

Dependently Typed Programming: an Agda introduction

Conor McBride

February 18, 2011

Chapter 1

Vectors and Finite Sets

```
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?

```
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

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

by structural induction [Burstall, 1969] on fs , say. Of course, we could just as well have concluded that $\text{length } (\text{zap } fs \ ss) = \text{length } ss$, 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.

¹by which I mean, not to a computer

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 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.

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

```

 $\_+_{\mathbf{N}}\_ : \mathbf{Nat} \rightarrow \mathbf{Nat} \rightarrow \mathbf{Nat}$ 
 $\mathbf{zero} \ +_{\mathbf{N}} \ y = y$ 
 $\mathbf{suc} \ x \ +_{\mathbf{N}} \ y = \mathbf{suc} \ (x \ +_{\mathbf{N}} \ y)$ 

```

We may define concatenation.

```

 $\_+_{\mathbf{L}}\_ : \{X : \mathbf{Set}\} \rightarrow \mathbf{List} \ X \rightarrow \mathbf{List} \ X \rightarrow \mathbf{List} \ X$ 
 $\langle \rangle \ +_{\mathbf{L}} \ ys = ys$ 
 $(x, xs) \ +_{\mathbf{L}} \ ys = x, (xs \ +_{\mathbf{L}} \ ys)$ 

```

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

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

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.*

```

 $\_ \times_{\mathbf{N}} \_ : \mathbf{Nat} \rightarrow \mathbf{Nat} \rightarrow \mathbf{Nat}$ 

```

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

```

 $\_ -_{\mathbf{N}} \_ : \mathbf{Nat} \rightarrow \mathbf{Nat} \rightarrow \mathbf{Nat}$ 

```

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:*

```

 $\_ \div_{\mathbf{N}} \_ : \mathbf{Nat} \rightarrow \mathbf{Nat} \rightarrow \mathbf{Nat}$ 
 $x \div_{\mathbf{N}} d = \mathbf{help} \ x \ d \ \mathbf{where}$ 
 $\mathbf{help} : \mathbf{Nat} \rightarrow \mathbf{Nat} \rightarrow \mathbf{Nat}$ 
 $\mathbf{help} \ x \ e = \ \_ \{!!\}$ 

```

*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*.

```

 $\mathbf{data} \ \mathbf{Vec} \ (X : \mathbf{Set}) : \mathbf{Nat} \rightarrow \mathbf{Set} \ \mathbf{where}$ 
 $\langle \rangle : \mathbf{Vec} \ X \ \mathbf{zero}$ 
 $\_, - : \{n : \mathbf{Nat}\} \rightarrow X \rightarrow \mathbf{Vec} \ X \ n \rightarrow \mathbf{Vec} \ X \ (\mathbf{suc} \ n)$ 

 $\mathbf{vap} : \{n : \mathbf{Nat}\} \{S \ T : \mathbf{Set}\} \rightarrow \mathbf{Vec} \ (S \rightarrow T) \ n \rightarrow \mathbf{Vec} \ S \ n \rightarrow \mathbf{Vec} \ T \ n$ 
 $\mathbf{vap} \ \langle \rangle \ \langle \rangle = \langle \rangle$ 
 $\mathbf{vap} \ (f, fs) \ (s, ss) = f \ s, \mathbf{vap} \ fs \ ss$ 

```

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.

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

`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.

```
vec : { n : Nat } { X : Set } → X → Vec X n
vec { zero } x = ⟨ ⟩
vec { suc n } x = x, vec x
```

```
-+v+- : { m n : Nat } { X : Set } → Vec X m → Vec X n → Vec X (m +N n)
⟨ ⟩ +v+ ys = ys
(x, xs) +v+ ys = x, (xs +v+ ys)
```

```
vrevapp : { m n : Nat } { X : Set } → Vec X m → Vec X n → Vec X (m +N n)
vrevapp ⟨ ⟩ ys = ys
vrevapp (x, xs) ys = vrevapp xs (x, ys)
```

This is the ‘traverse’ function from the ‘idiom paper’ []

```
vtraverse : { F : Set → Set } →
  ({ X : Set } → X → F X) →
  ({ S T : Set } → F (S → T) → F S → F T) →
  { n : Nat } { X Y : Set } →
  (X → F Y) → Vec X n → F (Vec Y n)
vtraverse pure _/- f ⟨ ⟩ = pure ⟨ ⟩
vtraverse pure _/- f (x, xs) = (pure _/- f x) / vtraverse pure _/- f xs
```

```
ι : { X : Set } → X → X
ι x = x
κ : { X Y : Set } → X → Y → Y
κ x y = x
```

```
vsum : { n : Nat } → Vec Nat n → Nat
vsum = vtraverse (κ zero) _+N- { Y = Nat } ι
```

Here’s a stinker. Of course, you can rejig `vec` to be tail recursive and make `+v+` a stinker.

