Dependently Typed Metaprogramming: (in Agda)

Conor McBride

August 5, 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 = ?
```

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.

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 1.5 (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 1.6 (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 1.7 (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 1.8 (Applicative product) *Show by construction that the pointwise product of Applicatives is Applicative.*

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

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

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

Exercise 1.9 (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:

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

```
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
```

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 1.10 (Traversable functors) *Show that* Traversable *is closed under identity and composition. What other structure does it preserve?*

1.4 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
  ⟦_⟧N X = Σ Shape λ s → Vec X (size s)
open Normal public
infixr 0 _◁_
```

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

```
VecN : ℕ → Normal
VecN n = One / pure n
ListN : Normal
ListN = ℕ / id
```

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 $\wedge$ to be a low precedence uncurrying operator, so that $\wedge\ \lambda\ s\ t \to \ldots$ 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.

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. Developing a library of normal functors will, however, necessitate arithmetic on sizes as well as shapes.

Exercise 1.11 (unary arithmetic) *Implement addition and multiplication for numbers.*

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

Let us construct sums and products of normal functors.

```
_+N_ : Normal → Normal → Normal
(ShF / szF) +N (ShG / szG) = (ShF + ShG) / ∧ szF ⟨?⟩ szG
_×N_ : Normal → Normal → Normal
(ShF / szF) ×N (ShG / szG) = (ShF × ShG) / ∧ λ f g → szF f +ℕ szG g
```

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

```
nInj : forall {X} (F G : Normal) → ⟦ F ⟧N X + ⟦ G ⟧N X → ⟦ F +N 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 : forall {X} F G (s : ⟦ F +N 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 smoe 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 1.12 (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 1.13 (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 $+_{\mathbb{N}}$ 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 → Λ 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.

Check this.

1.5 Proving Equations

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

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

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 eta-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.

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

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

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

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

Now that we have some sort of equality, we can specify laws for our structures, e.g., for 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 1.14 (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* λ *n xs* → *..., which will allow you to abstract over separated places with* n *and* xs*.*

Exercise 1.15 (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* ≃ *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 1.16 (Vec functor laws) *Show that vectors are functorial.*

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

1.6 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, $x \,\square$ marks the end of the chain.

Meanwhile, we may need to rewrite in a context whilst building these proofs. In the expression syntax, we have nothing like **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
```

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

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

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

```
    lawHom : forall {S T} (f : S → T) (s : S) →
      pure {{AF}} 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 1.17 (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
  }
```

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 1.18 *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.

1.7 Fixpoints of Normal Functors

```
data Tree (N : Normal) : Set where
  ⟨_⟩ : ⟦ N ⟧N (Tree N) → Tree N
```

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.

```
⟦_⟧Cx : Cx ⋆ → Set
⟦ ℰ ⟧Cx     = One
⟦ Γ , σ ⟧Cx = ⟦ Γ ⟧Cx × ⟦ σ ⟧⋆
⟦_⟧∈ : forall {Γ τ} → τ ∈ Γ → ⟦ Γ ⟧Cx → ⟦ 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 Del = forall {τ} → τ ∈ Γ → τ ∈ Del
Sub Γ Del = forall {τ} → τ ∈ Γ → Del ⊢ 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)
```

Chapter 3

Containers and W-types

Chapter 4

Indexed Containers (Levitated)

Chapter 5

Induction-Recursion

Chapter 6

Observational Equality

Chapter 7

Type Theory in Type Theory

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.