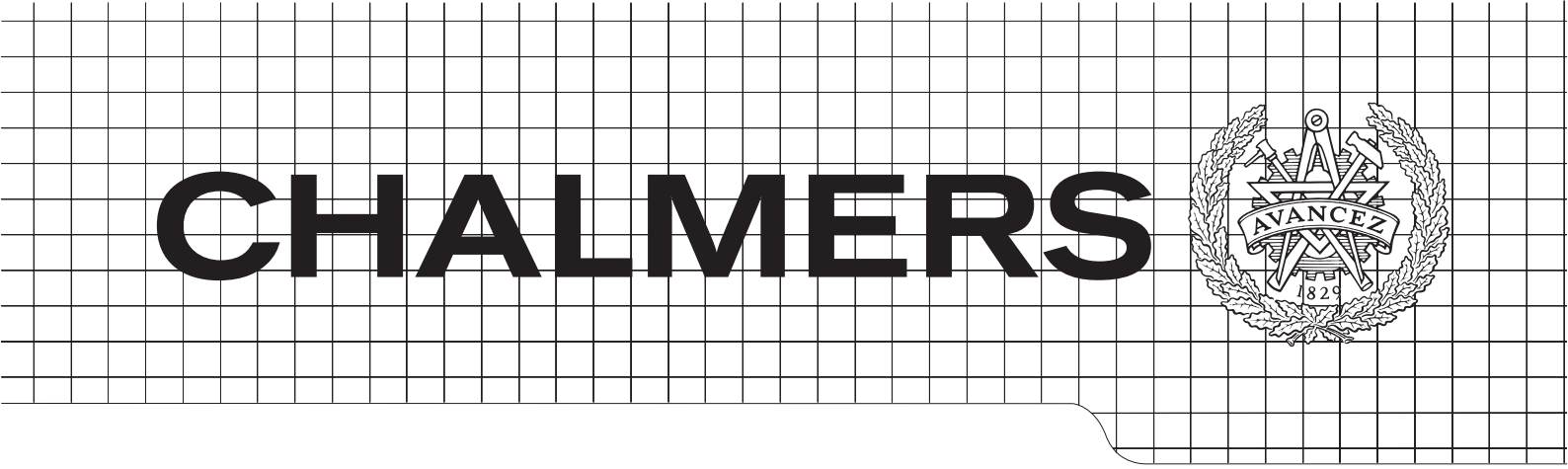


An Agda proof of the correctness of Valiant's algorithm for context free parsing

Master's Thesis in Computer Science – algorithms, languages and logic

Thomas Bååth Sjöblom

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden May 2013

An Agda proof of the correctness of Valiant's algorithm for context free parsing

Thomas Bååth Sjöblom

Examiner: Patrik Jansson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31 772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2013

Abstract

Parsing is an important problem with applications ranging from compilers to bioinformatics. To perform the parsing more quickly, it would be desirable to be able to parse in parallel. Valiant's algorithm [Valiant, 1975] is a divide and conquer algorithm for parsing and can be used to perform a large part of the work in parallel. It is fairly easy to implement in a functional programming languages, using pattern matching. Agda is a dependently typed functional programming language that doubles as a proof assistant and is hence very suitable for implementing and proving the correctness of Valiant's algorithm.

In this thesis, we provide a very natural specification for the parsing problem and prove that it is equivalent to the specification of the transitive closure used in [Valiant, 1975], that is further removed from both parsing and proving. We compare the two specifications and use our specification to derive Valiant's algorithm. We then implement it in Agda and prove our implementation correct (we prove that it satisfies our specification).

We also give short introductions to parsing, programming and proving with Agda and to using algebraic structures in Agda.

Keywords: Dependently typed programming, Formal proof, Agda, Valiant, Parallel parsing, Transitive closure, Nonassociative multiplication, Algebra

Acknowledgements

I would like to thank my supervisor Jean-Phillipe Bernardy and my examiner Patrik Jansson for their support, patience and optimism during the creation of this master's thesis.

I would also like to thank Anders Martinsson for listening to me failing to explain what parsing is on three separate occasions.

Thomas Bååth Sjöblom, Gothenburg, May 29 2013

Contents

Chapter 1

Introduction

Agda is a dependently typed functional programming language based on Martin–Löf type theory that can also be used as a proof assistant. Valiant's algorithm is a highly parallel algorithm for computing the transitive closure of a matrix in the time needed to multiply two matrices. Computing the transitive closure of an upper triangular matrix is a problem that appears when parsing a context free language.

In this thesis, we use Agda to formalise enough matrix algebra to formally prove the correctness of Valiant's algorithm. The full formalisation and proof is made up of around 3500 lines of code, partitioned into around 30 modules, all of which is available at `https://github.com/thobaa/Algebra-of-Parallel-Programming-in-Agda`.

Parsing is used as an early step in a many places where it is necessary to analyse some kind of text. Examples include compilers, where source code is turned into a tree structure that contains information about the properties a segment of code has (is it a statement, an if-expression, etc.) to help with type checking and machine code generation. With the advent of CPUs with multiple cores, it becomes increasingly attractive to try and parse in parallel. Because the functions in a functional programming language lack side effects, functional languages seem suitable for writing parallel programs. Additionally, working in Agda allows us to write an algorithm with syntax similar to Haskell and prove its correctness simultaneously.

The choice of Valiant's algorithm is due to the fact that, although it was initially discovered to prove that parsing can be done in the time needed to perform a matrix multiplication, it also happens to be highly parallelizable. Additionally, it is a simple enough algorithm to make it feasible to write a formal proof of its correctness.

We begin the report with Section 2, which a short introduction to Agda, where we use it to define and prove the correctness of a maximum function on lists. Next, in Section 3, we introduce the algebra relevant for parsing (some parts of which are fairly non-standard, in particular, we need to consider structures with 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. In Section 4, we then 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, Section 5, we first present Valiant's algorithm for computing the transitive closure and implement it in Agda. Then we combine the algebra with the ideas in the parsing section to prove the correctness of the algorithm.

This report 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.

Chapter 2

Agda

Agda is a dependently typed functional language based on Martin-Löf type theory [Martin-Löf, 1984]. The implementation of the current version of Agda, Agda 2, was started by Ulf Norell as a part of his PhD [Norell, 2007]. In this section, we give a short introduction to using Agda to write programs and proofs.

2.1 Introduction

That Agda is a functional programming language means that programs consist of a sequence of definitions of datatypes and functions. We begin with some general introduction to Agda in Section 2.1.1, and then begin with an extended example in 2.1.2, that we keep up for the remainder of Section 2, to motivate the introduction of new concepts.

2.1.1 General introduction

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:

```
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.

Moreover, everything has a type, and we generally need to provide the types as opposed to languages like Haskell or ML, where it is usually possible for the compiler to infer them. 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). The spacing in the above example is important. Agda allows identifiers to be almost any sequence of Unicode symbols, excluding spaces and parentheses (but including Unicode characters like $_1$ in Set_1). Because of this, we need to write spaces in Bool : 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 (as opposed to Haskell's requirement that constructors and types begin with an upper case letter and variables begin with a lower case letter). We could define a (different, but isomorphic) type bool:

```
data bool : Set where    -- this type is not used in this report,
  true  : bool           -- included here only as an example
  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 one Bool and returns the other one:

```
not : Bool → Bool   -- (1)
not True  = False   -- (2)
not False = True    -- (3)
```

Line (1) 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. On Line (2), we state that not applied to True is False, and on Line (3) that not applied to False is True. As in Haskell, function application is written without parentheses: 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 type X → Y → Z means X → (Y → Z) (i.e., the arrow → associates to the right), so that g x : Y → Z.

Agda is a total programming language, which means that every function terminates, and programs never crash. In particular, the following definitions are not legal Agda code. First,

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

is illegal, because if not' would be applied to False the program would crash. This is fairly easy to control: roughly speaking the system just needs to make sure that all available constructors appear in the definition. Second,

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

is illegal because trying to evaluate `not" e` would reduce it to `not" e`, 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 cannot be solved Turing [1936]). Agda sidesteps this problem by using a conservative termination-checker that only accepts a subset of terminating programs. Among other things, it requires that recursive calls are only made of subexpressions of the arguments (see Agda team [2011] for a more detailed discussion of the termination checker).

If we only want to use a function locally, we can define an anonymous function using a λ expression, as in $\lambda\ \mathsf{x} \to \mathsf{not}\ (\mathsf{not}\ \mathsf{x})$. Such functions are accepted whenever a function is needed. We cannot use pattern matching to define anonymous functions.

2.1.2 Starting the extended example

In this section, we will define a function that takes a list of natural numbers and returns its maximum. The return value should be greater than or equal to every element in the list. In Section 2.3.2, we state this property using Agda, and finally, in Section 2.4.3 we are going to prove in Agda that this property hold.

The first reason for doing this is to continue our introduction to Agda, by defining more complicated functions and pointing 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. 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`. The constructor `zero` is an element of ℕ, intended to be interpreted as 0, and the constructor `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 support a more standard notation for elements of ℕ using the pragmas

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

so that we may write definitions like:

```
two : ℕ
two = 2
```

Next, we define lists,

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

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 a parameter, which is 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, etc. 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 last line defines 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, but use them to give operations the bindings we expect (so that, for example, multiplication binds tighter than addition).

We give an example of a list of natural numbers:

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

Next, we define some functions on ℕ and [ℕ]. In particular, to define the maximum of 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 generalise 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 is what it means to be a dependent type. This is an example of a dependent function space. More generally, the fact that we can give any function argument a name and have the types following it depend on it means that Agda is a dependently typed language. We write (a : Set) instead if we want 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). When needed (when Agda cannot infer them), implicit arguments can be provided in curly brackets.

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.

2.2 The Curry–Howard Correspondence

To consider proofs and propositions in Agda, and to allow functions to depend on them and their existence, we use the Curry–Howard correspondence: propositions as types, proofs as programs (for a more detailed introduction to it, see for example Bove and Dybjer [2009]). The Curry–Howard correspondence states that a proposition P can be seen as the type containing all "proof objects", of P (we will refer to them simply as proofs in the remainder). 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 is at most 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 the placement of the ℕs in the first line. They are placed on the right side of the colon because they are indices of _⩽_. This means that we are defining a type

family (consisting of the types m ⩽ n for every m n : ℕ. We can see that we need to do this from the fact that the two constructors produce elements of different types, zero ⩽ n and suc m ⩽ suc n, respectively. We also make note of the names we have given the constructors. In the remainder of this report, we often use the convention that Pxy (without the spaces) is the name of an element of datatype P x y (with spaces), so m⩽n is a proof that m ⩽ n.

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 is at most n" is true. Or it is constructed by s⩽s, and we must have m = suc k, n = suc l for some k, l and an element of type k ⩽ l. But then, the proposition "k is at most than l" is true, and hence, again "m is at most n" is true. So providing an element of type m ⩽ n means providing a proof that m ⩽ n. Intuitively, we see that identifying propositions and types makes sense.

We now present the 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.

2.2.1 Propositional logic

We begin with concepts from propositional logic, and in the next section, we consider predicate logic.

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 to 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 predicate logic operation is negation. Constructively, the negation of a proposition means that the proposition implies falsity. We use the empty type to represent falsity:

```
data ⊥ : Set where
```

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

```
¬ : Set → Set
¬ P = P → ⊥
```

For convenience, we also define the true proposition $\top$, as a set with one constructor

```
data ⊤ : Set where
  tt : ⊤
```

To prove this proposition we simply use the element `tt`.

2.2.2 Predicate logic

Now we move on to define the quantifiers (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 `(x : X) → P x`, since to give a function of that type would mean providing a way to construct an element of `P x` (that is, a proof of $P(x)$) for every `x : X`, which is what $\forall x.P(x)$ means. Agda includes the syntax `∀ x` for `(x : _)` in type definitions (where the underscore indicates that the type should be inferred), so that `∀ x → P 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 $P(x_0)$. Like for conjunction, we use a pair. But this time, the second element of the pair depends on the first:

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

2.2.3 Decidability

Finally we discuss decidable propositions. Constructively, the law of excluded middle—saying that for any proposition P, $P \vee \neg P$ is true—is not valid. There is no algorithm that takes an arbitrary proposition and returns either a proof of it, or a proof that it implies $\bot$. However, there are many propositions for which it is valid. These propositions are said to be *decidable*. In Agda, if `P` is a proposition, we define the proposition that `P` is decidable as `Dec P`:

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

So an element of `Dec P` is a proof that `P` is decidable, since it contains either a proof of `P` or a proof of `¬ P`.