Which other things work badly? Filter?

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

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

Why is `Y` undetermined?

1.1.1 Matrix Exercises

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

```
Matrix : Nat → Nat → Set → Set
Matrix m n X = Vec (Vec X n) m
```

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

```
vvec : { m n : Nat } { X : Set } → X → Matrix m n X
vvap : { m n : Nat } { S T : Set } →
  Matrix m n (S → T) → Matrix m n S → Matrix m n T
```

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 : Nat } → Matrix m n Nat → Matrix m n Nat → Matrix m n Nat
```

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

`transpose` : $\{m\ n : \text{Nat}\} \{X : \text{Set}\} \rightarrow \text{Matrix } m\ n\ X \rightarrow \text{Matrix } n\ m\ X$

Exercise 1.7 (Identity Matrix) Define a function

`idMatrix` : $\{n : \text{Nat}\} \rightarrow \text{Matrix } n\ n\ \text{Nat}$

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

`_×M_` : $\{l\ m\ n : \text{Nat}\} \rightarrow \text{Matrix } l\ m\ \text{Nat} \rightarrow \text{Matrix } m\ n\ \text{Nat} \rightarrow \text{Matrix } l\ n\ \text{Nat}$

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  $\Sigma$  (S : Set) (T : S → Set) : Set where
  constructor →, _
  field
    fst : S
    snd : T fst
  open  $\Sigma$  public
  _×_ : Set → Set → Set
  S × T =  $\Sigma$  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
```

```
data == { X : Set } (x : X) : X → Set where
  ⟨⟩ : x == x
```

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

```
VecP : Set → Nat → Set
VecP X n = Σ (List X) λ xs → len xs == n
```

```
vnil : { X : Set } → VecP X zero
vnil = ⟨⟩, ⟨⟩
```

```
vcons : { X : Set } { n : Nat } → X → VecP X n → VecP X (suc n)
vcons x (xs, p) = (x, xs), --{!!}
```

```
vapP : { n : Nat } { S T : Set } →
      VecP (S → T) n → VecP S n → VecP T n
vapP (⟨⟩, ⟨⟩) (⟨⟩, ⟨⟩) = ⟨⟩, ⟨⟩
vapP ((f, fs), ⟨⟩) ((s, ss), p) = (f s, vap (fs, ?) (ss, ?)), ?
```

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

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

But this really is toxic.

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.

```
data Fin : Nat → Set where
  zero : { n : Nat } → Fin (suc n)
  suc   : { n : Nat } → (i : Fin n) → Fin (suc n)
foo : { X : Set } { n : Nat } → Fin (zero +N zero) → X
foo ()
```

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

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

Do you resent writing this function? You should.

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

```
vproj : { n : Nat } { X : Set } → Vec X n → Fin n → X
vproj ⟨⟩ ()
vproj (x, xs) zero   = x
vproj (x, xs) (suc i) = vproj xs i
```

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

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.

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

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 `sucs` 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*

$$\text{OPI} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Set}$$

such that $\text{OPI } m \ n$ gives a unique first-order representation to exactly the order-preserving injections from $\text{Fin } m$ to $\text{Fin } n$, 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**?

```
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)
```

How many different
kinds of trouble are
we in?

Simultaneous substitution?

```
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)
```

Notoriously not
structurally recursive.

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  $\rho$  (suc  $i$ ) = suc ( $\rho$   $i$ )
ren : {  $m$   $n$  : Nat }  $\rightarrow$  (Fin  $m \rightarrow$  Fin  $n$ )  $\rightarrow$  Tm  $m \rightarrow$  Tm  $n$ 
ren  $\rho$  (var  $i$ ) = var ( $\rho$   $i$ )
ren  $\rho$  ( $f$  $  $a$ ) = ren  $\rho$   $f$  $ ren  $\rho$   $a$ 
ren  $\rho$  (lam  $b$ ) = lam (ren (wkr  $\rho$ )  $b$ )

```

And you can define substitution, given renaming.

