Approaching Agda

or

Coq'n'Agda'n'Idris
(Especially the Agda Part)

Adrian King
16 May 2016

Part 0 (Magic for Beginners): Dependent Types and Types as Propositions

Non-Dependent Types Parameterized by Types

- In a language like Haskell, you can declare a type constructor that takes a type as a parameter:

```
data List a =
    Nil | Cons a (List a)
```

- a can be bound to any type.
- List isn't a type itself, but a sort of type-level function that returns a type when given a type argument, like:

```
List Int
```

Dependent Types Parameterized by Values

- An Agda type can be parameterized by a value that is not a type, such as a natural number (ℕ):

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

- A Vec is a list of a specified length. Vec Nat 3 is a list of 3 Nats.
- Being able to refer to values in types makes types more expressive.

But Wait, There's More!

- Values in dependent types are not limited to constant arguments to datatype constructors:

```
_++_ :
  {A : Set} {m n : ℕ} →
    Vec A m → Vec A n → Vec A (m + n)
[] ++ as = as
(h :: t) ++ as = h :: (t ++ as)
```

- ++ is a function that takes two vectors of sizes m and n and returns one of size m + n.
- Note that m + n is a function call—you can do arbitrary computation inside a type expression.

As If That Weren't Enough: The Curry-Howard Correspondence

- Haskell Curry and William Alvin Howard (among others) developed the idea that a type corresponds to a logical proposition, and values of the type to proofs of the proposition.
- 42 is a proof of ℕ.
- Less trivially, if we are given:

```
data composite : ℕ → Set where
  p22 : (m n : ℕ) → composite ((suc (suc m)) * (suc (suc n)))
```

then composite 6 is a type (and therefore a proposition) with proof:

```
p22 (suc zero) zero
```

Why You Are Thrilled

- Dependent types are expressive enough to describe what your code does in a lot more detail than other type systems.
- Dependent types let you do rigorous mathematical proofs.
- But so what?
- The promise is this: you can use dependent types to write detailed, machine- (and human-!) readable specifications for your code, and prove the code correct.
- No bugs, ever!

Why You Are Cynical

- The dependent type promise has two catches:
  - The compiler may need help proving that your code meets your specifications (sometimes a lot of help).
  - Specifications may be harder to write than code (sometimes a lot harder).
- Large-scale programming with dependent types is still a work in progress.

Part 1:
A Brief History

Computers Prove Theorems

- The first uses of computers to prove theorems date back to the 1950s when the first general-purpose computers appeared.
- Mathematical reasoning, when completely formalized, is an obvious target for automation.
- Early theorem provers were not based on dependent types, but on simpler systems like Presburger arithmetic.

Computers Prove Theorems about Computer Programs

- Writing correct programs is hard (duh).
- We'd like to make computer programs themselves the objects of automated reasoning.
- From the *Stanford Pascal Verifier User Manual*, 1979:

```
FUNCTION G(X0,Y0: INTEGER): INTEGER;
ENTRY X0>0 ∧ Y0>0;
EXIT G=GCD(X0,Y0);
VAR X,Y,R: INTEGER;
BEGIN
   X ← X0; Y ← Y0;
   REPEAT R ← MOD(X,Y);
            X ← Y;
            Y ← R
   UNTIL Y = 0
   INVARIANT GCD(X0,Y0) = GCD(X,Y) ∧ X > 0 ∧ Y ≥ 0;
   G ← X
END;
```

Proof Search

- *Proof search* tries to construct proofs without human input.
- The earliest computer theorem provers did proof search, but general proof search is really slow.
- Prolog (Colmerauer, 1972) and subsequent logic programming languages can be viewed as doing proof search for clauses of a particular form.

Proof Checking

- *Proof checking* relies on a human being to construct a proof. The automated proof checker verifies that the proof is correct.
- In most type theories, it is much easier to check a proof than to construct one.

