Dependently Typed Programming: an Agda introduction

Conor McBride

February 21, 2011

Chapter 1

Vectors and Finite Sets

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.

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

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

That's the usual 'garbage in? garbage out!' deal. Logically, we might want to ensure the inverse: if we supply meaningful input, we want 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?

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 decimal numerals.

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

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

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

$$\forall fs, ss.\ \mathsf{length}\ fs = \mathsf{length}\ ss \Rightarrow \mathsf{length}\ (\mathsf{zap}\ fs\ ss)\ =\ \mathsf{length}\ fs$$

by structural induction [Burstall, 1969] on fs, say. Of course, we could just as well have concluded that length (zap fs ss) = length ss, and if we carry on zapping, we shall accumulate a multitude of expressions known to denote the same number.

What can we say about list concatenation? We may define addition.

How many ways to define $+_{\mathsf{N}}$?

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

$$
\begin{array}{l}
\_+_{\mathsf{N}}\_ \ :\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{zero} \quad +_{\mathsf{N}}\ y \ =\ y \\
\mathsf{suc}\ x\ +_{\mathsf{N}}\ y \ =\ \mathsf{suc}\ (x\ +_{\mathsf{N}}\ y)
\end{array}
$$

We may define concatenation.

$$
\begin{array}{l}
\_+_{\mathsf{L}}+\_ \ :\ \{X\ :\ \mathsf{Set}\} \to \mathsf{List}\ X \to \mathsf{List}\ X \to \mathsf{List}\ X \\
\langle\rangle \qquad +_{\mathsf{L}}+\ ys \ =\ ys \\
(x, xs)\ +_{\mathsf{L}}+\ ys \ =\ x, (xs\ +_{\mathsf{L}}+\ ys)
\end{array}
$$

It takes a proof by induction (and a convenient definition of $+_{\mathsf{N}}$) to note that

$$\mathsf{length}\ (xs\ +_{\mathsf{L}}+\ ys) = \mathsf{length}\ xs\ +_{\mathsf{N}}\ \mathsf{length}\ ys$$

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.

1.0.1 Peano Exercises

Exercise 1.1 (Go Forth and Multiply!) *Given addition, implement multiplication.*

$$\_\times_{\mathsf{N}}\_ \ :\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$

Exercise 1.2 (Subtract with Dummy) *Implement subtraction, with a nasty old dummy return when you take a big number from a small one.*

$$\_-_{\mathsf{N}}\_ \ :\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$

Exercise 1.3 (Divide with a Duplicate) *Implement division. Agda won't let you do repeated subtraction directly (not structurally decreasing), but you can do something sensible (modulo the dummy) like this:*

$$
\begin{array}{l}
\_\div_{\mathsf{N}}\_ \ :\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
x\ \div_{\mathsf{N}}\ d \ =\ \mathsf{help}\ x\ d\ \textbf{where} \\
\quad \mathsf{help}\ :\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\quad \mathsf{help}\ x\ e \ = \quad \text{-- } \{!!\}
\end{array}
$$

You can recursively peel sucs *from* e *one at a time, with the original* d *still in scope.*

1.1 Vectors

Here are lists, indexed by numbers which happen to measure their length: these are known in the trade as *vectors*.

Agda allows overloading of constructors, as its approach to typechecking is of a bidirectional character

Might want to say something about head and tail, and about how coverage checking works anyway.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Vec}\ (X\ :\ \mathsf{Set})\ :\ \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \langle\rangle \ : \qquad\qquad\qquad\qquad\qquad\quad \mathsf{Vec}\ X\ \mathsf{zero} \\
\quad \_,\_ \ :\ \{n\ :\ \mathsf{Nat}\} \to X \to \mathsf{Vec}\ X\ n \to \mathsf{Vec}\ X\ (\mathsf{suc}\ n)
\end{array}
$$

Not greatly enamoured of $S\ T\ :\ \mathsf{Set}$ notation, but there it is

$$
\begin{array}{l}
\mathsf{vap}\ :\ \{n\ :\ \mathsf{Nat}\}\ \{S\ T\ :\ \mathsf{Set}\} \to \mathsf{Vec}\ (S \to T)\ n \to \mathsf{Vec}\ S\ n \to \mathsf{Vec}\ T\ n \\
\mathsf{vap}\ \langle\rangle \qquad \langle\rangle \qquad =\ \langle\rangle \\
\mathsf{vap}\ (f, fs)\ (s, ss)\ =\ f\ s, \mathsf{vap}\ fs\ ss
\end{array}
$$

$$
\begin{array}{l}
\mathsf{vec} : \{n : \mathsf{Nat}\}\,\{X : \mathsf{Set}\} \to X \to \mathsf{Vec}\ X\ n \\
\mathsf{vec}\ \{\mathsf{zero}\}\ \ x = \langle\rangle \\
\mathsf{vec}\ \{\mathsf{suc}\ n\}\ x = x, \mathsf{vec}\ x
\end{array}
$$

vec is an example of a function with an indexing argument that is usually inferrable, but never irrelevant.

By now, you may have noticed the proliferation of listy types.

$$
\begin{array}{l}
\_{+_V+}\_ : \{m\ n : \mathsf{Nat}\}\,\{X : \mathsf{Set}\} \to \mathsf{Vec}\ X\ m \to \mathsf{Vec}\ X\ n \to \mathsf{Vec}\ X\ (m +_N n) \\
\langle\rangle \qquad +_V+\ ys = ys \\
(x, xs)\ +_V+\ ys = x, (xs\ +_V+\ ys)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{vrevapp} : \{m\ n : \mathsf{Nat}\}\,\{X : \mathsf{Set}\} \to \mathsf{Vec}\ X\ m \to \mathsf{Vec}\ X\ n \to \mathsf{Vec}\ X\ (m +_N n) \\
\mathsf{vrevapp}\ \langle\rangle \qquad ys = ys \\
\mathsf{vrevapp}\ (x, xs)\ ys = \mathsf{vrevapp}\ xs\ (x, ys)
\end{array}
$$

Here's a stinker. Of course, you can rejig $+_N$ to be tail recursive and make $+_V+$ a stinker.

This is the 'traverse' function from the 'idiom paper' []

Which other things work badly? Filter?

