Report

Thomas Bååth Sjöblom

March 18, 2013

1 Introduction

1.1 What?

$\mathbb{N}$ $\mathbb{N}$

1.2 Why?

2 Agda

Agda was invented at Chalmers! By Ulf Norell.

2.1 As a programming language

Agda is a dependently typed functional language. Functional means that programs are essentially a sequence of definitions of different functions (mathematical functions – meaning no side effects). Dependently typed means that data types can depend on *values* of other typed. The syntax is very similar to that of Haskell, with the biggest difference being that Agda uses : for typing: `f : a → b`, while Haskell instead uses ::, as in `f :: a → b` The reason for this is that in Haskell, lists are very important, and use : for the cons operation. In Agda, on the other hand, there are no built in types, so : is not used up already, and Agda can use it for type information, like it is used in Type Theory. The second syntactical difference is that Agda allows all unicode [TODO: ?] characters in programs.

That Agda doens't have built in types is another very big difference. One advantage is that it guarantees that all types are inductively defined, instead of as in Haskell, where the built in types behave differently from the user defined ones. But the fact that allows unicode in programs let's one define types similar to those of Haskell. For example, we could define Lists as

```
infixr 8 _::_
data [_] (a : Set) : Set where
  [] : [a]
  _::_ : (x : a) → (xs : [a]) → [a]
```

The notation here is very similar to the Haskell notation for lists, with the diffrerence that we need to use spaces between the brackets and a (the reason for this is that [a], without spaces is a valid type identifier in Agda.

We also define a type of natural numbers, so we have some type to make Lists of. Here we take advantage of Agdas ability to use any unicode symbols to give the type a short and familiar name:

```
data ℕ : Set where
  zero : ℕ
  suc : (n : ℕ) → ℕ
```

If we use the commands following commands

```
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC suc #-}
```

we can write the natural numbers with digits, and define a list

```
exampleList : [ℕ]
exampleList = 5 :: 2 :: 12 :: 0 :: 23 :: []
```

Agda is a dependently typed language, which means that types can depend on the *values* of other types. To some degree, this can be simulated in Haskell, using extensions like Generalized Algebraic Datatypes [TODO: What can't be done, is it relevant].

2.2 Agda as a proof assistant

The main use of Agda in this thesis is as a proof assistant. This use is based on the Curry Howard correspondence, which considers types as propositions, and their inhabitants as proofs of them.

2.3 The Curry Howard Correspondence

To define a conjunction between two Propositions P and Q, one uses the pair $P \times Q$ defined below

```
data _×_ (P Q : Set) : Set where
  _,_ : (p : P) → (q : Q) → P × Q
```

Because, to construct an element of $P \times Q$, one needs an element of both P and Q.

For disjunction, one uses the disjoint sum $P + Q$:

```
data _+_ (P Q : Set) : Set where
  inl : (p : P) → P + Q
  inr : (q : Q) → P + Q
```

For implication, one simply uses functions, $\mathsf{P} \to \mathsf{Q}$ [TODO: one???]

```
impl : {P Q : Set} → P → Q
```

because implication in constructive logic is a method for converting a proof of P to a proof of Q, and this is exactly what a function is. On the other hand, one might want a data type for implication, along with constructors for "canonical proofs" DUMMY.

```
data _⇒_ (P Q : Set) : Set where
  impl′ : (f : P → Q) → P ⇒ Q
```

However, this has the disadvantage that every time we want to access the function, we have to unwrap it, which clutters the code. In general, it's a good idea to use unwrapped functions when possible [TODO: other reason]. For example, we use this approach when defining the matrixes in 2.3.

For universal quantification, we again use functions, but this time functions from the variables quantified over, $(\mathsf{x} : \mathsf{X}) \to \mathsf{P}\ \mathsf{x}$:

```
all : {X : Set} {P : X → Set} → (x : X) → P x
```

Indeed, Agda allows the use of the syntax $\forall\ \mathsf{x}$ to mean $(\mathsf{x} : \_)$ in type definitions, so that $\forall\ \mathsf{x} \to \mathsf{P}\ \mathsf{x}$ means exactly what we expect it to mean (usnig the $\forall\ \mathsf{x}$ in definitions is nice even when not considering the types as propositions, because it lets us use Agda's type inference to shortedn the defniitions).

