Report

Thomas Bååth Sjöblom

May 15, 2013

Contents

THOMAS: Go through TODOList :)

1 Introduction

1.1 What?

In this thesis, we use the programming language Agda to formalize enough matrix algebra to formally prove the correctness of Valiant's algorithm for computing the transitive closure of a matrix.

We begin by giving a short introduction to Agda, where we use it to define and prove the correctness of a maximum function on lists. Then we introduce the algebra relevant for parsing (some parts of which are fairly non-standard, in particular, we need to consider non-associative multiplication). We give definitions of algebraic structures both as "mathematical" definitions (as seen in an algebra textbook) and as Agda definitions to display the similarity between Agda syntax and ordinary mathematics, and to provide the base for later chapters. The main algebraic structures we discuss are commutative monoids and nonassociative semirings. We also define matrices over the nonassociative semirings. Next, we give a short introduction to Parsing, mainly to relate it to the algebra we have just presented, and show that Parsing is equivalent to computing the transitive closure of an upper triangular matrix. In the final part of our thesis, we first present Valiant's algorithm and implement it in Agda, and then we combine the algebra with the ideas in the parsing section to prove the correctness of the algorithm.

1.2 Why?

Finally, we use Agda because it

The thesis is meant to be usable as an introduction to proving things in Agda for people not familiar with the programming language, but it is helpful to have some previous experience with either functional programming or abstract algebra.

2 Agda

Agda is a dependently typed functional language based on Martin-Löf type theory **?**.

The current version of Agda, Agda 2, was implemented by Ulf Norell as a part of his PhD thesis **?**.

In this section, we give a short introduction to using Agda to write programs and proofs.

THOMAS: Emacs mode?

2.1 Programming in Agda

That Agda is a functional programming language means that programs consist of a sequence of definitions of datatypes and functions. One of the simplest datatypes we can define is the type Bool of truth values, consisting of the elements True and False. In Agda, we define it like this:

Notes by JP: 1. Every binding can be given a name. (important?)

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

There are a couple of things to note about the definition:

- The word **data** states that we are defining a new datatype. The list of constructors follow the word **where**.
- Following **data**, we give the name of the new type, Bool.
- Everything has a type, and we generally need to provide the types (as opposed to in Haskell, where it is usually possible for the compiler can infer them) and statements of the form a : b mean that a is an element of type b. In this case, True and False are elements of type Bool, while Bool is an element of type Set, the type of small types (which itself is an element of Set_1, which is an element of Set_2, and so on).

Additionally, the spacing in the above example is important. Agda allows identifiers to be almost any sequence of unicode symbols ,excluding spaces and parenthesis (but including unicode characters like $_1$ in Set_1), so in particular, we need a spaces between Bool and : and Set, because Bool:Set is a valid identifier. In the same spirit, there are no rules specifying that some identifiers need to begin with upper or lower case letters (compared to Haskell's requirement that constructors and types begin with an upper case letter, while variables begin with a lower case letter. We could define a (different, but isomorphic) type bool:

```
data bool : Set where
  true : bool
  False : bool
```

Note that different types can have constructors with the same name (like False for Bool and bool), but this can lead to some hard to understand error messages from the type checker.

As an example of a function definition, we define a function not that takes a Bool and returns the other one:

```
not : Bool → Bool
not True = False
not False = True
```

The first line is the type definition. not has type Bool → Bool (function from Bool to Bool). Next, we define not, and this is done by pattern matching. not True = False states that not applied to True is False, and similarily with the last line. As in Haskell, function application is written without parenthesis: f x means f applied to x, and associates to the left: g x y means (g x) y, where g : X → Y → Z (the arrow → associates to the right, X → Y → Z means X → (Y → Z), so that g x : Y → Z).

Agda is a total programming language, which means that the type checker makes sure that a program never crashes, and every function terminates. In particular, the following definitions are not legal Agda code. First,

```
not' : Bool → Bool
not' True = False
```

is illegal, since if not' is applied to False, there is no case, and the program would crash. This is fairly easy to control, the type checker just needs to make sure that all available constructors appear in the definition. Second,

```
not'' : Bool → Bool
not'' x = not'' x
```

is illegal since evaluating not'' x would reduce it to not'' x, which then has to be evaluated, creating an infinite loop. This is more difficult to control, since it is well known that there is no program that can determine if an arbitrary program eventually terminates or not (the Halting problem **?**). Agda sidesteps this problem by ensuring that when a function is recursive, it can only call itself on arguments that are structurally smaller than the input.

In particualar, the fact that every Agda function terminates implies that not every computable (mathematical) function can be computed with Agda.

2.1.1 Proving in Agda

In this section, we will define a function that takes a list of natural numbers and returns the maximum. Then we are going to state two properties we expect the function to have:

- The return value should be in the list.
- The return value should be greater than every element in the list.

Finally, we are going to prove that these properties hold. All of this will be done in Agda code, which means that the Agda type checker guarantees that our proofs are correct!

The first reason for doing this is to continue our introduction to Agda, by defining more complicated functions and poining out additional features of the

language (and in particular proving things with it). The second reason is that a proof in Agda can require quite a bit of boilerplate code, and hence, in later sections, we only include parts of them, and we feel that there should be a complete proof written in Agda somewhere in this thesis.

2.1.2 Some datatypes and functions

First, we need to define the datatype of natural numbers, which we denote by the unicode character ℕ).

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

This datatype has two constructors, zero and suc. suc is a function taking a natural number as input and returning what is to be thought of as the successor of the number. For example, we can define

```
one : ℕ
one = suc zero

five : ℕ
five = suc (suc (suc (suc (suc zero))))
```

This is a fairly cumbersome way of writing numbers, and it is possible to make Agda supports more standard notation for ℕ, so that we may write:

```
two : ℕ
two = 2
```

Next, we define lists,

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

We have chosen the notation to be similar to the Haskell notation for lists. The (a : Set) before the colon means the type of lists depends on (is indexed by) an arbitrary (small) type a. The underscore denotes where the argument is placed, so [a] is a list of elements from a, [ℕ] a list of natural numbers, et cetera. In the same way, _::_ is a function of two arguments, the first of type a (written in the place of the first underscore), the second of type [a] (written in place of the second underscore). The line **infixr** 8 _::_ explains that the infix operator _::_ associates to the right (the r) and the 8 defines how tightly it binds (to determine whether an expression including other operators needs parentheses or not). In the remainder we omit the **infix** declarations from this text, and use them only to give operations their expected bindings (for example, multiplication binds tighter than addition).

We give an example of a list of natural numbers:

ALL: lhs2tex spacing in lists

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

Next, we define some functions on $\mathbb{N}$ and $[\,\mathbb{N}\,]$. In particular, to define the maximum over a list of natural numbers, we need to be able to find the maximum of a pair of natural numbers. We define this as follows:

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

Here, we need to pattern match on both variables. The first variable is either zero or suc m, for some m. In the first case, we know that the maximum is the second argument. In the second case, we must pattern match on the second variable. If it is zero, we are again done. If it is suc n for some n, we recursively find max m n (note that max is called with two arguments, both of which contain one fewer applications of suc, so the recursion will terminate, eventually), and increase it.

Next, we want to define a function maxL that returns the maximum of a list. We decide to only define maxL on nonempty lists. It could be argued that maxL [ ] = 0 is sensible, but if we were to generalize the definition to an arbitrary total order on an arbitrary type, in general, there is no least element (just consider the integers $\mathbb{Z}$. We still want to use the same datatype, of lists though (we could assume that we have built a large library that depends on them).

To force the list to be non-empty, we want maxL to take two arguments, the first of which is a list xs, and the second of which is a proof that the length of xs is greater than 0. Writing the function length for $\mathbb{N}$ should be easy by now, so we decide to write it for arbitrary types a:

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

Here, the {a : Set} means that a is an implicit argument to the function (that Agda can infer by looking at the type of x in this case). The fact that a is given a name and appears in the types following it means that the types depend on the value of a, and this, in part is what it means to be a dependently typed language (more generally, the fact that we can give any function argument a name and have the types following it contain it), this is an example of a dependent function space. We write (a : Set) if we wanted a to be an explicit named argument. It is also possible to define multiple elements, say a, b, c, of the same type A by writing {a b c : A} or (a b c : A).

Implicit arguments are provided in curly brackets, so we can define a length-function for $\mathbb{N}$ by partial application:

```
lengthℕ : [ ℕ ] → ℕ
lengthℕ = length {ℕ}
```

In the next section, we discuss the second part of what we need to give the type of maxL, that is, a way to define proofs in Agda.

2.1.3 The Curry–Howard Correspondence

To consider proofs and propositions in Agda, and to allow functions to depend on them and their existence, we make use of the Curry–Howard correspondence (for a longer, more detailed introduction to the Curry–Howard correspondence with Agda, see for example **?**). It states that a proposition P can be seen as the type containing all proofs of P. To prove P, then means to give an element of the type corresponding to P (i.e., a proof of P).

To give an example of viewing propositions as types, we take a look at the proposition $m \leq n$. In Agda, we make the following definition:

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

Here we note that we placed the types of the arguments to _⩽_ on the right hand side of the :. This is because we are defining a lot of types at the same time, the types m ⩽ n, for every m n : ℕ. We see this from the fact that the two constructors produce elements of different types, zero ⩽ n and suc m ⩽ suc n, respectively.