```

wks : {  $m$   $n$  : Nat }  $\rightarrow$  (Fin  $m \rightarrow$  Tm  $n$ )  $\rightarrow$  Fin (suc  $m$ )  $\rightarrow$  Tm (suc  $n$ )
wks  $\sigma$  zero = var zero
wks  $\sigma$  (suc  $i$ ) = ren suc ( $\sigma$   $i$ )
sub : {  $m$   $n$  : Nat }  $\rightarrow$  (Fin  $m \rightarrow$  Tm  $n$ )  $\rightarrow$  Tm  $m \rightarrow$  Tm  $n$ 
sub  $\sigma$  (var  $i$ ) =  $\sigma$   $i$ 
sub  $\sigma$  ( $f$  $  $a$ ) = sub  $\sigma$   $f$  $ sub  $\sigma$   $a$ 
sub  $\sigma$  (lam  $b$ ) = lam (sub (wks  $\sigma$ )  $b$ )

```

How repetitive! Let's abstract out the pattern.

```

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

wk : {  $I$  : Nat  $\rightarrow$  Set }  $\rightarrow$  Kit  $I \rightarrow$  {  $m$   $n$  : Nat }  $\rightarrow$ 
  (Fin  $m \rightarrow$   $I$   $n$ )  $\rightarrow$  Fin (suc  $m$ )  $\rightarrow$   $I$  (suc  $n$ )
wk  $k$   $\tau$  zero = mkv  $k$  zero
wk  $k$   $\tau$  (suc  $i$ ) = wki  $k$  ( $\tau$   $i$ )

act : {  $I$  : Nat  $\rightarrow$  Set }  $\rightarrow$  Kit  $I \rightarrow$  {  $m$   $n$  : Nat }  $\rightarrow$ 
  (Fin  $m \rightarrow$   $I$   $n$ )  $\rightarrow$  Tm  $m \rightarrow$  Tm  $n$ 
act  $k$   $\tau$  (var  $i$ ) = mkt  $k$  ( $\tau$   $i$ )
act  $k$   $\tau$  ( $f$  $  $a$ ) = act  $k$   $\tau$   $f$  $ act  $k$   $\tau$   $a$ 
act  $k$   $\tau$  (lam  $b$ ) = lam (act  $k$  (wk  $k$   $\tau$ )  $b$ )

```

2.1.1 Exercises

Exercise 2.1 (Renaming Kit) Define the renaming kit.

```
renk : Kit Fin
```

Exercise 2.2 (Substitution Kit) Define the substitution kit.

