU = One
  ⟦ Two′        ⟧_PU = Two
  ⟦ Σ′ S T      ⟧_PU = Σ ⟦ S ⟧_PU λ s → ⟦ T s ⟧_PU
  ⟦ Π′ S T      ⟧_PU = (s : ⟦ S ⟧_PU) → ⟦ T s ⟧_PU
  ⟦ Tree′ I F i ⟧_PU = ITree
    ( (λ i → ⟦ fst (F i) ⟧_PU)
    ◁ (λ i s → ⟦ fst (snd (F i) s) ⟧_PU)
    $ (λ i s p → snd (snd (F i) s) p)
    ) i
  ⟦ Prf′ P      ⟧_PU = ⟦ P ⟧_PU
```

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.*

```
_∧_ : Set prop → Set prop → Set prop
P ∧ Q = Σ′ P λ _ → Q
_⇒_ : Set prop → Set prop → Set prop
P ⇒ Q = Π′ (Prf′ P) λ _ → Q
mutual
  PEQ : (X Y : Set set) → Set prop × (⟦ X ⟧PU → ⟦ Y ⟧PU → Set prop)
  _⇔_ : Set set → Set set → Set prop
  X ⇔ Y = fst (PEQ X Y)
  PEq : (X : Set set) (x : ⟦ X ⟧PU) → (Y : Set set) (y : ⟦ Y ⟧PU) → Set prop
  PEq X x Y y = snd (PEQ X Y) x y
  PEQ (Prf′ P) (Prf′ Q) = ((P ⇒ Q) ∧ (Q ⇒ P)), λ _ _ → One′
    -- more code goes here
  PEQ _ _ = Zero′, λ _ _ → One′
```

Chapter 7

Type Theory in Type Theory

A while ago, we defined the simply typed λ-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

Rod Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 230–266. Springer, 2008.