Report

Thomas Bååth Sjöblom

April 1, 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 characters in programs.

Notes by JP: 1. Every binding can be given a name. 2. Explain _. 3. Use of spacing. 4. Totality.

?

That Agda doesn'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
```

make clear list/section about agda-haskell differences. see afp lectures

also make a section about what we're going to do: write an extended example of a proof, showing all code

mention that we only use Set here, in

```
[] : [a]
_::_ : (x : a) → (xs : [a]) → [a]
```

The notation here is very similar to the Haskell notation for lists, with the difference 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 Agda's 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 .

What can't be done, is it relevant – especially: can one point to some example later in the report that can't be done

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 ∧ Q defined below

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

Because, to construct an element of P ∧ Q, one needs an element of both P and Q.

For disjunction, one uses a disjoint sum:

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

For implication, one simply uses functions, $P \to Q$, 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 : (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 . For example, we use this approach when defining the matrixes in 2.3.

is there another reason

The last of the predicate logic concepts is negation. Constructively, the negation of a proposition means that the proposition implies falsity. To define this, we first define the empty type as a type with no constructors to represent falsity:

```
data ⊥ : Set where
```

We then define negation with

```
¬ : (P : Set) → Set
¬ P = P → ⊥
```

Constructively, the law of excluded middle—saying that for every proposition P, either P or $\neg P$ is true—is not valid. However, there are propositions for which it is valid (trivially, for all true propositions). These are said to be *decidable*, and are propositions such that there exists an algorithm producing either a proof of the proposition, or a proof of the negation. In Agda, if X is a collection of statements, we define this with a helper type $\mathsf{Dec}\ X$ that has two constructors, one taking a proof of an instance $x : X$ and one a proof $\neg x : \neg X$:

what is it really that is decidable, proposition or relation (think a bit)

```
data Dec (P : Set) : Set where
  yes : (p : P) → Dec P
  no : (¬p : ¬ P) → Dec P
```

Then, a set of propositions P is proven to be decidable if we have a function $P \to \mathsf{Dec}\ P$, that is, and algorithm that takes an arbitrary instance of P and decides whether it is true.

Example 2.1. One example of a decidable proposition is inequality between natural numbers, which we consider in Section 2.3, since, given two natural numbers, we can determine which is smaller by repeatedly subtracting 1 from both until one is zero.

Next (the part of the Curry–Howard correspondence), we look at universal and existential quantification from predicate logic.

expand above section (the Dec section) a bit

CUrry or Howard? (or is this something I imagined I heard someone say?)

For universal quantification, we again use functions, but this time the variable is bound to a name $\mathsf{x} : \mathsf{X}$, and appears again, and the proposition can depend on the value x: $\mathsf{P} : \mathsf{X} \to \mathsf{Set}$, that is universal quantification is a function $(\mathsf{x} : \mathsf{X}) \to \mathsf{P}\ \mathsf{x}$:

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

That is, for any element $\mathsf{x} : \mathsf{X}$, we have $\mathsf{P}\ \mathsf{x}$. 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 (using 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 shorten the definitions).

Finally, existential quantification, $\exists x.P(x)$, which in constructive logic is interpreted to be true if there is a pair (x_0, Px_0) of an element x_0 along with a proof of $P(x_0)$, so we can model it by a dependent pair (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.

lookup $\exists$ in standard library

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

Fix the below labels (move)

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 $\mathbb{N}$.