If we have an element of type m ⩽ n, it is either constructed by z⩽n, which means that m is zero, so that the proposition $m \leq n$ is true. Otherwise, it was constructed by s⩽s, and we must have m = suc m', n = suc n' for some m', n' and an element of type m' ⩽ n'. But then, the proposition $m' \leq n'$ is true, and hence, again $m \leq n$ is true. So providing an element of type m ⩽ n means providing a proof that m ⩽ n, and it seems like identifying propositions and (some) types makes sense.

We now present the basic logical operations (as interpreted in constructive logic) that are done on propositions to generate new propositions, and their implementations in Agda, using syntax similar to the one used in logic, through the Curry–Howard correspondence.

To define a conjunction between two propositions P and Q, we use the pair, defined as

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

This coincides with the logical notion of a conjunction, which requires a proof of both conjuncts, because as seen above, to construct an element of P ∧ Q, one needs an element of each of P and Q.

For disjunction, we use a disjoint sum:

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

The two constructors mean that we can construct an element of P ∨ Q we need either an element of P or of Q.

For implication, one simply uses functions, P → Q, because implication in constructive logic means a method for converting a proof of P to a proof of Q, and this is exactly what a function is.

The last of the predicate logic operations 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
```

This can thus be seen as a proposition with no proof, which is exactly what falsity is. We thus define negation by

```
¬ : Set → Set
¬ P = P → ⊥
```

For convenience, we also define the true proposition ⊤, as a set with one constructor

```
data ⊤ : Set where
  tt : ⊤
```

To prove this proposition we simply use the element tt.

MOVE THIS!!

To reason about the maxL function, we are going to want to do different things depending on if 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. These are said to be *decidable*, and are propositions for which 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   → Dec P
  no  : ¬ 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 **??**, since, given two natural numbers, we can determine which is smaller by repeatedly subtracting 1 from both until one is zero.

expand above section (the Dec section) a bit

Next we move on to define the quantifications (universal and existential) in predicate logic.

For universal quantification, we again use functions, but this time, dependent functions: If P is a predicate on X (a function that takes elements of X to propositions $P(x)$), the proposition $\forall x.P(x)$ corresponds to the type $(\mathsf{x} : \mathsf{X}) \to \mathsf{P}\ \mathsf{x}$, since to give a function of that time would mean providing a way to construct an element of $\mathsf{P}\ \mathsf{x}$ (that is, a proof of $P(x)$) for every $\mathsf{x} : \mathsf{X}$, which is what $\forall x.P(x)$ means. Agda includes the syntax $\forall\ \mathsf{x}$ for $(\mathsf{x} : \_)$ in type definitions (where the underscore indicates that the type should be infered), so that $\forall\ \mathsf{x} \to \mathsf{P}\ \mathsf{x}$ means exactly what we expect it to mean.

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 a witness x_0 along with a proof of $P(x_0)$. So like for conjunction, we use a pair. But this time, the second element of the pair should depend on the first:

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

2.1.4 More datatypes and functions

We now go back to defining the maxL function. First, for convenience, we define a strictly less than relation:

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

We do not need to create a new datatype using **data** for this since we can use the fact that $\mathsf{m} < \mathsf{n}$ should be equivalent to $\mathsf{suc\ m} \leqslant \mathsf{n}$. In fact, with this definition, Agda will replace any occurence of $\mathsf{m} < \mathsf{n}$ by $\mathsf{suc\ m} \leqslant \mathsf{n}$ internally, which helps us when we write proofs.

Now, we can define the type of the maxL function. First, we give the type:

```
maxL : (xs : [ ℕ ]) → (0 < length xs) → ℕ
```

That is, maxL takes a list of natural numbers xs and a proof that the length of xs is greater than zero and returns the maximum of the list. To define the function, we pattern match on the first argument:

```
maxL [ ] ()
maxL (x :: [ ]) _ = x
maxL (x :: (x′ :: xs)) _ = max x (maxL (x′ :: xs) (s⩽s z⩽n))
```

On the first line, we use the absurd pattern () to denote the empty case resulting from pattern matching on the proof (there are no cases when pattern matchin on an element of $1 \leqslant 0$, and () is used to denote this, since Agda does not allow us to just leave out a case). 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 write _, which takes the place of the variable but doesn't allow it to be used in the definition to signify that it's not important).

We also need an indexing function (to specify that maxL xs _ is in the list), and again, we only define it for sensible inputs (nonempty lists). The simplest definition would probably be:

```
index : ∀ {a} → (xs : [ a ]) → (n : ℕ) → (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
```

Where we needed the proof in the last line, to call the index function recursively.

ALL: revised to about here

However, the above definition leads to a bit of trouble later on when we want to specify things about maxL. 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 n was less than the length of the list. One attempt to do this is with the proposition (n≤xs: n ≤ length xs) → index xs n n≤xs≡ maxL xs 0 <xs|, but this is wrong, since this proposition 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 ≤ 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 ℕ:

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

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.

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

For our purposes, this very strong concept of equality is suitable (it is the smallest relation satisfying $x \equiv x$). 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]

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 ⩽ maxL xs pf)
    ∧
  ∃ (λ n → xs !! n ≡ maxL 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 2.2 we prove the first part of the disjunct, and in Section 2.3, we prove the second.

2.2 First part of proof

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

```
max-greatest : {xs : [ ℕ ]} → {pf : 0 < length xs} →
  (n : Fin (length xs)) → xs !! n ⩽ maxL 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 ⩽ maxL 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 ⩽ maxL xs pf when giving the type of max-spec, because while the type of max-greatest is the proposition that maxL 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 ⩽ maxL (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 ⩽ maxL (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, maxL (x :: (x′ :: xs)) pf expands to max x (maxL (x′ :: xs) pf′), and for any x and y, we have y ⩽ max 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:

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

Where the subscript 2 refers to the fact that it is the second argument of max 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 max-increasing₂, 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 max-increasing₂. 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 max, we know that max 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 max-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′ ⩽ max m′ n′, so we use s⩽s to get suc n′ ⩽ suc (max m′ n′), and from the definition of max, we (and more importantly Agda) get that suc (max m′ n′) == max (suc m′) (suc n′), so we are in fact done.

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

We also prove the similar proposition, max-increasing$_1$: {m n : ℕ} → m ⩽ max m n, that max 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 max-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 ⩽ maxL (x :: (x′ :: xs)) pf, and expanding this using the definition of max, we find that we need to prove that x ⩽ max x (maxL (x′ :: xs) pf′), which is exactly what max-increasing$_1$ does.

```
max-greatest-initial {x} {[ ]}  = ⩽-refl
max-greatest-initial {x} {x′ :: xs}  = max-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 ⩽ maxL (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 ⩽ maxL (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) ⩽ maxL (x′ :: xs) pf, and additionally, from the definition of max,

```
maxL (x :: (x′ :: xs)) pf == maxℕ x (maxL (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) (max-increasing₂ {x})
```

2.3 Second part of proof

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

```
max-in-list : {xs : [ ℕ ]} → {pf : 0 < length xs} →
  ∃ (λ n → xs !! n ≡ maxL 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: 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 2.4.

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.

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

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?

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

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 ⩽ maxL (x′ :: xs) (s⩽s z⩽n) → ∃ (λ i → (x′ :: xs) !! i ≡ maxL (x′
```

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≡maxx0 : {x : ℕ} → x ≡ max x 0
x≡maxx0 {zero} = refl
x≡maxx0 {suc n} = refl

l″ : ∀ {x y} → y ⩽ x → x ≡ max x y
l″ {x} {zero} z⩽n = x≡maxx0
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-maxL : (x : ℕ) (xs : [ ℕ ]) (pf : 0 < length xs) → maxL xs pf ⩽ x → x ≡ maxL (x :: xs) (s⩽s z⩽n)
x-maxL x [ ] pf pf′ = refl
x-maxL x (x′ :: xs) (s⩽s z⩽n) pf′ = l″ pf′

  -- yes case
max-is-max : (x y : ℕ) → x ⩽ y → y ≡ max x y
max-is-max zero y pf = refl
max-is-max (suc m) zero ()
max-is-max (suc m) (suc n) (s⩽s m⩽n) = ≡-cong suc (max-is-max m n m⩽n)

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

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

min-finder : (x : ℕ) → (xs : [ ℕ ]) → ∃ (λ i → (x :: xs) !! i ≡ maxL (x :: xs) (s⩽s z⩽n))
min-finder x [ ] = fzero, refl
min-finder x (x′ :: xs) with x ⩽? maxL (x′ :: xs) _
min-finder x (x′ :: xs) | yes x⩽maxL = move-right x⩽maxL (min-finder x′ xs)
min-finder x (x′ :: xs) | no max⩽x = fzero, x-maxL 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
```

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 implementa-

2.4 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 (only visible in source)

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.

```
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).

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

```
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 _∼_
```

In the remainder of the report, we use the slightly more general IsEquivalence definition from the Agda Standard Library, because it lets us use the standard library modules for equational reasoning.

ALL: is it possible to make underlines less ugly?

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).

THOMAS: Should we give these things names, or use them anyway? (i.e., OP2 vs A \to A \to A

Definition 3.3. A binary operation $\cdot$ on a set X is associative if $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ for all x, y, $z \in X$.

In Agda code, the proposition that _•_ is associative, with respect to a given equivalence relation _≈_ is given by the type:

```
∀ x y z → (x • (y • z)) ≈ ((x • y) • z)
```

Most familiar basic mathematical operations, like addition and multiplication of numbers, are associative. On the other hand, operations like subtraction and division are not, since $x - (y - z) = x - y + z \neq (x - y) - z$, but this is because they are in some sense the combination of addition and inversion: $x - y = x + (-y)$.

In this thesis, we are very interested in a non-associative operation related to parsing.

Example 3.1. Informally, we can define a multiplication on sequences of words, where ?

THOMAS: write, maybe

Definition 3.4. A binary operation $\cdot$ on a set X is commutative if $x \cdot y = y \cdot x$ for all x, $y \in X$.

In Agda, this proposition becomes the type

```
∀ x y → (x • y) ≈ (y • x)
```

Again, many familiar basic mathematical operations are commutative, like addition and multiplication of numbers, are commutative, but matrix multiplication for matrices of size $n \times n$, $n \geq 2$ is an example of an operation that is not commutative.

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 $x \cdot (y + z) = x \cdot y + x \cdot z$, 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 X *distributes over* a binary operation $+$ if, for all x, y, $z \in X$,

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

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

In Agda, given binary operations `_+_` and `_•_`, we define the proposition that `_•_` distributes over `_+_`, with respect to a given equivalence relations `_≈_` as the conjunction:

```
(∀ x y z → (x • (y + z)) ≈ ((x • y) + (x • z)))
∧
(∀ x y z → ((y + z) • x) ≈ ((y • x) + (z • x)))
```

We quantify over `x`, `y` and `z` in both conjuncts, to make our code compatible with the Agda Standard Library, and because the two conjuncts make sense as individual propositions, it might be the case that some operation `_•_` only distributes from the left, for example. Additionally, the parenthesis are needed to guarantee the scoping of ∀.

The second such interaction axiom we will consider comes from the fact that 0 absorbs when involved in a multiplication of numbers: $0 \cdot x = x \cdot 0 = 0$. In particular, if we have an operation $+$ to which we have an inverse and a neutral element 0, we automatically get $0 \cdot x = 0$:

THOMAS: Should zero be here, it only considers one operation

$$0 \cdot x = (0 + 0) \cdot x = 0 \cdot x + 0 \cdot x,$$

where the first equality follows from the fact that 0 is a neutral element for $+$, and the second from that $\cdot$ distributes over $+$. We can then cancel $0 \cdot x$ on both sides to get $0 = 0 \cdot x$.

If we don't have inverses, we can't perform the final step (for example, if $+$ would happen to be idempotent: $x + x = x$ for all $x \in X$, like set union $\cup$, we could not conclude that $0 = 0 \cdot x$), and then, it makes sense to have the following as an axiom:

Definition 3.6. An element $0 \in X$ is a *zero element* (also known as an *absorbing element*) of a binary operation $\cdot$ if $0 \cdot x = x \cdot 0 = 0$ for every $x \in X$.

In Agda, we again give the proposition that 0 is a zero element as a conjunction:

```
(∀ x → (0 • x) ≈ 0)
∧
(∀ x → (x • 0) ≈ 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 3.7. A group is a set G together with a binary operation $\cdot$ on G, satisfying the following:

- $\cdot$ is associative, that is, for all x, y, $z \in G$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.
- 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. The set G is sometimes called the *carrier set* of the group. This is the name used in the Agda Standard Library for all algebraic structures.

Remark. One usually refers to a group $(G, \cdot, e)$ simply by the name of the set G, in particular when the operation considered is fairly obvious.

Definition 3.8. An group $(G, \cdot, e)$ is *Abelian* if the binary operation $\cdot$ is commutative.

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

Example 3.3. The non-zero rational numbers $\mathbf{Q}\backslash 0$, non-zero real numbers $\mathbf{R}\backslash 0$, and non-zero complex numbers $\mathbf{C} \setminus 0$, 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 _≈_
assoc         : ∀ x y z → ((x • y) • z) ≈ (x • (y • z))
•-cong        : ∀ {x x' y y'} → x ≈ x' → y ≈ y' → (x • y) ≈ (x' • y')
identity      : (∀ x → (e • x) ≈ x) ∧ (∀ x → (x • e) ≈ x)
inverse       : (∀ x → (x ⁻¹ • x) ≈ e) ∧ (∀ x → (x • (x ⁻¹)) ≈ e)
⁻¹-cong       : ∀ {x x'} → x ≈ x' → x ⁻¹ ≈ x' ⁻¹
```

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
```

Where we have both defined the various elements required in a group, along with the axioms they need to satisfy (which are hidden in IsGroup). To be able to use the group for reasoning, we open the "module" IsGroup:

```
open IsGroup isGroup public
```

This way, if we open the record Group, we have immediate access to the corresponding record IsGroup.

If we are to prove that something (a collection of an equivalence relation, an operation and a function) is a group, we prove IsGroup. Additionally,

3.2.1 Cayley Table / Examples / Finite groups

We are mostly concerned with finite structures (although not groups) in this thesis, and these can be generated by a multiplication table (also known as a cayley table. We here give two examples of finite groups, to examplify the equational reasoning for algebraic structures.

3.3 Monoids and related structures

Here, we are going to discuss algebraic structures that are slightly more general than groups. These are important for programming, because it is rare that a datatype satisfies all the axioms of a group. First, we define monoids, which are a generalization of groups, and then we define magmas, which are an even further generalization.

3.3.1 Definition

Like a group, a monoid is a set with a binary operation, that satisfies some axioms. The difference is that now, we don't require that there is an inverse for each element.

Definition 3.9. A monoid is a set M, together with a binary operation $\cdot$ on M, that satisfies the following:

- $\cdot$ is associative, that is, for all x, y, $z \in M$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.
- There is an element $e \in M$ such that $e \cdot x = x \cdot e = x$ for every $x \in M$. This element is the *neutral element* of M.

In Agda, we again define a record datatype for the proposition IsMonoid:

```
record IsMonoid {M : Set} (_≈_ : M → M → Set) (_•_ : M → M → M) (e : M) : Set where
  field
    isEquivalence : IsEquivalence _≈_
    assoc : ∀ x y z → ((x • y) • z) ≈ (x • (y • z))
    identity : (∀ x → (e • x) ≈ x) ∧ (∀ x → (x • e) ≈ x)
    •-cong : ∀ {x x' y y'} → x ≈ x' → y ≈ y' → (x • y) ≈ (x' • y')
```

THOMAS: Remove CARRIER

THOMAS: order of axioms in record

When we define our monoids, we add a derived property, setoid, which lets us use monoids in equational reasoning, see Section **??**.

```
record Monoid : Set₁ where
  infix 7 _•_
  infix 4 _≈_
  field
    M         : Set
    _≈_       : M → M → Set
    _•_       : M → M → M
    e         : M
    isMonoid  : IsMonoid _≈_ _•_ e
  open IsMonoid isMonoid public
  setoid : Setoid _ _
  setoid = record {isEquivalence = isEquivalence}
```

THOMAS: what are they called (derived property??)?

THOMAS: do we do this with groups too?, or if not, say so!

An example of a monoid (that are not groups) used in mathematics is the natural numbers, with dot as either multiplication or addition. Another is sets

(M is a collection of sets, dot is union). An example from computer science is lists (M is a collection of lists, dot is list concatenation), which are very common in functional programming.

THOMAS: where should this be

We also define commutative monoids, that is, monoids where $a \cdot b = b \cdot a$ for every $a, b \in M$. We begin with the proposition that something is a commutative monoid:

```
record IsCommutativeMonoid {A : Set} (_≈_ : A → A → Set) (_•_ : A → A → A) (e : A) : Set where
  field
    isMonoid : IsMonoid _≈_ _•_ e
    comm     : ∀ x y → (x • y) ≈ (y • x)
  open IsMonoid isMonoid public
```

In the above definition, we take a slightly different approach than the Agda Standard Library in that we require the user to provide a proof that the operations form a monoid, which in turn requires a proof that e is an identity element of _•_:

```
(∀ x → (e • x) ≈ x) ∧ (∀ x → (x • e) ≈ x)
```

while thanks to commutativity, it would be enough to prove only one of the conjuncts.

Next we define the datatype of commutative monoids:

```
record CommutativeMonoid : Set₁ where
  infixl 7 _•_
  infix 4 _≈_
  field
    M     : Set
    _≈_   : M → M → Set
    _•_   : M → M → M
    e     : M
    isCommutativeMonoid : IsCommutativeMonoid _≈_ _•_ e
  open IsCommutativeMonoid isCommutativeMonoid public
  monoid : Monoid
  monoid = record {isMonoid = isMonoid}
  open Monoid monoid public using (setoid)
```

Commutative monoids are one of the datatypes we will use the most, in particular, we will want to prove equalities among members of a commutative monoid. To allow easy access to the equational reasoning module from the standard library, we import the EqReasoning-module and give it an the alias EqR:

```
import Relation.Binary.EqReasoning as EqR
```

Then we make the following definition:

THOMAS: GLOBAL: make sure all alignment is good

```
module CM-Reasoning (cm : CommutativeMonoid) where
  open CommutativeMonoid cm
  open EqR setoid
```

To prove a simple lemma, we then do the following . For more complicated lemmas, or when there is a large number of them, there are automated approaches to proving equalities in commutative monoids in Agda, but they were not used for this thesis.

