Dependently Typed Programming: an Agda introduction

Conor McBride

January 31, 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.

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

```
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 $+_{\mathsf{N}}$?

```
_+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.

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

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

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

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

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

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

vec is an example of

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

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

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

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

Here's a stinker. Of course, you can rejig $+_{\mathsf{N}}$ to be tail recursive and make $+\mathsf{V}+$ a stinker.

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?

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

Why is Y undetermined?

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

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.*

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

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
```

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

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

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

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

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

But this really is toxic.

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

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.

$$\textbf{data}\ \mathsf{Fin} : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where}$$
$$\quad \mathsf{zero} : \{n : \mathsf{Nat}\} \to \qquad\qquad \mathsf{Fin}\ (\mathsf{suc}\ n)$$
$$\quad \mathsf{suc}\ \ : \{n : \mathsf{Nat}\} \to \mathsf{Fin}\ n \to \mathsf{Fin}\ (\mathsf{suc}\ n)$$

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

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

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

Chapter 2

Lambda Calculus with de Bruijn Indices

Chapter 3

Views

If you resent writing fog, your instincts are sound!

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

```
data _-Bounded?_ (u : Nat) : Nat → Set where
  yes : (i : Fin u) →  u -Bounded? (fog i)
  no  : (x : Nat) →    u -Bounded? (u <+> 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 <+> x))      |  no x   = no x
```

Bibliography

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