```
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 less than, $\_\leqslant\_$. 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 $m \leqslant n$ 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 $m = \mathsf{suc}\ m'$ and $n = \mathsf{suc}\ n'$ and we have a proof of $m' \leqslant n'$. 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 : Set} → [a] → ℕ
length [] = 0
length (x :: xs) = suc (length xs)
```

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 $1 \leqslant 0$. 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).

Explain why () can be used

We also need an indexing function, and again, we can only define it for sensible inputs. The simplest definition would probably be:

NAmes of _-pattern – if there is one

```
index : ∀ {a} → (xs : [a]) → (n : ℕ) → suc n ≤ length xs → a
index [] n ()
index (x :: xs) zero _ = x
index (x :: xs) (suc n) (s≤s m≤n) = index xs n m≤n
```

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 nth 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 `P = (proof : n ≤ length xs) → index xs n proof ≡ 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 n ≤ length xs, then we need to have that all n > 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 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 $n \leqslant$ length xs, so we can't use a pair (because the second element would have to depend on the first . Instead, we choose to define datatype Fin n containing the numbers less than n, and change the index function to use it instead of $\mathbb{N}$:

expand on this, and clean up: curry howard says some things, can move away from it, or state that there is a pair, but the existence must be on the left of the implication

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

That is, f0 (representing 0, but given a different name for clarity—it is not equal to the natural number 0, they don't even have the same type) is less than any number greater than or equal to 1, and for any number i, less than some number n, fsuc i is less than n + 1. Note that we have put the index n on the right side of the colon n the definition of Fin, this is so that [todo: is there a reason??? something with it being indexed (doesn't work if we move it). Alternatively, we could define 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 side of the colon, because otherwise the type would be too big [todo : Huh?] And then use it to define 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
```

Here, 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 example [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)
    ∧
  ∃ (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.This is actually quite a substantial task, that we accomplish in the following two sections. In Section 2.4 we prove the first part of the disjunct, and in Section 2.5, we prove the second.

2.4 First part of proof

This is actually quite a substantial task. We begin by proving the first disjunct (n : Fin (length xs)) → xs !! n ⩽ max xs pf

```
max-greatest : {xs : [ℕ]} → {pf : 0 < length xs} →
  (n : Fin (length xs)) → xs !! n ⩽ max xs pf
```

We have made the list and the proof that the length is greater than 0 implicit arguments because they can be infered from the resulting type `xs !! n ⩽ max xs pf`. However, when we prove the lemma, we are going to need to pattern match on those arguments many times.

We make the simple, but important observation that we cannot use `max-greatest` in the place of `(n : Fin (length xs)) → xs !! n ⩽ max xs pf` when giving the type of `max-spec`, because while the type of `max-greatest` is the proposition that `max xs pf` is the greatest element of the list, `max-greatest` itself is just one specific proof of that proposition.

is max-greatest a good name for it?

To prove a proposition in Agda, it is important to look at the structure of the proposition, and the structures of the involved parts. Then one needs to determine which structure should be pattern matched on, depending on what the inductive step in the proposition is.

To prove `max-greatest`, we begin by formulating the proof informally. The main idea we use is pattern matching the index into the list, if it is 0, we want to prove the simpler proposition that `x ⩽ max (x :: xs) pf`, which we call `max-greatest-initial`, because it is essentially the initial step in an induction on the index:

```
max-greatest-initial : {x : ℕ} {xs : [ℕ]} → x ⩽ max (x :: xs) (s⩽s z⩽n)
```

On the other hand, if the index is $i + 1$, we must have that the list must has length at least 2, we proceed by doing noting:

1. By induction, the ith element of the tail is less than the greatest element of the tail.
2. The ith element of the tail equals the $i + 1$th element of the list.
3. By the definition of `max`, `max (x :: (x′ :: xs)) pf` expands to `maxℕ x (max (x′ :: xs) pf′)`, and for any `x` and `y`, we have `y ⩽ maxℕ x y`.