```
subk : Kit Tm
```

Exercise 2.3 (Substitute zero) `sub0` : { n : Nat } \rightarrow Tm $n \rightarrow$ Tm (suc n) \rightarrow 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 }  $\rightarrow$  Tm  $n \rightarrow$  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).

$\text{develop} : \{n : \text{Nat}\} \rightarrow \text{Tm } n \rightarrow \text{Tm } n$

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.

$\text{iterate} : \text{Nat} \rightarrow \{X : \text{Set}\} \rightarrow (X \rightarrow X) \rightarrow X \rightarrow X$

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

$\text{-}+_a\text{-} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$
 $\text{zero } +_a n = n$
 $\text{suc } m +_a n = m +_a \text{suc } n$

let

$\text{Sub} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Set}$
 $\text{Sub } m n = (w : \text{Nat}) \rightarrow \text{Fin } (w +_a m) \rightarrow \text{Tm } (w +_a n)$

be the type of substitutions which can be weakened. Define

$\text{subw} : \{m n : \text{Nat}\} \rightarrow \text{Sub } m n \rightarrow \text{Tm } m \rightarrow \text{Tm } n$

Now show how to turn a renaming into a Sub .

$\text{renSub} : \{m n : \text{Nat}\} \rightarrow (\text{Fin } m \rightarrow \text{Fin } n) \rightarrow \text{Sub } m n$

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

$\text{subSub} : \{m n : \text{Nat}\} \rightarrow (\text{Fin } m \rightarrow \text{Tm } n) \rightarrow \text{Sub } m n$

2.1.2 How to Hide de Bruijn Indices

$\text{max} : \{n : \text{Nat}\} \rightarrow \text{Fin } (\text{suc } n)$
 $\text{max } \{\text{zero}\} = \text{zero}$
 $\text{max } \{\text{suc } n\} = \text{suc } (\text{max } \{n\})$
 $\text{embed} : \{n : \text{Nat}\} \rightarrow \text{Fin } n \rightarrow \text{Fin } (\text{suc } n)$
 $\text{embed } \text{zero} = \text{zero}$
 $\text{embed } (\text{suc } n) = \text{suc } (\text{embed } n)$

 $\text{shifty} : (m : \text{Nat}) \{n : \text{Nat}\} \rightarrow \text{Fin } (\text{suc } (m +_N n))$
 $\text{shifty } \text{zero} = \text{max}$
 $\text{shifty } (\text{suc } m) = \text{embed } (\text{shifty } m)$
 $\text{lambda} : \{m : \text{Nat}\} \rightarrow$
 $\quad ((\{n : \text{Nat}\} \rightarrow \text{Tm } (\text{suc } (m +_N n))) \rightarrow \text{Tm } (\text{suc } m)) \rightarrow$
 $\quad \text{Tm } m$
 $\text{lambda } \{m\} f = \text{lam } (f \lambda \{n\} \rightarrow \text{var } (\text{shifty } m \{n\}))$
 $\text{myTest} : \text{Tm } \text{zero}$
 $\text{myTest} = \text{lambda } \lambda f \rightarrow \text{lambda } \lambda x \rightarrow f \$ (f \$ x)$

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.

$$\begin{aligned} \llbracket _ \rrbracket_v &: \forall \{ F T \} \rightarrow F \dashv T \rightarrow \llbracket F \rrbracket_c \rightarrow \llbracket T \rrbracket_\tau \\ \llbracket \text{zero} \rrbracket_v &(-, t) = t \\ \llbracket \text{suc } i \rrbracket_v &(g, -) = \llbracket i \rrbracket_v g \end{aligned}$$

Interpret terms, plumbing the environment.

$$\begin{aligned} \llbracket _ \rrbracket_t &: \forall \{ F T \} \rightarrow F \vdash T \rightarrow \llbracket F \rrbracket_c \rightarrow \llbracket T \rrbracket_\tau \\ \llbracket \text{var } x \rrbracket_t &= \llbracket x \rrbracket_v \\ \llbracket \text{lam } b \rrbracket_t &= \lambda g s \rightarrow \llbracket b \rrbracket_t (g, s) \\ \llbracket f \$ s \rrbracket_t &= \lambda g \rightarrow \llbracket f \rrbracket_t g (\llbracket s \rrbracket_t g) \\ \text{eval} &: \forall \{ T \} \rightarrow \langle \rangle \vdash T \rightarrow \llbracket T \rrbracket_\tau \\ \text{eval } t &= \llbracket t \rrbracket_t \langle \rangle \end{aligned}$$

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 Exercises

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*

$$\begin{aligned} \text{tsub} &: \forall \{ F \Delta \} \rightarrow (\forall \{ T \} \rightarrow F \dashv T \rightarrow \Delta \vdash T) \\ &\quad \rightarrow (\forall \{ T \} \rightarrow F \vdash T \rightarrow \Delta \vdash T) \end{aligned}$$

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
  UNK      : CTy → VTy -- a suspended computation is a value
  ONE TWO  : VTy       -- primitive value types
data CTy : Set where
  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 $\mathbb{1} \rightarrow \mathbb{2}$, 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', `>>=` 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.

$$\begin{aligned} _ \gg _ &: \forall \{S\ T\} \rightarrow \text{Eff } S \rightarrow (S \rightarrow \text{Eff } T) \rightarrow \text{Eff } T \\ \text{ret } x &\gg f = f\ x \\ \text{toss } u\ k &\gg f = \text{toss } u\ \lambda b \rightarrow k\ b \gg f \end{aligned}$$

You may wish to modify the signature of operations available, but the general structure of $\text{Eff } X$ 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 .

$$\begin{aligned} \llbracket _ \rrbracket_{\text{VT}} &: \text{VTy} \rightarrow \text{Set} \\ \llbracket \text{UNK } C \rrbracket_{\text{VT}} &= \llbracket C \rrbracket_{\text{CT}} \\ \llbracket \text{ONE} \rrbracket_{\text{VT}} &= \mathbf{1} \\ \llbracket \text{TWO} \rrbracket_{\text{VT}} &= \mathbf{2} \\ \llbracket _ \rrbracket_{\text{CT}} &: \text{CTy} \rightarrow \text{Set} \\ \llbracket C \rrbracket_{\text{CT}} &= \text{Args } C \rightarrow \text{Eff } (\text{Return } C) \\ \text{Args} &: \text{CTy} \rightarrow \text{Set} \\ \text{Args } (\text{EFF } V) &= \mathbf{1} \\ \text{Args } (V \triangleright C) &= \llbracket V \rrbracket_{\text{VT}} \times \text{Args } C \\ \text{Return} &: \text{CTy} \rightarrow \text{Set} \\ \text{Return } (\text{EFF } V) &= \llbracket V \rrbracket_{\text{VT}} \\ \text{Return } (V \triangleright C) &= \text{Return } C \end{aligned}$$

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

- $\llbracket _ \rrbracket_{\text{CT}}$ gives the subcategory of Set containing just those named by elements of CTy , with morphisms given by

$$\begin{aligned} \rightarrow_{\text{C}} &: \text{CTy} \rightarrow \text{CTy} \rightarrow \text{Set} \\ C \rightarrow_{\text{C}} C' &= \llbracket C \rrbracket_{\text{CT}} \rightarrow \llbracket C' \rrbracket_{\text{CT}} \end{aligned}$$

and the usual functional identity and composition;

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

$$\begin{aligned} \rightarrow_{\text{V}} &: \text{VTy} \rightarrow \text{VTy} \rightarrow \text{Set} \\ V \rightarrow_{\text{V}} V' &= \llbracket V \rrbracket_{\text{VT}} \rightarrow \text{Eff } \llbracket V' \rrbracket_{\text{VT}} \end{aligned}$$

with ret as the identity and \gg inducing a composition

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

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.

$$\begin{aligned} \text{EFF}^{\$} &: \{V\ V' : \text{VTy}\} \rightarrow (V \rightarrow_{\text{V}} V') \rightarrow (\text{EFF } V \rightarrow_{\text{C}} \text{EFF } V') \\ \text{UNK}^{\$} &: \{C\ C' : \text{CTy}\} \rightarrow (C \rightarrow_{\text{C}} C') \rightarrow (\text{UNK } C \rightarrow_{\text{V}} \text{UNK } C') \end{aligned}$$

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

Exercise 2.11 (Up the Adjunction) *Now show*

$$C \rightarrow_C \text{EFF } V \cong \text{UNK } C \rightarrow_V V$$

$$\text{c2v} : \forall \{C \ V\} \rightarrow (C \rightarrow_C \text{EFF } V) \rightarrow (\text{UNK } C \rightarrow_V V)$$

$$\text{v2c} : \forall \{C \ V\} \rightarrow (\text{UNK } C \rightarrow_V V) \rightarrow (C \rightarrow_C \text{EFF } V)$$

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.

