Report

Thomas Bååth Sjöblom

April 22, 2013

1 Introduction

1.1 What?

ℕ 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 → 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 ¬ 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 Dec X that has two constructors, one taking a proof of an instance x : X and one a proof ¬x : ¬ 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 → 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 x : X, and appears again, and the proposition can depend on the value x: P : X → Set, that is universal quantification is a function (x : X) → P x:

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

That is, for any element x : X, we have P x. Agda allows the use of the syntax ∀ x to mean (x : _) in type definitions, so that ∀ x → P x means exactly what we expect it to mean (using the ∀ 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 ∃ 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 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 less than, _⩽_. 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 : 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 ⩽ 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

NAmes of _-pattern – if there is one

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

```
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:

$$\mathsf{max\mathbb{N}\text{-}increasing}_2 : \{\mathsf{m\ n} : \mathbb{N}\} \to \mathsf{n} \leqslant \mathsf{max\mathbb{N}\ m\ n}$$

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, i ⩽ j. If it is z⩽n, Agda infers that i is 0, so the resulting proof if z⩽n:

```
⩽-trans z⩽n j⩽k  =  z⩽n
```

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

```
⩽-trans (s⩽s i′⩽j′) (s⩽s j′⩽k′)  =  s⩽s (⩽-trans i′⩽j′ j′⩽k′)
```

We continue by proving maxℕ-increasing$_2$, for this, we introduce a lemma: ⩽-refl, stating that for any n, n ⩽ n (that is, ⩽ is reflexive), which is very easy to prove (if n == 0, a proof is given by the constructor z⩽n, and if n == suc n′, by induction, n′ ⩽ n′ and s⩽s takes the proof of this to a proof that n ⩽ n:

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

Now we prove maxℕ-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 z⩽n, regardless of what the first argument is. If it is suc 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 maxℕ, we know that maxℕ zero (suc n′) == suc n′, so we want to prove that suc n′ ⩽ suc n′, which we do by using the lemma ⩽-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 suc m′, we know by induction (we call maxℕ-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 n′ ⩽ maxℕ m′ n′, so we use s⩽s to get suc n′ ⩽ suc (maxℕ m′ n′), and from the definition of maxℕ, we (and more importantly Agda) get that suc (maxℕ m′ n′) == maxℕ (suc m′) (suc n′), so we are in fact done.

```
maxℕ-increasing₂ {m} {zero}  =  z⩽n
maxℕ-increasing₂ {zero} {suc n′}  =  ⩽-refl
maxℕ-increasing₂ {suc m′} {suc n′}  =  s⩽s (maxℕ-increasing₂ {m′} {n′})
```

We also prove the similar proposition, maxℕ-increasing$_1$: {m n : ℕ} → m ⩽ maxℕ m n, that maxℕ is greater than its first argument, in essentially the same way (we pattern match first on the first argument instead).

check that variable names are reasonably consistent

Using maxℕ-increasing$_1$ and ⩽-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 x ⩽ x, which is done with ⩽-refl, and if it is x′ :: xs, we need to prove that x ⩽ max (x :: (x′ :: xs)) pf, and expanding this using the definition of max, we find that we need to prove that x ⩽ maxℕ x (max (x′ :: xs) pf′), which is exactly what maxℕ-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 ⩽ 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 ⩽ 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)) !! (fsuc i) ⩽ max (x′ :: xs) pf, and additionally, from the definition of max,

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

So using maxℕ-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)

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

```
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

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 2.6.

So we want define the function _⩽?_ to return yes x⩽y, where x⩽y : x ⩽ y is a proof that x is less than or equal to y, or no ¬x⩽y, where ¬x⩽y : ¬ (x ⩽ y). If x == 0, this is straightforward, since we simply return yes z⩽n. If x == suc x′, we need to pattern match on y. The simples case is if y == 0, because then we need to derive ⊥ from suc x′ ⩽ 0.

More here (think about what the proof does, really) Also write that we curry/uncurry– whatever, actually ,this might be unneccessary

Next, in the case where x == suc x′ and y == suc y′, we need to know (with proof) which of x′ and y′ is greater. We need to pattern match on the Dec (x′ ⩽ y′), which is not part of the function arguments, and do this by introducing a new piece of Agda syntax, the **with** statement (we could of course use a helper function to do the pattern matching, but the with statement is simpler). After the function arguments, one writes **with** followed by a list of expressions to pattern match on, separated by vertical bars:|. Then on the line below, one writes either ... in place of the old arguments, followed by a bar, |, and the new arguments separated by bars, or (in case one wants to infer things about the old arguments based on the pattern matching), one repeats the function arguments in place of the .... We show both alternatives.

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

Above, we made a local definition of the proposition p⩽p stating that if both m and n are the successors of something, and if m ⩽ n, then the predecessor of m is less than or equal to the predecessor of n.