THOMAS: either find a lemma to prove here, or refer forward

3.4 Ring-like structures

3.4.1 Definitions

THOMAS: perhaps good note in discussion

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 3.10. A set R together with two binary operations $+$ and $*$ (called addition and multiplication) forms a *ring* if:

- It is an Abelian group with respect to $+$.
- It is a monoid with respect to $*$.
- $*$ distributes over $+$.

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, say, substring has no parse, then the whole string has no parse). We also do not have a unit element for multiplication (there is no guarantee that there is a string A such that A followed by X has the same parse as X for every string X). 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), as seen above. For this reason, we include as an axiom that zero absorbs.

There are a number of (fairly standard) generalizations of the ring definition, but none matches our requirements. One generalization is the semiring, which has the same axioms, except that addition need not have inverses, and so we add the requirement that zero absorbs:

THOMAS: Cite something

Definition 3.11. A set R together with two binary operations $+$ and $*$ forms a *semiring* if:

- It is a commutative monoid with respect to $+$.
- It is a monoid with respect to $*$.
- $*$ distributes over $+$.

- $0 * x = x * 0 = 0$ for all $x \in R$.

Another generalization is the nonassociative ring, which does away with the requirement that multiplication is associative and that there is an identity element for multiplication.

Definition 3.12. A set R together with two binary operations $+$ and $*$ forms a *non-associative ring* if:

- It is an Abelian group with respect to $+$.
- It is closed under $*$.
- $*$ distributes over $+$.

We take this to mean that the modifier *nonassociative* removes the requirement that the set together with $*$ is a monoidfrom the axioms of the structure. Hence, we make the following definition:

too horrible joke?

Definition 3.13. A set R together with two binary operations $+$ and $*$ forms a *nonassociative semiring* if:

- It is a commutative monoid with respect to $+$.
- $*$ distributes over $+$.
- $0 * x = x * 0 = 0$ for all $x \in R$.

In Agda, we begin by defining the proposition that something is a nonassociative semiring:

```
record IsNonAssociativeNonRing {R : Set} (_≈_ : R → R → Set) (_+_ _*_ : R → R → R) (R0 : R) :
  field
    +-isCommutativeMonoid : IsCommutativeMonoid _≈_ _+_ R0
    *-cong                : ∀ {x x' y y'} → x ≈ x' → y ≈ y' → (x * y) ≈ (x' * y')
    distrib               : (∀ x y z → (x * (y + z)) ≈ ((x * y) + (x * z))) ∧ (∀ x y z → ((y + z) * x) ≈
    zero                  : (∀ x → (R0 * x) ≈ R0) ∧ (∀ x → (x * R0) ≈ R0)
```

Next, we want the ability to use the axioms we get from the fact that `_+_` is a commutative monoid, we do this by openening the IsCommutativeMonoid-record, the **public** keyword opens it to the world when IsNonAssociativeNonRing is opened.

```
open IsCommutativeMonoid +-isCommutativeMonoid public
  renaming (assoc     to +-assoc
           ; •-cong   to +-cong
           ; identity to +-identity
           ; isMonoid to +-isMonoid
           ; comm     to +-comm
           )
```

Then, we want to open all the simpler structures involved, so that we can use the axioms related to the fact that addition is a commutative monoid.

Then, we define the record datatype containing all nonassociative semirings:

```
record NonAssociativeNonRing : Set₁ where
  infix 4  _≈_
  infix 5  _+_
  infix 10 _*_
  field
    R     : Set
    _≈_   : R → R → Set
    _+_   : R → R → R
    _*_   : R → R → R
    R0    : R
    isNonAssociativeNonRing : IsNonAssociativeNonRing _≈_ _+_ _*_ R0
  open IsNonAssociativeNonRing isNonAssociativeNonRing public
```

We want to be able to access the fact that it is a commutative monoid with _+_, so we give this a name and open it:

```
+-commutativeMonoid : CommutativeMonoid
+-commutativeMonoid = record {isCommutativeMonoid = +-isCommutativeMonoid}
open CommutativeMonoid +-commutativeMonoid public using (setoid)
  renaming (monoid to +-monoid)
```

As with the commutative monoids, we will spend a bit of time proving equalities of nonassociative semiringelements, so we define:

```
module NS-Reasoning (ns : NonAssociativeNonRing) where
  open NonAssociativeNonRing ns
  open EqR setoid
```

3.4.2 Matrices

We recall that a matrix is in some sense really just a set of numbers formed in a rectangle, so there is nothing stopping us from defining one over an arbitrary ring, or even over a nonassociative semiring, as opposed to over $\mathbf{R}$ or $\mathbf{C}$. 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, $i \in \mathbf{Z}_m$, $j \in \mathbf{Z}_n$, and hence, after currying define:

Definition 3.14. A *matrix* A over a set R is a function $A : \mathbf{Z}_m \to \mathbf{Z}_n \to R$.

When talking about matrices from a mathematical point of view, we will write A_{ij} for $A\,i\,j$

Fix formatting of 0#

In Agda, we will only define the type of a matrix over the nonassociative semiring. For simplicity, and to allow us to avoid adding the nonassociative semiringas an argument to every functino and proposition, we decide to parametrize the module the definition of the matrix by a nanring. Assuming that we have placed the definition of the nonassociative semiringin `NANRing.agda`, we import it:

```
open import NANRing.agda
```

Then, if the current module is placed in `Matrix.lagda`, we write:

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

The part (NAR : NonAssociativeNonRing) is an argument to the module matrix.

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≈_; zero to R-zero
```

If we didn't open the record, instead of, for example a R+ b for adding a to 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 → R
```

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 nonassociative semiringNAR: 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

```
infix 4 _M≈_
_M≈_ : {m n : ℕ} → Matrix m n → Matrix m n → Set
A M≈ B = ∀ i j → A i j R≈ B i j
```

We end by defining the zero matrix. It should be a matrix whose elements are all equal to the zero element of the nonassociative semiring.

```
zeroMatrix : {m n : ℕ} → Matrix m n
zeroMatrix i j = R0
```

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

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

and

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

To define the addition in Agda is straightforward:

```
infix 5 _M+_
_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:

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

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 nonassociative semiringelements:

note about difference between it and Vec?

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

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

```
_•_ : {n : ℕ} → Vector n → Vector n → R
_•_ {zero} u v = R0
_•_ {suc n} u v = (head u R* head v) R+ (tail u • tail v)
  where head : {n : ℕ} → Vector (suc n) → R
        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 (in Agda, we can't use AB or A B for matrix multiplication):

```
infixl 7 _M*_
_M*_ : {m n p : ℕ} → Matrix m n → Matrix n p → Matrix m p
(A M* 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 two propositions:

Proposition 3.15. *If R is a a ring (nonassociative semiring), then the matrices of size $n \times n$ over R also form a ring (nonassociative semiring). Actually, matrices over R of arbitrary size form an Abelian group (commutative monoid) under matrix addition. In both cases, the zero matrix plays the role of the neutral element.*

The proof is fairly easy but boring. We provide part of the proof when R, is particualar, we prove that addition is commutative, and that multiplication is ?. is a nonassociative semiringin Agda (because it is the case we will make use of later). We begin to prove that they form a commutative monoid. This is done by giving an element of the type

THOMAS: WHAT

```
IsCommutativeMonoid (_M≈_ {m} {n})
```

We give this type the name M+-isCommutativeMonoid, and expand the record datatype:

```
M+-isCommutativeMonoid : ∀ {m n} → IsCommutativeMonoid (_M≈_ {m} {n}) _M+_ zeroMatrix
```

```
M+-isCommutativeMonoid = record {isMonoid = {!!}; comm = {!!}}
```

We prove only part of the comm-axiom, the proofs involved in proving isMonoid are similar. We begin by writing the type:

```
M+-comm : ∀ {m n} → (x y : Matrix m n) → x M+ y M≈ y M+ x
```

```
M+-comm x y = {!!}
```

Then, we remember the definitions of _M+_ and _M≈_, and note that they are both pointwise operations, in fact, Agda sees this as well, if we position the cursor in the hole above and press `C-c C-,`, Agda tells us that the goal is of type:

```
Goal: (i : Fin .m) (j : Fin .n) →
  NonAssociativeNonRing._≈_ NAR
  (NonAssociativeNonRing._+_ NAR (x i j) (y i j))
  (NonAssociativeNonRing._+_ NAR (y i j) (x i j))
```

which after cleaning up is:

```
Goal: (i : Fin m) (j : Fin n) →
      (x i j) R+ (y i j) R≈ (y i j) R+ (x i j))
```

Hence, we should provide function that takes `i : Fin m` and `j : Fin n` to a proof that the ring elements `x i j` and `y i j` commute. But this is exactly what `R-comm` is.

```
M+-comm x y = λ i j → R+-comm (x i j) (y i j)
```

The proof is an element of the type

```
M-isNonAssociativeNonRing : ∀ {n} → IsNonAssociativeNonRing (_M≈_ {n} {n}) (_M+_) _M*_ zeroM
M-isNonAssociativeNonRing = record {
  +-isCommutativeMonoid = M+-isCommutativeMonoid;
  *-cong = {!!};
  distrib = {!!};
  zero = {!!}}
```

DO PROOF AND WRITE STUFF!

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

Since we are only interested in upper triangular matrices, we will sometimes refer to them as just *triangular* matrices.

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 nonassociative semiringto show that triangular matrices also form a nonassociative semiring(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.

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

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 definition. probably not

```
_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:

```
_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 !}}
```

4 Parsing

maybe time to switch to standard library things?