To translate the induction case into Agda code, we need to introduce two new lemmas. By induction, we already know that Point 1 is true. Additionally, Agda infers Point 2. However, we still need to prove the second part of Point 3:

```
maxℕ-increasing₂ : {m n : ℕ} → n ⩽ maxℕ m n
```

Where the subscript 2 refers to the fact that it is the second argument of `maxℕ` that is on the left hand side of the inequality. Finally, we need a way to piece together inequalities, if `i ⩽ j` and `j ⩽ k`, then `i ⩽ k` (that is, `⩽` is transitive):

```
⩽-trans : {i j k : ℕ} → i ⩽ j → j ⩽ k → i ⩽ k
```

Now we begin proving these lemmas, beginning with `⩽-trans`, since it does not depend on the others (all the other lemmas will require further sublemmas

to prove). We pattern match on the first proof, $\mathsf{i} \leqslant \mathsf{j}$. If it is $\mathsf{z}\leqslant\mathsf{n}$, Agda infers that i is 0, so the resulting proof if $\mathsf{z}\leqslant\mathsf{n}$:

$$\leqslant\text{-trans}\ \mathsf{z}\leqslant\mathsf{n}\ \mathsf{j}\leqslant\mathsf{k} \ = \ \mathsf{z}\leqslant\mathsf{n}$$

If it is $\mathsf{s}\leqslant\mathsf{s}\ \mathsf{i}'\leqslant\mathsf{j}'$, Agda infers that $\mathsf{i} == \mathsf{suc}\ \mathsf{i}'$, and $\mathsf{j} == \mathsf{suc}\ \mathsf{j}'$, and $\mathsf{i}'\leqslant\mathsf{j}'$ is a proof that $\mathsf{i}' \leqslant \mathsf{j}'$. We pattern match on the proof of $\mathsf{j} \leqslant \mathsf{k}$, which must be $\mathsf{s}\leqslant\mathsf{s}\ \mathsf{j}'\leqslant\mathsf{k}'$, because j is $\mathsf{suc}\ \mathsf{j}'$. Hence, we can use induction to get a proof that $\mathsf{i}' \leqslant \mathsf{k}'$, and apply $\mathsf{s}\leqslant\mathsf{s}$ to that proof:

$$\leqslant\text{-trans}\ (\mathsf{s}\leqslant\mathsf{s}\ \mathsf{i}'\leqslant\mathsf{j}')\ (\mathsf{s}\leqslant\mathsf{s}\ \mathsf{j}'\leqslant\mathsf{k}') \ = \ \mathsf{s}\leqslant\mathsf{s}\ (\leqslant\text{-trans}\ \mathsf{i}'\leqslant\mathsf{j}'\ \mathsf{j}'\leqslant\mathsf{k}')$$

We continue by proving $\mathsf{max}\mathbb{N}\text{-increasing}_2$, for this, we introduce a lemma: $\leqslant$-refl, stating that for any n, $\mathsf{n} \leqslant \mathsf{n}$ (that is, $\leqslant$ is reflexive), which is very easy to prove (if $\mathsf{n} == 0$, a proof is given by the constructor $\mathsf{z}\leqslant\mathsf{n}$, and if $\mathsf{n} == \mathsf{suc}\ \mathsf{n}'$, by induction, $\mathsf{n}' \leqslant \mathsf{n}'$ and $\mathsf{s}\leqslant\mathsf{s}$ takes the proof of this to a proof that $\mathsf{n} \leqslant \mathsf{n}$:

$$\begin{aligned}
&\leqslant\text{-refl}\ :\ \{\mathsf{n}\ :\ \mathbb{N}\} \to \mathsf{n} \leqslant \mathsf{n}\\
&\leqslant\text{-refl}\ \{\mathsf{zero}\} \ = \ \mathsf{z}\leqslant\mathsf{n}\\
&\leqslant\text{-refl}\ \{\mathsf{suc}\ \mathsf{n}\} \ = \ \mathsf{s}\leqslant\mathsf{s}\ \leqslant\text{-refl}
\end{aligned}$$