Finnally, existential quantification, $\exists x.P(x)$, which in constructive logic is interpreted to be true if there is a pair x_0 along with a proof of $P(x_0)$, so we can model it by a dependent product (similar to the cartesian product defined above but now we consider one of the sets a domain for the variables, and the other as a proposition). We use the same name for the constructor as above.

```
data ∃ (X : Set) (P : X → Set) : Set where
  _,_ : (x : X) → P x → ∃ X P
```

As a small example of using Agda, we will define a maximum function max for lists of natural numbers and prove that it satisfies a sensible specification. The specification we will use is that, max xs is greater than or equal to each element of xs, and equal to some element. As an example, we will define a maximum function max for lists of natural numbers and prove that it satisfies a sensible specification. The specification we will use is that, max xs is greater than or equal to each element of xs, and equal to some element. First, we define the maxℕ function on ℕ.

```
maxℕ : ℕ → ℕ → ℕ
maxℕ zero n = n
maxℕ n zero = n
maxℕ (suc m) (suc n) = suc (maxℕ m n)
```

We decide to only define the max function on nonempty lists (in the case of natural numbers, it might be sensible to define max [ ] = 0, but when it comes to other types, and orders, there is no least element). Second, we need to define . This is done with the following data type:

```
infix 3 _⩽_
data _⩽_ : ℕ → ℕ → Set where
  z⩽n : {n : ℕ} → zero ⩽ n
  s⩽s : {m n : ℕ} → (m⩽n : m ⩽ n) → suc m ⩽ suc n
```

Here we introduce another feature of Agda, that functions can take implicit arguments, the constructor z⩽n takes an argument n, but Agda can figure out what it is from the resulting type (which includes n), and hence, we don't need to include it.

Viewed through the Curry Howard Correspondence, the data type m ⩽ n represents the proposition that m is less than or equal to n, and the two possible proofs of this are, either m is zero, which is less than any natural number, or m = suc m′ and n = suc n′ and we have a proof of m′ ⩽ n′. Using the above definition, we can also define a less than function,

```
_<_ : ℕ → ℕ → Set
m < n = suc m ⩽ n
```

We note that we didn't need to create a new type using the **data** command to create this,

Now we define the length function for lists,

```
length : ∀ {a} → [a] → ℕ
length [ ] = 0
length (x :: xs) = suc (length xs)
```

Here, again, we introduce a new concept, preceeding a variable with ∀ means that Agda infers its type (in this case Set).

Now, we can define the max function:

```
max : (xs : [ℕ]) → (0 < length xs) → ℕ
max [ ] ()
max (x :: [ ]) _ = x
max (x :: (x' :: xs)) _ = maxℕ x (max (x' :: xs) (s⩽s z⩽n))
```

On the first line, we use the absurd pattern () to show that there is no proof that 1 ⩽ 0. On the second two lines, we don't care about what the input proof is (it is s⩽s z⩽n in both cases, so we use _ to signify that it's not important). [TODO: NAmes of _-pattern]

We also need an indexing function, and again, we can only define it for sensible inputs. The simplest definition would probably be: However, this leads to a bit of trouble later on, when we want to specify things about it, in particular when we want to say that the maximum is in the list. We want to say that there

is an index n such that the n:th element of the list is equal to the maximum. But to say this, we'd need to prove that the n was less than the length of the list, and the simple way to do this would be to attempt to use the proposition $\mathsf{P} = (\mathsf{proof} : \mathsf{n} \leqslant \mathsf{length\ xs}) \to \mathsf{index\ xs\ n\ proof} \equiv \mathsf{max\ xs\ lengthproof}$, but this is horribly wrong, because it states something completely different to what we want. It states that if there is a proof that $\mathsf{n} \leqslant \mathsf{length\ xs}$, then we need to have that all $\mathsf{n} > \mathsf{length\ xs}$ satisfy P, and this is clearly not what we want. The simplest way to fix this is to state that we want an integer that is n less than $\mathsf{length\ xs}$ and that the nth element of xs is equal to the max. However, there is a problem here too. To be able to index into the nth position, we need the proof that $\mathsf{n} \leqslant \mathsf{length\ xs}$, so we would can't use a pair (becuase the second element would have to depend on the first [todo: is there a way around this?]. Instead, we choose to define datatype $\mathsf{Fin\ n}$ containing the numbers less than n, and change the index function to use it instead of $\mathbb{N}$:

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

That is, $\mathsf{f0}$ (representing $\mathsf{0}$, but given a different name for clarity – it is not equal to the natural number $\mathsf{0}$, they don't even have the same type) is less than any number greater than or equal to $\mathsf{1}$, and for any number i, less than some number n, $\mathsf{fsuc\ i}$ is less than $\mathsf{n + 1}$. Note that we have put the index n on the right side of the colon n the definintion of Fin, this is so that [todo: is there a reason??? somethin with it being nidexed (doesn't work if we move it). Alternatively, we could define $\mathsf{Fin\ n}$ as a dependent pair of a natural number i and a proof that it is less than n. For future use, we define a dependent pair type first (we could of course have used it to define the regular pair for the Curry Howard Correspondence):

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (x : A) → B x → Σ A B
```

Here, on the other hand, we need to put the arguments to Σ on the left hand of the colon, because otherwise the type would be too big [todo : Huh?] And then use it to define $\mathsf{Fin'}$.

```
Fin' : (n : ℕ) → Set
Fin' n = Σ ℕ (λ i → i < n)
```

This second representation has the advantage that the natural number is close by (i is an actual natural number, that we can use right away, for the other Fin type, we would have to write and use a translation-function that replaces each fsuc by suc and fzero by zero) .

However, this would require us to always extract the proof when we need to use it, instead of having it "built into" the type. These two different ways of defining things are something we will use later when we define upper triangular

matrixes as their own data-type. For a concrete representation, we are going to use the first kind of representation, where we have built in the "proof" that the matrix is triangular – which lets us not worry about modifying the proof appropriately, or reprove that the product of two upper triangular matrixes is again upper triangular. While when representing matrixes abstractly (as functions from their indices), we will need to use the proofs and modify them, to strengthen some results from the concrete case.

We now redefine the indexing function, with different syntax, more familiar to Haskell users (and see already that not needing a separate proof argument makes things a lot clearer)

```
infix 10 _!!_
_!!_ : ∀ {a} → (xs : [a]) → (n : Fin (length xs)) → a
[] !! ()
(x :: xs) !! fzero = x
(x :: xs) !! fsuc i = xs !! i
```

The final step is defining equality, i.e., the proposition that two values x and y are equal. The basic equality is a data type whose only constructor refl is a proof that x is equal to itself.

```
infix 3 _≡_
data _≡_ {a : Set} : a → a → Set where
  refl : {x : a} → x ≡ x
```

Herre, we have an implicit argument to the *type*, to allow it to work for an type. For our purposes, this very strong concept of equality is suitable. However, if one wants to allow different "representations" of an object, for example if one defines the rational numbers as pairs of integers, $\mathbb{Q} = \mathbb{Z} \times \mathbb{Z} \setminus \{0\}$, one wants a concept of equality that considers (p, q) and $(m * p, m * q)$ to be equal. This could be taken care of by using equality defined as for eaxmple [TODO: what about division by 0]

```
data _≡′_ : ℚ → ℚ → Set where
  p/q≡mp/mq : {p : ℤ} {q : ℤ\0} (m : ℤ\0) → (p, q) ≡′ (m * p, m *′ q)
```

Another example is if we define a datatype of sets, we want two sets to be equal as long as they have the same elements, regardless if they were added in different orders, or if one set had the same element added multiple times.

Now we can finally express our specification in Agda:

```
max-spec : (xs : [ℕ]) → (pf : 0 < length xs) →
  ((n : Fin (length xs)) → xs !! n ⩽ max xs pf) ×
  ∃ (Fin (length xs)) (λ n → xs !! n ≡ max xs pf)
```

To prove the correctness of the max function, we must then find an implementation of max-spec, that is, we produce an element of its type, corresponding to a proof of the proposition it represents.

We do this in two parts, [todo : note that one should compare with the inductive definiotns. ALSO BREAK THIS STUFF UP WITH USEFUL COMMENTS (SPLIT CODE BLOCK INTO MANY)]