I wanted to make $\_/\_$ left-associative, but no such luck.

$$
\begin{array}{l}
\mathsf{vtraverse} : \{F : \mathsf{Set} \to \mathsf{Set}\} \to \\
\qquad (\{X : \mathsf{Set}\} \to X \to F\ X) \to \\
\qquad (\{S\ T : \mathsf{Set}\} \to F\ (S \to T) \to F\ S \to F\ T) \to \\
\qquad \{n : \mathsf{Nat}\}\,\{X\ Y : \mathsf{Set}\} \to \\
\qquad (X \to F\ Y) \to \mathsf{Vec}\ X\ n \to F\ (\mathsf{Vec}\ Y\ n) \\
\mathsf{vtraverse}\ pure\ \_/\_\ f\ \langle\rangle \qquad = pure\ \langle\rangle \\
\mathsf{vtraverse}\ pure\ \_/\_\ f\ (x, xs) = (pure\ \_,\_\ /\ f\ x)\ /\ \mathsf{vtraverse}\ pure\ \_/\_\ f\ xs
\end{array}
$$

When would be a good time to talk about universe polymorphism?

$$
\begin{array}{l}
\mathrm{I} : \{X : \mathsf{Set}\} \to X \to X \\
\mathrm{I}\ x = x \\
\mathrm{K} : \{X\ Y : \mathsf{Set}\} \to X \to Y \to X \\
\mathrm{K}\ x\ y = x
\end{array}
$$

Why is Y undetermined?

$$
\begin{array}{l}
\mathsf{vsum} : \{n : \mathsf{Nat}\} \to \mathsf{Vec}\ \mathsf{Nat}\ n \to \mathsf{Nat} \\
\mathsf{vsum} = \mathsf{vtraverse}\ (\mathrm{K}\ \mathsf{zero})\ \_{+_N}\_\ \{Y = \mathsf{Nat}\}\ \mathrm{I}
\end{array}
$$

1.1.1 Matrix Exercises

Let us define an m by n matrix to be a vector of m rows, each length n.

$$
\begin{array}{l}
\mathsf{Matrix} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set} \to \mathsf{Set} \\
\mathsf{Matrix}\ m\ n\ X = \mathsf{Vec}\ (\mathsf{Vec}\ X\ n)\ m
\end{array}
$$

Exercise 1.4 (Matrices are Applicative) *Show that* Matrix $m\ n$ *can be equipped with operations analogous to* vec *and* vap*.*

$$
\begin{array}{l}
\mathsf{vvec} : \{m\ n : \mathsf{Nat}\}\,\{X : \mathsf{Set}\} \to X \to \mathsf{Matrix}\ m\ n\ X \\
\mathsf{vvap} : \{m\ n : \mathsf{Nat}\}\,\{S\ T : \mathsf{Set}\} \to \\
\qquad \mathsf{Matrix}\ m\ n\ (S \to T) \to \mathsf{Matrix}\ m\ n\ S \to \mathsf{Matrix}\ m\ n\ T
\end{array}
$$

which, respectively, copy a given element into each position, and apply functions to arguments in corresponding positions.

Exercise 1.5 (Matrix Addition) *Use the applicative interface for* Matrix *to define their elementwise addition.*

$$
\_{+_M}\_ : \{m\ n : \mathsf{Nat}\} \to \mathsf{Matrix}\ m\ n\ \mathsf{Nat} \to \mathsf{Matrix}\ m\ n\ \mathsf{Nat} \to \mathsf{Matrix}\ m\ n\ \mathsf{Nat}
$$

Exercise 1.6 (Matrix Transposition) *Use* vtraverse *to give a one-line definition of matrix transposition.*

```
transpose : {m n : Nat} {X : Set} → Matrix m n X → Matrix n m X
```

Exercise 1.7 (Identity Matrix) *Define a function*

```
idMatrix : {n : Nat} → Matrix n n Nat
```

Exercise 1.8 (Matrix Multiplication) *Define matrix multiplication. There are lots of ways to do this. Some involve defining scalar product, first.*

```
_×M_ : {l m n : Nat} → Matrix l m Nat → Matrix m n Nat → Matrix l n Nat
```

1.1.2 Unit and Sigma types

Why do this with records?

```
record 𝟙 : Set where
  constructor ⟨⟩
open 𝟙 public
```

```
u0 : 𝟙
u0 = ⟨⟩
u1 : 𝟙
u1 = record {}
u2 : 𝟙
u2 = _
```

The **field** keyword declares fields, we can also add 'manifest' fields.

```
record Σ (S : Set) (T : S → Set) : Set where
  constructor _,_
  field
    fst : S
    snd : T fst
open Σ public
_×_ : Set → Set → Set
S × T = Σ S λ _ → T
```

1.1.3 Apocrypha

You would not invent dependent pattern matching if vectors were your only example.

```
VecR : Set → Nat → Set
VecR X zero    = 𝟙
VecR X (suc n) = X × VecR X n
```

The definition is logically the same, why are the programs noisier?

```
vconcR : {m n : Nat} {X : Set} →
         VecR X m → VecR X n → VecR X (m +N n)
vconcR {zero}  ⟨⟩       ys = ys
vconcR {suc m} (x, xs) ys = x, vconcR {m} xs ys
```

$$\textbf{data}\ \_{==}\_\ \{X : \mathsf{Set}\}\ (x : X) : X \to \mathsf{Set}\ \textbf{where}$$
$$\quad \langle\rangle : x == x$$

$$\begin{aligned} &\mathsf{len} : \{X : \mathsf{Set}\} \to \mathsf{List}\ X \to \mathsf{Nat}\\ &\mathsf{len}\ \langle\rangle \quad = \mathsf{zero}\\ &\mathsf{len}\ (x, xs) = \mathsf{suc}\ (\mathsf{len}\ xs) \end{aligned}$$

Agda's λ scopes rightward as far as possible, reducing bracketing. Even newer fancy binding sugar might make this prettier still.

$$\begin{aligned} &\mathsf{VecP} : \mathsf{Set} \to \mathsf{Nat} \to \mathsf{Set}\\ &\mathsf{VecP}\ X\ n = \Sigma\ (\mathsf{List}\ X)\ \lambda\ xs \to \mathsf{len}\ xs == n \end{aligned}$$

$$\begin{aligned} &\mathsf{vnil} : \{X : \mathsf{Set}\} \to \mathsf{VecP}\ X\ \mathsf{zero}\\ &\mathsf{vnil} = \langle\rangle, \langle\rangle \end{aligned}$$

It's already getting bad here, but we can match p against $\langle\rangle$ and complete.

$$\begin{aligned} &\mathsf{vcons} : \{X : \mathsf{Set}\}\ \{n : \mathsf{Nat}\} \to X \to \mathsf{VecP}\ X\ n \to \mathsf{VecP}\ X\ (\mathsf{suc}\ n)\\ &\mathsf{vcons}\ x\ (xs, p) = (x, xs), \quad \text{-- }\{!\,!\} \end{aligned}$$

But this really is toxic.

$$\begin{aligned} &\mathsf{vapP} : \{n : \mathsf{Nat}\}\ \{S\ T : \mathsf{Set}\} \to\\ &\qquad \mathsf{VecP}\ (S \to T)\ n \to \mathsf{VecP}\ S\ n \to \mathsf{VecP}\ T\ n\\ &\mathsf{vapP}\ (\langle\rangle, \langle\rangle)\qquad (\langle\rangle, \langle\rangle)\qquad = \langle\rangle, \langle\rangle\\ &\mathsf{vapP}\ ((f, fs), \langle\rangle)\ ((s, ss), p) = (f\ s, \mathsf{vap}\ (fs, ?)\ (ss, ?)), ? \end{aligned}$$

1.2 Finite Sets

If we know the size of a vector, can we hope to project from it safely? Here's a family of *finite sets*, good to use as indices into vectors.

$$\begin{aligned} &\textbf{data}\ \mathsf{Fin} : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}\\ &\quad \mathsf{zero} : \{n : \mathsf{Nat}\} \to \qquad\qquad\qquad \mathsf{Fin}\ (\mathsf{suc}\ n)\\ &\quad \mathsf{suc}\ \ : \{n : \mathsf{Nat}\} \to (i : \mathsf{Fin}\ n) \to \mathsf{Fin}\ (\mathsf{suc}\ n)\\ &\mathit{foo} : \{X : \mathsf{Set}\}\ \{n : \mathsf{Nat}\} \to \mathsf{Fin}\ (\mathsf{zero} +_{\mathsf{N}} \mathsf{zero}) \to X\\ &\mathit{foo}\ () \end{aligned}$$

Finite sets are sets of bounded numbers. One thing we may readily do is forget the bound.