We note that in the above example, we didn't actually need to use the **with** construction, since we didn't use the result of the pattern matching (if we had pattern matched on m⩽n above, we could have inferred that whether the argument m to suc m was zero or suc m′, but that wasn't neccessary for this proof). We could instead have introduced a helper function (perhaps locally) that we call in place of the with statement:

include ref to where it is actually neccessary (if ever in this report)

```
helper : {m n : ℕ} → Dec (m ⩽ n) → Dec ((suc m) ⩽ (suc n))
helper (yes p) = yes (s⩽s p)
helper (no ¬p) = no (λ x → ¬p (p⩽p x))
```

So we write

```
min-finder x (x′ :: xs) with x ⩽? max (x′ :: xs) _
```

Now, we begin with the the case where x ⩽? max returns yes x⩽max. We thus have a proof that x ⩽ max (x′ :: xs), and by recursively calling min-finder x′ xs , we get an index i and a proof that the ith element of x′ :: xs is the greatest element there. Hence, the index of our maximum should be fsuc i, and we need to prove that given the above, max (x :: x′ :: xs) ≡ max (x′ :: xs), since then, the fsuc ith element in x :: x′ :: xs would be equal to max (x′ :: xs) by the definition of !!, and hence to max (x :: x′ :: xs).. We introduce the function move-right to move the proof one step to the right.

make sure I mention min-finder name when introducing it above

```
move-right : {x x′ : ℕ} {xs : [ℕ]} → x ⩽ max (x′ :: xs) (s⩽s z⩽n) → ∃ (λ i → (x′ :: xs) !! i ≡ max (x′ :: xs
```

We write out the arguments as

pattern match on the existence proof, getting (i, pf). We already know that the first part of the pair move-right should return (the witness) should be fsuc i,

```
  -- No CASE
≡-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)

¬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)))

x-max : (x : ℕ) (xs : [ℕ]) (pf : 0 < length xs) → max xs pf ⩽ x → x ≡ max (x :: xs) (s⩽s z⩽n)
x-max x [] pf pf′  =  refl
x-max x (x′ :: xs) (s⩽s z⩽n) pf′  =  l″ pf′

  -- yes case
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)

small-x⇒max-equal : (x x′ : ℕ) (xs : [ℕ]) → x ⩽ max (x′ :: xs) (s⩽s z⩽n) → max (x′ :: xs) (s⩽s z⩽n) ≡ m
small-x⇒max-equal zero x′ xs pf  =  refl
small-x⇒max-equal (suc n) x′ xs pf  =  maxℕ-is-max (suc n) (max (x′ :: xs) (s⩽s z⩽n)) pf

move-right {x} {x′} {xs} x⩽max (i, pf)  =  fsuc i, ≡-trans pf (small-x⇒max-equal x x′ xs x⩽max)

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) _
min-finder x (x′ :: xs) | yes x⩽max  =  move-right x⩽max (min-finder x′ xs)
min-finder x (x′ :: xs) | no max⩽x  =  fzero, x-max x (x′ :: xs) _ (¬x⩽y⇒y⩽x max⩽x)

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

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
```

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

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 (only visible in source)

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 Introductory defintions

Before covering some algebraic structures, we would like to define the things needed to talk about them in Agda. These are mainly propositions regarding functions and relations.

3.1.1 Relation properties

The first thing to discuss is the equivalence relation. It is a relation that acts like an equality.

Definition 3.1. A relation $R \subseteq X \times X$ is called an *equivalence relation* if it is

write sqiggly line instead of R

- Reflexive: for $x \in X$, xRx.
- Symmetric: for x, $y \in X$, if xRy, then yRx.
- Transitive: for x, y, $z \in X$, if xRy and yRz, then xRz.

We formalize the way it behaves like an equality in the following proposition:

Proposition 3.2. *An equivalence relation R partitions the elements of a set X into disjoint nonempty equivalence classes (subsets $[x] = \{y \in X : yRx\}$) satisfying:*

- *For every $x \in X$, $x \in [x]$.*
- *If $x \in [y]$, then $[x] = [y]$.*

If we replace the equality of elements with an equivalence relation, i.e., that two elements are "equal" if they belong to the same equivalence class, we get a coarser sense of equality. To see that it acts as the regular equality, we note that the equivalence relation is equality on equivalence classes.

Expand/clean up on induced equality from equivalence classes.

To define an equivalence relation in Agda, we first need to define what a relation (on a set) is.

write about definition – and think about whether it need s to be there as opposed to A → A → Set

```
Rel : Set → Set₁
Rel X = X → X → Set
```

Next, we need to define the axioms it should satisfy: reflexivity, symmetry and transitivity. The first thing to note is that they are all propositions (parametrized by a relation), so they should be functinos from Rel to Set (which is where propositions live).

```
Reflexive : {X : _} → (_∼_ : Rel X) → Set
Reflexive _∼_ = ∀ {x} → x ∼ x
Symmetric : {X : _} → (_∼_ : Rel X) → Set
Symmetric _∼_ = ∀ {x y} → x ∼ y → y ∼ x
Transitive : {X : _} → (_∼_ : Rel X) → Set
Transitive _∼_ = ∀ {x y z} → x ∼ y → y ∼ z → x ∼ z
```

Then, we define the record IsEquivalence, for expressing that a relation is an equivalence relation (we use a record to be able to extract the three axioms with using names)

```
record IsEquivalence {X : Set} (_∼_ : Rel X) : Set where
  field
    refl  : Reflexive  _∼_
    sym   : Symmetric _∼_
    trans : Transitive _∼_
```

3.1.2 Operation Properties

Next, we define some properties that binary operation (i.e., functions $X \to X \to X$) can have. These are functions that are similar to ordinary addition and multiplication of numbers (if they have all the properties we define below—but it can be useful to think of them that way even if they don't).

