An Introduction to Dependent Types and Agda

Andreas Abel

3 July 2009

1 Types

Types in programming languages serve several purposes:

1. Find runtime-errors at compile-time.
2. Enable compiler optimizations.
3. Allow the programmer to express some of his intention about the code.
4. Provide a machine-checked documenation.

Strongly typed languages include JAVA and Haskell. Dependent types allow the programmer to add even more redundant information about his code which leads to detection logical errors or full program verification.

1.1 What is a dependent type?

Dependent types mean that types can refer to values, not just to other types. From the mathematics, uses of dependent types are familiar. For instance, in linear algebra we define the type of vectors $\mathbb{R}^n$ of length n by

$$\mathbb{R}^n = \underbrace{\mathbb{R} \times \cdots \times \mathbb{R}}_{n \text{ times}}$$

and the inner product of two vectors of length n receives type $\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. This type is dependent on the value n. Its purpose is to make clear that the inner product is not defined on two arbitrary vectors of reals, but only of those of the same length. Most strongly typed programming languages do not support dependent types. Haskell has a rich type system that can simulate dependent types to some extent [McB02].

Dependently typed languages include Agda, Coq, Omega, ATS (replacing DML). Cayenne is no longer supported.

2 Core Features of Agda

In the following, we give a short introduction into dependent types using the language Agda. Agda is similar to Haskell in many aspects, in particular, *indentation matters*!

```
module DepTypes where
```

2.1 Dependent Function Type

As opposed to ordinary function types `A → B`, dependent function types `(x : A) → B` allow to give a name, `x`, to the domain `A` of the function space. The name `x` can be subsequently used in the codomain `B` to refer to the particular value `x` of type `A` that has been passed to the function. Using a dependent function type, we can specify the `inner` product function as follows:

```
inner : (n : Nat) → Vect Nat n → Vect Nat n → Nat
```

Herein `Vect Nat n` denotes the type of vectors of natural numbers of length `n`, which we will define later. An application `inner 5 v w` is only well-typed if `v` and `w` both have length `5`.

2.2 Inductive Types

As in Haskell, we can declare new data types by giving their constructors. Only the new **data** syntax of Haskell is supported by Agda:

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

This means that we introduce a new type, a `Set` in Agda terminology, with a nullary constructor `zero` and a unary, recursive constructor `suc`. Thus, natural numbers are possibly empty sequences of `suc` terminated by a `zero`, which is a unary presentation of natural numbers, as opposed to a binary representation.

Polymorphic data types, *parametric* data types in proper terminology, can be defined by providing a sequence of variable declarations just after the name of the data type. For example, polymorphic lists have one parameter `A : Set`, which is the list element type. All the parameter names are in scope in the constructor declarations.

```
data [_] (A : Set) : Set where
  []    : [A]
  _::_  : A → [A] → [A]
```

Agda supports pre-, post-, in-, and mixfix identifiers. Here, we have declared the type `[A]` of lists over `A` as mixfix identifier, the constructor `[]` for empty lists is an ordinary identifier but made up of special symbols, the constructor `_::_` is infix, to be used in the form `x :: xs`.

2.3 Inductive Families

Vectors are lists over a certain element type A of a certain length n. While the *parameter* A remains fixed for the whole list, the *index* n varies for each sublist. Indexed inductive types are called inductive families and declared like

```
data Vect (A : Set) : Nat → Set where
  vnil  : Vect A zero
  vcons : (n : Nat) → A → Vect A n → Vect A (suc n)
```

Note that Vect itself is of type Set → Nat → Set and Vect A is of type Nat → Set.

2.4 Recursive Definitions and Pattern Matching

In order to define the inner product of two vectors of natural numbers, we define addition and multiplication for natural numbers first.

```
infix 2 _+_
infix 3 _*_
_+_ : Nat → Nat → Nat
n + zero   = n
n + suc m  = suc (n + m)
_*_ : Nat → Nat → Nat
n * zero   = zero
n * suc m  = n * m + n
```

Both are defined by recursion and case distinction over the second argument.