```
[_]cv : Context VTy → Set
[⟨⟩]cv = 1
[Γ, V]cv = [Γ]cv × [V]vt
[_]vw : ∀ {Γ V} → Γ ⊢ V → [Γ]cv → [V]vt
[zero]vw (−, t) = t
[suc i]vw (g, −) = [i]vw 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.*

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 $Fin\ u$. We could define a function in $Nat \rightarrow Maybe\ (Fin\ u)$ to compute n 's representation in $Fin\ u$ 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 $Fin\ u$, or $u + x$ for some 'excess' value x . Let's code up those possibilities.

```
data  $\text{u-Bounded?}$  (u : Nat) : Nat  $\rightarrow$  Set where
  yes : (i : Fin u)  $\rightarrow$   $\text{u-Bounded?}$  (fog i)
  no  : (x : Nat)  $\rightarrow$   $\text{u-Bounded?}$  (u +N x)
```

If we had a value in $\text{u-Bounded?}\ n$, 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.

```
 $\text{u-bounded?}$  : (u n : Nat)  $\rightarrow$   $\text{u-Bounded?}\ n$ 
zero  $\text{u-bounded?}\ n$  = no n
(suc u)  $\text{u-bounded?}\ zero$  = yes zero
(suc u)  $\text{u-bounded?}\ (suc\ n)$  with  $\text{u-bounded?}\ n$ 
(suc u)  $\text{u-bounded?}\ (suc\ .(fog\ i))$  | yes i = yes (suc i)
(suc u)  $\text{u-bounded?}\ (suc\ .(u +_N x))$  | no x = no x
```

If we are to compare $\text{suc}\ u$ with $\text{suc}\ n$, 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.

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.