Now we prove $\mathsf{max}\mathbb{N}\text{-increasing}_2$. We do this by pattern matching on the second argument (because it is the one involved in the inequality, and depending on its value, we need different constructors for the inequality proof). If it is zero, we use the constructor $\mathsf{z}\leqslant\mathsf{n}$, regardless of what the first argument is. If it is $\mathsf{suc}\ \mathsf{n}'$, we need to know what the first argument was, so we pattern match on it. If the first argument is zero, then, from the definition of $\mathsf{max}\mathbb{N}$, we know that $\mathsf{max}\mathbb{N}\ \mathsf{zero}\ (\mathsf{suc}\ \mathsf{n}') == \mathsf{suc}\ \mathsf{n}'$, so we want to prove that $\mathsf{suc}\ \mathsf{n}' \leqslant \mathsf{suc}\ \mathsf{n}'$, which we do by using the lemma $\leqslant$-refl (we note here that we didn't actually need the fact that the second argument was non-zero). On the other hand, if the first argument is $\mathsf{suc}\ \mathsf{m}'$, we know by induction (we call $\mathsf{max}\mathbb{N}\text{-increasing}_2$ where we need to supply at least the first argument, since it doesn't appear anywhere, and hence Agda is unable to infer it) that $\mathsf{n}' \leqslant \mathsf{max}\mathbb{N}\ \mathsf{m}'\ \mathsf{n}'$, so we use $\mathsf{s}\leqslant\mathsf{s}$ to get $\mathsf{suc}\ \mathsf{n}' \leqslant \mathsf{suc}\ (\mathsf{max}\mathbb{N}\ \mathsf{m}'\ \mathsf{n}')$, and from the definition of $\mathsf{max}\mathbb{N}$, we (and more importantly Agda) get that $\mathsf{suc}\ (\mathsf{max}\mathbb{N}\ \mathsf{m}'\ \mathsf{n}') == \mathsf{max}\mathbb{N}\ (\mathsf{suc}\ \mathsf{m}')\ (\mathsf{suc}\ \mathsf{n}')$, so we are in fact done.

$$\begin{aligned}
&\mathsf{max}\mathbb{N}\text{-increasing}_2\ \{\mathsf{m}\}\ \{\mathsf{zero}\} \ = \ \mathsf{z}\leqslant\mathsf{n}\\
&\mathsf{max}\mathbb{N}\text{-increasing}_2\ \{\mathsf{zero}\}\ \{\mathsf{suc}\ \mathsf{n}'\} \ = \ \leqslant\text{-refl}\\
&\mathsf{max}\mathbb{N}\text{-increasing}_2\ \{\mathsf{suc}\ \mathsf{m}'\}\ \{\mathsf{suc}\ \mathsf{n}'\} \ = \ \mathsf{s}\leqslant\mathsf{s}\ (\mathsf{max}\mathbb{N}\text{-increasing}_2\ \{\mathsf{m}'\}\ \{\mathsf{n}'\})
\end{aligned}$$

We also prove the similar proposition, $\mathsf{max}\mathbb{N}\text{-increasing}_1 : \{\mathsf{m}\ \mathsf{n} : \mathbb{N}\} \to \mathsf{m} \leqslant \mathsf{max}\mathbb{N}\ \mathsf{m}\ \mathsf{n}$, that $\mathsf{max}\mathbb{N}$ is greater than its first argument, in essentially the same way (we pattern match first on the first argument instead).

Using $\mathsf{max}\mathbb{N}\text{-increasing}_1$ and $\leqslant$-refl, we are able to prove the initial step in the induction proof, max-greatest-initial. We pattern match on the remainter of the list, if it is $[]$, we need to show that $\mathsf{x} \leqslant \mathsf{x}$, which is done with $\leqslant$-refl, and if it is $\mathsf{x}' :: \mathsf{xs}$, we need to prove that $\mathsf{x} \leqslant \mathsf{max}\ (\mathsf{x} :: (\mathsf{x}' :: \mathsf{xs}))\ \mathsf{pf}$, and

check that variable names are reasonably consistent

expanding this using the definition of max, we find that we need to prove that $x \leqslant \mathsf{max}\mathbb{N}\ x\ (\mathsf{max}\ (x' :: xs)\ pf')$, which is exactly what $\mathsf{max}\mathbb{N}\text{-increasing}_1$ does.

```
max-greatest-initial {x} {[]}  =  ⩽-refl
max-greatest-initial {x} {x′ :: xs}  =  maxℕ-increasing₁
```