Our first attempt to define the inner product is:

```
inner : (n : Nat) → Vect Nat n → Vect Nat n → Nat
inner zero    vnil          vnil          = zero
inner (suc n) (vcons n x xs) (vcons n y ys) = x * y + inner n xs ys
  -- FAILS
```

However this fails. The second clause for inner violates the *linearity condition*. The variable n is mentioned thrice in the patterns on the left hand side. Each variable can however only be bound once. What we meant to express is that whatever the values of the three occurrences of n are, because of the type of inner we know they are equal. This can be expressed properly using Agda's *dot patterns*.

```
inner : (n : Nat) → Vect Nat n → Vect Nat n → Nat
inner zero    vnil          vnil          = zero
inner (suc .n) (vcons .n x xs) (vcons n y ys) = x * y + inner n xs ys
```

There is now only one binding occurrence of n, the other two occurrences have been dotted. A dot can be followed by *any* expression, it does not have to be a variable as in our case. What .n it means is: *whatever stands here, do not match against it, for I know it is equal to n.*

2.5 Omitted and Hidden Arguments

A special expression is the hole _ which stands for any expression we do not care about. Agda's unification procedure tries to find the correct expressions which fit into the holes for us. For instance we can leave the administration of the vector length to Agda, since it is inferable from the constructors vnil and vcons.

```
inner : (n : Nat) → Vect Nat n → Vect Nat n → Nat
inner ._ vnil             vnil             = zero
inner ._ (vcons ._ x xs) (vcons _ y ys) = x * y + inner _ xs ys
```

We can even hide things we do not care about completely. The hidden dependent function space {x : A} → B contains functions whose argument is declared as hidden, i.e., it is not written by default, but one can supply it enclosed in braces. It is convenient to hide the length annotations in vectors so that they look like lists.

```
data Vect (A : Set) : Nat → Set where
  vnil  : Vect A zero
  vcons : {n : Nat} → A → Vect A n → Vect A (suc n)

inner : {n : Nat} → Vect Nat n → Vect Nat n → Nat
inner vnil          vnil          = zero
inner (vcons x xs) (vcons y ys) = x * y + inner xs ys
```

Internally, the last definition of inner is read as:

```
inner : {n : Nat} → Vect Nat n → Vect Nat n → Nat
inner {_} vnil                vnil                = zero
inner {_} (vcons {._} x xs) (vcons {._} y ys) = x * y + inner {_} xs ys
```

Can we really not fool the type checker? Let us test it.

```
one = suc zero
two = suc one

v1  = vcons one vnil
v2  = vcons zero v1
```

Now the following program is rejected by the type checker:

```
foo = inner v1 v2   -- FAILS
```

3 Some Library Functions for Vectors

In the following, we gain some more experiences with Agda by playing around with vectors. Let us try concatenation of vectors first.

```
append : {A : Set} {n m : Nat} →
         Vect A n → Vect A m → Vect A (n + m)   -- FAILS
append vnil         ys = ys
append (vcons x xs) ys = vcons x (append xs ys)
```

This code is rejected by Agda with the error message .m != zero + .m of type Nat pointing at the right hand side of the first clause. What is going on here? Since vnil : Vect A n the type checker infers n = zero, thus it expects the right hand side ys to be of type Vect A (zero + m). It has type Vect A m, and zero plus something is something, so where is the problem? The problem is that we know knowledge about addition that Agda does not have. By definition, Agda knows that m + zero = m, but it does not know that addition is commutative.[1] Flipping the sum solves our problem:

```
append : {A : Set} {n m : Nat} →
         Vect A n → Vect A m → Vect A (m + n)
append vnil         ys = ys
append (vcons x xs) ys = vcons x (append xs ys)
```

We got away this time, but in other cases we might have to teach Agda that addition is commutative!

A nice application of vectors is that looking up the element at a certain index can be made safe statically. I.e., we can define a lookup function _!!_ that only accept indices that are below the length of the vector. The trick is done using finite sets:

```
data Fin : Nat → Set where
  fzero : {n : Nat} → Fin (suc n)
  fsuc  : {n : Nat} → Fin n → Fin (suc n)
```

The finite set Fin n contains exactly n elements. In particular, Fin 0 is empty, and Fin (suc n) contains the elements fzero, fsuc fzero, ..., fsuc^{n-1} fzero.