An example of a proposition that is decidable is the proposition that $\mathsf{m} \leqslant \mathsf{n}$, where m and n are natural numbers. To prove that this is decidable for any m and n, we give a function that takes m and n and returns an element of $\mathsf{Dec}\ (\mathsf{m} \leqslant \mathsf{n})$:

```
_≤?_ : (m n : ℕ) → Dec (m ≤ n)
```

We present this function case by case. If m is $\mathsf{0}$, we can construct a proof that $\mathsf{m} \leqslant \mathsf{n}$ with the constructor $\mathsf{z{\leqslant}n}$:

```
0       ≤? n = yes z≤n
```

if m is $\mathsf{suc\ k}$, we pattern match on n. If n is $\mathsf{0}$, there is no proof of $\mathsf{m} \leqslant \mathsf{n}$, since no constructor of $\mathsf{\_{\leqslant}\_}$ constructs an element of type $\mathsf{suc\ k} \leqslant \mathsf{0}$. The fact that there are no such proofs is denoted by $\lambda\ ()$ (we basically write an anonymous function of type $(\mathsf{suc\ k} \leqslant \mathsf{0}) \to \bot$ by pattern matching on the empty type $\mathsf{suc\ k} \leqslant \mathsf{0}$).

```
suc k ≤? 0 = no (λ ())
```

If n is $\mathsf{suc\ l}$, we use a **with** statement to add an extra argument to the function, to pattern match on $\mathsf{Dec}\ (\mathsf{k} \leqslant \mathsf{l})$, which is decidable by induction:

```
suc k ≤? suc l with k ≤? l
suc k ≤? suc l | yes k≤l = yes (s≤s k≤l)
suc k ≤? suc l | no ¬k≤l = no (λ sm≤sn → ¬k≤l (p≤p sm≤sn))
  where p≤p : {m n : ℕ} → suc m ≤ suc n → m ≤ n
        p≤p (s≤s m≤n) = m≤n
```

2.3 Continuing the example

We now go back to the example started in Section 2.1.2.

2.3.1 Defining maxL

First, we define the maxL function.

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 because 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 evaluate any occurrence of $\mathsf{m} < \mathsf{n}$ to $\mathsf{suc\ m} \leqslant \mathsf{n}$ internally, which helps us when we write proofs.

Now, we can define the type of the maxL function:

```
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 matching 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 do not 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 does not allow it to be used in the definition to signify that it is not important).

2.3.2 Indexing function and specification

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]) → (i : ℕ) → (i < length xs) → a
index []       i       ()
index (x :: xs) 0       _          = x
index (x :: xs) (suc i) (s≤s m≤n) = index xs i m≤n
```

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

However, we can shorten the function definition by including the fact that the index is less than the length of the list by using a datatype that combines the index and the proof. This datatype is known as Fin, where Fin n contains the set of all natural numbers strictly less than n. One way to define Fin would be to use a dependent pair, which we define again to give it a syntax for types (as opposed to the "logical" ∃):

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

The order these definitions should be done is, first define Σ, then define A ∧ B = Σ A (λ x → B) and ∃ P = Σ _ P, where the underscore is used to denote the fact that the first argument of Σ can be inferred from the type of the second. Then, we could define Fin as:

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

With Fin n defined we can use the Haskell notation !! for indexing:

```
_!!_ : ∀ {a} → (xs : [a]) → Fin (length xs) → a
[]       !! (i     , ())
(x :: xs) !! (0     , _)          = x
(x :: xs) !! (suc i, s≤s m≤n) = xs !! (i, m≤n)
```

We note, however, that we do not really use the proof part for anything important. This, along with the fact that $\mathbb{N}$ is inductively defined (and the structure of the definition of $\_\leqslant\_$) lets us use an even nicer formulation, where the proof is further embedded into the datatype we use.

Instead, we choose to define Fin as a type family, using the simple fact that if $n = 1+k$, then $0 \leq n$, and if $n = 1 + k$ and $i \leq k$, then $1 + i \leq n$:

```
data Fin : ℕ → Set where
  f0  : {n : ℕ} → Fin (suc n)
  fsuc : {n : ℕ} → Fin n → Fin (suc n)
```

That is, f0 (representing 0, but given a different name for clarity—it is not equal to the natural number 0, they do not even have the same type) is less than any number of the form suc n, and for any number i, less than some number n, fsuc i is less than suc n (we can see this definition as instantiating the second argument of $\_\leqslant\_$ to suc n). As with $\_\leqslant\_$, the $\mathbb{N}$ is on the right hand side of the colon since because we are defining a type family. One disadvantage of the choice of Fin is that we are not dealing with natural numbers, at all. Instead, we have to define functions like

```
toℕ : {n : ℕ} → Fin n → ℕ
toℕ f0     = 0
toℕ (fsuc i) = suc (toℕ i)
```

and

```
fromℕ : (n : ℕ) → Fin (suc n)
fromℕ 0     = f0
fromℕ (suc y) = fsuc (fromℕ y)
```

and prove that they do what we expect (like that toℕ (fromℕ i) equals i).

These two different ways of defining things will be used later when we define upper triangular matrices as a datatype. When we represent matrices abstractly (as functions from their indices) in Section 3.3.3 with the datatype Triangle, we do not have a nice inductive definition of them, so we have to use the pairing of a matrix and a proof that it is upper triangular. In Section 5.2.3, on the other hand, we define the datatype Tri of a concrete representation of upper triangular matrices which have a built in "proof" that the matrix is triangular. Then, we do not need to worry about the proof when defining multiplication, for example. If our definition returns a Tri, then the result is upper triangular.

We now define the indexing function using the inductive family Fin:

```
_!!_ : ∀ {a} → (xs : [a]) → (i : Fin (length xs)) → a
[]       !! ()
(x :: xs) !! f0     = x
(x :: xs) !! fsuc i = xs !! i
```

Now we can finally express our specification in Agda.

```
max-greatest : (xs : [ℕ]) → (pf : 0 < length xs) →
  (i : Fin (length xs)) → xs !! i ⩽ maxL xs pf
```

To prove this property of the maxL function, we must produce an inhabitant of the above type. We do this in the next section.

2.4 Proving the correctness

To prove a proposition in Agda, it is important to look at the structure of the proposition. Then one needs to determine which part of the proposition one should pattern match on. To do this, it is a good idea to have a plan for the proof.

2.4.1 Informal proof

We formulate the proof informally. The main idea we use is pattern matching on the index into the list. If the index is 0, we want to prove the simpler proposition that `x ⩽ maxL (x :: xs) pf`, which we call max-greatest-base, because it is the base case in an induction on the index:

```
max-greatest-base : (x : ℕ) (xs : [ℕ]) → x ⩽ maxL (x :: xs) (s⩽s z⩽n)
```

On the other hand, if the index is $i + 1$, the list has length at least 2, and we proceed by 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 maxL, we get that `maxL (x :: (x' :: xs)) pf` reduces to `max x (maxL (x' :: xs) pf')`, and for any x and y, we should have `y ⩽ max x y`.

2.4.2 Lemmas

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, so there is nothing to prove. However, we still need to prove the second part of Point 3:

```
max-⩽₂ : {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. Then, to combine the three points, we need a way to piece together inequalities, if i ⩽ j and j ⩽ k, then i ⩽ k (i.e., _⩽_ is transitive, see Section 3.1):

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

Now we begin proving these lemmas, starting with ⩽-trans as it does not depend on the others (all the other lemmas will require further sub-lemmas 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 a⩽b, Agda infers that i is suc a, and j is suc b for some a, b, where a⩽b is a proof that a ⩽ b. We then pattern match on the proof of j ⩽ k, which has to be s⩽s b⩽c since j is suc b. Hence, we can use induction to get a proof that a ⩽ c, and apply s⩽s to that proof:

```
⩽-trans (s⩽s a⩽b) (s⩽s b⩽c) = s⩽s (⩽-trans a⩽b b⩽c)
```

and we are done.

We also introduce another lemma about _⩽_: ⩽-refl, stating that for any n, n ⩽ n (i.e., _⩽_ is reflexive, see Section 3.1), which is very easy to prove (if n is 0, a proof is given by the constructor z⩽n, and if n is suc m, by induction, we find a proof of m ⩽ m and s⩽s takes that proof to a proof that n ⩽ n:

```
⩽-refl : {n : ℕ} → n ⩽ n
⩽-refl {0}     = z⩽n
⩽-refl {suc n} = s⩽s ⩽-refl
```

Now we prove max-⩽₂. We pattern matching on the second argument. If it is 0, we use the constructor z⩽n, regardless of what the first argument is:

```
max-⩽₂ {m}     {0}     = z⩽n
```

If it is suc l, we pattern match on the first argument. If it is 0, then, max 0 (suc l) reduces to suc l, so we prove suc l ⩽ suc l using ⩽-refl (we can note that we did not actually need the fact that the second argument was non-zero, since max 0 n reduces to n no matter what n is):

```
max-⩽₂ {0}     {suc l} = ⩽-refl
```

On the other hand if the first argument is suc k, we find a proof of l ⩽ max k l using max-⩽₂—we need to supply the first implicit argument which Agda is unable

to infer—and use s≤s on it to get suc l ≤ suc (max k l), which Agda reduces max (suc k) (suc l) to suc (max k l), and we are done:

```
max-≤₂ {suc k} {suc l} = s≤s (max-≤₂ {k})
```

We also prove the similar proposition, that max is greater than its first argument, in essentially the same way. We pattern match first on the first argument instead, and this time, Agda is able to infer the arguments of max-≤₁ in the induction case, so we leave them out:

```
max-≤₁ : {m n : ℕ} → m ≤ max m n
max-≤₁ {0}     {n}     = z≤n
max-≤₁ {suc k} {0}     = ≤-refl
max-≤₁ {suc k} {suc l} = s≤s max-≤₁
```

2.4.3 Assembling the proof

Using max-≤₁ and ≤-refl, we are able to prove the initial step in the induction proof, max-greatest-base. We pattern match on xs. If it is [], we need to show that x ≤ x, which we do with ≤-refl, again:

```
max-greatest-base x []       = ≤-refl
```

If it is x' :: xs, we need to prove that x ≤ maxL (x :: (x' :: xs)) _. Agda reduces the right hand side to max x (maxL (x' :: xs) _), so we just need to use max-≤₁ does:

```
max-greatest-base x (x' :: xs) = max-≤₁
```

Finally, we finish our proof, 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:

```
max-greatest []       ()      _
```

Otherwise, Fin (length xs) is non-empty, and can pattern match on the index. If the index is f0, we use the initial step max-greatest-base, to prove that x ≤ maxL (x :: xs) pf:

```
max-greatest (x :: xs) (s≤s z≤n) f0     = max-greatest-base x xs
```

If the index is fsuc i, we pattern match on the tail of the list. If it is empty, we know that the index cannot be fsuc i, because we would have i : Fin 0, so we fill in i with the absurd pattern ():

```
max-greatest (x :: []) (s≤s z≤n) (fsuc ())
```

The last case 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) _. By the definition of !!, (x :: (x' :: xs)) !! (fsuc i) reduces to (x' :: xs) !! i. So by induction, max-greatest i proves that (x :: (x' :: xs)) !! (fsuc i) ⩽ maxL (x' :: xs) pf. From the definition of maxL, maxL (x :: (x' :: xs)) _ reduces to max x (maxL (x' :: xs) _). So using max-⩽$_2$, and ⩽-trans to put things together, we finish the proof:

```
max-greatest (x :: (x' :: xs)) (s⩽s z⩽n) (fsuc i) = ⩽-trans
                                                    (max-greatest _ _ i)
                                                    (max-⩽₂ {x})
```

We put the whole proof in Figure 2.1.

2.5 Final remarks about Agda

We end the section about Agda by going over a few parts of Agda that we have not mentioned but will be used in the remainder of the report.

First, Agda has Standard Library [Agda team, 2013] that contains most of the definitions we have made above (sometimes under slightly different names, for example, _∧_ is called _×_, and in more generality—definitions are made to work for all of Set$_1$, Set$_2$, ...). In the remainder of the report, and in our library proving the correctness of Valiant's algorithm, we use the Standard Library definitions whenever possible.

Our second comment is about the structure of Agda programs. Agda code is partitioned into modules, which contain a sequence of function and datatype definitions. Modules can be imported, and an imported module can be opened to bring all definitions into scope in the current module. Additionally, modules can be parametrised by elements of a datatype, which basically means that all functions in the module take an extra argument of that type. To open a parametrised module, an element of the parameter type is needed. We use parametrised modules frequently in this report and in our library, starting in Section 3.3.2.

Finally, there is another way to define a datatype: as a record. A record is similar to a product type, but each field is given a name. This is useful when there is a lot of fields and there is no natural ordering of them. Records behave like small modules, they can contain function definitions, and they can be parametrised and opened, like modules, bringing all their fields and definitions into scope. As an example, we define a record type of a pair:

```
record Pair (A B : Set) : Set where
  field
    fst : A
    snd : B
```

When defining algebraic structures in Section 3, records are very useful for handling the axioms needed, since they have no natural ordering.