Finally, we are able to finish our proof of max-greatest. As we said above, we want to pattern match on the index, however, this is not possible to do right away, since the available constructors (if any) for Fin (length xs) depends on the length of xs. Therefore, we begin by pattern matching on the list. If the list is empty, we fill in the absurd pattern () for the proof that it is nonempty. Otherwise, we pattern match on the index. If the index is fzero, we use the initial step max-greatest-initial, to prove that $x \leqslant \mathsf{max}\ (x :: xs)\ pf$. If the index is fsuc i, we pattern match on the tail of the list. If it is empty, we know that the index shouldn't have been fsuc i, because we'd have i : Fin zero, so we fill in i with the absurd pattern (). The case we have left is when the list is $x :: (x' :: xs)$, and the index is fsuc i. As we said above, we use induction to prove that $(x' :: xs) !!\ i \leqslant \mathsf{max}\ (x' :: xs)\ pf$. By the definition of !!, we have that

```
(x :: (x′ :: xs)) !! (fsuc i) == (x′ :: xs) !! i
```

So by induction, max-greatest i proves that $(x :: (x' :: xs)) !!\ (\mathsf{fsuc}\ i) \leqslant \mathsf{max}\ (x' :: xs)\ pf$, and additionally, from the definition of max,

```
max (x :: (x′ :: xs)) pf == maxℕ x (max (x′ :: xs) pf′)
```

So using $\mathsf{max}\mathbb{N}\text{-increasing}_2$, and ⩽-trans to put things together, we get the desired result:

clean up the proofs "pf" that are input to max

```
max-greatest {[]} {()} _
max-greatest {x :: xs} {s⩽s z⩽n} fzero  =  max-greatest-initial {x} {xs}
max-greatest {x :: []} {s⩽s z⩽n} (fsuc ())
max-greatest {x :: (x′ :: xs)} {s⩽s z⩽n} (fsuc i)  =  ⩽-trans (max-greatest i) (maxℕ-increasing₂ {x})
```

2.5 Second part of proof

In this section, we will prove the disjunction in the specification, that is:

Make first part of proof, making of specification, etc subsections (or something)

```
max-in-list : {xs : [ℕ]} → {pf : 0 < length xs} →
  ∃ (λ n → xs !! n ≡ max xs pf)
```

This is a bit different from the previous proof, because the definition of the existential quantifier ∃ in constructive mathematics states that we actually need a function that finds the maximum in the list and remembers that it is the maximum. So proving that something exists is in mainly a programming problem—in particular

Is pf a good name for a proof, or should they be more descriptive?

To find a function that does this, we begin by getting rid of the case when the list is empty, since then, there is no proof that it is non-empty. Then we

look at the definition of max. If the list contains only one element, we can return (fzero, refl), since the first element is returned by max and also by indexing at fzero, and refl proves that an element is equal to itself. That was the base case. If the list has at least two elements, we can find the maximum in the remaining list by induction. Depending on whether the first element is greater than this maximum or not, we then either return (fzero, refl) again, or increase the returned value and modify the proof returned by the maximum function.

The fact that we need the proof means that we can't simply define a type Bool and an if statement:

```
data Bool : Set where
  True : Bool
  False : Bool

if_then_else : {a : Set} → Bool → a → a → a
if True then x else y = x
if False then x else y = y
```

Along with a check like (we use the prime because we want the similar function we will actually use to be named _⩽?_):

```
_⩽?′_ : ℕ → ℕ → Bool
_⩽?′_ zero n = False
_⩽?′_ (suc m) zero = True
_⩽?′_ (suc m) (suc n) = m ⩽?′ n
```