Using Fin, we construct the following total lookup function

```
_!!_ : {A : Set} {n : Nat} → Vect A n → Fin n → A
vnil       !! ()
vcons x xs !! fzero   = x
vcons x xs !! fsuc m = xs !! m
```

The first clause uses the absurd pattern (). If the vector is vnil, then n = 0 and Fin n is empty, so there is no match for the index. Of course, no right hand side has to be given in this case.

Again, we cannot fool the type checker:

```
foo = vnil !! fzero   -- FAILS
```

[1] Agda is dumb as a chicken!

4 Sorted Lists and Logic in Agda

Let us introduce booleans and comparison of naturals:

```
data Bool : Set where
  true  : Bool
  false : Bool

if_then_else_ : {A : Set} → Bool → A → A → A
if true  then b1 else b2 = b1
if false then b1 else b2 = b2

_⩽_ : Nat → Nat → Bool
zero    ⩽ n       = true
(suc m) ⩽ zero    = false
(suc m) ⩽ (suc n) = m ⩽ n
```

An idea to implement a type of descendingly sorted lists of natural numbers is to index the list type by a lower bound for the next element which can be prepended.

```
data SList : Nat → Set where
  snil  : SList zero
  scons : {n : Nat} (m : Nat) → (n ⩽ m) → SList n → SList m
  -- FAILS
```

The empty list is indexed by zero so any natural number can be prepended without violating sorting. The list scons {n} m _ l is only sorted if n ⩽ m. The placeholder needs to be filled with some evidence for this fact, or in mathematical terms we need to provide a proof that can be checked by Agda!

However, the way we wrote it above does not make sense. The term n ⩽ m is a Boolean, i.e., either true or false, thus, it is a value and not a set and we cannot form a function space from a value to a set.

4.1 The Curry Howard Correspondence

We need to find a representation of truth values as sets, such that we can integrate evidence for valid conditions, such as n ⩽ m into data structures. The solution is based on an observation by Haskell Curry (1934 and 1958) and William A. Howard (1969) that a proposition can be viewed as the set of its proofs, and proofs correspond to programs, at least in intuitionistic logic, which is a logic without the principle of the excluded middle. The correspondence is called Curry-Howard-Correspondence or -Isomorphism, sometimes also Curry-Howard-de Bruijn Correspondence.

```
Proposition = Set
```

An empty set corresponds to a unprovable, i.e., false proposition, an inhabited (i.e., non-empty) set to a true proposition.

```
data Absurd : Proposition where
data Truth  : Proposition where
  tt : Truth
```

Consequently, absurdity can be modelled as a data type with no constructors, and truth as a data type with a constructor that has no arguments. Pure truth needs no further evidence (but also conveys no information).

```
data _∧_ (A B : Proposition) : Proposition where
  _,_ : A → B → A ∧ B
```

To prove the conjunction $A \wedge B$, we need a proof $a : A$ of A and a proof $b : B$ of B, which we put together to form the pair a, b.

```
fst : {A B : Proposition} → A ∧ B → A
fst (a , b) = a
snd : {A B : Proposition} → A ∧ B → B
snd (a , b) = b
```

If we have a proof of $A \wedge B$, we can obtain proofs of A and B by projection. Viewed as set, the conjunction is just the Cartesian product.

```
_×_ = _∧_
```

A disjunction $A \vee B$ is proven by either providing a proof of A or a proof of B. In Agda this is modelled by a data type with two constructors inl and inr, the first to turn a proof of A into a proof of $A \vee B$ and the second to turn a proof of B into a proof of $A \vee B$.

```
data _∨_ (A B : Proposition) : Proposition where
  inl : A → A ∨ B
  inr : B → A ∨ B
```

A proof of a disjunction $A \vee B$ can be used by performing case analysis, i.e., distinguishing whether a proof of A was given or a proof of B.

```
case : {A B C : Proposition} → A ∨ B → (A → C) → (B → C) → C
case (inl a) f g = f a
case (inr b) f g = g b
```

So if we have a method $f : A \to C$ to turn a proof of A into a proof of C, and we have a method $g : B \to C$ to obtain a proof of C from a proof of B, then given a proof of $A \vee B$ we obtain a proof of C by case analysis.