Do you resent writing this function? You should.

$$\begin{aligned} &\mathsf{fog} : \{n : \mathsf{Nat}\} \to \mathsf{Fin}\ n \to \mathsf{Nat}\\ &\mathsf{fog}\ \mathsf{zero} \quad = \mathsf{zero}\\ &\mathsf{fog}\ (\mathsf{suc}\ i) = \mathsf{suc}\ (\mathsf{fog}\ i) \end{aligned}$$

Now let's show how to give a total projection from a vector of known size.

Here's our first Aunt Fanny. We could also swap the arguments around.

$$\begin{aligned} &\mathsf{vproj} : \{n : \mathsf{Nat}\}\ \{X : \mathsf{Set}\} \to \mathsf{Vec}\ X\ n \to \mathsf{Fin}\ n \to X\\ &\mathsf{vproj}\ \langle\rangle\ ()\\ &\mathsf{vproj}\ (x, xs)\ \mathsf{zero} \quad = x\\ &\mathsf{vproj}\ (x, xs)\ (\mathsf{suc}\ i) = \mathsf{vproj}\ xs\ i \end{aligned}$$

It's always possible to give enough Aunt Fannies to satisfy the coverage checker.

Suppose we want to project at an index not known to be suitably bounded. How might we check the bound? We shall return to that thought, later.

1.2.1 Renamings

We'll shortly use Fin to type bounded sets of de Bruijn indices. Functions from one finite set to another will act as 'renamings'.

Extending the context with a new assumption is sometimes known as 'weakening': making more assumptions weakens an argument. Suppose we have a function from Fin m to Fin n, renaming variables, as it were. How should weakening act on this function? Can we extend the function to the sets one larger, mapping the 'new' source zero to the 'new' target zero? This operation shows how to push a renaming under a binder.

Categorists, what should we prove about weaken?

```
weaken : {m n : Nat} → (Fin m → Fin n) → Fin (suc m) → Fin (suc n)
weaken f zero    = zero
weaken f (suc i) = suc (f i)
```

One operation we'll need corresponds to inserting a new variable somewhere in the context. This operation is known as 'thinning'. Let's define the order-preserving injection from Fin n to Fin (suc n) which misses a given element

```
thin : {n : Nat} → Fin (suc n) → Fin n → Fin (suc n)
thin          zero    = suc
thin {zero}   (suc ())
thin {suc n}  (suc i) = weaken (thin i)
```

1.2.2 Finite Set Exercises

Exercise 1.9 (Tabulation) *Invert* vproj*. Given a function from a* Fin *set, show how to construct the vector which tabulates it.*

```
vtab : {n : Nat} {X : Set} → (Fin n → X) → Vec X n
```

Exercise 1.10 (Plan a Vector) *Show how to construct the 'plan' of a vector—a vector whose elements each give their own position, counting up from* zero*.*

```
vplan : {n : Nat} → Vec (Fin n) n
```

Exercise 1.11 (Max a Fin) *Every nonempty finite set has a smallest element* zero *and a largest element which has as many* suc*s as allowed. Construct the latter*

```
max : {n : Nat} → Fin (suc n)
```

Exercise 1.12 (Embed, Preserving fog) *Give the embedding from one finite set to the next which preserves the numerical value given by* fog*.*

```
emb : {n : Nat} → Fin n → Fin (suc n)
```

Exercise 1.13 (Thickening) *Construct* thick i *the partial inverse of* thin i*. You'll need*

```
data Maybe (X : Set) : Set where
  yes : X → Maybe X
  no  :     Maybe X
```

Which operations on Maybe *will help? Discover and define them as you implement:*

```
thick : {n : Nat} → Fin (suc n) → Fin (suc n) → Maybe (Fin n)
```

Note that thick *acts as an inequality test.*

Exercise 1.14 (Order-Preserving Injections) *Define an inductive family*

$$\mathsf{OPI} \;:\; \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set}$$

such that $\mathsf{OPI}\ m\ n$ *gives a unique first-order representation to exactly the order-preserving injections from* $\mathsf{Fin}\ m$ *to* $\mathsf{Fin}\ n$*, and give the functional interpretation of your data. Show that* OPI *is closed under identity and composition.*

Chapter 2

Lambda Calculus with de Bruijn Indices

I'm revisiting chapter 7 of my thesis here.

Here are the λ-terms with n available de Bruijn indices [de Bruijn, 1972].

```
data Tm (n : Nat) : Set where
  var : Fin n →               Tm n
  _$_ : Tm n → Tm n → Tm n
  lam : Tm (suc n) →        Tm n
infixl 6 _$_
```

Which operations work?

Substitute for zero?

How many different kinds of trouble are we in?

```
sub0 : {n : Nat} → Tm n → Tm (suc n) → Tm n
sub0 s (var zero)      = s
sub0 s (var (suc i))   = var i
sub0 s (f $ a)         = sub0 s f $ sub0 s a
sub0 s (lam b)         = lam (sub0 ? b)
```

Simultaneous substitution?

Notoriously not structurally recursive.

```
ssub : {m n : Nat} → (Fin m → Tm n) → Tm m → Tm n
ssub σ (var i)  = σ i
ssub σ (f $ a)  = ssub σ f $ ssub σ a
ssub {m} {n} σ (lam b)  = lam (ssub σ b) where
  σ : Fin (suc m) → Tm (suc n)
  σ zero     = var zero
  σ (suc i)  = ssub (λ i → var (suc i)) (σ i)
```

At this point, Thorsten Altenkirch and Bernhard Reus [Altenkirch and Reus, 1999] reached for the hammer of wellordering, but there's a cheaper way to get out of the jam.