```
  -- general lemmas about _⩽_:
⩽-refl : {n : ℕ} → n ⩽ n
⩽-refl  {0}     = z⩽n
⩽-refl  {suc n} = s⩽s ⩽-refl

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

  -- properties of max
max-⩽₁ : {m n : ℕ} → m ⩽ max m n
max-⩽₁ {0}     {n}     = z⩽n
max-⩽₁ {suc k} {0}     = ⩽-refl
max-⩽₁ {suc k} {suc l} = s⩽s max-⩽₁

max-⩽₂ : {m n : ℕ} → n ⩽ max m n
max-⩽₂ {m}     {0}     = z⩽n
max-⩽₂ {0}     {suc l} = ⩽-refl
max-⩽₂ {suc k} {suc l} = s⩽s (max-⩽₂ {k})

  -- base case
max-greatest-base : (x : ℕ) (xs : [ℕ]) → x ⩽ maxL (x :: xs) (s⩽s z⩽n)
max-greatest-base x []       = ⩽-refl
max-greatest-base x (x' :: xs) = max-⩽₁

  -- the proof
max-greatest : (xs : [ℕ]) → (pf : 0 < length xs) →
               (i : Fin (length xs)) → xs !! i ⩽ maxL xs pf
max-greatest []           ()       _
max-greatest (x :: xs)    (s⩽s z⩽n) f0      = max-greatest-base 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-⩽₂ {x})
```

Figure 2.1: Proof that the maxL function finds a maximal element in the list.

Chapter 3

Algebra

In this section, we are going to introduce a number of algebraic definitions. Some (like commutative monoid and nonassociative semiring) will be useful later in the report, while other (like group and ring) are mentioned as possibly familiar examples and for comparison.

In Section 3.1, we introduce a number of well known propositions that show up in the definitions of algebraic structures as axioms and comment on differences between defining an algebraic structure in mathematics and in Agda. Then, in Sections 3.2 and 3.3, we use these properties to define algebraic structures consisting of sets with one and two binary operations, respectively.

3.1 Introductory definitions

When defining an algebraic structure (consisting of just one set), one gives the set of objects, a number of binary operations on the objects and a number of axioms that the set and the operations are required to satisfy. In this section, we are going to introduce common such axioms.

3.1.1 Equivalence relations

The axioms usually refer to equalities between different sequences of operation applications, like the axiom in a group that $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ for all x, y and z. However, to define these things in Agda, we note that we do not have a concept of equality for terms of an arbitrary datatype. Further, the most "basic" equality definition (called propositional equality), stating just that $x = x$ for all x and defined by

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

is often not what we want. For the datatype $\mathbb{N}$, it coincides with the "mathematical" equality of natural numbers, because $\mathbb{N}$ is inductively defined (in particular, if two things

are built in different ways by zero and suc, they cannot be equal). But if we were to define the integers $\mathbb{Z}$ as differences between natural numbers:

```
data ℤ : Set where
  _-_ : ℕ → ℕ → ℤ
```

we want a concept of equality that considers `5 - 3` as equal to `0 - 2`, for example. Also, if we define a datatype of sets, we want sets where the elements have been added in different orders to be equal. Hence, we need to generalise the propositional equality to some kind of relation that behaves well. We cannot choose an arbitrary relation, for example, $<$ does not behave as we expect equality to.

It turns out that there are three properties we want a concept of equality to have. First, it should be reflexive: every element should be equal to itself. Second, it should be symmetric: if a is equal to b, b should be equal to a. Third, it should be transitive: if a is equal to b and b is equal to c, then a should be equal to c. A relation that satisfies this is called an equivalence relation:

Definition 3.1.1. A relation $\sim \subseteq X \times X$ is called an *equivalence relation* if it is

- Reflexive: for $x \in X$, $x \sim x$.
- Symmetric: for x, $y \in X$, if $x \sim y$, then $y \sim x$.
- Transitive: for x, y, $z \in X$, if $x \sim y$ and $y \sim z$, then $x \sim z$.

The following proposition formalises the way it behaves like an equality:

Proposition 3.1.2. *An equivalence relation $\sim$ partitions the elements of a set X into disjoint nonempty equivalence classes (subsets $[x] = \{y \in X \mid y \sim x\}$) satisfying:*

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

This means that if we use an equivalence relation as "equality" on a set, we are saying that two elements are equal if they generate the same equivalence class, so we let actual equality on the equivalence classes give us an "equality" on the members of the set.

To define an equivalence relation in Agda, use the fact that a relation (on a single set X) is an element of type `X → X → Set`. That is, it takes two elements of X and produces the proposition (recall that propositions are types) that the elements are related.

Next, we give the types for the propositions it should satisfy. Given a relation `_∼_`, reflexivity is given by the type

```
Reflexive _∼_ = ∀ {x} → x ∼ x
```

symmetry by the type

```
Symmetric _∼_ = ∀ {x y} → x ∼ y → y ∼ x
```

and transitivity by

```
Transitive _∼_ = ∀ {x y z} → x ∼ y → y ∼ z → x ∼ z
```

That we decide to make the arguments x, y and z implicit is somewhat arbitrary, they can be inferred from the types appearing later, and we follow the definitions from the Standard Library.

Then, we define the record IsEquivalence, for expressing the proposition that a relation is an equivalence relation (we use a record so that we can give names to the three axioms, for use in proofs)

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

In the remainder of the report, we actually use the slightly more general IsEquivalence definition in the Agda Standard Library, because it lets us use the EqReasoning module from the Standard Library for equational reasoning, as exemplified in Section 3.2.2. When a relation is an equivalence relation, we will usually denote it by $\approx$ instead of $\sim$.

3.1.2 Propositions about one operation

Next, we define some propositions that binary operations (i.e., functions $X \to X \to X$) can satisfy. These are properties that ordinary addition and multiplication of numbers satisfy.

Definition 3.1.3. 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, the proposition that _•_ is associative, with respect to a given equivalence relation _≈_ is given by the type:

```
Associative _≈_ _•_ = ∀ x y z → (x • (y • z)) ≈ ((x • y) • z)
```

Many 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 functional programming, a very important example of an associative operation is list concatenation. In this thesis, we are very interested in a non-associative operation related to parsing, defined in Section 4.2

Definition 3.1.4. 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 is given by the type

```
Commutative _≈_ _•_ = ∀ x y → (x • y) ≈ (y • x)
```

Again, many familiar basic mathematical operations are commutative, like addition and multiplication of numbers, but matrix multiplication (for matrices of size $n \times n$, $n \geq 2$) is an example of an operation that is not commutative. In this thesis, we are interested in the commutative operation set union, $\cup$.

Now, we present two properties that relate elements to the operations:

Definition 3.1.5. An element $e \in X$ is an *identity element* of a binary operation $\cdot$ if $x \cdot e = e \cdot x = x$ for all $x \in X$.

In Agda, the type of this proposition is

```
Identity _≈_ e _•_ = (∀ x → (e • x) ≈ x) ∧ (∀ x → (x • e) ≈ x)
```

We quantify over x in both conjuncts to make our code compatible with the Agda Standard Library and because the two conjuncts make sense as individual propositions: an element can be just a left identity or a right identity. It might be the case that some element e is only an identity of _•_ when multiplied on the left, for example. The parentheses in the type are needed to give ∀ the correct scope.

The identity element of addition of numbers is zero, and the identity element of multiplication is one.

Definition 3.1.6. An element $x^{-1} \in X$ is the *inverse* of x with respect to a binary operation $\cdot$ if $x^{-1} \cdot x = x \cdot x^{-1} = e$.

When discussing inverses, it is usually required that every (or nearly every) element has an inverse. In Agda, since the proposition that every element has an inverse is a universal quantification, it is proved by a function _⁻¹ that takes an element to its inverse. Given such a function, the statement that x ⁻¹ is the inverse of x is given by the type

```
Inverse _≈_ _⁻¹ e _•_ = (∀ x → ((x ⁻¹) • x) ≈ e) ∧ (∀ x → (x • (x ⁻¹)) ≈ e)
```

If the operation is addition of numbers, the inverse of x is given by $-x$, and if the operation is multiplication of numbers, the inverse is given by $1/x$. In computer science, inverses occur more rarely. Indeed, none of the algebraic structures we have used to prove the correctness of Valiant's algorithm include inverses.

3.1.3 Propositions about two operations

When we have two different binary operations on the same set, we often want them to interact with each other sensibly. Here we define two such ways of interaction.

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, respectively and generalise it to two arbitrary operations:

Definition 3.1.7. 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 tighter than $+$.

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

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

In the Agda Standard Library, the module containing the function properties is parametrised by an equivalence relation, so the properties do not need the argument _≈_. Instead of Distributive, the property is called _DistributesOver_ in the standard library, giving it a very readable syntax.

The second such interaction we will consider comes from the fact that 0 absorbs when involved in a multiplication of numbers: $0 \cdot x = x \cdot 0 = 0$. The reason we consider this a property of a pair of operations is that if we have an operation $+$ for which we have an identity element 0 and every element has an inverse, and an operation $\cdot$ which distributes over $+$, we automatically get $0 \cdot x = 0$:

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

where the first equality follows from the fact that 0 is an identity 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 do not have inverses, we cannot perform the final step (for example, if $+$ would happen to be idempotent: $x + x = x$ for all $x \in X$, like set union $\cup$ is, we could not conclude that $0 = 0 \cdot x$), and then, it makes sense to have the following as an axiom:

Definition 3.1.8. An element $z \in X$ is a *zero element* (also known as an *absorbing element*) of a binary operation $\cdot$ if $z \cdot x = x \cdot z = z$ for every $x \in X$.

In Agda, we give the proposition that z is a zero element as the conjunction

```
Zero _≈_ z _•_ = (∀ x → (z • x) ≈ z) ∧ (∀ x → (x • z) ≈ z)
```

3.2 Sets with one operation

In this section, we are going to discuss algebraic structures made up of a set and a binary operation on the set.

3.2.1 Monoid-like structures

In mathematics, the most common such structure is the group. Many mathematical objects form groups, including the integers $\mathbf{Z}$, the rational numbers $\mathbf{Q}$, the real numbers $\mathbf{R}$ and the complex numbers $\mathbf{C}$, with addition as the binary operation, and the non-zero rational numbers $\mathbf{Q} \setminus 0$, non-zero real numbers $\mathbf{R} \setminus 0$, and non-zero complex numbers $\mathbf{C} \setminus 0$, with multiplication as the binary operation. Although groups satisfy too many axioms to be useful to us, we give the definition below, to clarify the difference between them and the perhaps less familiar structure, the monoid, that we present next.

Definition 3.2.1. A *group* is a set G together with a binary operation $\cdot$ on G, satisfying the following:

- The operation $\cdot$ is associative: for all x, y, $z \in G$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.
- There is an identity element $e \in G$: for all $x \in G$, $e \cdot x = x \cdot e = x$.
- For every $x \in G$, there is an inverse element x^{-1}: $x \cdot x^{-1} = x^{-1} \cdot x = e$.

A group where the binary operation is commutative is said to be *Abelian*.

We move on to discuss monoids, which are slightly more general than groups in that they do not require the existence of inverses. Monoids are important for programming because it is rare that a datatype satisfies all the axioms of a group.

Example of monoids (that are not also groups) include the natural numbers, with $\cdot$ as either multiplication or addition, sets (M is a collection of sets, $\cdot$ is union and the identity element is the empty set), lists (M is a collection of lists, $\cdot$ is list concatenation and the identity element is the empty list).

Definition 3.2.2. A monoid is a set M, together with a binary operation $\cdot$ on M, that satisfies the following:

- The operation $\cdot$ is associative: x, y, $z \in M$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.
- There is an identity element $e \in M$: for all $x \in M$, $e \cdot x = x \cdot e = x$.

A monoid where the binary operation is commutative is a *commutative monoid*.

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 _≈_
    •-cong  : ∀ {x x' y y'} → x ≈ x' → y ≈ y' → (x • y) ≈ (x' • y')
    assoc   : Associative _≈_ _•_
    identity : Identity _≈_ e _•_
               -- (∀ x → (e • x) ≈ x) ∧ (∀ x → (x • e) ≈ x)
```

We add the line

```
open IsEquivalence isEquivalence public
```

in the record to put the fields from IsEquivalence in scope when the IsMonoid record is opened.

The set M is sometimes called the *carrier* of the monoid (or any other structure), and the Agda Standard Library uses this name.

We note that we need to include the the equality _≈_ along with the fact that it should be an equivalence relation in the definition. We also want a proof that •-cong that the operation and the equality interact nicely, if x and x' are equal and y and y' are equal, then x • y and x' • y' should be equal.