Because while if (x ⩽?′ y) then x else y does return the maximum, it doesn't return a proof, and we cannot use it to convince Agda that x ⩽ y or vice versa. Instead, we need a function like that along with a Bool-like answer returns a proof that it is correct. This is exactly the point of the data type Dec we defined above **??**.

```
≡-cong : {a b : Set} {x y : a} → (f : a → b) → x ≡ y → f x ≡ f y
≡-cong f refl = refl

≡-trans : ∀ {a b c} → a ≡ b → b ≡ c → a ≡ c
≡-trans refl 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′

x<y⇒¬y⩽x : {x y : ℕ} → (x < y) → ¬ (y ⩽ x)
x<y⇒¬y⩽x (s⩽s m⩽n) = λ x → x<y⇒¬y⩽x x m⩽n
```

```
¬x⩽y⇒y⩽x : {x y : ℕ} → ¬ (x ⩽ y) → (y ⩽ x)
¬x⩽y⇒y⩽x {x} {zero} pf = z⩽n
¬x⩽y⇒y⩽x {zero} {suc n} pf with pf z⩽n
...| ()
¬x⩽y⇒y⩽x {suc m} {suc n} pf = s⩽s (¬x⩽y⇒y⩽x (λ x → pf (s⩽s x)))

p⩽p : {m n : ℕ} → (suc m ⩽ suc n) → m ⩽ n
p⩽p (s⩽s m⩽n) = m⩽n

_⩽?_ : (x : ℕ) → (y : ℕ) → Dec (x ⩽ y)
zero ⩽? n = yes z⩽n
suc m ⩽? zero = no (x<y⇒¬y⩽x (s⩽s z⩽n))
suc m ⩽? suc n with m ⩽? n
...| yes m⩽n = yes (s⩽s m⩽n)
...| no n⩽m = no (λ x → n⩽m (p⩽p x))

maxℕ-is-max : (x y : ℕ) → x ⩽ y → y ≡ maxℕ x y
maxℕ-is-max zero y pf = refl
maxℕ-is-max (suc m) zero ()
maxℕ-is-max (suc m) (suc n) (s⩽s m⩽n) = ≡-cong suc (maxℕ-is-max 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⩽
lemma″ zero x′ xs pf = refl
lemma″ (suc n) x′ xs pf = maxℕ-is-max (suc n) (max (x′ :: xs) (s⩽s z⩽n)) pf

increase : ∀ x x′ xs → x ⩽ max (x′ :: xs) (s⩽s z⩽n) → ∃ (λ i → (x′ :: xs) !! i ≡ max (x′ :: xs) (s⩽s z⩽n)) → ∃
increase x x′ xs pf′ (i, pf) = fsuc i, ≡-trans pf (lemma″ x x′ xs pf′)

  -- en max funktion som tar värde och kanske tom lista

min-finder : (x : ℕ) → (xs : [ℕ]) → ∃ (λ i → (x :: xs) !! i ≡ max (x :: xs) (s⩽s z⩽n))
min-finder x [] = fzero, refl
min-finder x (x′ :: xs) with x ⩽? max (x′ :: xs) (s⩽s z⩽n)
min-finder x (x′ :: xs) | yes x⩽y = increase x x′ xs x⩽y (min-finder x′ xs)
min-finder x (x′ :: xs) | no y⩽x = fzero, lemma x (x′ :: xs) (s⩽s z⩽n) (¬x⩽y⇒y⩽x y⩽x)

max-in-list {[]} {()}
max-in-list {(x :: xs)} {s⩽s z⩽n} = min-finder x xs
```

note that min″ wouldn't work, because Agda can't see that the structure gets smaller (could reformulate this wrt max-in-list, give different implementation

⩽-trans repeatedly leads to introduction of equational syntax, trap is trying to

2.6 Finish

Now, we are able to finish our proof of the specification by putting together the parts of the two previous sections.