2.1 Simultaneous Renaming and Substitution

You can define simultaneous renaming really easily.

```
wkr : {m n : Nat} → (Fin m → Fin n) → Fin (suc m) → Fin (suc n)
wkr ρ zero     = zero
```

```
wkr ρ (suc i) = suc (ρ i)
ren : {m n : Nat} → (Fin m → Fin n) → Tm m → Tm n
ren ρ (var i) = var (ρ i)
ren ρ (f $ a) = ren ρ f $ ren ρ a
ren ρ (lam b) = lam (ren (wkr ρ) b)
```

And you can define substitution, given renaming.

```
wks : {m n : Nat} → (Fin m → Tm n) → Fin (suc m) → Tm (suc n)
wks σ zero    = var zero
wks σ (suc i) = ren suc (σ i)
sub : {m n : Nat} → (Fin m → Tm n) → Tm m → Tm n
sub σ (var i) = σ i
sub σ (f $ a) = sub σ f $ sub σ a
sub σ (lam b) = lam (sub (wks σ) b)
```

How repetitive! Let's abstract out the pattern.

```
record Kit (I : Nat → Set) : Set where
  constructor mkKit
  field
    mkv : {n : Nat} → Fin n → I n
    mkt : {n : Nat} → I n → Tm n
    wki : {n : Nat} → I n → I (suc n)
open Kit public

wk : {I : Nat → Set} → Kit I → {m n : Nat} →
     (Fin m → I n) → Fin (suc m) → I (suc n)
wk k τ zero    = mkv k zero
wk k τ (suc i) = wki k (τ i)
act : {I : Nat → Set} → Kit I → {m n : Nat} →
      (Fin m → I n) → Tm m → Tm n
act k τ (var i) = mkt k (τ i)
act k τ (f $ a) = act k τ f $ act k τ a
act k τ (lam b) = lam (act k (wk k τ) b)
```

2.1.1 Exercises

Exercise 2.1 (Renaming Kit) *Define the renamimg kit.*

```
renk : Kit Fin
```

Exercise 2.2 (Substitution Kit) *Define the substitution kit.*

```
subk : Kit Tm
```

Exercise 2.3 (Substitute zero) `sub0 : {n : Nat} → Tm n → Tm (suc n) → Tm n`

Exercise 2.4 (Reduce One) *Define a function to contract the leftmost redex in a λ-term, if there is one.*

```
leftRed : {n : Nat} → Tm n → Maybe (Tm n)
```

Exercise 2.5 (Complete Development) *Show how to compute the complete development of a λ-term, contracting all its visible redexes in parallel (but not the redexes which thus arise).*

$$\mathsf{develop} : \{n : \mathsf{Nat}\} \to \mathsf{Tm}\ n \to \mathsf{Tm}\ n$$

Exercise 2.6 (Gasoline Alley) *Write an iterator, computing the n-fold self-composition of an endofunction, effectively interpreting each* Nat *as its corresponding Church numeral.*

$$\mathsf{iterate} : \mathsf{Nat} \to \{X : \mathsf{Set}\} \to (X \to X) \to X \to X$$

You can use iterate *and* develop *to run λ-terms for as many steps as you like, as long as you are modest in your likes.*

Exercise 2.7 (Another Substitution Recipe) *It occurred to me at time of writing that one might cook substitution differently. Using abacus-style addition*

$$\begin{array}{l}
_{+_\mathsf{a}}_ : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{zero}\ \ +_\mathsf{a}\ n = n \\
\mathsf{suc}\ m +_\mathsf{a}\ n = m +_\mathsf{a} \mathsf{suc}\ n
\end{array}$$

let

$$\begin{array}{l}
\mathsf{Sub} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set} \\
\mathsf{Sub}\ m\ n = (w : \mathsf{Nat}) \to \mathsf{Fin}\ (w +_\mathsf{a} m) \to \mathsf{Tm}\ (w +_\mathsf{a} n)
\end{array}$$

be the type of substitions which can be weakened. Define

$$\mathsf{subw} : \{m\ n : \mathsf{Nat}\} \to \mathsf{Sub}\ m\ n \to \mathsf{Tm}\ m \to \mathsf{Tm}\ n$$

Now show how to turn a renaming into a Sub.

$$\mathsf{renSub} : \{m\ n : \mathsf{Nat}\} \to (\mathsf{Fin}\ m \to \mathsf{Fin}\ n) \to \mathsf{Sub}\ m\ n$$

Finally, show how to turn a simultaneous substitution into a Sub.

$$\mathsf{subSub} : \{m\ n : \mathsf{Nat}\} \to (\mathsf{Fin}\ m \to \mathsf{Tm}\ n) \to \mathsf{Sub}\ m\ n$$

2.1.2 How to Hide de Bruijn Indices

$$\begin{array}{l}
\mathsf{max} : \{n : \mathsf{Nat}\} \to \mathsf{Fin}\ (\mathsf{suc}\ n) \\
\mathsf{max}\ \{\mathsf{zero}\} \ \ = \mathsf{zero} \\
\mathsf{max}\ \{\mathsf{suc}\ n\} = \mathsf{suc}\ (\mathsf{max}\ \{n\}) \\
\mathsf{embed} : \{n : \mathsf{Nat}\} \to \mathsf{Fin}\ n \to \mathsf{Fin}\ (\mathsf{suc}\ n) \\
\mathsf{embed}\ \mathsf{zero} \ \ \ \ = \mathsf{zero} \\
\mathsf{embed}\ (\mathsf{suc}\ n) = \mathsf{suc}\ (\mathsf{embed}\ n)
\end{array}$$

$$\begin{array}{l}
\mathsf{shifty} : (m : \mathsf{Nat})\ \{n : \mathsf{Nat}\} \to \mathsf{Fin}\ (\mathsf{suc}\ (m +_\mathsf{N} n)) \\
\mathsf{shifty}\ \mathsf{zero} \ \ \ \ = \mathsf{max} \\
\mathsf{shifty}\ (\mathsf{suc}\ m) = \mathsf{embed}\ (\mathsf{shifty}\ m) \\
\mathsf{lambda} : \{m : \mathsf{Nat}\} \to \\
\qquad ((\{n : \mathsf{Nat}\} \to \mathsf{Tm}\ (\mathsf{suc}\ (m +_\mathsf{N} n))) \to \mathsf{Tm}\ (\mathsf{suc}\ m)) \to \\
\qquad \mathsf{Tm}\ m \\
\mathsf{lambda}\ \{m\}\ f = \mathsf{lam}\ (f\ \lambda\ \{n\} \to \mathsf{var}\ (\mathsf{shifty}\ m\ \{n\})) \\
\mathsf{myTest} : \mathsf{Tm}\ \mathsf{zero} \\
\mathsf{myTest} = \mathsf{lambda}\ \lambda\ f \to \mathsf{lambda}\ \lambda\ x \to f\ \$\ (f\ \$\ x)
\end{array}$$

2.1.3 Simply Typed Lambda Calculus

Altenkirch and Reus carry on to develop simultaneous type-preserving substitution for the *simply-typed* λ-calculus. Let's see how.

```
infixr 4 _▷_
infixr 3 _⊢_
infixr 3 _⊣_

data Ty : Set where
  ι   :                Ty
  _▷_ : (S T : Ty) → Ty
```

I'll have a bunch of variations, so it will help if I make context a general type of snoc-list.

```
data Context (X : Set) : Set where
  ⟨⟩  : Context X
  _,_ : (G : Context X) (S : X) → Context X
```

Variables become typed references into the context.

```
data _⊣_ {X : Set} : Context X → X → Set where
  zero : ∀ {G T}               → G , T ⊣ T
  suc  : ∀ {G S T} (x : G ⊣ T) → G , S ⊣ T
```