Typechecking vs. Inference

- Compilers for statically typed languages do proof checking when they verify that type annotations in the program are correct.
- Type (or term) inference is a form of proof search, not proof checking.
- Many statically typed languages perform type inference, at least in some contexts.
- In Agda mode in emacs, typing CTRL-C CTRL-A in a hole makes Agda perform a (limited) search for a term that works there (term inference).

Proof Assistants Based on Dependent Types

- N. G. de Bruijn started the Automath project in 1967. It was the first to use dependent types.
- Automath's the type theory differed from more recent systems, which incorporate ideas of Per Martin-Löf (1960s–1980s).
- De Bruijn argued that proofs should be represented as *proof objects*, that is, terms of a dependently typed language whose correctness can be verified by a typechecker.

Coq

- Coq is the oldest of the languages we're looking at (1984).
- Developed by Thierry Coquand and Gérard Huet.
- Written in OCaml.
- Underlying type theory is called the Calculus of Constructions.
- Coq is a big system that has been worked on by many people, with lots of libraries.
- Several well-known proofs have been done in Coq (e.g., the four color map theorem).

Agda and Idris

- Agda was developed by Catarina Coquand and Ulf Norell; documented in Norell's PhD thesis (2007) at Chalmers University in Sweden.
- Underlying type theory derives from Zhaohui Luo's UTT.
- Idris was created by Edwin Brady at the University of St. Andrews in Scotland.
- Work in progress since 2011.
- Brady is also an author of the language Whitespace, in which printing characters are ignored.

Part 2:
Installation

Not by Language Alone

- The usual way of running each of these languages depends on more than just the compiler and standard library.

Coq 8.4pl3	Agda 2.4.2.2	Idris 0.11
• emacs • Proof General (seems to be more reliable than coqtop)	• Haskell (GHC) • emacs • Agda emacs mode • DejaVu fonts (or any font with enough weird math symbols)	• C compiler (back end)

Installing Agda on Windows

- Agda has an MSI installer for Windows that includes Haskell, emacs, and all necessary fonts.
- It takes an hour to download.
- The MSI didn't seem to do anything except mess up my existing emacs installation.
- Solution: go through the 25-step (!) installation script by hand (about 45 minutes).

Installing Agda on Ubuntu

- There is a standard `agda` package for Ubuntu.
- Less than 5 minutes to install.
- Includes all the same pieces as the Windows installation script.

Not-So-Standard Library

- Agda has a standard library, but it isn't installed by default.
- An Agda program starts out with *nothing* defined for you.
- Download and untar the standard library version that matches your version of Agda, and follow the instructions to tell emacs (!) where you put it.

Part 3:
Tiny Examples

Natural Numbers

- Inductively defined natural numbers and addition in each language:

Nats.v (Coq)	Nats.agda (Agda)	Nats.idr (Idris)

Inductive Nat: Type :=
  | Z: Nat
  | S: Nat -> Nat.
 	
Fixpoint plus (m n: Nat) :=
  match m with
  | Z => n
  | S m' => S (plus m' n)
  end.
 	
Notation "x + y" :=
  (plus x y)
  (at level 50,
  left associativity).
	module Nats where
	
	data Nat : Set where
	  Z : Nat
	  S : Nat → Nat
	
	_+_ : Nat → Nat → Nat
	Z + n = n
	(S m) + n = S (m + n)
		module Nats
		
		%access public export
		%default total
		
		-- The name "Nat" conflicts
		-- with built-in Nat
		
		data Nat' = Z' | S' Nat'
		
		(+): Nat' -> Nat' -> Nat'
		(+) Z' n = n
		(+) (S' m) n = S' (m + n)

Natural Numbers and Addition Don't Require Dependent Types

- This game is so easy, even Haskell and Scala can play.

Nats.hs (Haskell)	Nats.scala (Scala)

module Nats where
 	
data Nat = Z | S Nat
 	
(+):: Nat -> Nat -> Nat
(+) Z n = n
-- Haskell is confused about
-- whether I mean Prelude.+
(+) (S m) n = S ((Nats.+) m n)
	object Nats {
	
	  sealed trait Nat {
	  def + (that: Nat): Nat = this match {
	  case Z => that
	  case S(n) => S(n + that)
	  }
	  }
	  object Z extends Nat
	  case class S (n: Nat) extends Nat
	
	}

Odds and Evens

- You need dependent types to express a proposition that an inductively declared natural number is odd or even.