Parsing is the process of annoting a sequence of tokens from some alphabet with with structural properties of a language the sequence comes from. 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 no Agda

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

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.

4.1 Definitions

The goal of parsing is to first decide if a given sequence 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* for the language, which contains rules that can either be used to build sequences belonging to the language, or to try assign structural properties to a sequences of tokens.

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

- N is a finite set of nonterminals.
- Σ, is a finite set of terminals, with $N \cap \Sigma = \emptyset$.
- P, is a finite set of production rules, written as $\alpha \to \beta$.
- $S \in N$ is the start symbol.

We use upper case letters to denote nonterminals, lower case letters to denote terminals and Greek letters to denote sequences of both terminals and nonterminals.

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

Example 4.1. We present a simple example grammar for a language of arithmetic expressions:

-

Definition 4.2. A grammar is *context free* if the left hand side of every production rule is a single nonterminal: $A \to \omega$.

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 4.3: 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 4.3, 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 Transitive Closure

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

When we consider the

4.3 Specification of non-associative transitive closure

In Section **??**, we showed that parsing can be considered as computing the (non-associative) transitive closure of a matrix. To get an algorithm from this, we want to create a specification for the problem of finding the (non-associative) transitive closure of a matrix. As before, we let C^+ denote the transitive closure of C. In Valiant [1975], the specification is given as

THOMAS: check if valiant has a word for this, also if he uses ∞, or not

$$C^+ = \sum_{i=1}^{\infty} C^{(i)} \tag{2}$$

where $C^{(n)}$ is defined recursively by:

$$C^{(1)} = C, \tag{3}$$

$$C^{(n)} = \sum_{i=1}^{n} (C^{(i)})(C^{(n-i)}). \tag{4}$$

Hence, $C^{(n)}$ is the sum of all possible bracketings of products containing n copies of C (this is the nth Catalan number, $(2n)!/(n!(n+1)!)$ **?**). The idea behind the specification, and the justification that it specifies the transitive closure of C, is that $c_{ij} \geq x$ if there is ? an element belongs in the matrix if it .

We note that it is enough to only consider the sum from 1 to n, since the matrix is upper triangular and hence $C^{(m)} = \mathbf{0}$ when $m > n$.

However, working with this specification is complicated, and seems to require considering individual matrix elements, and how an element was formed in previous steps (for example, when Valiant proves the correctness of his algorithm, he looks at an arbitrary matrix element and its bracketing Valiant [1975]).

This specification is not easy to work with in Agda for a number of reasons:

- The sum (2) is finite and doesn't make sense.
- The other source of recursion: (**??**) is complicated.
- The proof by Valiant would probably be hard to adapt since it includes even more consepts, moving away from our algebraic structure view—considers bracketings of elements.
- Other reasons?

A big problem with the above specification along with Valiant's proof using it, is that it was too concerned by syntactical matters (how a particular element of the transitive closure was built with regards to where the parentheses were placed), which we want to abstract away by considering algebraic structures (which, once a product is formed, destroy the information as to how it was formed). . So we move towards a more semantical specification of the problem of computing the transitive closure.

The simple fact we use is that if an element belongs to the transitive closure C^+, then it either belonged to C, or it must also belong to C^+C^+.This gives us the following specification:

$$C^+ = C^+C^+ + C \tag{5}$$

We note that this specification doesn't explicitly mention the non-associativity of multiplication. If we expand the equation once (by replacing C^+ on the right hand side by $C^+C^+ + C$), we get

$$C^+ = (C^+C^+ + C)(C^+C^+ + C) + C = (C^+C^+)(C^+C^+) + (C^+C^+)C + C(C^+C^+) + CC + C \tag{6}$$

THOMAS: Continue — mention reduction when associative, looks good, because it looks lika a calculation

THOMAS: Check valiant — and expand on bracketing stuff

JPPJ: Deep-/Shallow embedding

Expand / prove

THOMAS: words to use to refer to the elements of the matrices

and if we repeat this again, we get

$$C^+ = ((C^+C^+{+}C)(C^+C^+{+}C))((C^+C^+{+}C)(C^+C^+{+}C))+((C^+C^+{+}C)(C^+C^+{+}C))C+C((C^+C^+{+}C)(C^+C \tag{7}$$

We prove that the two specifications are equivalent (2) and (**??**):

THOMAS: Expand, maybe, or remove equation

Theorem 4.4. *The transitive closure C^+ of C satisfies*

$$C^+ = C^+C^+ + C \tag{8}$$

if and only if it satisfies

THOMAS: Need upper triang here?

$$C^+ = \sum_{i=1}^{\infty} C^{(i)} \tag{9}$$

Proof. We prove this by showing by induction on the index i in the sum (9) that if C^+ satisfies (8), then the products of i copies of C contain all possible bracketings of the factors. For $i = 1$, this is obvious since the terms resulting from C^+C^+ contain at least two Cs. If it is true for $i < k$, then for $i = k$, □

THOMAS: finish

This specification turns out to work very well with Agda (once we define appropriate concrete datatypes for the matrices—which will be different from the abstract ones given in Section **??**—we will do this in Section **??**).

To end this section, we note that other possible specifications of the transitive closure, similar to (**??**), that are equivalent to it if we assume associativity (and that addition is idempotent) fail to be correct without associativity, one such is:

$$C^+ = C^+C + C, \tag{10}$$

and adding the extra term CC^+, to get

$$C^+ = C^+C + CC^+ + C \tag{11}$$

fails to make the specification correct, since when expanding them, these two only ever produce bracketings of the form $(\cdots(CC)\cdots)C$ (and $C(\cdots(CC)\cdots)$.

fix references to other sections

5 Valiant's Algorithm

In his paper Valiant [1975], 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 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 **?**.

THOMAS: - or –

5.1 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:

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

THOMAS: remove the 2×2 stuff!

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

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

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

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:

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

since $\mathbf{0}$ 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:

$$U^+ = U^+U^+ + U \tag{12}$$
$$R^* = U^+R^* + R^*L^+ + R \tag{13}$$
$$L^+ = L^+L^+ + L, \tag{14}$$

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 (13).

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.

5.1.1 The overlap case

To do this, we need to separate four cases, depending on the dimensions of R. We will give a recursive function, for computing R^* from R, U^+ and L^+, that we will call Overlap(U^+, R, L^+) , because when used for parsing, .

First, if R is a 1×1 matrix, in which case, we must have that U^+ and L^+ are also 1×1 matrices, and since they are upper triangular, they are both equal to the zero matrix. Hence, by (13), $R^* = R$, so we define

$$\text{Overlap}((0), (r), (0)) = r. \tag{15}$$

THOMAS: come up with good reason for name (overlap). Also, it should be U^+ everywhere!

Second, if R is a $1 \times n$ matrix, with $n > 1$, we must have that U^+ is a 1×1 matrix (the zero matrix), and L^+ is a $n \times n$ upper triangular matrix. Then, the rectangular specification (13) gives us the rectangular specification for a row vector:

$$R^* = R^* L^+ + R \tag{16}$$

. We can subdivide R along the middle into two nonempty matrices: $R = (u, v)$, where u is an $1 \times i$ vector, v and $1 \times j$ vector, $i + j = n$, and in the same way, split L^+ into four blocks

$$L^+ = \begin{pmatrix} L_U^+ & L_R^* \\ 0 & L_L^+ \end{pmatrix},$$

where L_U^+ is an $i \times i$ upper triangular matrix and L_L^+ is a $j \times j$ upper triangular matrix. Inserting this in the specification for R^*, we get

$$R^* = (u^*, v^*) \begin{pmatrix} L_U^+ & L_R^* \\ 0 & L_L^+ \end{pmatrix} + (u, v) = (u^* L_U^+ + u, rcvL_L^+ + u^* L_U^* + v),$$

so that

$$\begin{aligned} u^* &= u^* L_U^+ + u \\ v^* &= u^* L_U^* + v^* L_L^+ + v, \end{aligned}$$

that is, u^* satisfies the rectangular specification for a row vector, (16), with $R = u$, $L^+ = L_U^+$, and v^* satisifes the specification with $R = u^* L_U^* + v$, $L_U^+ = L_L^+$

$$\text{Overlap}\left((0),(u,v),\begin{pmatrix} L_U^+ & L_R^* \\ 0 & L_L^+ \end{pmatrix}\right) = (\text{Overlap}((0),u,L_U^+),\text{Overlap}((0),u^*L_U^*+v,L_L^+)), \tag{17}$$

and we can compute u^* first, to use in the second computation, to avoid repeating anything.

Next, if R is a $n \times 1$ matrix, as in the case above, we can subdivide R into $(u,v)^T$ and U^+ into

$$U^+ = \begin{pmatrix} U_U^+ & U_R^* \\ 0 & U_L^+ \end{pmatrix},$$

and since $L = 0$, the rectangular specification turns into

$$R^* = U^+ R^* + R, \tag{18}$$

which gives us

$$\begin{pmatrix} u^* \\ v^* \end{pmatrix} = \begin{pmatrix} U_U^+ & U_R^* \\ 0 & U_L^+ \end{pmatrix} \begin{pmatrix} u^* \\ v^* \end{pmatrix} + \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} U_U^+ u^* + U_R^* v^* + u \\ U_L^+ v^* + v \end{pmatrix},$$

so u^* and v^* satisfy the rectangular specification for a column vector (18) with $R = U_R^* v^* + u$, $U = U_U^+$ and $R = v$, $U = U_L^+$, respectively. Hence, we have

$$\text{Overlap}\left(\begin{pmatrix} U_U^+ & U_R^* \\ 0 & U_L^+ \end{pmatrix}, \begin{pmatrix} u \\ v \end{pmatrix}, (0)\right) = \begin{pmatrix} \text{Overlap}(U_U^+, U_R^* v^* + u, (0)) \\ \text{Overlap}(U_L^+, v, (0)) \end{pmatrix}, \tag{19}$$

where, we can compute v^* first, so as not to repeat anything.