Definition 3.3. A function is

Example 3.1.

Definition 3.4. A function is

Example 3.2.

3.1.3 Properties of pairs of operations

When we have two different binary operations on the same set, we often want them to interact with each other sensibly, where sensibly means as much as multiplication and addition of numbers interact as possible. We recall the distributive law $a \cdot (b + c) = a \cdot b + a \cdot c$, where x, y, and z are numbers and $\cdot$ and $+$ are multiplication and addition and generalize it to arbitrary operations:

Definition 3.5. A binary operation $\cdot$ on A *distributes over* a binary operation $+$ if, for all x, y, z,

- $x \cdot (y + z) = x \cdot y + x \cdot z$,
- $(y + z) \cdot x = y \cdot x + z \cdot x$,

where we assume that $\cdot$ binds its arguments harder than $+$.

In Agda, we begin by defining what a binary operation is:

```
Op₂ : Set → Set
Op₂ A = A → A → A
```

In Agda, we define the two requirements separately, as _DistributesOverˡ_ and _DistributesOverʳ_, for left and right distributive repectively, so the statement that * distributes over + on the right becomes the very readable * DistributesOverˡ + (sadly, the definition cannot be written in this readable syntax due to the fact that we need to include an implicit argument to express equality):

```
_DistributesOverˡ_ : {A : Set} {_≈_ : Rel A} → Op₂ A → Op₂ A → Set
_DistributesOverˡ_ {A} {_≈_} _*_ _+_ = ∀ x y z → (x * (y + z)) ≈ ((x * y) + (x * z))
_DistributesOverʳ_ : {A : Set} {_≈_ : Rel A} → Op₂ A → Op₂ A → Set
_DistributesOverʳ_ {A} {_≈_} _*_ _+_ = ∀ x y z → ((y + z) * x) ≈ ((y * x) + (z * x))
```

And then we combine them to make the proposition _DistributesOver_:

```
_DistributesOver_ : {A : Set} {_≈_ : Rel A} → Op₂ A → Op₂ A → Set
_DistributesOver_ {A} {_≈_} _*_ _+_ = (_DistributesOverˡ_ {A} {_≈_} _*_ _+_) ∧ (_DistributesOverʳ_ {A}
```

The second such interaction axiom we will consider comes from the fact that 0 annihilates things when involved in a multiplication of numbers: $0 * x = 0$.

3.2 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 3.6. 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.

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 3.3. 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 3.4. The non-zero rational numbers $\mathbb{Q} \setminus 0$, non-zero real numbers $\mathbb{R} \setminus 0$, 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

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

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

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).

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

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).

```
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

3.2.1 Cayley Table

One

3.3 Monoid-like structures

3.3.1 Definition

3.3.2 Cayley Table

3.4 Ring-like structures

3.4.1 Definitions

As we discussed above, in Section **??**, when one has two operations on a set, one often wants them to interact sensibly. The basic example from algebra is a Ring:

Definition 3.7. A set R together with two binary operations $+$ and $*$ forms a ring if

- It is an abelian group with respect to $+$.

- It is a monoid with respect to $*$.

- Multiplication distributes over addition.

From these facts, it is in fact possible to prove that the additive identity 0 is an absorbing element with respect to multiplication ($0 * x = x * 0 = 0$ for every $x \in X$).

include proof? – useful exercize – proof exists in standard library

However, for the applications we have in mind, Parsing, the algebraic structure in question (see section **??**) does not even have associative multiplication (see section **??**), and does not have inverses for addition. We still have an additive 0 (the empty set—representing no parse), and want it to be an absorbing element with regard to multiplication (if the left or right substring has no parse, then the whole string has no parse). But the proof that 0 is an absorbing element depends crucially on the existence of the ability to cancel (implied by the existence of additive inverses in a group).

For this reason, we include as an axiom that zero annihilates. We this, we define a non-associative non-ring(which doesn't have :

Definition 3.8. A set X together with two operations $+$ and $*$ called addition and multiplication forms a *non-associative non-ring* if:

- It is a commutative monoid with respect to addition.
- It is a magma with respect to multiplication.
- Multipliciation distributes over addition.
- The additive identity is a multiplicative zero.

more text?

3.4.2 Matrices

We recall that a matrix is really just a square set of numbers, so there is nothing stopping us from defining one over an arbitrary ring, or even over a non-associative non-ring, as opposed to over $\mathbb{R}$ or $\mathbb{C}$. To be similar to the definition we will make in Agda of an abstract matrix (one without a specific implementation in mind), we consider a matrices of size $m \times n$ as a functions from a pair of natural numbers (i, j), with $0 \leq i < m$, $0 \leq j < n$ (or more specifically, $i \in \mathbb{Z}_m$, $j \in \mathbb{Z}_n$, and hence, after currying define:

uncurrying?

Definition 3.9. A matrix A over a set R is a function $A : \mathbb{Z}_m \to \mathbb{Z}_n \to R$.

When talking about matrices mathematically, we write A_{ij} for Aij

Fix formatting of 0#

In Agda, we will only define the type of a matrix over the non-associative non-ring. For simplicity, and to allow us to avoid adding the non-associative non-ringas an argument to every functino and proposition, we decide to parametrize the module the definition of the matrix by a nanring. We must first ensure that we have it imported, by starting the module file (named `Matrix.lagda`) with