Odds.v (Coq)	Odds.agda (Agda)	Odds.idr (Idris)

Require Export Nats.
 	
Inductive Odd: Nat -> Prop :=
  | Odd1: Odd (S Z)
  | OddSS:
  forall n: Nat,
  Odd n -> Odd (S (S n)).
 	
Theorem Odd_5:
  Odd (S (S (S (S (S Z))))).
Proof.
  constructor. constructor.
  constructor.
Qed.
	open import Nats
	
	data Odd : Nat → Set where
	  Odd1 : Odd (S Z)
	  OddSS :
	  {n : Nat} → Odd n →
	  Odd (S (S n))
	
	Odd5 :
	  Odd (S (S (S (S (S Z)))))
	Odd5 = OddSS (OddSS Odd1)
		module Odds
		
		import Nats
		
		data Odd: Nat' -> Type where
		  Odd1: Odd (S' Z')
		  OddSS:
		  Odd n -> Odd (S' (S' n))
		
		Odd5:
		  Odd (S' (S' (S' (S' (S' Z')))))
		Odd5 = OddSS (OddSS Odd1)

Part 4:
Style

Characters

- Agda's standard library makes heavy use of Unicode.
- Increase the font size in your text editor, or distinguishing the characters will give you eyestrain.
- Coq and Idris have provisions for Unicode, but don't prefer it. The standard libraries are mostly readable in ASCII.

Aesthetic Coherence

- Coq is really an amalgam of 3 languages:
  - Gallina, a dependently typed functional programming language
  - The Vernacular, for top-level definitions and REPL commands
  - Ltac, an imperative language for tactics
- Agda and Idris (being newer) are simpler and more uniform.
- There is not much to them beyond the dependently typed functional language.

Proof Construction

- Nontrivial proofs in Coq are usually built with tactics, not Gallina code.
- The tactic language may be a better match for the way mathematicians think about proofs, but it can be quite verbose.
- Agda has no tactics, but emacs commands that help you fill in holes in a program play a similar role.
- Idris has a tactic language, but it's poorly documented. At least for now, you usually write proofs directly in Idris.

Part 5:
The Proof of the Pudding

Examples

- Following are a couple of problems and how you might go about solving them in Agda.
- I'll post the Agda code along with the slides at https://github.com/archontophoenix/agda-intro-talk.
- The first example also has translations into Coq and Idris for comparison.

Part 5a:
Evens and Odds

Separating Odds from Evens

- In this example, we start with an inductive definition of even and odd numbers, and show that any natural number can be classified as one or the other.
- Distinguishing even and odd numbers isn't the most pressing problem in building software, but it's quick and easy and shows off the style of mathematical proofs in Agda.

Imports and Module Declaration

- First, import the parts of the standard library we use (remember, Agda does not import a prelude by default):

```
open import Data.Nat
open import Data.Sum
open import Relation.Binary.Core
open import Relation.Nullary

module EvensAndOdds where
```

Datatype Definitions

- There are different ways to encode even and odd numbers, including mutually recursive definitions. This one isn't mutually recursive; it defines odd in terms of even.

```
data even : ℕ → Set where
  even0 : even 0
  evenSS : {n : ℕ} → even n → even (suc (suc n))

data odd : ℕ → Set where
  oddS : {n : ℕ} → even n → odd (suc n)
```