Finally, if R is an $m \times n$ matrix, with $m > 1$, $n > 1$, we can subdivide R along both rows and colums, into four (nonempty) blocks:

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

and subdivide U^+ and L^+ along the same rows and columns into three parts:

$$U^+ = \begin{pmatrix} U_U^+ & U_R^* \\ 0 & U_L^+ \end{pmatrix},$$

$$L = \begin{pmatrix} L_U^+ & L_R^* \\ 0 & L_L^+ \end{pmatrix}$$

where the numbering of the parts are selected so as to make the final specifications symmetric.

Inserting this in the specification for R, (13), gives us,

$$\begin{pmatrix} A^* & B^* \\ C^* & D^* \end{pmatrix} = \begin{pmatrix} U_U^+ A^* + U_L^+ C^+ & U_U^+ B^* + U_R^* D^+ \\ U_L^+ C^* & U_L^+ D^* \end{pmatrix} + \begin{pmatrix} A^* L_U^+ & A^* L_R^* + B^* L_L^+ \\ C^* L_U^+ & C^* L_R^* + D^* L_L^+ \end{pmatrix} + \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

where we have again written A^+, B^+, C^+ and D^+, for the parts of R^* corresponding to A, B, C and D, which a priori are not the transitive closures of anything, while U_U^+, U_L^+, L_U^+ and L_L^+ *are* the transitive closures of U_U, U_L, L_U and L_L, respectively, and U_R^* and L_R^* satisfy the rectangular specification for U_R and L_R, respectively, since we assume that we have computed the transitive closures of U and L recursively. Hence, after rearranging, we get the equations

THOMAS: maybe give better names to parts

$$\begin{aligned}
A^* &= U_U^+ A^* + A^* L_U^+ + U_L^+ C^* + A \\
B^* &= U_U^+ B^* + B^* L_L^+ + U_R^* D^* + A^* L_R^* + B \\
C^* &= U_L^+ C^* + C^* L_U^+ + C \\
D^* &= U_L^+ D^* + D^* L_L^+ + C^* L_R^* + D.
\end{aligned}$$

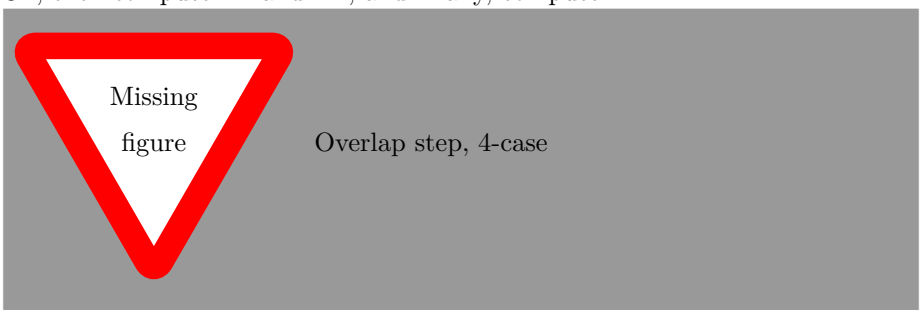
When looking at them carefully and comparing them to the rectangular specification (13), $R^* = U^+ R^* + R^* L^+ + R$, we see that, for example, A^* satisfies the rectangular specification with $U = U_U$, $L = L_U$, $R = U_L^+ C^*$.

THOMAS: update to use tc and rc commands

Hence, we can finish the definition of Overlap.

$$\text{Overlap}\left(\begin{pmatrix} U_U^+ & U_R^* \\ 0 & U_L^+ \end{pmatrix}, \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \begin{pmatrix} L_U^+ & L_R^* \\ 0 & L_L^+ \end{pmatrix}\right) = \begin{pmatrix} \text{Overlap}(U_U^+, U_L^+ C^* + A, L_U^+) & \text{Overlap}(U_U^+, U_R^* D^* + A^* L \\ \text{Overlap}(U_L^+, C, L_U^+) & \text{Overlap}(U_L^+, C^* L_R^* + \end{pmatrix} \tag{20}$$

where again, we note that there is an order to compute the parts that avoids repeating work (and there are no mutual dependencies), namely first compute C^*, then compute A^* and D^*, and finally, compute B^*.



5.1.2 The Algorithm (or Valiant's Algorithm, if we rename the subsection)

Summing up, and going back to the original matrix for the parse chart, C, and dividing it into parts in two steps, we get

$$C = \begin{pmatrix} U_U & U_R & A & B \\ 0 & U_L & C & D \\ 0 & 0 & L_U & L_R \\ 0 & 0 & 0 & L_L \end{pmatrix}. \tag{21}$$

We can then compute the transitive closure of C by the following algorithm, which is (roughly) the algorithm introduced by Valiant Valiant [1975]:

1. Recursively compute the transitive closures U^+ of U and L^+ of L.

2. Compute $U^* = \text{Overlap}(U, R, L)$ by recursively computing $C^* = \text{Overlap}(U_L, C, L_U)$, then $A^* = \text{Overlap}(U_U^+, U_L^+ C^* + A, L_U^+)$ and $D^* = \text{Overlap}(U_L^+, C^* L_R^* + D, L_L^+)$, and finally $B^* = \text{Overlap}(U_U^+, U_R^* D^* + A^* L_R^* + B, L_L^+)$ and putting them together into
$$U^* = \begin{pmatrix} A^* & B^* \\ C^* & D^* \end{pmatrix}.$$

5.2 Implementation

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

5.2.1 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:

THOMAS: make note about us using matrix for Mat here.

$$\text{quad}(A, B, C, D) = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, \tag{22}$$

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:

$$\text{quad}(A, B, e_{0\,m}, e_{0\,n}) = \begin{pmatrix} A & B \\ e_{0\,m} & e_{0\,n} \end{pmatrix} = \begin{pmatrix} A & B \end{pmatrix}, \tag{23}$$

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:

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

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.

THOMAS: 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

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

```
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)
```

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.

Finally, we define Tri′ as

```
data Tri′ : ℕ → Set where
  one : Tri′ 1
  two : {m n : ℕ} → (U : Tri′ m) → (R : Mat′ m n) → (L : Tri′ n) → Tri′ (m + n)
```

While the above 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 .

THOMAS: 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

THOMAS: look for paper maybe – mentioned by JP at some point in time

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

change to bin in actual code

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:

THOMAS: maybe rename to |∘ |

```
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. Mim-

THOMAS: data type or datatype?

icking 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 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 in Section 5.1. 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 Section 5.1.1 can be done:

THOMAS: Check for number of constructors in JP's Tri definition

```
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 5.2.1), and correspondingly, empty Splittings, we might have needed an extra constructor for triangles also .

THOMAS: check if there should be a reference back here

Later, we are going to prove that Tri s is a nonassociative semiringfor any s, and that Vec s and Mat s_1 s_2 are commutative monoids (under addition). For this, we need to define their zero elements, that is a zero Vec, a zero Mat and a zero Tri. Even if we decided against doing this, we would need to define the zero elements somewhere, since we need multiplication to define our specification of the transitive closure, and multiplying a Tri one by a Mat one s should result in a Mat one s that is zero everywhere.

THOMAS: think, is this true???

We define them by pattern matching on the implicit splittings:

```
zeroVec : {s : Splitting} → Vec s
zeroVec {one} = one R0
```

```
zeroVec {bin s₁ s₂} = two zeroVec zeroVec

zeroMat : {s₁ s₂ : Splitting} → Mat s₁ s₂
zeroMat {one} {one} = sing R0
zeroMat {one} {bin s₁ s₂} = rVec zeroVec
zeroMat {bin s₁ s₂} {one} = cVec zeroVec
zeroMat {bin s₁ s₂} {bin s₁' s₂'} = quad zeroMat zeroMat zeroMat zeroMat

zeroTri : {s : Splitting} → Tri s
zeroTri {one} = one
zeroTri {bin s₁ s₂} = two zeroTri zeroMat zeroTri
```

5.2.2 Operations on our datatypes

Now, we will define operations, addition, multiplication and equality, for our datatypes, Vec, Mat and Tri. We only need the operations for Tri, but, for example, to multiply two Tris, we need to be able to multiply the rectangular parts with triangles, and to do this, in turn, we need to be able to multiply two matrices, which requires the ability to multiply vectors.