If the list is empty, the proof would be an element of $1 \leqslant 0$, and that type is empty, so we can put in the absurd patern (). On the other hand, if the list is x :: xs, we make a pair of the above proofs, and are done:

```
max-spec [] ()
max-spec (x :: xs) (s⩽s z⩽n) = max-greatest, max-in-list
```

To end this example, we note that proving even simple (obvious) propositions in Agda takes quite a bit of work, and a lot of code, but generally not

much thinking. After this extended example, we feel that we have illustrated most of the techniques that will be used later on in the report. As we wrote in the introduction to the section, we will often only give the types of the propositions, followed with the types of important lemmas and note what part of the arguments we pattern match on and in what order.

We also feel that we have illustrated the fact that proving something in Agda often requires a lot of code, but not much thinking, as the above proof essentially proceeds as one would intuitively think to prove the specification correct. Most of the standard concepts used are available in one form or another from the standard library, and we have attempted to keep our names consistent with it (the actual code given in later sections uses the standard library when possible, but we try to include simplified definitions in this report).

fix references below

Another practical difference is that all programs have to terminate. This is guaranteed by requiring 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 introduce 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 that 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. We give first the mathematical definition, and then define it in Agda:

Definition 3.1. A group is a set G (sometimes called the *carrier*) together with a binary operation $\cdot$ on G, satisfying the following:

- $\cdot$ is associative, that is,
- There is an element $e \in G$ such that $e \cdot g = g \cdot e = g$ for every $g \in G$. This element is the *neutral element* of G.

- For every $g \in G$, there is an element g^{-1} such that $g \cdot g^{-1} = g^{-1} \cdot g = e$. This element g^{-1} is the *inverse* of g.

Remark. One usually refers to a group $(G, \cdot, e)$ simply by the name of the set G.

Remark. G doesn't actually need to be a set.

should this be noted

An important reason to study groups that many common mathematical objects are groups. First there are groups where the set is a set of numbers:

Example 3.1. The integers $\mathbb{Z}$, the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$ and the complex numbers $\mathbb{C}$, all form groups when $\cdot$ is addition and e is 0.

Example 3.2. The non-zero rational numbers $\mathbb{Q}\backslash 0$, non-zero real numbers $\mathbb{R}\backslash 0$, and non-zero complex numbers $\mathbb{C} \setminus 0$, all form groups when $\cdot$ is multiplication and e is 1.

Second, the symmetries of a In Agda code, we define the proposition IsGroup, that states that something is a group. We define this using a record so that we can give names to the different lemmas, because when reasoning about an arbitrary group (which we will define shortly), the only thing we have are these lemmas.

include that Agda records somewhere in Agda section

```
record IsGroup {G : Set} (_≈_ : G → G → Set) (_•_ : G → G → G) (e : G) (_⁻¹ : G → G) : Set where
  field
    isEquivalence : IsEquivalence _≈_
```

make note that we have taken names from standard library but use less general/simpler definitions

We note that we need to include the equality in the definition of the group along with the fact that it should be an equivalence relation, this is usually not mentioned in a mathematical definitions of a group, but is necessary here, because the structural equality of the type G is not necessarily the equality we want for the group (as not all sets are inductively defined).

We can then define the type Group, containing all groups with a record, so that we can have names for the different fields. Note that the type of Group is Set_1, because like Set, Group is "too big" to be in Set (if we want to avoid things like Russel's Paradox).

is this the word, is it used before— should be mentioned when introducing refl

```
record Group : Set₁ where
  infix 7 _•_
  infix 4 _≈_
  field
    Carrier : Set
    _≈_ : Carrier → Carrier → Set
    _•_ : Carrier → Carrier → Carrier
    e   : Carrier
    _⁻¹ : Carrier → Carrier
    isGroup : IsGroup _≈_ _•_ e _⁻¹
```

The things to note here are that Where we have both defined the various elements required in a group, along with the axioms they need to satisfy.

To prove that something is a group, one would thus

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).

Todo list

References

DUMMY. *DUMMY*.