Order in Which Even and Odd Numbers Occur

- The oddS constructor tells us that an odd number occurs after an even number. It's also nice to know that an even number occurs after an odd number.
- What about the even0 case?

```
evenAfterOdd : {n : ℕ} → even (suc n) → odd n
evenAfterOdd (evenSS evenPredN) =
  oddS evenPredN

oddThenEven : {n : ℕ} → odd n → even (suc n)
oddThenEven (oddS evenPredN) =
  evenSS evenPredN
```

odd vs. Not even

- It's obvious to *us* that “even” and “not odd” are the same thing. Not so obvious to Agda.
- ¬ *x* is short for *x* → ⊥ (⊥ is the empty type).
- Again, Agda realizes that the 0 cases are absurd, and doesn't make us write them out.

```
evenNotOdd : {n : ℕ} → even n → ¬ (odd n)
evenNotOdd (evenSS evenPPN) (oddS evenPN) =
  evenNotOdd evenPN (oddS evenPPN)

oddNotEven : {n : ℕ} → odd n → ¬ (even n)
oddNotEven (oddS evenPN) (evenSS evenPPn) =
  oddNotEven (oddS evenPPn) evenPN
```

The Sheep from the Goats

- Every natural number is either even or odd.
- ⊎, inj_1, and inj_2 are Agda for Either, Left, and Right.
- The with clause lets you do a pattern match inside a pattern match.

```
evenOrOdd : (n : ℕ) → even n ⊎ odd n
evenOrOdd 0 = inj₁ even0
evenOrOdd (suc pN) with evenOrOdd pNC
evenOrOdd (suc pN)
  | inj₁ evenPN = inj₂ (oddS evenPN)
evenOrOdd (suc pN)
  | inj₂ oddPN  = inj₁ (oddThenEven oddPN)
```

Decidability of Evenness

- This is a proof of the *decidability* of evenness: you can create a function that tells whether even is true or false of any natural number.

```
evenDecidable : (n : ℕ) → even n ⊎ ¬ (even n)
evenDecidable n with evenOrOdd n
evenDecidable n
  | inj₁ evenN = inj₁ evenN
evenDecidable n
  | inj₂ oddN  = inj₂ (oddNotEven oddN)
```

Part 5b:
Towards MergeSort

All Is Not So Clear and Simple

- Proving Mergesort correct seems like a useful exercise.
- Frequently used library functions are the ones whose correctness we depend on most.
- There exist certified implementations of Mergesort in Agda.
- Problem: those implementations are not small or easy to explain in a single lecture.
- My own attempt was like a bad blind date with Agda. Here's the whole awful story.

Magic from the Standard Library

- Writing “ordinary code” requires importing some possibly surprising-sounding pieces of the standard library:

```
open import Data.List
open import Data.Nat hiding (_⊔_)
open import Data.Nat.Properties
open import Data.Product
open import Level hiding (suc)
open import Relation.Binary
open import Relation.Nullary
```

DecTotalOrder Is Parameterized by Universe Levels

- We'll sort only types that have a decidable total order, as expressed by the library type DecTotalOrder. DecTotalOrder is parameterized by the universe levels of its carrier type and equality and less-than-or-equal operators (!).

```
module MergeSort {ℓ ℓ₁ ℓ₂} (
  ord : DecTotalOrder ℓ ℓ₁ ℓ₂) where

A : Set ℓ
A = DecTotalOrder.Carrier ord
```

Fishing Out the Useful Bits of DecTotalOrder

- DecTotalOrder is a big bag of tricks.

```
_=A_ : Rel A ℓ₁
_=A_ = DecTotalOrder._≈_ ord

_≤A_ : Rel A ℓ₂
_≤A_ = DecTotalOrder._≤_ ord

_=A?_ : Decidable _=A_
_=A?_ =
  IsDecTotalOrder._≟_ (DecTotalOrder.isDecTotalOrder ord)

_≤A?_ : Decidable _≤A_
_≤A?_ =
  IsDecTotalOrder._≤?_ (DecTotalOrder.isDecTotalOrder ord)
```