Types reflect the typing rules (which are syntax-directed). I exploit comment syntax to write a suggestive line of dashes in the relevant places. I have not managed to persuade `lhs2TeX` to achieve that.

```
data _⊢_ : Context Ty → Ty → Set where
  var : ∀ {G T}   (x : G ⊣ T)
      →   --
                   G ⊢ T
    -- λ-abstraction extends the context
  lam : ∀ {G S T} (b : G, S ⊢ T)
      →   --
                   G ⊢ S ▷ T
    -- application demands a type coincidence
  _$_ : ∀ {G S T} (f : G ⊢ S ▷ T) (s : G ⊢ S)
      →   --
                   G ⊢ T
```

Implementing an evaluator is an exercise in denotational semantics. First, explain what types mean: functions are...functions!

```
⟦_⟧T : Ty → Set
⟦ ι ⟧T       = Nat
⟦ S ▷ T ⟧T   = ⟦ S ⟧T → ⟦ T ⟧T
```

Interpret contexts as types of environments.

```
⟦_⟧C : Context Ty → Set
⟦ ⟨⟩ ⟧C     = 𝟙
⟦ Γ , S ⟧C  = ⟦ Γ ⟧C × ⟦ S ⟧T
```

Interpret variables as projections from environments.

```
⟦_⟧v : ∀ { Γ T } → Γ ⊣ T → ⟦ Γ ⟧C → ⟦ T ⟧T
⟦ zero ⟧v  (_, t)  = t
⟦ suc i ⟧v (g, _)  = ⟦ i ⟧v g
```

Interpret terms, plumbing the environment.

```
⟦_⟧t : ∀ { Γ T } → Γ ⊢ T → ⟦ Γ ⟧C → ⟦ T ⟧T
⟦ var x ⟧t  = ⟦ x ⟧v
⟦ lam b ⟧t  = λ g s → ⟦ b ⟧t (g, s)
⟦ f $ s ⟧t  = λ g → ⟦ f ⟧t g (⟦ s ⟧t g)
eval : ∀ { T } → ⟨⟩ ⊢ T → ⟦ T ⟧T
eval t = ⟦ t ⟧t ⟨⟩
```

Here's an example term. You may notice that Agda cannot fully infer its type, but it is still willing to run it.

```
example : ⟨⟩ ⊢ _
example = (lam (var zero)) $ lam (var zero)
```

2.1.4 An Exercise

Exercise 2.8 (Simultaneous Substitution) *Using a technique of your choice and implementing auxiliary functions as needed, show how to adapt our implementation of scope-safe substitution to type-safe substitution. Define*

```
tsub : ∀ { Γ Δ } → (∀ { T } → Γ ⊣ T → Δ ⊢ T)
                 → (∀ { T } → Γ ⊢ T → Δ ⊢ T)
```

2.1.5 Robbing Peter to Pay Paul

Based on Paul Blain Levy's *call-by-push-value* calculus, here's a variation on the simply typed λ-calculus for you to play with and extend.

Types are separated into *value* types for 'being' and *computation* types for 'doing'. I've supplied some primitive value types.

```
mutual
  data VTy : Set where              -- value types for ways of being
    UNK      : CTy → VTy            -- a suspended computation is a value
    ONE TWO  : VTy                  -- primitive value types
  data CTy : Set where              -- computation types for ways of doing
    EFF  : VTy → CTy                -- making a value by doing effects
    _▷_  : VTy → CTy → CTy          -- abstract a computation
data 𝟚 : Set where tt ff : 𝟚
```

You may wish to add more value types.

To give semantics to these types, we'll need a type of command-response trees. They make a monad. Here I've added a command toss, whose 'ML type' would be $\mathbb{1} \to \mathbb{2}$, but it's really tossing a coin.

```
data Eff (X : Set) : Set where
  ret  : X →                  Eff X
  toss : 𝟙 → (𝟚 → Eff X) → Eff X
```

The ret constructor puts values at the leaves of the tree. Meanwhile, the 'bind', $\gg=$ acts like tree-grafting, pasting new command-response strategies onto the leaves of old.

Exercise 2.9 (Bind for tossing Trees) *Define* $\gg=$ *to graft strategy trees together.*

$$\_\gg=\_ : \forall \{S\ T\} \to \mathsf{Eff}\ S \to (S \to \mathsf{Eff}\ T) \to \mathsf{Eff}\ T$$

You may wish to modify the signature of operations available, but the general structure of Eff X trees will remain the same, with nodes carry commands and edges branching over possible responses.

To give you a better clue of what's going on, let me define the semantics of these types. Values are, er, values in the given type. By contrast, computations are Kleisli arrows—operations which produce Eff-strategies to compute a Return value, given a tuple of Args.

mutual

$$
\begin{aligned}
&[\![\_]\!]_{\mathsf{VT}} : \mathsf{VTy} \to \mathsf{Set}\\
&[\![\ \mathsf{UNK}\ C\ ]\!]_{\mathsf{VT}} = [\![\ C\ ]\!]_{\mathsf{CT}}\\
&[\![\ \mathsf{ONE}\ ]\!]_{\mathsf{VT}} = \mathbb{1}\\
&[\![\ \mathsf{TWO}\ ]\!]_{\mathsf{VT}} = \mathbb{2}\\
&[\![\_]\!]_{\mathsf{CT}} : \mathsf{CTy} \to \mathsf{Set}\\
&[\![\ C\ ]\!]_{\mathsf{CT}} = \mathsf{Args}\ C \to \mathsf{Eff}\ (\mathsf{Return}\ C)\\
&\mathsf{Args} : \mathsf{CTy} \to \mathsf{Set}\\
&\mathsf{Args}\ (\mathsf{EFF}\ V) = \mathbb{1}\\
&\mathsf{Args}\ (V \rhd C) = [\![\ V\ ]\!]_{\mathsf{VT}} \times \mathsf{Args}\ C\\
&\mathsf{Return} : \mathsf{CTy} \to \mathsf{Set}\\
&\mathsf{Return}\ (\mathsf{EFF}\ V) = [\![\ V\ ]\!]_{\mathsf{VT}}\\
&\mathsf{Return}\ (V \rhd C) = \mathsf{Return}\ C
\end{aligned}
$$

We have separated being and doing. There are two categories at work

- $[\![\_]\!]_{\mathsf{CT}}$ gives the subcategory of Set containing just those named by elements of CTy, with morphisms given by

  $$
  \begin{aligned}
  &\_\to_{\mathsf{C}}\_ : \mathsf{CTy} \to \mathsf{CTy} \to \mathsf{Set}\\
  &C \to_{\mathsf{C}} C' = [\![\ C\ ]\!]_{\mathsf{CT}} \to [\![\ C'\ ]\!]_{\mathsf{CT}}
  \end{aligned}
  $$

  and the usual functional identity and composition;

- we have the subcategory of Eff's Kleisli category induced by $[\![\_]\!]_{\mathsf{VT}}$, with objects named by elements of VTy and morphisms being

  $$
  \begin{aligned}
  &\_\to_{\mathsf{V}}\_ : \mathsf{VTy} \to \mathsf{VTy} \to \mathsf{Set}\\
  &V \to_{\mathsf{V}} V' = [\![\ V\ ]\!]_{\mathsf{VT}} \to \mathsf{Eff}\ [\![\ V'\ ]\!]_{\mathsf{VT}}
  \end{aligned}
  $$

  with ret as the identity and $\gg=$ inducing a composition

  $$
  \begin{aligned}
  &\_\circ_{\mathsf{V}}\_ : \{R\ S\ T : \mathsf{Set}\} \to (S \to \mathsf{Eff}\ T) \to (R \to \mathsf{Eff}\ S) \to R \to \mathsf{Eff}\ T\\
  &f \circ_{\mathsf{V}} g = \lambda\, r \to g\ r \gg= f
  \end{aligned}
  $$

  You may wish to check that this composition is associative and absorbs identity on either side.