We can then define a record for the type Monoid, containing all monoids. Note that the type of Monoid is Set_1, because like Set itself, Monoid is "too big" to be in Set.

```
record Monoid : Set₁ where
  field
    M         : Set
    _≈_       : M → M → Set
    _•_       : M → M → M
    e         : M
    isMonoid : IsMonoid _≈_ _•_ e
  open IsMonoid isMonoid public
```

We also add a definition setoid to our monoid record, which lets us use monoids in equational reasoning, see Section 3.2.2.

```
setoid : Setoid _ _
setoid = record {isEquivalence = IsMonoid.isEquivalence isMonoid}
```

To prove something is a monoid, we construct an inhabitant of the type IsMonoid. We usually also give the monoid record containing the object a name, to be able to use it in places where an object of type monoid is wanted.

We now define commutative monoids. We begin with the proposition that something is a commutative monoid. The proposition contains a proof that it is a monoid and a proof that the operation is commutative, and we open the IsMonoid record so that the proofs that the structure is a monoid are in scope.

```
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 definition of IsCommutativeMonoid, we are taking a slightly different approach than the Agda Standard Library: 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 because of commutativity, x • e ≈ e • x, so it would be enough to require a proof of just one of the conjuncts.

Next we define the datatype of commutative monoids, and open the records so that all definitions are in scope:

```
record CommutativeMonoid : Set₁ where
  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)
```

3.2.2 Equational Reasoning in Commutative Monoids

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. The Agda Standard Library contains a module EqReasoning that lets us reason about equalities using a very natural syntax. To allow easy access to this module, we we import it, give it the alias EqR:

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

and make the following module definition:

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

So that the module CM-Reasoning takes a commutative monoid as a parameter and brings into scope the commutative monoid axioms and the contents of the EqReasoning module. We often use this module locally, using a **where**-definition.

As an example of equational reasoning, we prove a simple lemma with it, that if two elements in a commutative monoid are equal to the identity element, then so is their

sum. We use a **let** statement in the type to open the record CommutativeMonoid locally and make the type more readable.

```
e'•e''≈e : ∀ cm → let open CommutativeMonoid cm in
  {e' e'' : M} → e' ≈ e → e'' ≈ e → e' • e'' ≈ e
e'•e''≈e cm {e'} {e''} e'≈e e''≈e = begin
  e' • e''
    ≈⟨ •-cong e'≈e e''≈e ⟩
  e • e
    ≈⟨ proj₁ identity e ⟩
  e ■
  where open CM-Reasoning cm
```

On the first line (after begin), we write the expression on the left hand side of the equality we want to prove. Then, we write a sequence of expressions and proofs that the expressions are equal, ending with the expression on the right hand side, followed by ■.

The syntax makes the proof a lot easier to follow, and even more so with longer proofs. Without it, the proof would be written as:

```
e'•e''≈e cm e'≈e e''≈e = trans (•-cong e'≈e e''≈e) (proj₁ identity e)
```

where proj_1 is the projection that takes a pair to its first element.

For more complicated lemmas, or in case of many lemmas, there are automated approaches to proving equalities in commutative monoids in Agda, but they were not used for this thesis, see for example `https://personal.cis.strath.ac.uk/conor.mcbride/pub/Hmm/CMon.agda`.

3.3 Sets with two operations

In this section, we discuss algebraic structures that consist of a set together with two binary operations. As we discussed in Section 3.1.3, we want them to interact sensibly.

3.3.1 Ring-like structures

The basic example of this kind of structure is a ring:

Definition 3.3.1. 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 $+$.

Examples of rings include the integers, real and complex numbers with the usual addition and multiplication.

However, for the applications we have in mind, parsing, the algebraic structure in question (see Section 4.2) does not even have associative multiplication, and does not have inverses for addition. We still have an additive 0 (the empty set—representing no parse), which is a zero 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). The usual proof that 0 is an absorbing element depends on the existence of the ability to cancel (which is implied by the existence of additive inverses in a group), as seen in section 3.1.3. So if we are to define an algebraic structure modelling this, we need to include as an axiom that zero absorbs.

There are a number of fairly standard generalisations of a ring, but none of these matches our requirements. One generalisation is the semiring, which has the same axioms as a ring, except that addition need not have inverses:

Definition 3.3.2. 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 generalisation is the nonassociative ring, which instead does away with the requirement that multiplication is associative and that there is an identity element for multiplication.

Definition 3.3.3. A set R together with two binary operations $+$ and $*$ forms a *nonassociative 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 monoid from the axioms of the structure. Hence, we make the following (nonstandard) definition:

Definition 3.3.4. 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, with operations _+_ and _*_ and additive identity 0_R. We open the IsCommutativeMonoid record for _+_, and prefix the ones referring to addition with +-.

```
record IsNonassociativeSemiring {R : Set} (_≈_ : R → R → Set)
                                          (_+_ _*_ : R → R → R)
                                          (0ᵣ : R) : Set₁ where
  field
    *-cong : ∀ {x x' y y'} → x ≈ x' → y ≈ y' → (x * y) ≈ (x' * y')
    +-isCommutativeMonoid : IsCommutativeMonoid _≈_ _+_ 0ᵣ
    distrib : Distributive _≈_ _*_ _+_
    zero : Zero _≈_ 0ᵣ _*_
  open IsCommutativeMonoid +-isCommutativeMonoid public
    renaming (assoc to +-assoc
             ; •-cong to +-cong
             ; identity to +-identity
             ; isMonoid to +-isMonoid
             ; comm to +-comm
             )
```

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

```
record NonassociativeSemiring : Set₁ where
  field
    R : Set
    _≈_ : R → R → Set
    _+_ : R → R → R
    _*_ : R → R → R
    0ᵣ : R
    isNonassociativeSemiring : IsNonassociativeSemiring _≈_ _+_ _*_ 0ᵣ
  open IsNonassociativeSemiring isNonassociativeSemiring 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 semiring elements, so we define a module similar to the one in Section 3.2.2:

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

3.3.2 Matrices

A matrix is in some sense really just a collection of numbers arranged in a rectangle, so there is nothing stopping us from defining such a matrix with entries from an arbitrary set, as opposed to from $\mathbf{R}$ 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 matrix of size $m \times n$ as a function from a pair of natural numbers (i, j), with $0 \leq i < m$, $0 \leq j < n$ (or, $i \in \operatorname{Fin} m$, $j \in \operatorname{Fin} n$), and hence, after currying, define:

Definition 3.3.5. A *matrix* A over a set R is a function $A : \operatorname{Fin} m \to \operatorname{Fin} n \to R$.

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

In our Agda development we only define the type of a matrix over a nonassociative semiring. For simplicity, and to allow us to avoid adding the nonassociative semiring as an argument to every function and proposition, we decide to parametrize the module we place the definition of a matrix in by a nonassociative semiring, and open the nonassociative semiring, renaming things so they start with "R-" to make it clear when we are referring to the concepts in the underlying nonassociative semiring:

```
module Matrix (NaSr : NonassociativeSemiring) where
open NonassociativeSemiring NaSr
  renaming (_+_          to _+_R_
           ; _*_         to _*_R_
           ; _≈_         to _≈_R_
           ; zero        to R-zero
           ; +-cong      to +_R-cong
           ; +-comm      to +_R-comm
           ; +-identity  to +_R-identity
           ; refl        to R-refl
           ; +-commutativeMonoid to +_R-CM
           )
```

If we had not opened the record, then instead of `a +_R b` for adding `a` to `b`, we would have to write the much less readable

```
(NonassociativeSemiring._+_ NaSr) a b
```

Now we define our matrix type in Agda:

```
Matrix : ℕ → ℕ → Set
Matrix m n = Fin m → Fin n → R
```

As with the algebraic structures previously, we want to be able to say that two matrices are equal. We will thus define matrix equality, which we denote by $\_\approx_M\_$ to disambiguate it from the regular equality. 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.

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

We also define the zero matrix. It should be a matrix whose elements are all equal to the zero in the nonassociative semiring.

```
zeroMatrix : {m n : ℕ} → Matrix m n
zeroMatrix i j = 0_R
```

If R is 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:

```
_+_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:

$$(AB)_{ij} = \mathbf{a_i} \cdot \mathbf{b_j}, \tag{3.1}$$

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 function from indices to elements of a nonassociative semiring:

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

We define the dot product by pattern matching on the length of the vector, making local definitions of head and tail for clarity:

```
_•_ : {n : ℕ} → Vector n → Vector n → R
_•_ {zero}    u v = 0_R
_•_ {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 cannot use AB or A B for matrix multiplication since juxtaposition means function application):

```
_*_M_ : {m n p : ℕ} → Matrix m n → Matrix n p → Matrix m p
(A *_M B) i j = row i A • col j B
  where row : {m n : ℕ} → Fin m → Matrix m n → Vector n
        row i A = λ k → A i k
        col : {m n : ℕ} → Fin n → Matrix m n → Vector m
        col j B = λ k → B k j
```

Here, Agda helps us 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, Agda more or less makes sure that our vectors are row vectors for A and column vectors for B. Alternatively, if we by writing down the formula (3.1) as the definition, Agda forces A to have as many rows as B has columns.

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

Proposition 3.3.6. *If R is a ring (nonassociative semiring), then the matrices of size $n \times n$ over R also form a ring (nonassociative semiring). Additionally, the matrices over R of size $m \times n$ form an Abelian group (commutative monoid) under matrix addition. In both cases, the zero matrix plays the role of the zero element.*

The proof is fairly easy but boring. We provide part of the proof when R is a nonassociative semiring in Agda: we prove that addition is commutative and that the zero matrix is a zero element of multiplication. The whole proof for a nonassociative semiring is available in our library.

We begin to prove that they form a commutative monoid. This is done by giving an element M+-isCommutativeMonoid of type

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

We need to supply the implicit arguments to _≈_M_ to make Agda understand what size of matrices the proposition concerns. To define this element, we need to give a proof

that it is a monoid, and a proof that it is commutative. Here, we only include the proof of the comm-axiom, the proofs involved in proving isMonoid are similar to it. The proof should be an element of type

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

Then, we recall the definitions of _+_M_ and _≈_M_, in particular that they are both pointwise operations. In fact, Agda tells us that the type reduces to:

```
(i : Fin .m) (j : Fin .n) →
NonassociativeSemiring._≈_ NaSr
(NonassociativeSemiring._+_ NaSr (x i j) (y i j))
(NonassociativeSemiring._+_ NaSr (y i j) (x i j))
```

which is the same as:

```
(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 nonassociative semiring elements x i j and y i j commute. But R-comm proves that any two elements of R commute, so we define M+-comm as:

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

To prove that the square matrices form a nonassociative semiring, we need to give an element M-isNonassociativeSemiring of type

```
M-isNonassociativeSemiring : ∀ {n} →
  IsNonassociativeSemiring (_≈_M_ {n}) (_+_M_) _*_M_ zeroMatrix
```

which includes giving proofs that matrices with addition form a commutative monoid, along with proofs that matrix multiplication respects matrix equality and distributes over matrix addition, and that zeroMatrix is a zero element. The last part consists of two conjuncts. We prove the left one here.

To prove this, we want to give an element:

```
M-zero^l : ∀ {n} → (x : Matrix n n) →
                   zeroMatrix {n} {n} *_M x ≈_M zeroMatrix
```

Agda tells us that the type of M-zero^l {n} reduces to

```
(x : Matrix n n) (i j : Fin n) → (zeroVector • col j x) ≈_R 0_R
```

where

```
zeroVector : {n : ℕ} → Vector n
zeroVector _ = 0_R
```

We prove that zeroVector • v ≈ 0_R for any v by induction on the length. The only interesting fact that appears is that the type of •-zero {suc n} {v} reduces to

$$(0_R \ *_R \ \text{head v}) \ +_R \ (\text{zeroVector} \bullet \text{tail v}) \approx_R 0_R$$

so we apply our lemma from Section 3.2.2:

```
•-zero : ∀ {n v} → zeroVector {n} • v ≈R 0R
•-zero {zero}          = R-refl
•-zero {suc n} {v} = e'•e''≈e +R-CM (proj1 R-zero (head v))
                                    (•-zero {n})
```

Then we define

```
M-zero^l {n} x i j = •-zero {n}
```

Finally, we give a name to the matrix nonassociative semiring, so we can supply it as an element of type NonassociativeSemiring (we do the same with commutative monoid):

```
M-nonassociativeSemiring : ∀ {n} → NonassociativeSemiring
M-nonassociativeSemiring {n} = record {
  isNonassociativeSemiring = M-isNonassociativeSemiring {n}}
```

3.3.3 Upper triangular matrices

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

Definition 3.3.7. A matrix is *upper triangular* if all elements on or below its diagonal are equal to zero.

The standard definition of upper triangular matrix allows nonzero elements on the diagonal (for example, the identity matrix is both upper and lower triangular), but we only consider matrices with zeros on the diagonal, so the above definition simplifies our language considerably. Since we are only interested in upper triangular matrices, we will usually refer to them as just *triangular* matrices. In Agda, there are two obvious ways to define a triangular matrix. The first is to use records, where a triangular matrix 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 illustrate these two approaches in Figure 3.1.

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 semiring to show that triangular matrices also form a nonassociative semiring (or a ring, if their elements come from a ring), under the obvious multiplication, addition and equality. The only problem we will have is to prove that the multiplication is closed.

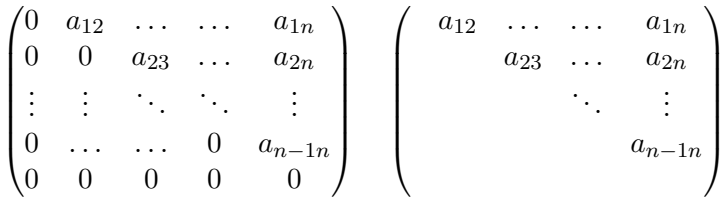


Figure 3.1: A figure showing an upper triangular matrix on the left and a "triangle" on the right.

One additional reason for not choosing the second approach is that it involves more inequalities between elements of Fin, and inequalities for Fin are a bit difficult to work with in Agda.

We define a datatype of triangular matrices, which we name Triangle as a record with a field mat for a matrix, and a field tri for a proof that it is zero on and below the diagonal:

```
record Triangle (n : ℕ) : Set where
  field
    mat : Matrix n n
    tri : (i j : Fin n) → toℕ j ⩽ toℕ i → mat i j ≈ᵣ 0ᵣ
```

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 and should not matter when comparing matrices.