```
open import NANRing.agda
```

Then we continue by writing

```
module Matrix (NAR : NonAssociativeNonRing) where
```

We open the record NonAssociativeNonRing with NAR so that we will be able to use the definitions in the ring easily, and rename things so that they do not clash with concepts we will define for the matrices (and also to help us figure out what operation we are using):

```
open NonAssociativeNonRing NAR renaming (_+_ to _R+_; _*_ to _R*_; _≈_ to _R≈_)
```

If we didn't open the record, instead of, for example a + b, we would have to write

```
(NonAssociativeNonRing._+_ NAR) a b
```

Now we are ready to define our matrix type in Agda:

```
Matrix : (m n : ℕ) → Set
Matrix m n = Fin m → Fin n → Carrier
```

As with the algebraic structures previously, we would like a way to speak of equality among matrices. First, we make a helpful definition of the equality in the non-associative non-ringNAR: We will thus define matrix equality, which we denote by _M≈_ to disambiguate it from the regular equality (in the library it is called _≈_, since it is in its own module). It should take two matrices to the proposition that they are equal, and two matrices A and B are equal if for all indices i and j, A i j and B i j are equal.

make sure that it really is called _≈_ in the library

note about extensionality ?

```
_M≈_ : {m n : ℕ} → Matrix m n → Matrix m n → Set
A M≈ B = ∀ i j → A i j R≈ B i j
```

If R is a ring or a non-associative non-ring, we can define addition and multiplication of matrices with the usual formulas:

$$(A + B)_{ij} = A_{ij} + B_{ij}$$

and

$$(A * B)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Above, we used $*$ to denote matrix multiplication, even though it is normally written simply as AB, because in Agda we need to give it a name.

To define the addition in Agda is straightforward:

```
_M+_ : {m n : ℕ} → Matrix m n → Matrix m n → Matrix m n
A M+ B = λ i j → A i j R+ B i j
```

To define multiplication, on the other hand, we consider the alternative definition of the product as the matrix formed by taking scalar products between the rows of A and the columns of B:

$$(A * B)_{ij} = \mathbf{a_i} \cdot \mathbf{b_j}, \tag{1}$$

where $\mathbf{a_i}$ is the ith row vector of A and $\mathbf{b_j}$ is the jth column vector of B.

For this, we define the datatype Vector of a (mathematical) vector, represented as a functino from indices to non-associative non-ringelements:

note about difference between it and Vec?

```
Vector : (n : ℕ) → Set
Vector n = Fin n → Carrier
```

We define the dot product by pattern matching on the length of the vector:

```
_•_ : {n : ℕ} → Vector n → Vector n → Carrier
_•_ {zero} u v = 0#
_•_ {suc n} u v = (head u R* head v) R+ (tail u • tail v)
  where head : {n : ℕ} → Vector (suc n) → Carrier
    head v = v fzero
    tail : {n : ℕ} → Vector (suc n) → Vector n
    tail v = λ i → v (fsuc i)
```

With it, we define matrix multiplication:

```
_*_ : {m n p : ℕ} → Matrix m n → Matrix n p → Matrix m p
(A * B) i j = (λ k → A i k) • (λ k → B k j)
```

Here, the type system of Agda really helps in making sure that the definition is correct. If we start from the fact that the product of a $m \times n$ matrix and an $n \times p$ matrix is an $m \times p$ matrix, then the type system makes sure that our vectors are row vectors for A and column vectors for B.

Alternatively, if we start from the formula (3.4.2), the type system forces A to have as many rows as B has columns.

fill in refer- ecnes below

The most interesting fact about matrices (to our application) is the following proposition:

Proposition 3.10. *If R is a a ring (non-associative non-ring), then the matrices of size $n \times n$ over R also form a ring (non-associative non-ring).*

The proof is fairly easy but boring. We prove the case where R is a non-associative non-ringin Agda (because it is the case we will make use of later).

As before, we put the proof in a parametrised module, so we always have access to our base non-associative non-ringcalled NAR.

Then, the proof is an element

```
MatrixIsNonAssociativeNonRing : {!!}
MatrixIsNonAssociativeNonRing = {!!}
```

DO PROOF AND WRITE STUFF!

write the proof that matrices are a ring

3.5 Triangular Matrices

For our applications, we will be interested in matrices that have no non-zero elements on or below the diagonal.

Definition 3.11. A matrix is *upper triangular* if all elements on or below its diagonal are equal to zero.

Missing figure

draw figure of two matrices, one with zeros below diagonal, one with nothing (or stars or somethign)

Since we are only interested in upper triangular matrices, we will sometimes refer to them as just *triangular* matrices. We generalize the defintion to matrices that have zeros on their super diagonals too.

In Agda, there are two obvious ways to define a triangular matrix. The first way would be to use records, where a triangular matix is a matrix along with a proof that it is triangular. The second way would be to use functions that take two arguments and return a ring element, but where the second argument must be strictly greater than the first. We show one difference between the two approaches in Figure 3.5

We choose the first approach here, because it will make it possible to use the majority of the work from when we proved that matrices form a non-associative non-ringto show that triangular matrices also form a non-associative non-ring(or a ring, if their elements form a ring), under the obvious multiplication, addition and equality. The only problem we will have is proving that the multiplication is closed. Here it is important repeat that by triangular, we mean upper triangular (although everything would work equally well if we used it to mean lower triangular, as long as it doesn't include both upper and lower) if both upper and lower triangular matrices were allowed, we would not get a ring, , since it is well known that any matrix can be factorized as a product of a lower and an upper triangular matrix.

One additional reason for not choosing the second approach is that inequalities among Fin are not very nice .

expand on this paragraph

Thus we define triangular matrices of triangularity d (and give them the name Triangle):

