# Report

Thomas Bååth Sjöblom

March 3, 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

As a programming language, Agda is fairly similar to Haskell. The syntax is very similar to that of Haskell, with the biggest difference being that Agda uses : for typing: $f : a \to b$, while Haskell instead uses ::, as in $f :: a \to b$ 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 $P + Q$:

For implication, functions

For universal quantification,

For existential quantification,

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 $max\mathbb{N}$ function on $\mathbb{N}$.

$$
\begin{aligned}
&max\mathbb{N} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&max\mathbb{N}\ zero\ n = n \\
&max\mathbb{N}\ n\ zero = n \\
&max\mathbb{N}\ (suc\ m)\ (suc\ n) = suc\ (max\mathbb{N}\ m\ n)
\end{aligned}
$$

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:

$$
\begin{aligned}
&\textbf{data}\ \_\leqslant\_ : \mathbb{N} \to \mathbb{N} \to Set\ \textbf{where} \\
&\quad z\leqslant n : \{\, n : \mathbb{N} \,\} \to zero \leqslant n \\
&\quad s\leqslant s : \{\, m\ n : \mathbb{N} \,\} \to (m \leqslant n : m \leqslant n) \to suc\ m \leqslant suc\ n
\end{aligned}
$$

Here we introduce another feature of Agda, that functions can take implicit arguments, the constructor $z\leqslant n$ 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 $m \leqslant n$ 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 $m' \leqslant n'$.

Now we define the *length* function for lists,

$$
\begin{aligned}
&length : \forall\{\, a \,\} \to [\, a \,] \to \mathbb{N} \\
&length\ [\,] = 0 \\
&length\ (x :: xs) = suc\ (length\ xs)
\end{aligned}
$$

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

Now, we can define the *max* function:

$$
\begin{aligned}
&max : (xs : [\mathbb{N}]) \to 1 \leqslant length\ xs \to \mathbb{N} \\
&max\ [\,]\ () \\
&max\ (x :: [\,])\ \_ = x \\
&max\ (x :: (x' :: xs))\ \_ = max\mathbb{N}\ x\ (max\ (x' :: xs)\ (s\leqslant s\ z\leqslant n))
\end{aligned}
$$

On the first line, we use the ()-pattern to show that there is no proof that $1 \leqslant 0$. On the second two lines, we don't care about what the input proof is

(it is $s \leqslant s\ z \leqslant n$ in both cases, so we use _ to signify that it's not important). [TODO: NAmes of ()-pattern and _]

We also need an indexing function, and again, we only define it for sensible inputs:

$$
\begin{aligned}
&\mathit{index} : \forall\{\, a \,\} \to (\mathit{xs} : [\, a \,]) \to (n : \mathbb{N}) \to \mathit{suc}\ n \leqslant \mathit{length}\ \mathit{xs} \to a \\
&\mathit{index}\ [\,]\ n\ () \\
&\mathit{index}\ (x :: \mathit{xs})\ \mathit{zero}\ \_ = x \\
&\mathit{index}\ (x :: \mathit{xs})\ (\mathit{suc}\ n)\ (s \leqslant s\ m \leqslant n) = \mathit{index}\ \mathit{xs}\ n\ m \leqslant n
\end{aligned}
$$

Now, we can write our specification of the *max* function.

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.

$$
\begin{aligned}
&\textbf{data}\ \_ \equiv \_\,\{\, a : \mathit{Set} \,\} : a \to a \to \mathit{Set}\ \textbf{where} \\
&\quad \mathit{refl} : \{\, x : a \,\} \to x \equiv x
\end{aligned}
$$

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, $\mathbb{Q} = \mathbb{Z} \times \mathbb{Z}$, one wants a concept of equality that considers $(p, q)$ and $(m * p, m * q)$ to be equal. would want two numbers two sets where the elements were added in different order, or if one defines the rational numbers two rational numbers, $p/q$ and $(mp/mq)$, defined as pairs of natural numbers) to be equal, one would need an equality type with more constructors. For example

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).