```
_≈ₜ_ : {n : ℕ} → Triangle n → Triangle n → Set
A ≈ₜ B = Triangle.mat A ≈ₘ Triangle.mat B
```

Next, we go on to define addition and multiplication of triangles. We apply the matrix operations to the mat fields and modify the tri proofs appropriately. For addition, the proof modification is straightforward (we take care of the $0_R + 0_R$ with e'•e"≈e):

```
_+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 → e'•e"≈e +ᵣ-CM (Triangle.tri A i j i⩽j)
                                    (Triangle.tri B i j i⩽j)
  }
```

For multiplication, the proof modification required is a bit more complicated, and requires a lemma related to dot-products. We first prove that the dot product of two vectors u and v is zero if for every i, either the ith component of u or the ith component of v is zero. To do this, we need a further (short) lemma that the product of two

elements, one of which is zero is zero. We include the case where the first element is zero:

```
r*s-zero : (r s : R) → (r ≈R 0R) ∨ (s ≈R 0R) → r *R s ≈R 0R
r*s-zero r s (inj₁ r≈0) = begin
  r *R s
    ≈⟨ *-cong r≈0 refl ⟩
  0R *R s
    ≈⟨ proj₁ R-zero s ⟩
  0R ■
  where open CM-Reasoning +R-CM
```

and the case where s is zero is similar.

```
u•v-zero : ∀ {n} (u v : Vector n) →
           ((i : Fin n) → u i ≈R 0R ∨ v i ≈R 0R) → u • v ≈R 0R
u•v-zero {zero}  u v one0 = R-refl
u•v-zero {suc n} u v one0 = begin
  (head u *R head v) +R (tail u • tail v)
    ≈⟨ +R-cong (r*s-zero (u fzero) (v fzero) (one0 fzero))
               (u•v-zero (tail u) (tail v) (λ i → one0 (fsuc i))) ⟩
  0R +R 0R
    ≈⟨ proj₁ +R-identity 0R ⟩
  0R ■
  where open CM-Reasoning +R-CM renaming (_•_ to _+_)
```

Next, we use the fact that inequalities among natural numbers are decidable, see the end of Section 2.2, to extract a proof that for an arbitrary k, if $j \leq i$, then either A_{ik} or B_{kj} is zero (if $k \leq i$ then A_{ik} is zero, and if $\neg(k \leq i)$ then $j \leq k$, so B_{jk} is zero):

```
one0-mat : {n : _} → (A B : Triangle n) → (i j : Fin n)
  → toℕ j ⩽ toℕ i → (k : Fin n)
  → Triangle.mat A i k ≈R 0R ∨ Triangle.mat B k j ≈R 0R
one0-mat A B i j j⩽i k with toℕ k ⩽? toℕ i
one0-mat A B i j j⩽i k | yes k⩽i = inj₁ (Triangle.tri A i k k⩽i)
one0-mat A B i j j⩽i k | no  k⩽̸i = inj₂ (Triangle.tri B k j (begin
  toℕ j
    ⩽⟨ j⩽i ⟩
  toℕ i
    ⩽⟨ n⩽1+n (toℕ i) ⟩
  suc (toℕ i)
    ⩽⟨ ⩽̸⇒> k⩽̸i ⟩
  toℕ k ■))
  where open Data.Nat.⩽-Reasoning
```

where the module ⩽-Reasoning lets us use syntax similar to the one we used in Section 3.2.2 to prove that inequalities among natural numbers.

Now, we combine these to give the proof of triangularity for A $*_T$ B.

