Dependently Typed Programming: an Agda introduction

Conor McBride

February 7, 2011

Chapter 1

Vectors and Finite Sets

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

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.

```
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.

```
data Nat : Set where
  zero :       Nat
  suc  : Nat → Nat
```

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

```
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 $+_N$?

```
_+N_ : Nat → Nat → Nat
zero  +N y = y
suc x +N y = suc (x +N y)
```

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

We may define concatenation.

```
_+L+_ : {X : Set} → List X → List X → List X
⟨⟩       +L+ ys = ys
(x, xs)  +L+ ys = x, (xs +L+ ys)
```

It takes a proof by induction (and a convenient definition of $+_N$) to note that

$$\mathsf{length}\ (xs\ +_L+\ ys) = \mathsf{length}\ xs\ +_N\ \mathsf{length}\ ys$$

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.*

```
_×N_ : Nat → Nat → 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.*

```
_-N_ : Nat → Nat → 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:*

```
_÷N_ : Nat → Nat → Nat
x ÷N d = help x d where
  help : Nat → Nat → Nat
  help x e =    -- {!!}
```

You can recursively peel suc*s 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

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

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

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

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

vec is an example of

```
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)
```

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

```
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) !} |
```

Here's a stinker. Of course, you can rejig $+_N$ 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.

```
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
```

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

```
I : {X : Set} → X → X
I x = x
K : {X Y : Set} → X → Y → X
K x y = x
```

Why is Y undetermined?

```
vsum : {n : Nat} → Vec Nat n → Nat
vsum = vtraverse (K zero) _+N_ {Y = Nat} I
```

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 : 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
```

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
```

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

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

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

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


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

It's already getting bad here, but we can match p against ⟨⟩ and complete.

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

But this really is toxic.

```
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, ?)), ?
```

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)
```

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.

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

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.

```
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
```

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*

```
OPI : Nat → Nat → Set
```

such that OPI m n *gives a unique first-order representation to exactly the order-preserving injections from* Fin m *to* 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).*

```
develop : {n : Nat} → Tm n → 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.*

```
iterate : Nat → {X : Set} → (X → X) → X → 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*

```
_+a_ : Nat → Nat → Nat
zero    +a n = n
suc m   +a n = m +a suc n
```

let

```
Sub : Nat → Nat → Set
Sub m n = (w : Nat) → Fin (w +a m) → Tm (w +a n)
```

be the type of substitions which can be weakened. Define

```
subw : {m n : Nat} → Sub m n → Tm m → Tm n
```

Now show how to turn a renaming into a Sub*.*

```
renSub : {m n : Nat} → (Fin m → Fin n) → Sub m n
```

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

```
subSub : {m n : Nat} → (Fin m → Tm n) → Sub m n
```

2.1.2 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 _⊣_
infixr 4 _,_

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

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

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


data _⊢_ : Context → 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


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


⟦_⟧C : Context → Set
⟦ ⟨⟩ ⟧C     = 𝟙
⟦ G, S ⟧C  = ⟦ G ⟧C × ⟦ S ⟧T
⟦_⟧v : ∀ { G T } → G ⊣ T → ⟦ G ⟧C → ⟦ T ⟧T
⟦ zero ⟧v  (_, t)  = t
⟦ suc i ⟧v (g, _)  = ⟦ i ⟧v g
⟦_⟧t : ∀ { G T } → G ⊢ T → ⟦ G ⟧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 ⟨⟩


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

Chapter 3

Views

```
data _-Bounded?_ (u : Nat) : Nat → Set where
  yes : (i : Fin u) →   u -Bounded? (fog i)
  no  : (x : Nat) →     u -Bounded? (u +N  x)
_-bounded?_ : (u n : Nat) → u -Bounded? n
zero     -bounded? n       =  no n
(suc u) -bounded? zero  =  yes zero
(suc u) -bounded? (suc n)               with u -bounded? n
(suc u) -bounded? (suc .(fog i))       |     yes i  =  yes (suc i)
(suc u) -bounded? (suc .(u +N  x))  |     no x   =  no x
```

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.