An Actual Sorting Algorithm

- This part looks like a real programming language.

```
-- Part 1: the sorting algorithm proper ------------

split : List A → List A × List A
split [] = [] , []
split (a :: []) = (a :: []) , []
split (a0 :: a1 :: t) with split t
... | tl , tr = (a0 :: tl) , (a1 :: tr)

merge : List A → List A → List A
merge [] right = right
merge left [] = left
merge (hl :: tl) (hr :: tr) with hl ≤A? hr
... | yes _ = hl :: (merge tl (hr :: tr))
... | no _ = hr :: (merge (hl :: tl) tr)
```

Our First Fight with the Termination Checker

- Agda can't see that the recursive calls to mergeSort are on smaller lists than the arguments. We'll need to do some proofs to get around this obstacle.

```
{- Traditional definition of mergeSort
-- Does not pass termination checker
mergeSort : List A → List A
mergeSort [] = []
mergeSort (a :: []) = a :: []
mergeSort (a0 :: a1 :: t) with split t
... | tl , tr =
  merge (mergeSort (a0 :: tl)) (mergeSort (a1 :: tr))
-}
```

The First of Two Annoyingly Similar Lemmas

- The left part of a split is no bigger than the input.

```
-- Lemma: length limit on left of split
splitLenLeft :
  {lim : ℕ} → (a0 a1 : A) →
  (t : List A) →
  (p : suc (length (a0 ∷ a1 ∷ t)) ≤ suc lim) →
  (suc (length (a0 ∷ (proj₁ (split t)))) ≤ lim)
splitLenLeft a0 a1 [] (s≤s p) = p
splitLenLeft a0 a1 (_ ∷ []) (s≤s p) = p
splitLenLeft
    a0 a1 (a2 ∷ a3 ∷ t) (s≤s (s≤s (s≤s p))) =
  ≤-step (s≤s (splitLenLeft a2 a3 t (s≤s p)))
```

The Second of Two Annoyingly Similar Lemmas

- And, as expected, the right part of a split is also no bigger than the input.

```
-- Lemma: length limit on right of split
splitLenRight :
  {lim : ℕ} → (a0 a1 : A) →
  (t : List A) →
  (p : suc (length (a0 ∷ a1 ∷ t)) ≤ suc lim) →
  (suc (length (a1 ∷ (proj₂ (split t)))) ≤ lim)
splitLenRight a0 a1 [] (s≤s p) = p
splitLenRight a0 a1 (_ ∷ []) (s≤s (s≤s p)) =
  ≤-step p
splitLenRight
    a0 a1 (a2 ∷ a3 ∷ t) (s≤s (s≤s (s≤s p))) =
  ≤-step (s≤s (splitLenRight a2 a3 t (s≤s p)))
```

A mergeSort with a Limit: Easy Cases

- lim decreases on each iteration. lim = 0 is absurd.

```
-- Decreasing limit placates the termination checker
mergeSortLimited :
  (lim : ℕ) → (l : List A) →
  (p : suc (length l) ≤ lim) →
  List A
mergeSortLimited 0 _ ()
mergeSortLimited (suc n) [] _ = []
mergeSortLimited (suc n) (a ∷ []) _ = a ∷ []
```

Agda Is Not Too Smart about Products and Projections

- I would have expected this to work. Am I asking too much?

```
{- This doesn't work:
-- Agda doesn't seem to know that
--   split t = proj₁ (split t) , proj₂ (split t)
mergeSortLimited (suc n) (a0 :: a1 :: t) p with split t
... | tl , tr =
  merge
   (mergeSortLimited n (a0 :: tl)
     (splitLenLeft a0 a1 t p))
   (mergeSortLimited n (a1 :: tr)
     (splitLenRight a0 a1 t p)) -}
```

Explicit Projections instead of Pattern Matching Actually Works

- This is awkward and looks as if it calls split too many times, but Agda doesn't reject it.

```
mergeSortLimited (suc n) (a0 :: a1 :: t) p =
  merge
   (mergeSortLimited n (a0 :: (proj₁ (split t)))
     (splitLenLeft a0 a1 t p))
   (mergeSortLimited n (a1 :: (proj₂ (split t)))
     (splitLenRight a0 a1 t p))
```