```
_*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 j⩽i → u•v-zero (row i (Triangle.mat A))
                              (col j (Triangle.mat B))
                              (one0-mat A B i j j⩽i)
  }
```

To prove that upper triangular matrices form a ring, all we need to do is apply the matrix results to Triangle.mat.

In our library, we use a more general definition of triangularity: a matrix is triangular of degree d if it is zero whenever $j - i \leq d$, that is, it is zero on the main diagonal and on $d - 1$ diagonals above it (an upper triangular matrix is triangular of degree 1). We prove there that if A is triangular of degree d_A and B is triangular of degree d_B, then AB is triangular of degree $d_A + d_B$. In particular, any product of at least $n - 1$ upper triangular matrices is equal to the zero matrix (there are $n-1$ diagonals above the main diagonal, and the product has to be zero on them all). We use this fact in Section 4.3 to prove (not in Agda) that the (seemingly infinite) sum (4.3) is actually finite and hence defines a unique upper triangular matrix.

Chapter 4

Parsing

Parsing is about analysing the structure of a sequence of tokens coming from some alphabet. We only give a brief overview of it here. We begin by introducing some concepts of parsing in Section 4.1. Then, in Section 4.2, we tie these concept together with the algebra from Section 3, and finally, in Section 4.3, we show that parsing is equivalent to computing the transitive closure of an upper triangular matrix. In Section 5, we then focus on a particular algorithm for computing the transitive closure, Valiant's algorithm, that we implement and prove correct using Agda.

4.1 Definitions

The goal of parsing is first to decide if a given sequence of tokens belongs to a given language, and second to describe its structure within the language. For this, we consider the process opposite to parsing: generating a string in a given language. To do this, one uses a grammar for the language, which contains rules that can be used to build all strings in the language. Here, we define concepts relevant to our thesis.

Definition 4.1.1. 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 $\alpha \to \beta$, where α and β are sequences of terminals and nonterminals, and α contains at least one nonterminal.
- $S \in N$ 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.

The terminals are the tokens that belong to the alphabet (and could be English words), while nonterminals are structural properties of sequence of tokens (for English words, they could stand for things like "noun" and "verb phrase").

Symbols	Explanation
E	Start symbol
$E + T$	2 on E
$E + T + T$	2 on E
$T + T + T$	1 on E
$T + T + T * F$	4 on rightmost T
$T + T + T * (E)$	5 on F
$T + T + T * (T + T)$	2, then 1 on E
$N + N + N * (N + N)$	4, then 6 on all Ts
$3 + 5 + 7 * (2 + 3)$	7 on all Ns

Figure 4.1: Generating the string $3 + 5 + 7 * (2 + 3)$ with the grammar in Example 4.1.1.

A grammar generates a string of terminals by repeatedly applying production rules to the start symbol, until there are no nonterminals left. The language generated by a grammar is the set of strings of terminals (or tokens) it generates.

Parsing is then the process of taking a string and figuring out what (if any) sequence of expansions might have produced it. Often, one creates a datastructure annotating the string with the nonterminals generating the parts of the string.

Example 4.1.1. We present a simple grammar for a language of arithmetic expressions (which appears in slightly modified form in [Lange and Leiß, 2009]):

- $\Sigma = \{1, \ldots, 9, +, *, (,)\}$,
- $N = \{E, T, F, N\}$, for "expression", "term", "factor" and "number", respectively.
- The production rules are
  1. $E \to T$
  2. $E \to E + T$
  3. $T \to F$
  4. $T \to T * F$
  5. $F \to (E)$
  6. $F \to N$
  7. $N \to i$, for $i = 1, \ldots 9$.
- $S = E$.

In Figure 4.1, we give an example of generating the string $3 + 5 + 7 * (2 + 3)$ with this grammar. To parse the string, we begin at the bottom of the figure, and apply the rules "backwards".

We are not going to consider arbitrary grammars in this report, so we give two restrictions to the definition above:

Definition 4.1.2. A grammar is *context free* if the left hand side of every production rule is a single nonterminal: $A \to \beta$.

Definition 4.1.3. A grammar is in (reduced) Chomsky Normal Form (CNF) [Chomsky, 1959] if the every production rule is of one of the following two forms:

$$A \to a$$
$$A \to BC$$

It is well known that any Context Free Grammar can be converted into a grammar in CNF (which generates the same language), with a size increase that is at most quadratic [Lange and Leiß, 2009]. In the remainder of the report, we only consider grammars in Chomsky Normal Form.

Example 4.1.2. We give a grammar in Chomsky Normal Form generating the same language as the one in Example 4.1.1 (which again appears, slightly modified, in [Lange and Leiß, 2009]) by introducing new nonterminals and production rules to get rid of the production rules which had the wrong form:

- $\Sigma = \{1, \ldots, 9, +, *, (,)\}$ (the same as before),
- $N = \{E, T, F, X, Y, Z, T_+, T_*, T_(, T_)\}$.
- The production rules (where $A \to \beta \mid \gamma$ is short for $A \to \beta$ and $A \to \gamma$, and $i = 1, \ldots, 9$) are:
  1. $E \to TX \mid TY \mid T_(\mid i$
  2. $T \to TY \mid T_(\mid i$
  3. $F \to T_(Z \mid i$
  4. $X \to T_+T$
  5. $Y \to T_*F$
  6. $Z \to ET_)$
  7. $T_+ \to +$
  8. $T_* \to *$
  9. $T_(\to ($
  10. $T_) \to)$
- $S = E$.

4.2 Grammar as a nonassociative semiring

The set of production rules for a grammar in Chomsky Normal Form which have the form $A \to BC$ could almost be used directly as a definition of multiplication on the nonterminals by replacing the arrows by equals signs:

$$BC = A$$

However, there are two problems with this. First, there can be nonterminals B and C with no production $A \to BC$. Second, there can be many different nonterminals that expand to the same thing: there can be A_1 and A_2 such that $A_1 \to BC$ and $A_2 \to BC$ are both productions.

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

$$x \cdot y = \{A \mid B \in x,\, C \in y,\, A \to BC \in P\}.$$

In general, this multiplication does not satisfy any algebraic axioms on its own, it is neither associative nor commutative. Since we are considering sets, it is natural to choose set union as addition and hence $\emptyset$ as 0, and with this, the above multiplication distributes over addition and the empty set is an absorbing (or zero) element for it. That is, the sets with set union and the above multiplication form a nonassociative semiring (see Definition 3.3.4). In the remainder of the report, we will prove things for an arbitrary nonassociative semiring.

4.3 A specification for parsing

In this section, we find a specification for the problem of parsing a string with a grammar in Chomsky Normal Form. Then, in the next section, we compare it to the specification used in [Valiant, 1975].

One approach to parsing a string w of length n is to form a matrix X containing the sets of all non-terminals that generate a substring: if we define $w[i, j)$ to be the substring starting at the ith symbol in w and ending at the $(j-1)$st, we let X_{ij} be the set of all nonterminals generating $w[i, j)$. The matrix formed this way is upper triangular since if $j \leq i$, X_{ij} is the set of all parses of an empty string.

Now, if we consider what nonterminals should be in the set X_{ij}, we note that:

- If $j = i + 1$, then $w[i, j)$ is a single token a. The only ways to generate $w[i, j)$ are using a production rules of the form $A \to a$, so X_{ij} is the set of all A such that $A \to a \in P$.

- If $j > i + 1$, then $w[i, j)$ contains more than one token. The only ways to generate $w[i, j)$ are thus using a production rule $A \to BC$, where B generates $w[i, k)$ and C generates $w[k, j)$, for some k. For a fixed k, we find all nonterminals A such that

$A \to BC \in P$, where B generates $w[i,k)$ and C generates $w[k,j)$ by computing $X_{ik} \cdot X_{kj}$. Hence,

$$X_{ij} = \bigcup_k X_{ik} \cdot X_{kj} = \sum_k X_{ik} X_{kj} = (XX)_{ij} \tag{4.1}$$

Combining the two points, we get:

$$X = XX + C, \tag{4.2}$$

where C is the matrix whose only nonzero entries are $C_{ii+1} = \{A \mid A \to w[i,i+1) \in P\}$. The above equation is the one we are going to use to derive Valiant's algorithm and prove it correct in Section 5.

We now present the specification used in [Valiant, 1975], and prove that it is equivalent to our specification above (4.2). In particular, this implies that our specification defines a unique X, and proving that Valiant's algorithm computes it (in Section 5.3.3) proves that it exists.

In [Valiant, 1975], the following definition of the transitive closure of a matrix is used:

Definition 4.3.1. The (non-associative) *transitive closure* of an upper triangular square matrix C is the matrix C^+ defined by

$$C^+ = \sum_{i=1}^{\infty} C^{(i)}, \tag{4.3}$$

where $C^{(n)}$ is the sum of all possible products of n factors C, defined recursively by:

$$C^{(1)} = C \quad \text{and} \quad C^{(n)} = \sum_{i=1}^{n-1} C^{(i)} C^{(n-i)}.$$

The sum in (4.3) is actually finite, because as we mentioned in the end of Section 3.3.3, any product of at least $n-1$ upper triangular matrices equals the zero matrix. Hence, if we show that X defined in (4.2) and C^+ defined in (4.3) are equal, we could use (4.3) to compute it, and it must therefore be unique.

Proposition 4.3.2. *The two equations* (4.2) *and* (4.3) *are equivalent: If C is upper triangular, then X is upper triangular and satisfies*

$$X = XX + C \tag{4.4}$$

if and only if

$$X = \sum_{i=1}^{\infty} C^{(i)}. \tag{4.5}$$

Proof. Assume first that $X = \sum_{i=1}^{\infty} C^{(i)} = \sum_{i=1}^{n-1} C^{(i)}$, where $n \times n$ is the size of X. It is clear that X is upper triangular since X is a sum of upper triangular matrices. Further,

$$XX = \left(\sum_{i=1}^{n-1} C^{(i)}\right)\left(\sum_{j=1}^{n-1} C^{(j)}\right) = \sum_{i=1}^{n-1}\sum_{j=1}^{n-1} C^{(i)}C^{(j)} = \sum_{i=1}^{n-1}\sum_{j=1}^{n-i-1} C^{(i)}C^{(j)},$$

where the last equality holds because the remaining terms contain more than $n-1$ factors C, and hence equal zero. The above sum contains one copy of $C^{(i)}C^{(j)}$ for each $i = 1, \ldots n-1$ $j = 1, \ldots, n-i-1$. Another way to sum these up (we can rearrange the sums since the addition is commutative) is to consider the sum $k = i + j$ and for each $k = 2, \ldots, n-1$ generate all products of k factors. Hence, the above sum equals

$$\sum_{k=2}^{n-1}\sum_{l=1}^{k-1} C^{(l)}C^{(k-l)} = \sum_{k=2}^{n-1} C^{(k)},$$

and so, $XX + C = \sum_{k=1}^{n-1} C^{(k)} = X$.

Next, we assume that X is upper triangular and satisfies

$$X = XX + C$$

We define R_k inductively by

$$R_1 = XX + C \quad \text{and} \quad R_{n+1} = R_n R_n + C.$$

Since $X = R_1$, and if $X = R_n$, then by inserting R_n in the right hand side of (4.3), $X = R_n R_n + C = R_{n+1}$, so by induction, $X = R_k$ for any k. We note also that after multiplying out (using the distributivity of multiplication over addition), R_k is a sum of terms consisting of products of X and C (only). Now, we want to prove:

1. For all i, $j \geq 1$, there is a k such that $C^{(i)}C^{(j)}$ is a term in R_k.
2. If $C^{(i)}C^{(j)}$, for i, $j \geq 1$, is a term in R_k, then it is also a term in R_{k+1}.
3. For every k, there are no structurally equal terms in R_k (terms that are products of the same factors, in the same order, including parentheses).
4. The number of factors in terms containing X in R_k is at least $k+1$.

From these facts, we can deduce that $X = \sum_{i=1}^{n-1} C^{(i)}$: By 1 and 2, there is a R_k that contains all products $C^{(i)}C^{(j)}$ with $i < n-1$, $j < n-1$, and hence contains the sum $\sum_{i=2}^{n-1} C^{(i)}$. By definition, it also contains C, so it contains $\sum_{i=1}^{n-1} C^{(i)}$. Thus, (for example) R_{k+n} also contains the whole sum. Since any term in R_{k+n} involving X contains at least $k+n+1$ factors by 4, those terms are zero. So any other terms in R_{k+n} are products of Cs only. But any such term would either be a product of more than $n-1$ Cs, and hence equal to zero, or a product of at most $n-1$ Cs, and hence nonexistent since by 3 there are no duplicates, and the term is in the sum $\sum_{i=1}^{n-1} C^{(i)}$, so $X = R_{k+n} = \sum_{i=1}^{n-1} C^{(i)}$.

We prove 2 by induction on $i+j$. If $i+j=2$, we get the statement that CC is a term in every R_k, $k \geq 2$ (since it is a term in R_2), and this is clearly true, since $R_{k-1} = Y+C$, so that $R_k = (Y+C)(Y+C)+C = YY+YC+CY+CC+C$. If $i+j = n+1$, and $C^{(i)}C^{(j)}$ is a term in R_k for some $k \geq 2$, then from the definition of R_k, $C^{(i)}$ and $C^{(j)}$ are terms in R_{k-1}, and hence are terms in every R_l, $l \geq k-1$ by induction (since each of $C^{(i)}$ and $C^{(j)}$ are either equal to C and hence a term in every R_l, or a sum of $C^{(i')}j'$, with $i'+j' = i$ or j which is at most n).

Next, we prove 1, again by induction on $i+j$. If $i+j=2$, then $C^{(i)}C^{(j)} = CC$, which is a term in R_2. If $i+j = n+1$, then there are k_i and k_j such that $C^{(i)}$ and $C^{(j)}$ are terms of R_{k_i} and R_{k_j} respectively (if i or j is 1, then R_1 will do, otherwise we use induction, since i and $j \leq n$). Then if we let $k = \max(k_i, k_j)$, by 2, R_k contains both, and hence, R_{k+1} contains their product.

We prove 3 by induction on k. First, R_1 contains no structurally equal terms. Second, if R_k contains no structurally equal terms, then if R_{k+1} contains two structurally equal terms, they must both be products, say X_1Y_1 and X_2Y_2, and then, R_k contains all four factors X_1, Y_1, X_2 and Y_2, but for X_1Y_1 to be structurally equal to X_2Y_2, their outermost parentheses must be equal, and so we must have that X_1 is structurally equal to X_2 and Y_1 is structurally equal to Y_2, since otherwise the placement of the parentheses would be different in the products.

We prove 4 by induction on k. In R_1, XX is the only term containing X, and contains 2 factors. If it is true for $k=n$, then when forming R_{n+1}, we multiply each term containing X, which contains at least $n+1$ factors, by something and the result thus contains at least $n+2$ factors. □

Since we have proven that our specification (4.2) is equivalent to Valiant's definition of transitive closure, (4.3), we will refer to our upper triangular matrix X as the transitive closure of C.

Although our specification (4.2) is seemingly less "computational" than (4.3), which could be used to compute the transitive closure of C by computing the value of the sum, ours is a lot simpler to use to derive Valiant's algorithm, as we do in Section 5.1. In particular, using our specification together with a block matrix makes Valiant's algorithm fall out almost immediately, while simply proving the correctness of it using (4.3) is a difficult task.

Additionally, we feel that our specification ties in with the problem of parsing a string much more naturally than (4.3) does. By considering elements of the parse chart, it is clear that the chart should satisfy (4.2), while to go from parsing to (4.3) involves using the fact that an element of the parse chart should contain all possible "formally distinct" sequences of nonterminals [Valiant, 1975].

Chapter 5

Valiant's Algorithm

In Valiant [1975], Leslie G. Valiant gave a divide and conquer algorithm for computing the transitive closure of an upper triangular matrix to prove that context free parsing has the same time complexity as matrix multiplication. The algorithm divides a string into two parts, 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 (where the combine step is also fairly parallelizable), it could potentially be used for parsing in parallel, as suggested in [Bernardy and Claessen, 2013].

5.1 Derivation

In this section, we are going to derive Valiant's algorithm from our specification (4.2).

5.1.1 Main structure

We want to compute the transitive closure of a parse chart. The main idea of the algorithm is to split the chart into two subcharts and a rectangular overlap region (containing parses of strings that overlap the corresponding splitting of the string). Next, compute the transitive closures of the subcharts, and combine them (somehow) to fill in the overlap part. A chart of size 1×1 contains just one element, which is zero because it is upper triangular, and the transitive closure of this is again the zero matrix.

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

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

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:

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

where U^+, R^*, L^+ are the part of $\mathcal{C}^+$s corresponding to U, R and L (a priori, we do not know if U^+ and L^+ are the transitive closures of U, L). Performing the multiplication and addition, we get:

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

Since $\mathbf{0}$ is a zero element and since all elements of two matrices need to be equal for the matrices to be equal, we get the set of equations:

$$U^+ = U^+U^+ + U \tag{5.1}$$
$$R^* = U^+R^* + R^*L^+ + R \tag{5.2}$$
$$L^+ = L^+L^+ + L, \tag{5.3}$$

We see that it the condition that $\mathcal{C}^+$ is the transitive closure of $\mathcal{C}$ is equivalent to the conditions that the upper and lower parts of $\mathcal{C}^+$ are the transitive closures of the upper and lower parts of $\mathcal{C}$, respectively and the rectangular part of $\mathcal{C}^+$ satisfies the equation (5.2), which we will refer to as the overlap specification. 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 do not depend on the second part, and vice versa, since the matrix is upper triangular. To parse a subset of the of the first part of a string, it does not matter what the second part of the string is, because the grammar is context free.

5.1.2 The overlap part

To compute the overlap part, we need to consider four separate cases, depending on the dimensions of R. We will derive a recursive algorithm for computing R^* from R, U^+ and L^+.

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 both equal the zero matrix. Hence, by (5.2), $R^* = R$.

We leave out the cases where R is $1 \times n$ or $n \times 1$. They are similar to the case when R is $m \times n$ with both m and n greater than 1 (and could be derived from it by allowing blocks with 0 width or height, which we discuss in Section 5.2.1).

Now, if R is an $m \times n$ matrix, with $m > 1$, $n > 1$, we can subdivide R along both rows and columns, into four blocks (A is an $i \times k$ matrix, B an $i \times l$ matrix and so on):

$$R = \begin{array}{c} \\ i \\ j \end{array} \begin{array}{c} \begin{array}{cc} k & l \end{array} \\ \begin{pmatrix} A & B \\ C & D \end{pmatrix} \end{array}$$

We subdivide U^+ and L^+ along the same rows and columns:

$$U^+ = \begin{matrix} \\ i \\ j \end{matrix} \begin{matrix} \begin{matrix} i & j \end{matrix} \\ \begin{pmatrix} U_U^+ & U_R^* \\ \mathbf{0} & U_L^+ \end{pmatrix} \end{matrix} \quad \text{and} \quad L^+ = \begin{matrix} \\ k \\ l \end{matrix} \begin{matrix} \begin{matrix} k & l \end{matrix} \\ \begin{pmatrix} L_U^+ & L_R^* \\ \mathbf{0} & L_L^+ \end{pmatrix} \end{matrix}$$

Inserting this in the overlap specification, (5.2), and performing the multiplications gives us,

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

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 do not satisfy the overlap specification for anything). Hence, after rearranging (and inserting parentheses to make things clearer), we get the equations

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

Now, recall that the overlap specification of R^* (5.2) contains three parts, something to multiply with on the left, denoted by U^+, something to multiply with on the right, denoted by L^+ and something to add, denoted by R. Looking at the above equations, we see that they state that A^*, B^*, C^* and D^* satisfy the overlap specification for some particular choices of R, U^+ and L^+.

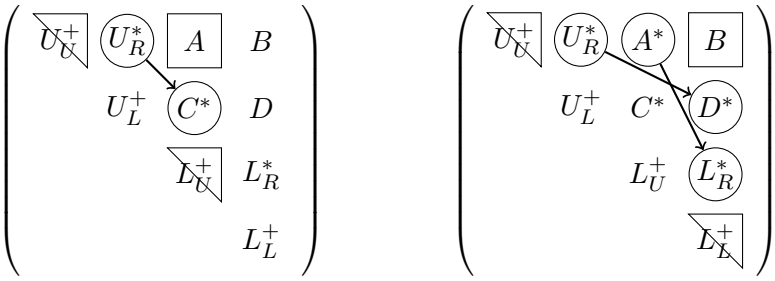
The third equation states that C^* satisfies the overlap specification with $U^+ = U_L^+$, $L^+ = L_U^+$ and $R = C$. The first equation states that A^* satisfies it with $U^+ = U_U^+$, $L^+ = L_U^+$, $R = U_R^* C^* + A$. Similarly for the second and fourth equations.

We see also that the equation for B^* contains A^* and D^*. The equations for A^* and D^* in turn both contain C^*, while C^* does not contain any of the other parts. So we can compute them recursively, starting with C^*, and then computing A^* and D^*, finishing with B^*. We illustrate the recursion paths in Figure 5.1.

5.1.3 Summary of Valiant's algorithm

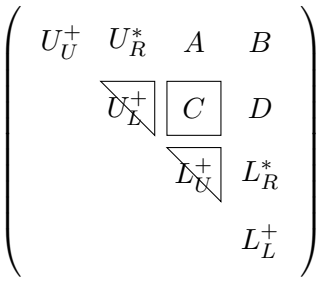
Summing up, we give the algorithm for computing the transitive closure $\mathcal{C}^+$ of an upper triangular matrix $\mathcal{C}$ here (including the cases where the rectangular part is a row or column vector) as a function Valiant($\mathcal{C}$):

- If $\mathcal{C}$ has size 1×1, return $\mathcal{C}$.

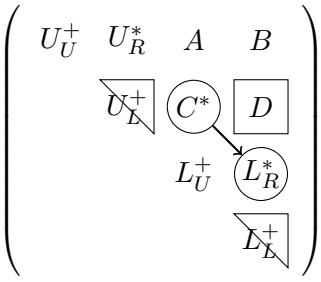


(a) Computation of A^*.

(b) Computation of B^*.



(c) Computation of C^*.



(d) Computation of D^*.

Figure 5.1: Illustration of the overlap step. Overlap is called on the two triangles and the square to which we add any circled parts, multiplied according to the arrows.

- Otherwise $\mathcal{C}$ has size $n \times n$, with $n > 1$. Split $\mathcal{C}$ into

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

and then:

1. Recursively compute $U^+ = \text{Valiant}(U)$ and $L^+ = \text{Valiant}(L)$.
2. Compute the overlap R^* with a function $\text{Overlap}(U^+, R, L^+)$ defined by:
   - If R has size 1×1, return R.
   - If R has size $n \times 1$, $n > 1$, then L^+ is the 1×1 zero matrix. Split U^+ and R into
   $$U^+ = \begin{pmatrix} U_U^+ & U_R^* \\ \mathbf{0} & U_L^+ \end{pmatrix} \quad \text{and} \quad R = \begin{pmatrix} u \\ v \end{pmatrix}$$
   and recursively compute
   $$v^* = \text{Overlap}(U_L^+, v, \mathbf{0})$$
   $$u^* = \text{Overlap}(U_U^+, U_R^* v^* + u, \mathbf{0}).$$
   Return $\begin{pmatrix} u^* \\ v^* \end{pmatrix}$.

– If R has size $1 \times n$, $n > 1$, then U^+ is the 1×1 zero matrix. Split R and L^+ into

$$R = \begin{pmatrix} u & v \end{pmatrix} \quad \text{and} \quad L^+ = \begin{pmatrix} L_U^+ & L_R^* \\ \mathbf{0} & L_L^+ \end{pmatrix}$$

and recursively compute

$$\begin{aligned} u^* &= \text{Overlap}(\mathbf{0}, u, L_U^+) \\ v^* &= \text{Overlap}(\mathbf{0}, u^* L_R^* + v, L_L^+). \end{aligned}$$

Return $\begin{pmatrix} u^* & v^* \end{pmatrix}$.

– Otherwise, R has size $m \times n$ with $m > 1$ and $n > 1$. Split U^+, R and L^+ into

$$U^+ = \begin{pmatrix} U_U^+ & U_R^* \\ \mathbf{0} & U_L^+ \end{pmatrix} \quad \text{and} \quad R = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \quad \text{and} \quad L^+ = \begin{pmatrix} L_U^+ & L_R^* \\ \mathbf{0} & L_L^+ \end{pmatrix}$$

and recursively compute

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

Return $\begin{pmatrix} A^* & B^* \\ C^* & D^* \end{pmatrix}$.

3. Return $\begin{pmatrix} U^+ & R^* \\ \mathbf{0} & L^+ \end{pmatrix}$.

5.2 Datatypes

In this section, we are going to define datatypes to use when we implement Valiant's algorithm in 5.3.

5.2.1 Discussion

To implement this in Agda using the Matrix and Triangle datatypes from Sections 3.3.2 and 3.3.3 would be very complicated since we would have to handle the splitting and triangularity proofs manually. Instead, we define concrete representations for the matrices and triangle that have the way we split them built in. We 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. We begin with Mat since it is needed to define Tri.

By the above reasoning, we have one constructor, which we will call quad that takes four smaller matrices and puts them together into a big one. Written mathematically, we want the meaning to be:

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

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 should definitely have a constructor sing for single element matrices. We realize that it is not enough to simply allow these 1×1 matrices, as it would imply that any matrix is a $2^n \times 2^n$ matrix, where n is the number of times we use quad.

One way to allow arbitrary dimensions for the matrices is to have a constructor empty 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:

$$\text{quad}(A, B, e_{0\,m}, e_{0\,n}) = \begin{pmatrix} A & B \\ e_{0\,m} & e_{0\,n} \end{pmatrix} = (A, B), \tag{5.4}$$

where e_m and e_n are empty $m \times 0$ and $n \times 0$ matrices respectively. Similarly, empty $n \times 0$ matrices are used to put two matrices with the same number of columns on top of each others. Second, an empty $m \times n$ matrix, $m > 0$, $n > 0$, represents a $m \times n$ matrix whose entries are all zero. One advantages of this method is that we could give fast implementations of addition and multiplication of "empty" matrices:

$$\begin{aligned} e_{m\,n} + A &= A \\ A + e_{m\,n} &= A \\ e_{m\,n} A &= e_{m\,p} \\ A e_{n\,p} &= e_{m\,p}, \end{aligned}$$

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 we generally pattern match on the structures, and this gives us a case for each constructor.

One downside is that there are multiple constructors for matrices of the same size (there is always empty and a nonempty way when $m > 0$, $n > 0$), removing some of the advantage we get from having few constructors.

Another approach, which is the one we take in this report, is to have constructors for row and column vectors, $1 \times n$ and $n \times 1$ matrices for arbitrary $n > 1$, along with the single element matrices. 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 specialise algorithms when the input is a vector, but introduces one extra constructor (sing, rVec, cVec and quad compared to empty, sing and quad).

We then need a concrete representation Vec of vectors. Since we want to be able to split vectors along the middle (to implement Valiant's algorithm in the case where R is a vector), we give them a constructor two that takes a vector of length m and one of length n and concatenates them. For our base cases, we need to be able to build single element vectors, this is enough to build any vector. To implement this approach, we need to define the datatypes Vec of vectors and Mat of matrices (that should be concrete representations of Vector and Matrix).

5.2.2 A first attempt at an Agda implementation

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