Exercise 2.10 (Functorial EFF and UNK) *Show that the* EFF *and* UNK *constructors, which turn value types into computation types and vice versa, extend to functors.*

$$\mathsf{EFF}^{\$} : \{V\ V' : \mathsf{VTy}\} \to (V \to_{\mathsf{V}} V') \to (\mathsf{EFF}\ V \to_{\mathsf{C}} \mathsf{EFF}\ V')$$
$$\mathsf{UNK}^{\$} : \{C\ C' : \mathsf{CTy}\} \to (C \to_{\mathsf{C}} C') \to (\mathsf{UNK}\ C \to_{\mathsf{V}} \mathsf{UNK}\ C')$$

Feel free to prove that identity and composition are suitably respected.

Exercise 2.11 (Up the Adjunction) *Now show*

$$C \to_{\mathsf{C}} \mathsf{EFF}\ V \quad \cong \quad \mathsf{UNK}\ C \to_{\mathsf{V}} V$$

$$\mathsf{c2v} : \forall\{C\ V\} \to (C \to_{\mathsf{C}} \mathsf{EFF}\ V) \to (\mathsf{UNK}\ C \to_{\mathsf{V}} V)$$
$$\mathsf{v2c} : \forall\{C\ V\} \to (\mathsf{UNK}\ C \to_{\mathsf{V}} V) \to (C \to_{\mathsf{C}} \mathsf{EFF}\ V)$$

in such a way that the two are mutually inverse.

We've split our monad into an adjunction, connecting distinct notions of value and computation.

Now, let's have some language.

```
mutual
  data Value (Γ : Context VTy) : VTy → Set where
    var   : ∀ {V} → Γ ⊣ V →          Value Γ V
    ⟨⟩    :                          Value Γ ONE
    tt ff :                          Value Γ TWO
    ⌊_⌋   : ∀ {C} → Compt Γ C → Value Γ (UNK C)
  data Compt (Γ : Context VTy) : CTy → Set where
    toss : Compt Γ (EFF TWO)
    lam  : ∀ {V C} → Compt (Γ, V) C →                 Compt Γ (V ▷ C)
    _$_  : ∀ {V C} → Compt Γ (V ▷ C) → Value Γ V → Compt Γ C
    ret  : ∀ {V}   → Value Γ V →                      Compt Γ (EFF V)
    bind : ∀ {V C} → Compt Γ (EFF V) → Compt (Γ, V) C →
                                                      Compt Γ C
    if   : ∀ {C}   → Value Γ TWO → Compt Γ C → Compt Γ C →
                                                      Compt Γ C
```

Here are contexts, interpreted as environment types, with variables represented as value projections.

$$[\![\_]\!]_{\mathsf{CV}} : \mathsf{Context}\ \mathsf{VTy} \to \mathsf{Set}$$
$$[\![\ \langle\rangle\ ]\!]_{\mathsf{CV}} = \mathbb{1}$$
$$[\![\ \Gamma, V\ ]\!]_{\mathsf{CV}} = [\![\ \Gamma\ ]\!]_{\mathsf{CV}} \times [\![\ V\ ]\!]_{\mathsf{VT}}$$
$$[\![\_]\!]_{\mathsf{vv}} : \forall\{\Gamma\ V\} \to \Gamma \dashv V \to [\![\ \Gamma\ ]\!]_{\mathsf{CV}} \to [\![\ V\ ]\!]_{\mathsf{VT}}$$
$$[\![\ \mathsf{zero}\ ]\!]_{\mathsf{vv}}\ (\_, t) = t$$
$$[\![\ \mathsf{suc}\ i\ ]\!]_{\mathsf{vv}}\ (g, \_) = [\![\ i\ ]\!]_{\mathsf{vv}}\ g$$

Now your turn.

Exercise 2.12 (Interpreter) *Define mutually recursive interpreters for values and computations. You should interpret* toss *via the* toss *constructor of* Eff.

```
mutual
  ⟦_⟧vt : ∀ {Γ V} → Value Γ V → ⟦ Γ ⟧CV → ⟦ V ⟧VT
  ⟦_⟧ct : ∀ {Γ C} → Compt Γ C → ⟦ Γ ⟧CV → ⟦ C ⟧CT
```

Exercise 2.13 (Natural Numbers) *Extend the language with a* VTy *of natural numbers, adding* zero *and* suc *constructors to* Value *and an effectful primitive recursor to* Compt*.*

Exercise 2.14 (Input/Output) *Extend* Eff*, and your extended language with* get *and* put *operators, respectively reading and writing natural numbers.*

Exercise 2.15 (State) *Implement an interpreter for* Eff *strategies, treating the* get *and* put *operations as reading and writing a* Nat*-valued state. Feel free to make* toss *work any way you like.*

Next, an exercise received with gratitude from Peter Hancock.

Exercise 2.16 (Interlopers) *Implement an operator which combines two communicating processes* $alice$: Eff $\mathbb{1}$ *and* bob : Eff X *to make a single demand-driven* Eff X *process. Here's the plan:* bob*'s activities should be prioritised; his* puts *should be* put *to the world, but his* gets *should come from* $alice$*'s* puts*;* $alice$ *should run only when* bob *needs input, and should* get *from the world; if* $alice$ *terminates before* bob returns *an* X*,* bob *should* get *the rest of his inputs directly from the world.*

Implement a similar but supply-driven combinator, connecting bob : Eff X *and* $charlie$: Eff $\mathbb{1}$*, where* $charlie$ gets *in the way of* bob*'s* puts*.*

2.1.6 Compare and Swap

Here's a little recursor for pairs of numbers, generalizing a pattern I learned from James McKinna.

$$
\begin{array}{l}
\mathsf{commRec} : \{X : \mathsf{Set}\} \to (\mathsf{Nat} \to X) \to (X \to X) \to \mathsf{Nat} \to \mathsf{Nat} \to X \\
\mathsf{commRec}\ z\ s\ \mathsf{zero}\qquad n \qquad\quad = z\ n \\
\mathsf{commRec}\ z\ s\ m \qquad\ \ \mathsf{zero} \quad = z\ m \\
\mathsf{commRec}\ z\ s\ (\mathsf{suc}\ m)\ (\mathsf{suc}\ n) = s\ (\mathsf{commRec}\ z\ s\ m\ n)
\end{array}
$$

Exercise 2.17 (Commutativity) *Show that* commRec z s *is commutative.*

$$
\begin{array}{l}
\mathsf{commRecComm} : \{X : \mathsf{Set}\}\ (z : \mathsf{Nat} \to X)\ (s : X \to X)\ (m\ n : \mathsf{Nat}) \to \\
\qquad\qquad\qquad \mathsf{commRec}\ z\ s\ m\ n == \mathsf{commRec}\ z\ s\ n\ m
\end{array}
$$