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+_ : {s₁ s₂ : Splitting} → Mat s₁ s₂ → Mat s₁ s₂ → Mat s₁ s₂
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')
```

The overall structure used above when defining addition for the different datatypes is fairly typical of what needs to be done when defining something that is essentially a lifting of an operation (as it is for the abstract matrix **??**).

For multiplication, which is not simply a lifting, 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:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} A' & B' \\ C' & D' \end{pmatrix} = \begin{pmatrix} AA' + BC' & AB' + BD' \\ CA' + DC' & CB' + DD' \end{pmatrix} \tag{24}$$

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

$$\begin{pmatrix} u \\ v \end{pmatrix} (u', v') = \begin{pmatrix} uu' & uv' \\ vu' & vv' \end{pmatrix} \tag{25}$$

We now begin defining these multiplications in Agda. There is some dependency between them, for example, to define outer product, we need both kinds of scalar–vector multiplication (although we don't need anything to define the dot product). We hence begin with the simplest kinds of multiplication, first scalar–vector mutliplication:

```
_sv*_ : {s : Splitting} → Carrier → Vec s → Vec s
x sv* one x' = one (x R* x')
x sv* two u v = two (x sv* u) (x sv* v)
```

and then vector–scalar multiplication:

```
_vs*_ : {s : Splitting} → Vec s → Carrier → Vec s
one x vs* x' = one (x R* x')
two u v vs* x = two (u vs* x) (v vs* x)
```

Then we move on to the dot product:

```
_•_ : {s : Splitting} → Vec s → Vec s → Carrier
one x • one x' = x R* x'
two u v • two u' v' = u • u' R+ v • v'
```

next, we move on to scalar–matrix and matrix–scalar multiplication (the definition of which we leave out since it is essentially the same as scalar–matrix multiplication):

```
_sm*_ : {s₁ s₂ : Splitting} → Carrier → Mat s₁ s₂ → Mat s₁ s₂
x sm* sing x' = sing (x R* x')
x sm* rVec v = rVec (x sv* v)
x sm* cVec v = cVec (x sv* v)
x sm* quad A B C D = quad (x sm* A) (x sm* B) (x sm* C) (x sm* D)
_ms*_ : {s₁ s₂ : Splitting} → Mat s₁ s₂ → Carrier → Mat s₁ s₂
```

Next we define the outer product:

```
_⊗_ : {s₁ s₂ : Splitting} → Vec s₁ → Vec s₂ → Mat s₁ s₂
one x ⊗ one x' = sing (x R* x')
one x ⊗ two u v = rVec (two (x sv* u) (x sv* v))
two u v ⊗ one x = cVec (two (u vs* x) (v vs* x))
two u v ⊗ two u' v' = quad (u ⊗ u') (u ⊗ v') (v ⊗ u') (v ⊗ v')
```

and note that we could have defined the multiplications by a single element vector, using sv* on one level higher:

```
one x ⊗ two u v = rVec (x sv* (two u v))
```

but either way, we need to pattern match on the vector to tell if it is one x' or two u v, since we need to use different constructors for the matrix (and using a smart multiplication function doesn't help much, since we need to do the same thing when proving things anyway). Next, we give the types of vector–matrix and matrix–vector multiplication (but leave out the implementation):

```
_vm*_ : {s₁ s₂ : Splitting} → Vec s₁ → Mat s₁ s₂ → Vec s₂
_mv*_ : {s₁ s₂ : Splitting} → Mat s₁ s₂ → Vec s₂ → Vec s₁
```

Finally, we can define matrix multiplication:

```
_m*_ : {s₁ s₂ s₃ : Splitting} → Mat s₁ s₂ → Mat s₂ s₃ → Mat s₁ s₃
sing x m* sing x' = sing (x R* x')
sing x m* rVec v = rVec (x sv* v)
rVec v m* cVec v' = sing (v • v')
rVec (two u v) m* quad A B C D = rVec (two (u vm* A v+ v vm* C) (u vm* B v+ v vm* D))
cVec v m* sing x = cVec (v vs* x)
cVec v m* rVec v' = v ⊗ v'   --
quad A B C D m* cVec (two u v) = cVec (two (A mv* u v+ B mv* v) (C mv* u v+ D mv* v))
quad A B C D m* quad A' B' C' D' = quad (A m* A' m+ B m* C') (A m* B' m+ B m* D') (C m* A' m+ D
```

To define triangle multiplication is a lot simpler, since we only need to consider one index. However, we need matrix multiplication in its full generality, because in general, the Splitting involved is not a balanced binary tree, and hence, the row splitting and the column splitting differs (the case two U R L doesn't require that U and L have the same splitting, since the constructor bin takes two splittings).

We also need to define multiplication between Vec and Tri and between Mat and Tri, all of which are straight-forward to define:

```
_vt*_ : {s : Splitting} → Vec s → Tri s → Vec s
_tv*_ : {s : Splitting} → Tri s → Vec s → Vec s
_mt*_ : {s₁ s₂ : Splitting} → Mat s₁ s₂ → Tri s₂ → Mat s₁ s₂
_tm*_ : {s₁ s₂ : Splitting} → Tri s₁ → Mat s₁ s₂ → Mat s₁ s₂
```

Using these, we can define triangle–triangle multiplication:

```
_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') (U tm* R' m+ R mt* L') (L t* L')
```

The final part needed to express the transitive closure specification in Agda is a concept of equality among triangles (and for this, we need equality for matrices and vectors, as before). In all cases, we want to lift the nonassociative semiringequality to the datatype in question. As before (see section 5.2.2), equality takes two objects of a datatype to a proposition (a member of Set). We begin with equality among Vec, where two one element vectors are equal if their only elements are equal, and vectors that are made up of two parts are equal if both parts are equal (as vectors):

```
_v≈_ : {s : Splitting} → Vec s → Vec s → Set
one x v≈ one x' = x R≈ x'
two u v v≈ two u' v' = (u v≈ u') ∧ (v v≈ v')
```

Note that this (and the remaining equality definitions only apply to vectors with the same splitting) so vectors which contain the same elements can be unequal.

THOMAS: have I not written this already somewhere

We move on to equality for Mat:

```
_m≈_ : {s₁ s₂ : Splitting} → Mat s₁ s₂ → Mat s₁ s₂ → Set
Valiant.MatAndTri.sing x m≈ Valiant.MatAndTri.sing x' = x R≈ x'
Valiant.MatAndTri.rVec v m≈ Valiant.MatAndTri.rVec v' = v v≈ v'
Valiant.MatAndTri.cVec v m≈ Valiant.MatAndTri.cVec v' = v v≈ v'
Valiant.MatAndTri.quad A B C D m≈ Valiant.MatAndTri.quad A' B' C' D' = A m≈ A' ∧ B m≈ B' ∧ C m≈
```

And to finish this section, equality for Tri:

```
_t≈_ : {s : Splitting} → Tri s → Tri s → Set
Valiant.MatAndTri.one t≈ Valiant.MatAndTri.one = ⊤
Valiant.MatAndTri.two U R L t≈ Valiant.MatAndTri.two U' R' L' = U t≈ U' ∧ R m≈ R' ∧ L t≈ L'
```

THOMAS: Make the actual code have these defs for equality (as opposed to the **data** definitions it now uses

5.2.3 Proof that they are NANRings

We will now prove that Vec, Mat and Tri are commutative monoids with _v+_, _m+_ and _t+_, and Tri is a nonassociative semiringwith _t+_ and _t*_ as defined above. One big reason for doing this is that it will make it possible, and easier, to reason about equations containing elements from the different datatypes. We can use something similar to the equational reasoning used in Section **??**, with the added help of the axioms of a commutative monoid or a nonassociative semiring. That is, we will prove

```
Vec-isCommutativeMonoid : {s : Splitting} → IsCommutativeMonoid _v≈_ _v+_ (zeroVec {s})
Mat-isCommutativeMonoid : {s₁ s₂ : Splitting} → IsCommutativeMonoid _m≈_ _m+_ (zeroMat {s₁} {s
Tri-isCommutativeMonoid : {s : Splitting} → IsCommutativeMonoid _t≈_ _t+_ (zeroTri {s})
Tri-isNonAssociativeNonRing : {s : Splitting} → IsNonAssociativeNonRing _t≈_ _t+_ _t*_ (zeroTri {s}
```

the reason we include the Splitting in the zero element is that we need to make Agda infer what datatype we are talking about. To prove these things is generally very easy, but requires a lot of code. The interested reader is referred to the full code at **?**.

The approach for proving things about addition involves lifting statements into the ground nonassociative semiring. We examplify by proving the identity^l-lemma for Vec, that is, that zeroVec is the left identity of _v+_:

```
Vec-identityˡ : {s : Splitting} → (x : Vec s) → zeroVec v+ x v≈ x
Vec-identityˡ (one x) = proj₁ R+-identity x
Vec-identityˡ (two u v) = (Vec-identityˡ u), (Vec-identityˡ v)
```

Then, we can define instances of CommutativeMonoid and NonAssociativeNonRing:

```
Vec-CommutativeMonoid : {s : Splitting} → CommutativeMonoid
Vec-CommutativeMonoid {s} = record {isCommutativeMonoid = Vec-isCommutativeMonoid {s}}
Mat-CommutativeMonoid : {s₁ s₂ : Splitting} → CommutativeMonoid
Mat-CommutativeMonoid {s₁} {s₂} = record {isCommutativeMonoid = Mat-isCommutativeMonoid {s₁
Tri-CommutativeMonoid : {s : Splitting} → CommutativeMonoid
Tri-CommutativeMonoid {s} = record {isCommutativeMonoid = Tri-isCommutativeMonoid {s}}
Tri-NonAssociativeNonRing : {s : Splitting} → NonAssociativeNonRing
Tri-NonAssociativeNonRing {s} = record {isNonAssociativeNonRing = Tri-isNonAssociativeNonRing {s}
```

To partly motivate proving that they are (and further, using algebraic structures) algebraic structures, we prove two simple lemmas about CommutativeMonoids. The first is used repeatedly when proving that zeroTri is an annihilating element:

fix the all a b below