One More Teeny Lemma

- Sometimes repeated code is just easier.

```
-- Reflexivity of ≤ is predefined, but it's buried
-- so deep in Nat.decTotalOrder that it's more
-- concise just to prove it again from scratch:
n≤n : (n : ℕ) → n ≤ n
n≤n 0 = z≤n
n≤n (suc n) = s≤s (n≤n n)
```

And Finally, mergeSort Itself

- This looks innocent enough.

```
mergeSort : List A → List A
mergeSort l =
  mergeSortLimited
    (suc (length l)) l (n≤n (suc (length l)))
```

With a Whimper, Not a Bang

- The file `MergeSort.agda` has a little more stuff in it, but really, getting `mergeSort` past the Agda termination checker was my only victory.
- (Agda supports another approach to termination checking called sized types, but it seems like cheating.)

Part 6:
Standard Libraries

For Mathematicians or Engineers?

- The standard libraries differ in flavor from one language to another.

Coq	Agda	Idris
• Large standard library • Oriented towards mathematical proof • Organization rather ad hoc • Includes tactics as well as terms	• Emphasis on mathematical structures and relations • Systematically organized • Sometimes difficult to navigate deeply nested records	• More like Haskell library: emphasis on working data structures • Relatively little emphasis on abstract mathematics • Smaller than Coq's or Agda's

Part 7:
Interfacing with the Real World

Foreign Languages

- None of these languages is mature or complete enough to do much without resorting to some other language.

Coq	Agda	Idris
• Can extract pure Gallina code to OCaml (most common). • Haskell and Scheme are also said to be available.	• Foreign Function Interface supports calling Haskell functions. • Dealing with Haskell typeclasses is awkward.	• Original Foreign Function Interface talks to C. • Javascript FFI said to be available now (fully supported?).

I/O? Fuhgeddaboudit

- Did my program actually *do* anything? I/O is a nice way to find out.

Coq	Agda	Idris
• No built-in I/O; extract to OCaml and use OCaml's.	• No built-in I/O (?); use Haskell's through the Foreign Function Interface.	• I/O is built in as part of the Effects library.

Part 8:
The Theoretical Fine Print

Not All Type Systems Are Judgmentally Equal

- Coq, Agda, and Idris implement closely related type theories.
- All the theories are believed to be sound, in the sense that you can trust any proofs you derive in these languages (provided you turn on the totality check in Idris).
- Minor differences in the theories can affect how easy it is to state and prove some theorems.

Welcome to the Multiverse

- The type of types cannot be an element of itself without creating a paradox (Girard's paradox) that would make proofs unreliable.
- Consequently, all 3 languages we're looking at support stratification in type universes.
- In Agda, that means:

  $\mathsf{Set}_0 : \mathsf{Set}_1 : \mathsf{Set}_2 \ldots$

- Set is just a synonym for Set_0.

Layers upon Layers

- Universe polymorphism in Coq and Idris is implicit. You don't need to mention levels explicitly; the language figures out whether the (mostly invisible) indices to Type can be assigned consistently in a given program, and complains if they can't.
- Universe polymorphism in Agda is explicit: at any level above zero, you must provide an index to Set.
- If you don't give the right index, you get the dreaded Set ℓ_1 != Set ℓ_2 error message.

Propped Up

- In addition to the predicative Set/Type hierarchy of universes, Coq provides an impredicative Prop.
- This lets you formulate propositions about propositions without worrying about levels.
- You can't pattern match on a value of Prop (this avoids paradoxes).
- Agda and Idris get by without Prop.
- How useful is Prop when you're talking about programs, as opposed to abstract mathematics?

Part 9:
Status

Patience Is a Virtue

- If you want to develop real-world software *today* with any of these 3 languages, prepare for a bumpy ride.

Coq	Agda	Idris
• Very mature, but not really oriented towards writing runnable programs (as opposed to doing proofs).	• Ostensibly for writing programs, but mostly used by mathematicians. • Standard library not oriented towards usable workaday data structures.	• Most oriented towards real-world programming, but still quite immature.