```
n⩽n : {n : ℕ} → n ⩽ n
n⩽n {zero} = z⩽n
n⩽n {suc n} = s⩽s n⩽n

maxℕ-increasing₁ : {x y : ℕ} → x ⩽ maxℕ x y
maxℕ-increasing₁ {zero} = z⩽n
maxℕ-increasing₁ {suc n} {zero} = s⩽s n⩽n
maxℕ-increasing₁ {suc n} {suc n'} = s⩽s maxℕ-increasing₁

maxℕ-increasing₂ : {x y : ℕ} → y ⩽ maxℕ x y
maxℕ-increasing₂ {x} {zero} = z⩽n
maxℕ-increasing₂ {zero} {suc n} = s⩽s n⩽n
maxℕ-increasing₂ {suc n} {suc n'} = s⩽s (maxℕ-increasing₂ {n} {n'})

max-greatest-one-step : {x : ℕ} {xs : [ℕ]} → x ⩽ max (x :: xs) (s⩽s z⩽n)
max-greatest-one-step {x} {[]} = n⩽n
max-greatest-one-step {x} {x' :: xs} = maxℕ-increasing₁

max-greatest-one-step₂ : {x : ℕ} {xs : [ℕ]} → x ⩽ max (x :: xs) (s⩽s z⩽n)
max-greatest-one-step₂ {x} {[]} = n⩽n
max-greatest-one-step₂ {x} {x' :: xs} = maxℕ-increasing₁

⩽-trans : {i j k : ℕ} → i ⩽ j → j ⩽ k → i ⩽ k
⩽-trans z⩽n j⩽k = z⩽n
⩽-trans (s⩽s i⩽j) (s⩽s j⩽k) = s⩽s (⩽-trans i⩽j j⩽k)

max-greatest : {xs : [ℕ]} → {pf : 0 < length xs} →
  (n : Fin (length xs)) → xs !! n ⩽ max xs pf
max-greatest {[]} {()} _
max-greatest {x :: xs} {s⩽s z⩽n} fzero = max-greatest-one-step {x} {xs}
max-greatest {x :: []} (fsuc ())
max-greatest {x :: (x' :: xs)} {s⩽s z⩽n} (fsuc i) = ⩽-trans (max-greatest i) (maxℕ-increasing₂ {x})
  -- max är antingen först, eller inte först.
  -- om x ⩾ max xs pf så är x == max (x :: xs) pf
≡-cong : {a b : Set} {x y : a} → (f : a → b) → x ≡ y → f x ≡ f y
≡-cong f refl = refl

x≡maxℕx0 : {x : ℕ} → x ≡ maxℕ x 0
x≡maxℕx0 {zero} = refl
x≡maxℕx0 {suc n} = refl

l'' : ∀ {x y} → y ⩽ x → x ≡ maxℕ x y
l'' {x} {zero} z⩽n = x≡maxℕx0
l'' (s⩽s m⩽n) = ≡-cong suc (l'' m⩽n)

lemma : ∀ x xs pf → max xs pf ⩽ x → x ≡ max (x :: xs) (s⩽s z⩽n)
lemma x [] pf pf' = refl
lemma x (x' :: xs) (s⩽s z⩽n) pf' = l'' pf'
```