```
data Vec' : ℕ → Set where
  one : R → Vec' 1
  two : {m n : ℕ} → Vec' m → Vec' n → Vec' (m + n)
```

and then defining Mat' as

```
data Mat' : ℕ → ℕ → Set where
  sing  : R → 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)
```

Finally, to define Tri' is straightforward. There is only one base case, that of the 1×1 zero triangle, with constructor zer. We only need one size argument since it is a square matrix.

```
data Tri' : ℕ → Set where
  zer : Tri' 1
  tri : {m n : ℕ} → Tri' m → Mat' m n →
                                Tri' n
                                → Tri' (m + n)
```

While the above looks very natural, it will not work well when we want to prove things about the matrices. 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, and that both r_1 and r_2 (say) have to be at least 1. It is possible to write lemmas proving this, and use them at every step, but this clutters the proofs. Additionally, when Agda will be unable to infer for example that the different parts have been built the same way. For example when trying to define the overlap row step, $m = 1$, $n > 1$ we pattern match

on R and L, and Agda infers that they have sizes x + y and x' + y', but cannot infer (since it need not be true) that their x equals x' and y equals y', which is required to compute the overlap for the parts of R recursively.

5.2.3 Mat and Tri

Instead we want to use an approach for indexing our matrices where we build the splitting further into the datatypes. Looking at the first attempt to define quad, we see that we only use constant ℕs and _+_ (we never use suc to increase the size of a matrix). So we could use a datatype like ℕ, but, instead of having suc as a constructor, it has _+_. We define such a datatype and call Splitting, since it determines the splitting of the matrix, and define it as follows

```
data Splitting : Set where
  one : Splitting
  bin : Splitting → Splitting → Splitting
```

where one plays the role of suc zero (since there is 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 the one element type, then Splitting is the type of binary trees with elements from the one element type).

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

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

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

```
data Vec : Splitting → Set where
  one : R → Vec one
  two : {s₁ s₂ : Splitting} → Vec s₁ → 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 R (with elements in the leaves). We move on to defining Mat as:

```
data Mat : Splitting → Splitting → Set where
  sing : R → Mat one one
  rVec : {s₁ s₂ : Splitting} → Vec (bin s₁ s₂) → Mat one (bin s₁ s₂)
  cVec : {s₁ s₂ : Splitting} → Vec (bin s₁ s₂) → Mat (bin s₁ s₂) one
  quad : {r₁ r₂ c₁ c₂ : Splitting} → Mat r₁ c₁ → Mat r₁ c₂ →
                                     Mat r₂ c₁ → Mat r₂ c₂ →
                                     Mat (bin r₁ r₂) (bin c₁ c₂)
```

Finally, Tri:

```
data Tri : Splitting → Set where
  zer : Tri one
  tri : {s₁ s₂ : Splitting} → Tri s₁ → Mat s₁ s₂ →
                                Tri s₂
                              → Tri (bin s₁ s₂)
```

Later, we are going to prove that Tri s is a nonassociative semiring for 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 (also, we need these to define multiplication, since multiplying a Tri one by a Mat one n gives a zero matrix):

We define them by pattern matching on splittings:

```
zeroVec : {s : Splitting} → Vec s
zeroVec {one}       = one 0_R
zeroVec {bin s₁ s₂} = two zeroVec zeroVec

zeroMat : {s₁ s₂ : Splitting} → Mat s₁ s₂
zeroMat {one}       {one}       = sing 0_R
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}       = zer
zeroTri {bin s₁ s₂} = tri zeroTri zeroMat zeroTri
```

5.2.4 Operations on our datatypes

In this section, we will define operations: addition, multiplication and equality, for Vec, Mat and Tri.

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

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

then for Mat:

```
_+m_ : {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')
```

and finally for Tri:

```
_+t_ : {s : Splitting} → Tri s → Tri s → Tri s
zer        +t zer          = zer
tri U R L  +t tri U' R' L' = tri (U +t U') (R +m R') (L +t L')
```

For multiplication, we need to do a bit more work. 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, matrix multiplication works out nicely with regard to the block structures:

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

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 the simplicity of it is very helpful.