```
record Triangle (n : ℕ) : Set where    -- (d : Fin n) : Set where
  field
    mat : Matrix n n
    tri : (i j : Fin n) → i ⩽ j → mat i j R≈ 0 #
```

We also define two Triangles to be equal if they have the same underlying matrix, since the proof is only there to ensure us that they are actually upper triangular.

do we actually want arbitrary triangularity? Pros: makes going from sum to different spec easier, do we want that? Cons: trickier defi-

```
_T≈_ : {n : ℕ} → Triangle n → Triangle n → Set
```

```
A T≈ B  =  Triangle.mat A M≈ Triangle.mat B
```

Now, we go on to define addition and multiplication of triangles. We apply matrix addition on their matrices and modify their proofs. For addition, the proof modification is straightforward:

maybe time to switch to standard library things?

```
_T+_ : {n : ℕ} → Triangle n → Triangle n → Triangle n
A T+ B  =  record
  {mat  =  Triangle.mat A M+ Triangle.mat B;
   tri  =  λ i j i⩽j → {!A i j + B i j !}}
```

write triangle multiplication. Then the end of the algebra chapter is hit

4 Parsing

Parsing is the process of annoting a string of tokens with structural properties. We will only give a fairly general overview of the process, to tie it in with the algebra we discussed in Section **??**. We note that this section contains very little Agda code, instead we move back to mathematical notation. In Section **??**, we will then focus on a particular algorithm for parsing, Valiant's Algorithm, that we implement and prove the correctness of using Agda.

very little or no

4.1 Definitions

The goal of parsing is to first decide if a given string of tokens belongs to a given language, and second to describe its structure in the language.

To do these two things, one uses a *grammar*, which contains rules for assigning structural properties to tokens and sequences of tokens.

Definition 4.1. A *grammar* G is a tuple (N, T, P, S), where

- N is a finite set of nonterminals,
- T,
-
- ,

A grammar can generate a string of terminals by repeatedly applying production rules to the start symbol. A grammar is used to describe a language (a set of strings of tokens). We say that grammar G generates a language L if the strings in L are exactly the strings that can be generated from G by repeatedly applying

this paragraph is a mess

Definition 4.2. Context Free Grammar

Definition 4.3. Chomsky Normal Form (or reduced CNF) — probably only consider languages that don't contain the empty string, for simplicity.

Any Context Free Grammar can be converted into one in Chomsky Normal Form. Hence we only consider grammars in Chomsky Normal Form in the rest of the report.

reference, and size increase

4.2 Grammar as an algebraic structure

When looking at the set of production rules for a grammar in Chomsky Normal Form, we see some similarities with the definition of a multiplication in a magma in Section 4.2.1: If we only consider the production rules involving only nonterminals:

$$A \to BC,$$

and if we further reverse places of A and BC, replace the arrow $\to$ by an equals sign $=$, we get

$$BC = A,$$

which we can consider as defining the product of B and C to be equal to A, giving us a multiplication table similar to (**??**).

Note also that as in Section 4.2.1, the multiplication is little more than a binary operation. Grammars are usually not associative:

what does associative represent here? also, commutative, have inverse, have unit — write down the equations for each.

This looks very nice, but we note that we only considered a single production $A \to BC$. When we try to apply this to the whole of P, there are two problems:

1. What happens if P contains $A \to BC$ and $D \to BC$, where A and D are different nonterminals?

2. What happens if, for some pair B and C of nonterminals, P contains no rule $A \to BC$?

The first problem is related to the fact that some strings have different many parses, and the second problem is related to the fact that some strings have none (i.e., they don't belong to the language).

The solution to these two problems is to consider *sets* of nonterminals, with the following multiplication:

$$\{A_1, \ldots, A_n\} \cdot \{B_1 \ldots B_m\} = \{A_1B_1, \ldots, A_nB_m, A_2B_1 \ldots, A_2B_m, \ldots, A_nB_n\}$$

4.2.1 Parsing as Trasitive Closure

When we consider the

was it valiant who came up with this idea? – include reference to whoever

5 Valiant's Algorithm

In his paper **?**, Leslie G. Valiant gave a divide and conquer algorithm for chart parsing that has the same time complexity as matrix multiplication. The algorithm divides a string into two parts, and parses them recursively, and then puts

fix references to other sections

them together through a fairly complicated procedure that requires a constant number of matrix multiplications.

Since the algorithm is a divide and conquer algorithm (and the combining step is also fairly paralellizable), it could potentially be used for parsing in parallel, as suggested by Jean-Philippe Bernardy and Koen Claessen **?**.

- or –

5.1 Specification of transitive closure

should this be here? might not make sense

5.2 The Algorithm

We want to compute the transitive closure of the parse chart. The main idea of the algorithm is to split the chart along the diagonal, into two subcharts and a rectangular overlap region, see Figure **??**. Next, compute the transitive closures of the subcharts, and combine them (somehow) to fill in the rectangular part. We note that charts of size 1×1 are the zero matrix, where the transitive closure is also the zero matrix. We also note that it is easy to compute the transitive closure of subcharts that have size 2×2, since all charts are nonzero on the diagonal, they have only one nonzero element:

note that we want to compute the transitive closure here, not *parse*.

$$\begin{pmatrix}0x\\00\end{pmatrix},$$

and hence the specification (**??**) is:

$$\begin{pmatrix}0c00\end{pmatrix} = \begin{pmatrix}0 & c\\0 & 0\end{pmatrix}\begin{pmatrix}0 & c\\0 & 0\end{pmatrix} + \begin{pmatrix}0 & x0 & 0\end{pmatrix}$$

which turns into $c = x$, since $CC = \mathbf{0}$.

When the chart X is $n \times n$, with $n > 1$, we can write it down as a block matrix

$$C = \begin{pmatrix}U & R\\0 & L\end{pmatrix}$$

where U is upper triangular and is the chart corresponding to the first part of the string (the *upper* part of the chart), L is upper triangular and is the chart corresponding to the second part of the string (the *lower* part of the chart, and R corresponds to the parses that start in the first string and end in the second string (the *rectangular* part of the chart).