Exercise 2.18 (Arithmetic Operations) *Implement addition and multiplication by suitably instantiating* commRec*. Multiplication is a bit tricky: you may find that you need to compute an extra quantity, alongside the product, in order to make the recursion go through.*

Exercise 2.19 (Comparison Operations) *Implement maximum and minimum by suitably instantiating* commRec*. Implement the equality test.*

I finally gave in and defined the following operation to help with the next exercise. It's the 'uncurry' operation, but with a dependent type which effectively makes it the *dependent case analysis* principle for $\Sigma\ A\ B$: not only do you split a pair ab into pieces a and b, you also learn that ab is a, b. I write it as v as it depicts the splitting of one into two.

$$
\begin{array}{l}
\mathsf{v} : \{A : \mathsf{Set}\}\ \{B : A \to \mathsf{Set}\}\ \{C : \Sigma\ A\ B \to \mathsf{Set}\} \to \\
\qquad ((a : A)\ (b : B\ a) \to C\ (a, b)) \to \qquad \text{-- two on top} \\
\qquad (ab : \Sigma\ A\ B) \to C\ ab \qquad\qquad\qquad \text{-- one below} \\
\mathsf{v}\ f\ ab = f\ (\mathsf{fst}\ ab)\ (\mathsf{snd}\ ab)
\end{array}
$$

Exercise 2.20 (Compare-and-swap) *Use* commRec *to implement* cas*, the operation which sorts a pair of numbers into increasing order.*

```
cas : Nat × Nat → Nat × Nat
```

But where is the λ-calculus? It's on its way. Here are today's *linear* types.

```
data LTy : Set where
  ONE TWO KEY :               LTy
  LIST TREE   :        LTy → LTy
  _⊸_ _⊗_ _&_ : LTy → LTy → LTy
```

Let's consider a *linear* context to be a list of Maybe-types which indicate availability.

```
LCx : Set
LCx = Context (Maybe LTy)
```

We can make a variable reference record the usage by indexing by contexts, before and after.

```
data LV : LCx → LTy → LCx → Set where
  zero : ∀ {Γ T} →                        LV (Γ, yes T) T (Γ, no)
  suc  : ∀ {Γ₀ Γ₁ T S} → LV Γ₀ T Γ₁ → LV (Γ₀, S) T (Γ₁, S)
```

```
data LTm : LCx → LTy → LCx → Set where
  var : ∀ {Γ₀ Γ₁ T} → LV Γ₀ T Γ₁ → LTm Γ₀ T Γ₁
  lam : ∀ {Γ₀ Γ₁ S T} → LTm (Γ₀, yes S) T (Γ₁, no) → LTm Γ₀ (S ⊸ T) Γ₁
  _$_ : ∀ {Γ₀ Γ₁ Γ₂ S T} → LTm Γ₀ (S ⊸ T) Γ₁ → LTm Γ₁ S Γ₂ → LTm Γ₀ T Γ₂
```

Sorry folks, I've got to stop preparing this exercise and prepare the lectures instead. I'll finish it later, but let me tell you where it's going. The plan is to deliver a domain-specific language for transforming containers which neither copy nor delete elements, so that a function of type LIST KEY ⊸ LIST KEY must deliver as output a permutation of its input. Equipped with compare-and-swap for KEY, one should be able to implement sorting functions, guaranteeing the permutation property by construction.

Chapter 3

Views

Views [Wadler, 1987] provide a way to give an alternative interface to an existing type.

We can write a program which transforms our original data to its alternative representation, but in the dependently typed setting, we may and we should get a little more. Let's see how.

Given some 'upper bound', a number u, we may check if any other number, n is below it or not, for example, to check if n may be used to index a vector of size u. However, a Boolean answer alone will not help, if our indexing function vproj demands an element of $\mathsf{Fin}\ u$. We could define a function in $\mathsf{Nat} \to \mathsf{Maybe}\ (\mathsf{Fin}\ u)$ to compute n's representation in $\mathsf{Fin}\ u$ if it exists, but that simple type does not tell us what that representative has to do with n. Alternatively, we can express what it means for n to be *bounds-checkable*: it must be detectably either representable in $\mathsf{Fin}\ u$, or $u + x$ for some 'excess' value x. Let's code up those possibilities.

$$
\begin{array}{l}
\textbf{data}\ \_{-}\mathsf{Bounded?}\_\ (u\ :\ \mathsf{Nat})\ :\ \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{yes}\ :\ (i\ :\ \mathsf{Fin}\ u) \to u\,{-}\mathsf{Bounded?}\ (\mathsf{fog}\ i) \\
\quad \mathsf{no}\ \ :\ (x\ :\ \mathsf{Nat}) \to\ u\,{-}\mathsf{Bounded?}\ (u\ +_{\mathsf{N}}\ x)
\end{array}
$$

If we had a value in $u\,{-}\mathsf{Bounded?}\ n$, inspecting it would tell us which of those two possibilities applies. Let us show that we can always construct such a value. The base cases are straight forward, but something rather unusual happens in the step.

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\_{-}\mathsf{bounded?}\_\ :\ (u\ n\ :\ \mathsf{Nat}) \to u\,{-}\mathsf{Bounded?}\ n} \\
\mathsf{zero}\quad {-}\mathsf{bounded?}\ n & & = & \mathsf{no}\ n \\
(\mathsf{suc}\ u)\,{-}\mathsf{bounded?}\ \mathsf{zero} & & = & \mathsf{yes}\ \mathsf{zero} \\
(\mathsf{suc}\ u)\,{-}\mathsf{bounded?}\ (\mathsf{suc}\ n) & \textbf{with}\ u\,{-}\mathsf{bounded?}\ n & & \\
(\mathsf{suc}\ u)\,{-}\mathsf{bounded?}\ (\mathsf{suc}\ .(\mathsf{fog}\ i)) & \mid\quad \mathsf{yes}\ i & = & \mathsf{yes}\ (\mathsf{suc}\ i) \\
(\mathsf{suc}\ u)\,{-}\mathsf{bounded?}\ (\mathsf{suc}\ .(u\ +_{\mathsf{N}}\ x)) & \mid\quad \mathsf{no}\ x & = & \mathsf{no}\ x
\end{array}
$$

If we are to compare $\mathsf{suc}\ u$ with $\mathsf{suc}\ n$, we surely need to know the result of comparing u with n. The **with** construct McBride and McKinna [2004] allows us to grab the result of that comparison and add a column for it to our pattern match. You can see that the subsequent lines tabulate the possible outcomes of the match, as well as showing patterns for the original arguments. Moreover, something funny happens to those patterns: n becomes instantiated with the non-constructor expressions corresponding to the in- and out-of-bounds cases, marked with a dot. Operationally, there is *no need to check* that n takes the form indicated by the dotted pattern: the operational check is a constructor case analysis on the result of the recursive call, and the consequent analysis of n is forced by the types of those constructors. We work hard to make values in precise types, and we get repaid with information when we inspect those values!