```
data Dec (x : ℕ) (y : ℕ) : Set where
  yes : (x⩽y : x ⩽ y) → Dec x y
  no : (y⩽x : y ⩽ x) → Dec x y
  -- yes pf
  -- no pf of opposite
_⩽?_ : (x : ℕ) → (y : ℕ) → Dec x y
zero ⩽? n = yes z⩽n
suc m ⩽? zero = no z⩽n
suc m ⩽? suc n with m ⩽? n
suc m ⩽? suc n | yes m⩽n = yes (s⩽s m⩽n)
suc m ⩽? suc n | no n⩽m = no (s⩽s n⩽m)
  --
–m ax : (xs : [ℕ]) → (0 < length xs) → ℕ
–m ax [] ()
–m ax (x :: []) _ = x
–m ax (x :: (x' :: xs)) _ = maxℕ x (max (x' :: xs) (s⩽s z⩽n))
data Bool : Set where
  True : Bool
  False : Bool
≡-trans : ∀ {a b c} → a ≡ b → b ≡ c → a ≡ c
≡-trans refl refl = refl
maxℕ- : (x y : ℕ) → x ⩽ y → y ≡ maxℕ x y
maxℕ- zero y pf = refl
maxℕ- (suc m) zero ()
maxℕ- (suc m) (suc n) (s⩽s m⩽n) = ≡-cong suc (maxℕ- m n m⩽n)
lemma'' : ∀ x x' xs → x ⩽ max (x' :: xs) (s⩽s z⩽n) → max (x' :: xs) (s⩽s z⩽n) ≡ maxℕ x (max (x' :: xs) (s⩽s
lemma'' zero x' xs pf = refl
lemma'' (suc n) x' xs pf = maxℕ- (suc n) (max (x' :: xs) (s⩽s z⩽n)) pf
  -- [todo: move to CH]
witness : {A : Set} {B : A → Set} → ∃ A B → A
witness (x, y) = x
–p roof : : {A : Set} {B : A → Set} → ∃ A B → B
increase : ∀ x x' xs → x ⩽ max (x' :: xs) (s⩽s z⩽n) → ∃ (Fin (suc (length xs)))
      (λ i → (x' :: xs) !! i ≡ max (x' :: xs) (s⩽s z⩽n)) → ∃ (Fin (suc (suc (length xs))))
      (λ i → (x :: x' :: xs) !! i ≡ maxℕ x (max (x' :: xs) (s⩽s z⩽n)))
increase x x' xs pf' (i, pf) = fsuc i, ≡-trans pf (lemma'' x x' xs pf')
min' : (xs : [ℕ]) → (pf : 0 < length xs) → ∃ (Fin (length xs)) (λ i → xs !! i ≡ max xs pf)
min' [] ()
min' (x :: []) pf = fzero, refl
min' (x :: x' :: xs) pf with x ⩽? max (x' :: xs) (s⩽s z⩽n)
min' (x :: (x' :: xs)) (s⩽s z⩽n) | yes x⩽x' = increase x x' xs x⩽x' {!prod!}
  where prod = min' (x' :: xs) (s⩽s z⩽n)
min' (x :: x' :: xs) pf | no x'⩽x = fzero, lemma x (x' :: xs) (s⩽s z⩽n) x'⩽x
```

```
max-in-list : {xs : [ℕ]} → {pf : 0 < length xs} →
  ∃ (Fin (length xs)) (λ n → xs !! n ≡ max xs pf)
max-in-list {[]} {()}
max-in-list {xs} {pf} = min' xs pf
max-spec [] ()
max-spec (x :: xs) (s≤s m≤n) = max-greatest, max-in-list
```

[todo : ⩽-trans repeatedly leads to introduction of equational syntax, trap is trying to expand variables too many times]

Another practical difference is that all programs have to terminate. This is guaranteed by requiering that some argument of the function gets smaller at each step. This means that recursive programs written in Agda should be structurally recursive in some way, or include some kind of proof term on which they recurse structurally. Thanks to the dependent types, it is possible to encode also properties of programs.

The first thing to do this is

, for example, as in the above example, we could express that the length of the list after the

3 Algebra

We are going to indroduce a bunch of algebraic things that will be useful either later or as point of reference. They will also be useful as an example of using agda as a proof assistant!

The first two sections are about algebraic structures taht are probably already known. Both for reference, and as examples. Then we go on to more general algebraic structures, more common in Computer Science, since they satisfy fewer axioms (more axioms mean more interesting structure – probably – but at the same time, it's harder to satisfy all the axioms.

3.1 Groups

The first algebraic structure we will discuss is that of a group.

Definition 3.1. A group is a

In Agda code, this is defined using a record:

```
_≡_ : Set → Set → Set
A ≡ B = A
record Test : Set₁ where
  infix 4 _≈_
  field
    Carrier : Set
    _≈_ : Set → Set → Set
```

To prove that something is a group, one would thus

3.2 Rings

3.2.1 Definition

3.2.2 Matrixes

3.3 Monoids

3.3.1 Definition

3.3.2 Cayley Table

3.4 Monoid-like structures

3.4.1 Definition

3.4.2 Cayley Table

3.5 Ring-like structures

3.5.1 Definition

3.5.2 Matrixes

4 Parsing

4.1 General stuff

4.1.1 Parsing as Trasitive Closure

5 Valiant's Algorithm

5.1 Specification of transitive closure

6 Discussino

6.1 Related work

6.2 Future work

Some future work: Expand on the algebraic structures in Agda, perhaps useful to learn abstract algebra (proving that zero in a ring annihilates is a fun(!) exercise!). Also expand on it so that it becomes closer to what is doable in algebra packages – create groups by generators and equations, for example.

Fit into Algebra of Programming (maybe).

References

DUMMY. *DUMMY*.