If we put this into the specification, we get:

$$\begin{pmatrix}U^+ & R^+\\0 & L^+\end{pmatrix} = \begin{pmatrix}U^+ & R^+\\0 & L^+\end{pmatrix}\begin{pmatrix}U^+ & R^+\\0 & L^+\end{pmatrix} + \begin{pmatrix}U & R\\0 & L\end{pmatrix}$$

where U^+, R^+, L^+ are the corresponding parts of (a priori, we don't know if U^+ and L^+ are the transitive closures of U, L). Multiplying together C^+C^+, and adding C, we get:

$$\begin{pmatrix}U^+ & R^+\\0 & L^+\end{pmatrix} = \begin{pmatrix}U^+U^+ + R^+\mathbf{0} + U & U^+R^+ + R^+L^+ + R\\0 & \mathbf{0}R + L^+L^+ + L\end{pmatrix} = \begin{pmatrix}U^+U^+ + U & U^+R^+ + R^+L^+ + R\\0 & L^+L^+ + L\end{pmatrix},$$

since **0** is an absorbing element. Since all elements of two matrices need to be equal for the matrices to be equal, we get the set of equations:

$$\begin{aligned} U^+ &= U^+U^+ + U && (2)\\ R^+ &= U^+R^+ + R^+L^+ + R && (3)\\ L^+ &= L^+L^+ + L, && (4) \end{aligned}$$

so we see that it the condition that C^+ is the transitive closure of C is equivalent to the conditions that the upper and lower parts of C^+ are the transitive closures of the upper and lower parts of C, respectively (intuitively, this makes sense, since the transitive closure of the first part describes the ways to get between nodes in the first part, and these don't depend on the second part, and vice versa, since the matrix is upper triangular—i.e., while parsing a subset of the of the first part of a string, it doesn't matter what the second part of the string is, because the grammar is context free) and the rectangular part of C^+ satisfies the equation (3).

Hence, if we compute the transitive closures of the upper and lower part of the matrix recursively, we only need to put them together and compute the rectangular part of the matrix. To do this, we subdivide R into four blocks:

$$R = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

So assuming

So Valiant's algorithm is:

should it be written like this?

5.3 Implementation

In this section, we are going to implement Valiant's Algorithm.

5.3.1 Data types

To implement this in Agda using the Matrix and Triangle datatype from Section **??** would be very complicated since we would have to handle the splitting manually. Instead, we define concrete representations for the matrices and triangle that have the way we split them built inWe will call the datatypes we use Mat and Tri for general matrices and upper triangular matrices, respectively. To build the split into the data types, we give them constructors for building a large Mat or Tri from four smaller Mats or two Tri and one Mat respectively. Since we need Mat to define Tri, it should appear earlier on in the Agda code, and we begin by reasoning about it. By the above, we have one constructor ready, which we will call quad, and which takes four smaller matrices and puts them together into a big one. Written mathematically, we want the meaning to be:

$$\text{quad}(A, B, C, D) = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \qquad (5)$$

make note about us using matrix for Mat here.

where A has the same number of rows as B, C has the same number of rows as D, A has the same number of columns as C and B has the same number of columns as D. Thinking about what "small" structures should have constructors, we we realize that it is not enough to simply allow 1×1 matrices, since then, any matrix would be a $2^n \times 2^n$ matrix, where n is the number of times we use quad.

One way to to solve this problem is to have a constructor for "empty" matrices of any dimension, that play two different roles. First, empty $0 \times n$ matrices are used to allow quad to put two matrices with the same number of rows next to each others:

$$\mathrm{quad}(A, B, e_{0\,m}, e_{0\,n}) = \begin{pmatrix} A & B \\ e_{0\,m} & e_{0\,n} \end{pmatrix} = \begin{pmatrix} A & B \end{pmatrix}, \qquad (6)$$

where e_m and e_n are empty $m \times 0$ and $n \times 0$ matrices respectively. Similarily, empty $n \times 0$ matrices are used to put two matrice with the same number of columns on top of each others. Second, an empty $m \times n$ matrix, $m \neq 0$, $n \neq 0$, represents a $m \times n$ matrix whose entries are all zero. This approach is taken in **?**. One advantages of this method is that one can probably get some speedup when adding and multiplying with "empty" matrices:

$$e_{m\,n} + A = A + e_{m\,n} = A e_{m\,n} * A = e_{m\,p} A * e_{n\,p} = e_{m\,p},$$

where A is an arbitrary $m \times n$, $n \times p$ and $m \times n$ matrix, respectively. Another is that it keeps the number of constructors down (three constructors for the matrix type), and this is desirable when proving things with Agda, since one often has to deal separately with each constructor, to establish the base cases in an induction.

One (potential) downside with this approach is that while it allows easy construction of zero-matrices of arbitrary size, non-zero matrices still require many constructor application. For example, to make a $2^k \times 1$ vector, we'd have to build a tree of "n" applications of quad.

count

Another approach, which we take in this report, is to allow row and column vectors, that is $1 \times n$ and $n \times 1$ matrices for arbitrary $n > 1$, along with the single element matrices. That is, we define rVec and cVec to take a vector of length $n > 1$ and turn it into a $1 \times n$ or $n \times 1$ matrix respectively. This approach has the advantage that we can define all matrices in a simple way, and that we could potentially specialize algorithms when the input is a vector, but introduces one extra constructor (one for rows, one for columns and one for single elements and quad, as opposed to one for empty matrices, one for single element matrices and quad).

Similarily to the matrices, we then want a concrete representation Vec of vectors. Since we (probably) want to be able to split vectors too along the middle, we give them a constructor two that takes a vector of length m and one of length n and appends them. For our base cases, we need to be able to build single element vectors, and this turns out to be enough, since we can then build any vector. To implement this approach, we need to define the datatypes Vec of vectors and Mat of matrices (that shoud be concrete representations of Vector and Matrix).

The naive (and not the way we finally decide on, for reasons that become clear later, hence we add a ′ to the datatypes) way, which stays close to the Vector and Matrix datatypes would be to define Vec′ as something like