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

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

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

$$\begin{pmatrix} u \\ v \end{pmatrix} \begin{pmatrix} u & v \end{pmatrix} = \begin{pmatrix} uu' & uv' \\ vu' & vv' \end{pmatrix}.$$

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 do not need anything to define the dot product). We hence begin with the simplest kinds of multiplication, first scalar–vector multiplication:

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

Vector–scalar, scalar–matrix and matrix–scalar multiplication are similar, so we leave them out. We define the dot product:

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

and 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 R*v u) (x R*v v))
two u v ⊗ one x  = cVec (two (u v*R x) (v v*R x))
two u v ⊗ two u' v' = quad (u ⊗ u') (u ⊗ v') (v ⊗ u') (v ⊗ v')
```

We give the types of vector–matrix and matrix–vector multiplication (but leave out their straightforward implementation):

```
_ v*m _ : {s1 s2 : Splitting} → Vec s1 → Mat s1 s2 → Vec s2
_ m*v _ : {s1 s2 : Splitting} → Mat s1 s2 → Vec s2 → Vec s1
```

Finally, using all of the above definitions, 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 R*v v)
rVec v          *m cVec v'            = sing (v • v')
rVec (two u v)  *m quad A B C D       = rVec
  (two (u v*m A +v v v*m C) (u v*m B +v v v*m D))
cVec v          *m sing x             = cVec (v v*R x)
cVec v          *m rVec v'            = v ⊗ v'
quad A B C D    *m cVec (two u v)     = cVec (two (A m*v u +v B m*v v)
                                                  (C m*v u +v D m*v 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 *m C') (C *m B' +m D *m D')
```

To define triangle multiplication is 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 and column splittings differ. We also need to define multiplication between Vec and Tri and between Mat and Tri, all of which are straight-forward to define:

```
_*v t_ : {s      : Splitting} → Vec  s      → Tri  s      → Vec  s
_*t v_ : {s      : Splitting} → Tri  s      → Vec  s      → Vec  s
_*m t_ : {s₁ s₂ : Splitting} → Mat  s₁ s₂ → Tri  s₂     → Mat  s₁ s₂
_*t m_ : {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
zer  *t  zer = zer
tri  U  R  L  *t  tri  U'  R'  L' = tri  (U  *t  U')  (U  t*m  R'  +m  R  m*t  L')
                                                      (L  *t  L')
```

The final part needed to express the transitive closure specification (4.2) 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 semiring-equality to the datatype in question. We begin with Vec, where one element vectors if they contain the same element, and two element vectors are equal if their left parts are equal and their right parts are equal:

```
_≈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 other equality definitions) only apply to vectors with the same splitting, so vectors which contain the same elements can be unequal and have the same length.

We move on to equality for Mat:

```
_≈m_ : {s₁ s₂ : Splitting} → Mat  s₁  s₂ → Mat  s₁  s₂ → Set
sing  x           ≈m  sing  x'           = x  ≈R  x'
rVec  v           ≈m  rVec  v'           = v  ≈v  v'
cVec  v           ≈m  cVec  v'           = v  ≈v  v'
quad  A  B  C  D  ≈m  quad  A'  B'  C'  D' = (A  ≈m  A') ∧ (B  ≈m  B') ∧
                                            (C  ≈m  C') ∧ (D  ≈m  D')
```

and to finish this section, equality for Tri:

```
_≈t_ : {s : Splitting} → Tri  s → Tri  s → Set
zer          ≈t  zer          = ⊤
tri  U  R  L  ≈t  tri  U'  R'  L' = (U  ≈t  U') ∧ (R  ≈m  R') ∧ (L  ≈t  L')
```

where two zer are always equal (⊤ is the true proposition).

5.2.5 Nonassociative Semirings

We will now prove that Vec, Mat and Tri are commutative monoids with $\_+_v\_$, $\_+_m\_$ and $\_+_t\_$, and Tri is a nonassociative semiring with $\_+_t\_$ and $\_*_t\_$ as defined above. One

big reason for doing this is that it will make it possible to use the equational reasoning introduced in Section 3.2.2. We will prove

```
∀ {s}      → IsCommutativeMonoid _≈ᵥ_ _+ᵥ_ (zeroVec {s})
∀ {s₁ s₂} → IsCommutativeMonoid _≈ₘ_ _+ₘ_ (zeroMat {s₁} {s₂})
∀ {s}      → IsCommutativeMonoid _≈ₜ_ _+ₜ_ (zeroTri {s})
∀ {s}      → IsNonassociativeSemiring _≈ₜ_ _+ₜ_ _*ₜ_ (zeroTri {s})
```

The reason we include the Splitting in the zero elements 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. Complete proofs are available in our library. We also define instances of CommutativeMonoid (for Vec, Mat and Tri) and NonassociativeSemiring (for Tri).

Proving things about addition means pushing the statements into the algebraic structure below. We exemplify by proving 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)
```

Proving things about multiplication also means moving the properties down to the nonassociative semiring below, but here, the path is longer. We exemplify 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-zeroˡ : {s : Splitting} → (x : Tri s) → zeroTri *ₜ x ≈ₜ zeroTri
*ₜ-zeroˡ zer = tt
*ₜ-zeroˡ {bin s₁ s₂} (tri U R L) =
  ( *ₜ-zeroˡ U
  , e'•e''≈e Mat-commutativeMonoid {zeroTri *ₜₘ R} {zeroMat *ₘₜ L}
      ( *ₜₘ-zeroˡ R) ( *ₘₜ-zeroˡ {s₁} L)
  , *ₜ-zeroˡ L
  )
```

where tt is the constructor of the true proposition (we want to prove that any two zer are equal), and

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

are the proofs (that we would need to write) that zeroTri is a "left zero" of $\_*_{t}\_*_{m}\_$, and that zeroMat is a "left zero" of $\_*_{m}\_*_{t}\_$ (if the concept of a zero is slightly generalised to "operations" $f : A \to B \to A$ and $f : A \to B \to B$).

5.3 Implementation and proof of correctness

We are now ready to implement the algorithm in Agda.

5.3.1 Implementing the algorithm

Using the above operations, we can now define Valiant's algorithm in Agda, following the outline in Section 5.1.3. First we define functions for the overlap part (we introduce two new functions overlapRow and overlapCol for the cases when one dimension of the matrix is 1):

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

we use **where** definitions to avoid having to repeat code and to show that things are not mutually recursive. Then we define the actual algorithm

```
valiant : {s : Splitting} → Tri s → Tri s
valiant zer           = zer
valiant (tri U R L) = tri U⁺ (overlap U⁺ R L⁺) L⁺
  where U⁺ = valiant U
        L⁺ = valiant L
```

In the next section, we give a specification for the algorithm, and in Section 5.3.3, we prove it correct.

5.3.2 Specification in Agda

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 proposition that C^+ is the transitive closure of C, which is true if C^+ and C satisfy the specification (4.2) as implemented in Agda:

```
_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 three sub-specifications: (5.2) and its restriction to the case when one dimension is 1, where the first one is considered as a relation between a Mat and a Tri (the Mat satisfies the specification when inserting the parts of the Tri):

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

For the row and column part, instead consider a relation between a Vec and a pair of a Vec and a Tri, with the intent that the Vec is put on top of the Tri. We do this to avoid dealing separately with the 1×1-vector case:

```
_is-tcRow-of_ : {s : Splitting} → Vec s → Vec s × Tri s → Set
v× is-tcRow-of (v, L⁺) = v× ≈ᵥ v× *ᵥₜ L⁺ +ᵥ v
```

```
_is-tcCol-of_ : {s : Splitting} → Vec s → (Tri s × Vec s) → Set
v× is-tcCol-of (U⁺, v) = v× ≈ᵥ U⁺ *ₜᵥ v× +ᵥ v
```

5.3.3 The proof

In this section, we are going to prove the correctness of Valiant's algorithm, as defined in the previous section, 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 types of the different propositions, so we can use them in an arbitrary order later. The first is the main proposition:

```
v-tc  : {s : Splitting} (C : Tri s) → (valiant C) is-tc-of C
v-mat : {s₁ s₂ : Splitting} (U⁺ : Tri s₁) (R : Mat s₁ s₂) (L⁺ : Tri s₂) →
  (overlap U⁺ R L⁺) is-tcMat-of (tri U⁺ R L⁺)
v-row : {s : Splitting} (v : Vec s) (L⁺ : Tri s) →
  (overlapRow v L⁺) is-tcRow-of (v , L⁺)
v-col : {s : Splitting} (U⁺ : Tri s) (v : Vec s) →
  (overlapCol U⁺ v) is-tcCol-of (U⁺, v)
```

giving an object of the first type is easy:

```
v-tc zer = tt
v-tc (tri U R L) = (v-tc U , v-mat (valiant U) R (valiant L) , v-tc L)
```

The other parts take a bit more code, but that code is for shuffling nonassociative semiring objects around and is easy to write. We include the proof of the correctness of overlapRow here:

```
v-row (one x) zer = R-sym (proj₁ +_R-identity x)
v-row {bin s₁ s₂} (two u v) (tri U R L) = (v-row u U) , (begin
  overlapRow (overlapRow u U *_v*_m R +_v v) L
    ≈⟨ v-row (overlapRow u U *_v*_m R +_v v) L ⟩
  v₁ +_v (v₂ +_v v)
    ≈⟨ sym (assoc v₁ v₂ v) ⟩
  (v₁ +_v v₂) +_v v
    ≈⟨ •-cong (comm v₁ v₂) refl ⟩
  (v₂ +_v v₁) +_v v
    ≈⟨ refl ⟩
  overlapRow u U *_v*_m R +_v
  (overlapRow (overlapRow u U *_v*_m R +_v v) L) *_v*_t L +_v v ■)
  where open CM-Reasoning (Vec-commutativeMonoid {s₂})
        v₁ = overlapRow (overlapRow u U *_v*_m R +_v v) L *_v*_t L
        v₂ = overlapRow u U *_v*_m R
```

the other proofs are similar.

Chapter 6

Concluding remarks

This thesis was initially intended to put Valiant's algorithm into the Algebra of programming framework [Bird and Moor, 1997]. Algebra of programming is about expressing specifications as a relation, using relational catamorphisms and the converses of catamorphisms, and then derive a function that refines the relation using universal properties of catamorphisms. We wanted to use Agda for this, because there has been some work on doing Algebra of programming derivations in Agda [Mu et al., 2009], and because of the ability to implement the algorithm and prove it correct in the same language, ensuring us that we made no implementation errors. We see two main reasons for failing to do this.

First, Valiant's algorithm is a catamorphism on the Tri datatype (it recurses on the two sub-triangles and then combines them), but we did not find a way to express the overlap step using ideas from resembled Algebra of programming. In part, this was because the recursion is fairly complicated (first compute the lower left part recursively, then use that result to compute the upper left and lower right parts recursively, and finally use those results to compute the upper right part). So we did not know what we were supposed to aim for in our derivation.

Second, the relation we have used as specification: X is the transitive closure of an upper triangular matrix C if X is upper triangular and satisfies $X = XX + C$, is very nice to work with mathematically. Indeed, as we saw in Section 5.1, by combining the specification with block matrices, the algorithm falls out immediately. However, the relation does contain a lot of hidden information: mainly the fact that matrix multiplication is a fairly complicated function when defined on our datatypes (see Section 5.2.4), that we have failed to express within Algebra of programming. Hence, we have neither a starting point, nor an end point of our derivation. It might be the case that other specifications or other datatypes for matrices are more suitable to do Algebra of programming-style derivations.

The algebra involved in this thesis is fairly trivial. From an algebraic point of view, the most interesting fact is perhaps that the multiplication involved in parsing (see Section 4.2) is an example of a "basic" nonassociative operation. It is not the combination of more basic operations and functions that satisfy axioms that force it to be nonassociative

(as is the case with subtraction in a group, for example).

Doing algebra in Agda has the advantage of making it visible where the axioms are used, and hence, whether they are needed or not. In particular, since the origin of the algebraic structure for parsing (the nonassociative semiring) is as a set with set union and a multiplication, early on, we thought that we might need the axiom that addition is idempotent ($x + x = x$ for all x), which holds for set union. But after writing the proof, we found that we had never used the axiom, and so we removed it from our algebraic structure.

A big problem of doing algebra in Agda is that if we change the definition of some datatype slightly (for example, if we allow empty matrices, or if we decide to remove the sing constructor and use one element vectors instead), almost all our proofs break, and we will have to redo them. To solve this problem, we could make our programs more modular, using interface-like records for nonassociative semirings that have splittings which respect addition and multiplication (to represent the triangles and matrices)—similar to the way the algebraic structures provide an interface to the addition and multiplication. However, doing this has two drawbacks in our case. First, we would have to write almost all the code to prove that the splittings respect addition and multiplication (so we gain little from doing it). Second, using these interfaces would make it impossible to use Agda's built in pattern matching tools, which would make proof writing take longer. We are not sure whether a better solution to this exists.

Bibliography

The Agda team. The Agda reference manual: Totality. `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Totality`, December 2011. Accessed: 2013–05–30.

The Agda team. The Agda standard library, version 0.7. `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.StandardLibrary`, January 2013. Accessed: 2013–05–30.

Jean-Philippe Bernardy and Koen Claessen. Efficient parallel and incremental parsing of practical context-free languages. `http://publications.lib.chalmers.se/publication/175113`, March 2013.

Richard Bird and Oege de Moor. *Algebra of programming*. Prentice Hall, London, 1997. ISBN 0-13-507245-X.

Ana Bove and Peter Dybjer. Dependent types at work. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development*, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03152-6. doi: 10.1007/978-3-642-03153-3_2.

Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959. ISSN 0019-9958. doi: 10.1016/S0019-9958(59)90362-6.

Martin Lange and Hans Leiß. To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica*, (8), 2009. ISSN 1615-1771.

Per Martin-Löf. *Intuitionistic type theory : notes by Giovanni Sambin of a series of lectures given in Padua, June 1980*. Bibliopolis, Napoli, 1984. ISBN 88-7088-105-9.

Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in Agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19(05):545–579, 2009. doi: 10.1017/S0956796809007345.

Ulf Norell. *Towards a practical programming language based on dependent type theory.* Chalmers University of Technology, Göteborg, 2007. ISBN 978-91-7291-996-9. Diss. Göteborg : Chalmers tekniska högskola, 2007.

Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

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.