The possibility that that inspecting one value might induce knowledge of another is a phenomenon new with dependent types, and it necessitates some thought about our programming notation, and also our selection of what programs to write. When we write functions to inspect data, we should ask what the types of those functions tell us about what the inspection will learn.

3.0.7 Finite Set Structure

The natural numbers can be thought of as names for finite types. We can equip these finite types with lots of useful structure.

Let's start with the *coproduct* structure, corresponding to addition. We can see Fin $(m +_{\mathsf{N}} n)$ as the disjoint union of Fin m (at the left, low end of the range) and Fin n (at the right, high end of the range). Let us implement the injections. Firstly, finl embeds Fin m, preserving numerical value. I am careful to make the value of m visible, as you can't easily guess it from $m +_{\mathsf{N}} n$.

```
finl : (m : Nat) {n : Nat} → Fin m → Fin (m +N n)
finl zero ()
finl (suc m) zero  =  zero
finl (suc m) (suc i)  =  suc (finl m i)
```

Secondly, finr embeds Fin n by shifting its values up by m.

```
finr : (m : Nat) {n : Nat} → Fin n → Fin (m +N n)
finr zero i  =  i
finr (suc m) i  =  suc (finr m i)
```

Landin: if a job's worth doing, it's worth half-doing.

Injections leave the job half done We need to be able to tell them apart. We can certainly split Fin $(m +_{\mathsf{N}} n)$ as a disjoint union.

```
data _+_ (S T : Set) : Set where
  inl : S → S + T
  inr : T → S + T
```

```
finlr : (m : Nat) {n : Nat} → Fin (m +N n) → Fin m + Fin n
finlr zero    k              =  inr k
finlr (suc m) zero           =  inl zero
finlr (suc m) (suc k) with finlr m k
...                  |  inl i  =  inl (suc i)
...                  |  inr j  =  inr j
```

However, that still leaves work undone. Here's another function of the same type.

```
badlr : (m : Nat) {n : Nat} → Fin (m +N n) → Fin m + Fin n
badlr zero {zero}  ()
badlr zero {suc n} _  =  inr zero
badlr (suc m)      _  =  inl zero
```

As you can see, it ignores its argument, except where necessary to reject the input, and it returns the answer that's as far to the left as possible under the circumstances.

The type of our testing function finlr makes no promise as to what the test will tell us about the value being tested. We compute a value in a disjoint union, but we *learn* nothing about the values we already possess. There's still time to change all that. We can show that the finl and finr injections *cover* Fin $(m +_{\mathsf{N}} n)$ by consrtucting a *view*. Firstly, let us state what it means to be in the image of finl or finr.

```
data FinSum (m n : Nat) : Fin (m +N n) → Set where
  isFinl : (i : Fin m) → FinSum m n (finl m i)
  isFinr : (j : Fin n) → FinSum m n (finr m j)
```

Then let us show that every element is in one image or the other.

```
finSum : (m : Nat) {n : Nat} (k : Fin (m +N n)) → FinSum m n k
finSum zero      k                                 = isFinr k
finSum (suc m) zero                                = isFinl zero
finSum (suc m) (suc k)            with finSum m k
finSum (suc m) (suc .(finl m i)) | isFinl i  = isFinl (suc i)
finSum (suc m) (suc .(finr m j)) | isFinr j  = isFinr j
```

Note that the case analysis on the result of finSum m k exposes which injection made k, directly in the patterns.

3.0.8 Finish the Job

Exercise 3.1 (Products) *Equip* Fin *with its product structure. Implement the constructor*

```
fpair : (m n : Nat) → Fin m → Fin n → Fin (m ×N n)
```

then show that it covers by constructing the appropriate view. Use your view to implement the projections.

Exercise 3.2 (Exponentials) *Implement the exponential function for* Nat*.*

```
_^N_ : Nat → Nat → Nat
```

Now implement the abstraction operator which codifies the finitely many functions between Fin m *and* Fin n*. (You know how to tabulate a function; you know that a vector, like an exponential, is an iterated product.)*

```
flam : (m n : Nat) → (Fin m → Fin n) → Fin (n ^N m)
```

Show that flam *covers, and thus implement application. You will not be able to show that every function is given by applying a code, for that is true only up to an extensional equality which is not realised in Agda.*

Exercise 3.3 (Masochism) *Implement dependent functions and pairs!*

3.0.9 One Song to the Tune of Another (with James McKinna)

Let's define positive binary numbers as snoc-lists of bits.

```
Bin = Context 2
```

We can define a 'one' and a 'successor' operation for these numbers.

```
bone : Bin
bone = ⟨⟩
```

$$
\begin{array}{lcl}
\mathsf{bsuc} : \mathsf{Bin} \to \mathsf{Bin} & & \\
\mathsf{bsuc}\ \langle\rangle & = & \langle\rangle, \mathsf{ff} \\
\mathsf{bsuc}\ (b, \mathsf{ff}) & = & b, \mathsf{tt} \\
\mathsf{bsuc}\ (b, \mathsf{tt}) & = & \mathsf{bsuc}\ b, \mathsf{ff}
\end{array}
$$

It's fun to write binary arithmetic operations, but our mission just now is to establish that we can still *reason* about these numbers as we did with unary numbers. To do so, we must establish Peano's induction principle for binary numbers. That is, we need to implement the following:

$$
\begin{array}{l}
\mathsf{peanoBin} : (P : \mathsf{Bin} \to \mathsf{Set}) \to \\
\qquad\qquad (P\ \mathsf{bone}) \to \\
\qquad\qquad ((b : \mathsf{Bin}) \to P\ b \to P\ (\mathsf{bsuc}\ b)) \to \\
\qquad\qquad (b : \mathsf{Bin}) \to P\ b \\
\mathsf{peanoBin}\ P\ \mathit{pone}\ \mathit{psuc} = \mathsf{help}\ \mathbf{where} \\
\quad \mathsf{help} : (b : \mathsf{Bin}) \to P\ b \\
\quad \mathsf{help}\ b = ?
\end{array}
$$

This goes horribly wrong. How to fix?

Chapter 4

Generic Programming

A *universe* is a collection of types, given as the image of a function. A simple example is the universe

```
data Zero : Set where    -- no constructors!

TT : 2 → Set
TT tt = 1
TT ff = Zero
```

TT gives you a universe of sets corresponding to *decidable* propositions. You can use TT to attach decidable preconditions to functions. The standard example is this

```
le : Nat → Nat → 2
le zero     n            = tt
le (suc m) zero          = ff
le (suc m) (suc n)       = le m n

_-N_ : (m n : Nat) {p : TT (le n m)} → Nat
(m      -N zero)         = m
(zero   -N suc _) {()}
(suc m -N suc n) {p}     = (m -N n) {p}

exampleSubtraction      : Nat
exampleSubtraction      = 42 -N 37

exampleNonSubtraction   : Nat
exampleNonSubtraction   = 37 -N 42
```

Bibliography

Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *CSL*, volume 1683 of *LNCS*, pages 453–468. Springer, 1999.

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

Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of POPL '87*. ACM, 1987.