```
data Vec′ : ℕ → Set where
  one : (x : Carrier) → Vec′ 1
  two : {m n : ℕ} → Vec′ m → Vec′ n → Vec′ (m + n)
```

and then defining Mat′ as

```
data Mat′ : ℕ → ℕ → Set where
  sing : (x : Carrier) → Mat′ 1 1
  rVec : {n : ℕ} → Vec′ (suc (suc n)) → Mat′ 1 (suc (suc n))
  cVec : {n : ℕ} → Vec′ (suc (suc n)) → Mat′ (suc (suc n)) 1
  quad : {r₁ r₂ c₁ c₂ : ℕ} → Mat′ r₁ c₁ → Mat′ r₁ c₂ →
    Mat′ r₂ c₁ → Mat′ r₂ c₂ → Mat′ (r₁ + r₂) (c₁ + c₂)
```

should we call Vec′ Vec or Vec′ in text, and code?

Where we name the indices r_1, r_2, c_1 and c_2 to for rows and columns of the involved matrices, and the ordering is so that we can write it on two rows.

While this looks like a very natural way to define the datatypes, it will not work well when we want to prove things about the matrices. As we have mentioned before, the main way to prove things in Agda is to use structural induction by pattern matching on the structures involved. However, if we pattern match on a Mat′, one problem that appears is that Agda is unable to see that in the quad case, both indices must be at least 2, nor that both terms a and b have to be at least 1. It is possible to write lemmas proving this, and use them at every step. However, there are worse cases, when Agda's ability to unify indices won't help us when doing more complicated things, like realizing that some integer n is equal to a + b, also, we can't tell whether a is a sum or not, so the second splitting step is complicated, for example .

include short example

Instead we want to use a different approach for indexing our matrices, by building the splitting further into the data structures. Looking at the first attempt to define quad, we can perhaps guess that the indexing should have a constructor that puts two sub-indices together to form a new index (as in a + b), because then, quad would result in a Mat whose indices are clearly distinguishable from the single index (that is 1 above). Hence, we want something like ℕ, but, instead of having suc as a constructor, it should have +. We call this datatype Splitting, since it indexes the splitting of the matrix, and define it as follows

ask JP where the idea is from

```
data Splitting : Set where
  one : Splitting
  bin : (s₁ : Splitting) → (s₂ : Splitting) → Splitting
```

where one plays the role of suc zero (since there's no reason to have dimensions 0 for matrices, and bin plays the role of + (we have chosen the name bin to connect it to binary trees: we can think of ℕ as the type of list with elements

change to bin in actual code

from ⊤, where ⊤ is the one element type; then Splitting is the type of binary trees with elements ⊤).

We also define the translation function that takes a Splitting to an element of ℕ, by giving the one-splitting the value 1 and summing the sub splittings otherwise:

```
splitToℕ : Splitting → ℕ
splitToℕ one = 1
splitToℕ (bin s₁ s₂) = splitToℕ s₁ + splitToℕ s₂
```

Using this data type we can finally define our data types Mat and Tri. Mimicking the above, but using Splittings as indices (the code is essentially the same, with every instance of "ℕ" replaced by "Splitting"), we first define Vec as:

data type or datatype?

probably have Tri above too