```
0′+0″≈0 : (cm : CommutativeMonoid) → let open CommutativeMonoid cm renaming (_•_ to _+_) in
0′+0″≈0 cm = {!!}
  where open CM-Reasoning cm
```

The second is used very frequently when proving that _t*_ distributes over _t+_:

```
rearr : {!!}
rearr = {!!}
```

Proving things about multiplication also involves moving the properties down to the ground nonassociative semiring, but here, the path is longer. We examplify the beginning of this path by giving the proof that zeroTri is a left zero of _t*_, and that _t*_ distributes over _t*_, on the left:

```
t*-zero^l : {s : Splitting} → (x : Tri s) → zeroTri t* x t≈ zeroTri
t*-zero^l one = tt
t*-zero^l {bin s₁ s₂} (two U R L) = t*-zero^l U, 0 ′+0″≈0 Mat-CommutativeMonoid {zeroTri tm* R} {zeroM
```

where

```
tm*-zero^l : {s₁ s₂ : Splitting} → (x : Mat s₁ s₂) → zeroTri tm* x m≈ zeroMat
mt*-zero^r : {s₁ s₂ : Splitting} → (x : Tri s₂) → (zeroMat {s₁} {s₂}) mt* x m≈ zeroMat
```

are the proofs that zeroTri is a left zero of _tm*_, and that zeroMat is a leftzero of _mt*_ (where the concept of a zero is slightly generalized to allow "operations" f : A → B → A or f : A → B → B).

```
t*-distrib^l : {s : Splitting} → (x y z : Tri s) → x t* (y t+ z) t≈ x t* y t+ x t* z
t*-distrib^l = {!!}
```

THOMAS: Find and prove lemmas (rearrangeLemma)

5.2.4 Specification and Proof in Agda

With the above operations, and the fact that they form nonassociative semirings, we are now ready to express the transitive closure problem in Agda. It is a relation between two Tris, that is, a function that takes two Tris, C^+ and C, and returns the propostion that C^+ is the transitive closure of C, which is true if C^+ and C satisfy the specification (**??**), with multiplication, addition and equality replaced by their triangle versions:

ALL: the specification part is very small – either move up to previous, or keep with proof here.

```
_is-tc-of_ : {s : Splitting} → Tri s → Tri s → Set
C⁺ is-tc-of C = C⁺ t≈ C⁺ t* C⁺ t+ C
```

Additionally, for use in our proof, we want to define the various subspecifications we used, from equations (**??**), (**??**) and (**??**), which are relations between a Mat or Vec and a Tri. Each of these concerns a different kind of Tri, and we are able to pattern match to get out the parts used in the abovementioned equations.

THOMAS: chech out how the spacing differs when using different code blocks

```
_is-tcMat-of_ : {s₁ s₂ : Splitting} → Mat s₁ s₂ → Tri (bin s₁ s₂) → Set
R× is-tcMat-of (two U⁺ R L⁺) = R× m≈ U⁺ tm* R× m+ R× mt* L⁺ m+ R
```

THOMAS: byt ut R^* mot $R^\times$ or something

```
_is-tcRow-of_ : {s : Splitting} → Vec s → Tri (bin one s) → Set
(one x) is-tcRow-of two one (sing x') one = x R≈ x'
```

```
v× is-tcRow-of two one (rVec v) L+ = v× v≈ v× vt* L+ v+ v
–v × is-tcRow-of (two one (rVec v) L+) = v× v≈ v× vt* L+ v+ v


_is-tcCol-of_ : {s : Splitting} → Vec s → Tri (bin s one) → Set
one x is-tcCol-of two one (sing x') one = x R≈ x'
v× is-tcCol-of (two U+ (cVec v) one) = v× v≈ U+ tv* v× v+ v
```

Where we restrict the types in the Vec specifications to vector with length at least 2, since this is the only case they need to handle.

Now, we begin to define Valiant's algorithm in Agda, and in the next section, we prove the proposition (valiant C) is-tc-of C. We begin by defining the overlap part, and to do this, we define the overlap part for row vectors and column vectors—as separate functions that work on Vecs, since their recursive calls only affect vectors:

THOMAS: Clear up reason

THOMAS: Fix discussion about overlap case in different section, it is wrong!

```
overlapRow : {s : Splitting} → Vec s → Tri s → Vec s
overlapRow (one x) one = one x
overlapRow (two u v) (two U+ R× L+) = two u× v×
  where u× = overlapRow u U+
        v× = overlapRow (u vm* R× v+ v) L+
overlapCol : {s : Splitting} → Tri s → Vec s → Vec s
overlapCol one (one x) = one x
overlapCol (two U+ R× L+) (two u v) = two u× v×
  where v× = overlapCol L+ v
        u× = overlapCol U+ (R× mv* v× v+ u)
```

Then we define the function the function that calculates the overlap part for arbitrary Mats, as defined in Section **??**. It should take as input a Tri, a Mat, and a Tri, and return a Mat, that should satisfy (overlap U R L) is-tcMat-of (two U R L):

THOMAS: expand overlap step in algorithm (in other section), so it summarizes things

```
overlap : {s₁ s₂ : Splitting} → Tri s₁ → Mat s₁ s₂ → Tri s₂ → Mat s₁ s₂
overlap one (sing x) one = sing x
overlap one (rVec v) L+ = rVec (overlapRow v L+)
overlap U+ (cVec v) one = cVec (overlapCol U+ v)
overlap (two U+ R× L+) (quad A B C D) (two U'+ R'× L'+) = quad A× B× C× D×
  where C× = overlap L+ C U'+
        A× = overlap U+ (A m+ R× m* C×) U'+
        D× = overlap L+ (D m+ C× m* R'×) L'+
        B× = overlap U+ (B m+ R× m* D× m+ A× m* R'×) L'+
```

Where we have used **where** blocks to avoid repeating computations (and avoid repeating the code for them).

Finally, we define Valiant's algorithm:

```
valiant : {s : Splitting} → Tri s → Tri s
valiant Valiant.MatAndTri.one = one
```

```
valiant (Valiant.MatAndTri.two U R L) = two U⁺ (overlap U⁺ R L⁺) L⁺
  where U⁺ = (valiant U)
        L⁺ = (valiant L)
```

we note that the algorithm is fairly straightforward, since some of the complexity of it is hidden in the definitions of matrix multiplication. Now, we are ready to prove the correctness.

5.2.5 Proof

In this section, we are going to prove the correctness of Valiant's algorithm, as defined in the previous section, that is, we will prove that it satisfies the specification we gave above, or, in words, for every splitting s and every upper triangular matrix C : Tri s, valiant C is the transitive closure of C. We begin by giving the proofs of the different propositions, so we can use them in an arbitrary order later. The first is the main proposition:

```
valiant-correctness : {s : Splitting} (C : Tri s) → (valiant C) is-tc-of C
valiant-mat-correctness : {s₁ s₂ : Splitting} (U⁺ : Tri s₁) (R : Mat s₁ s₂) (L⁺ : Tri s₂) → overlap U⁺ R L
rvec : {s : Splitting} → Vec s → Mat one s
rvec {one} (Valiant.MatAndTri.one x) = Valiant.MatAndTri.sing x
rvec {bin s₁ s₂} v = Valiant.MatAndTri.rVec v
cvec : {s : Splitting} → Vec s → Mat s one
cvec {one} (Valiant.MatAndTri.one x) = Valiant.MatAndTri.sing x
cvec {bin s₁ s₂} v = Valiant.MatAndTri.cVec v
valiant-row-correctness : {s : Splitting} (v : Vec s) (L⁺ : Tri s) → overlapRow v L⁺ is-tcRow-of two one (r
  -- s₁ s₂ : Splitting (v : Vec (bin s₁ s₂)) (L⁺ : Tri (bin s₁ s₂)) → overlapRow v L⁺ is-tcRow-of two one
–v aliant-row-correctness' : {s : Splitting} (v : Vec s) (L⁺ : Tri s) → overlapRow v L⁺ is-tcRow-of two one
–v aliant-row-correctness' = {!!}
valiant-col-correctness : {s : Splitting} (U⁺ : Tri s) (v : Vec s) → overlapCol U⁺ v is-tcCol-of two U⁺ (cve


valiant-correctness one = tt
valiant-correctness (two U R L) = valiant-correctness U, valiant-mat-correctness (valiant U) R (valiant L), val
valiant-mat-correctness Valiant.MatAndTri.one (Valiant.MatAndTri.sing x) Valiant.MatAndTri.one = {!!}
valiant-mat-correctness one (rVec v) L = {!valiant-row-correctness v L!}
  -- where open EqR
valiant-mat-correctness (Valiant.MatAndTri.two U R L) (Valiant.MatAndTri.cVec v) Valiant.MatAndTri.one =
valiant-mat-correctness U (Valiant.MatAndTri.quad A B C D) L = {!!}
valiant-row-correctness (Valiant.MatAndTri.one x) Valiant.MatAndTri.one = R-refl
valiant-row-correctness (Valiant.MatAndTri.two u v) (Valiant.MatAndTri.two U R L) = ({!valiant-row-correc
valiant-col-correctness U v = {!!}
```

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

THOMAS: smart Constructors

5.3 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

Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–314, April 1975. ISSN 0022-0000. doi: 10.1016/S0022-0000(75)80046-8. URL `http://dx.doi.org/10.1016/S0022-0000(75)80046-8`.