Implication is just the functions space. Thus, a proof $f : A \to B$ of the implication "A implies B", is a function which computes a proof of B from a proof of A. For example, we can prove the following two (trivial) propositions:

```
lemma : {A : Proposition} → A → A ∧ Truth
lemma a = (a, tt)
comm∧ : {A B : Proposition} → A ∧ B → B ∧ A
comm∧ (a, b) = (b, a)
```

The first lemma states that A implies A ∧ Truth, its proof is a program taking a : A and producing the pair a, tt. The second shows that conjunction is commutative, the proof just swaps the two components of the pair.

4.2 Booleans and Propositions

We can translate booleans into propositions by mapping true to Truth and false to Absurd.

```
True : Bool → Proposition
True true  = Truth
True false = Absurd
```

The opposite translation is not possible, since there are more propositions than just the trivial ones (truth and absurdity).

4.3 Proof by Induction

Now we can show that comparison ⩽ is reflexive.

```
refl⩽ : (n : Nat) → True (n ⩽ n)
refl⩽ zero    = tt
refl⩽ (suc n) = refl⩽ n
```

The proof of True (n ⩽ n) proceeds by induction on n : Nat. In case zero, the term zero ⩽ zero simplifies to true by definition of ⩽. Thus, it remains to show True true which in turn simplifies to Truth and this has the trivial proof tt. In case suc n we have to show True (suc n ⩽ suc n) which simplifies to True (n ⩽ n) by definition. We conclude by the induction hypothesis, which is obtained via the induction hypothesis refl⩽ n : True (n ⩽ n).

By the Curry-Howard correspondence, a proof by induction is a recursive function over the natural numbers, with one important condition: It needs to be terminating on all inputs. Termination of refl⩽ is checked by Agda using the *structural ordering* on the natural numbers. The term n is structurally smaller than suc n since n is an argument to the constructor suc. The structural ordering is wellfounded for any data type, thus, termination checking using the structural ordering is sound.

Using non-terminating programs, we can prove false propositions:

```
{-# NO_TERMINATION_CHECK #-}

bla   :  (n : Nat) → True (one ⩽ zero)
bla n = bla (suc n)
```

```
{-# NO_TERMINATION_CHECK #-}

foo :  Absurd
foo = foo
```

The Agda termination checker marks all programs that fail the termination check in red color, indicating that it cannot guarantee logical consistency in the presence of non-termination. However, it does not report an error, since the program might terminate even if Agda cannot guarantee it. (Remember the halting problem: termination is undecidable in general.)

Another source of logical inconsistency are incomplete pattern matches, for instance:

```
A : Bool → Proposition
A true  = Truth
A false = True (one ⩽ zero)

prf : (b : Bool) → A b
prf true = tt
  -- FAILS

bar : True (one ⩽ zero)
bar = prf false
```

The proof prf is incomplete the case b = false has been omitted, which happens to be unprovable. Now bar is a proof of an inconsistency, which if executed, would result in a runtime error. However, Agda already rejects prf since the pattern matching is incomplete. By declaring the options

```
{-# OPTIONS –no-termination-check –no-coverage-check #-}
```

termination and case coverage check can be turned off, which might be convenient during drafting a program/verification project.

4.4 Statically Sorted Lists

We can now come back to the data type of descending lists.

```
data SList : Nat → Set where
  snil  : SList zero
  scons : {n : Nat} (m : Nat) → True (n ⩽ m) → SList n → SList m

slist1 = scons one tt (scons zero tt snil)
```

Whenever we prepend a new element to the list with scons, we need to provide a proof that the new element m is greater or equal to the head element n of the List (or zero if the list is empty). For concrete numbers, these proofs are always trivial, since ⩽ is decidable.

This completes the crash course in Agda. More information can be found in Ulf Norell's tutorial or the LERNET summer school notes by Ana Bove and Peter Dybjer.

References

[McB02] Conor McBride. Faking it: Simulating dependent types in haskell. *J. Func. Program.*, 12(4&5):375–392, 2002.