```
data Vec : Splitting → Set where
  one : (x : Carrier) → Vec one
  two : {s₁ s₂ : Splitting} → (u : Vec s₁) → (v : Vec s₂) → Vec (bin s₁ s₂)
```

We can note that where Splitting is a binary tree of elements of the unit type, Vec is instead a binary tree of Carrier (with elements in the leaves). We move on to defining Mat as:

```
data Mat : Splitting → Splitting → Set where
  sing : (x : Carrier) → Mat one one
  rVec : {s₁ s₂ : Splitting} → (v : Vec (bin s₁ s₂)) → Mat one (bin s₁ s₂)
  cVec : {s₁ s₂ : Splitting} → (v : Vec (bin s₁ s₂)) → Mat (bin s₁ s₂) one
  quad : {r₁ r₂ c₁ c₂ : Splitting} → (A : Mat r₁ c₁) → (B : Mat r₁ c₂) →
    (C : Mat r₂ c₁) → (D : Mat r₂ c₂) → Mat (bin r₁ r₂) (bin c₁ c₂)
```

The definition of the last datatype involved, Tri is straightforward from the subdivision made above **??**. There is only one base case, that of the 1 × 1 zero triangle (equal to the 1 × 1 zero matrix when viewed as an upper triangular marix), and putting together Tris is straightforward since the upper triangular matrices need to be square, now that our matrices can have any shape, and the definition guarantees that the two step splitting in **??** can be done:

```
data Tri : Splitting → Set where
  one : Tri one
  two : {s₁ s₂ : Splitting} → (U : Tri s₁) → (R : Mat s₁ s₂) → (L : Tri s₂) → Tri (bin s₁ s₂)
```

Where again, the ordering of the arguments to two (it takes *two* Tris) is such that if we introduce a line break after Mat s_1 s_2, and indent Tri s_2 so it is below Mat s_1 s_2, they have the shape of an upper triangular matrix.

Here, we note that if we had chosen the approach with empty matrices (see **??**), and correspondingly, empty Splittings, we might have needed an extra constructor for triangles also .

think, is this true???

To end this section, we define addition and multiplication for Mat and then for Tri.

Addition is straightforward, since matrix addition is done pointwise, so we just recurse on the subparts, first we need to define it for Vec:

```
_V+_ : {s : Splitting} → Vec s → Vec s → Vec s
one x V+ one x' = one (x R+ x')
two u v V+ two u' v' = two (u V+ u') (v V+ v')
```

Then for Mat:

```
_M+_ : {s_1 s_2 : Splitting} → Mat s_1 s_2 → Mat s_1 s_2 → Mat s_1 s_2
sing x M+ sing x' = sing (x R+ x')
rVec v M+ rVec v' = rVec (v V+ v')
cVec v M+ cVec v' = cVec (v V+ v')
quad A B C D M+ quad A' B' C' D' = quad (A M+ A') (B M+ B') (C M+ C') (D M+ D')
```

Finally for Tri:

```
_T+_ : {s : Splitting} → Tri s → Tri s → Tri s
one T+ one = one
two U R L T+ two U' R' L' = two (U T+ U') (R M+ R') (L T+ L')
```

For multiplication, we need to do a bit more work (and in particular, we need to have already defined addition). The first thing to note is that if we have two matrices split into blocks, where the splitting of the columns of the first matrix equals the splitting of the rows of the second (similar to the fact that to multiply matrices A and B, A must have as many columns as B has rows), matrix multiplication works out nicely with regard to the block structures:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} A' & B' \\ C' & D' \end{pmatrix} = \begin{pmatrix} AA' + BC' & AB' + BD' \\ CA' + DC' & CB' + DD' \end{pmatrix} \tag{7}$$

We will use this formula to define multiplication for Mat. We will therefore not define multiplication for Mats where the inner splittings are not equal—so our Mat multiplication is less general that arbitrary matrix multiplication, but it is all we need, and its simplicity is very helpful.

Nevertheless, the definition takes quite a bit of work (we need to define multiplication of Mat s_1 s_2 and an Mat s_2 s_3, for all cases of s_1, s_2 and s_3, in all, 8 different cases). The above equation takes care of the case when s_1 s_2 and s_3 are all bin of something. To take care of the remaining cases, we should consider vector–vector multiplication (two cases, depending on whether we are multiplying a row vector by a column vector or a column vector by a row vector), vector–matrix multiplication, matrix–vector multiplication, scalar–vector multiplication, vector–scalar multiplication, and finally scalar–scalar multiplication. All of which are different, but all can be derived from the above equation, if we allow the submatrices to have 0 as a dimension, for example, vector–matrix multiplication is given by

$$\begin{pmatrix} u & v \end{pmatrix} \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} uA + vC & uB + vD \end{pmatrix},$$

and column vector–row vector multiplication (the outer product) is given by

$$\begin{pmatrix}u\\v\end{pmatrix}\begin{pmatrix}u' & v'\end{pmatrix} = \begin{pmatrix}uu' & uv'\end{pmatrix} \tag{8}$$

fix these references

That is, we want The way to do this in a way that works well with Agda is by using

when getting to Tri, comment on the fact that Tri probably needs another constructor if we're doing things JP's way (does it have that in his papers / code?)

smart Constructors

5.4 Correctness Proof

Here we prove the correctness of Valiant's Algorithm.

6 Discussion

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).

Todo list

References

DUMMY. *DUMMY.*