Certifying Time Complexity of Agda Programs Using Complexity Signatures

Christian Bach Møllnitz
Johannes Elgaard
Simon Rannes

Department of Computer Science
Aalborg University

June 14, 2020

Abstract

Danielsson and others have shown that by annotating functions it becomes possible to perform time complexity analysis in functional languages, however the idea of how little annotation is necessary has not been explored. This paper describes a small library which enables a method for minimal annotation of functions to certify time complexity with a `Timed` monad. Using this method we develop a library of matrix functions with certified time complexity and implement the Cocke-Younger-Kasami (CYK) parsing algorithm for context-free-grammars which we certify to have a complexity of $\mathcal{O}(n^5)$ in Agda, a dependently typed programming language.

Preface

This report was made as a 10th semester project at Aalborg University. We would like to thank our supervisor Hans Hüttel for his steadfast support, without which this report would not have been possible.

Johannes Elgaard
<jelgaa15@student.aau.dk>

Christian Møllnitz
<cmolln10@student.aau.dk>

Simon Rannes
<sranne13@student.aau.dk>

Danish Summary

I dette kapitel giver vi et kort resume af denne rapport.

I kapitel 1 presenterer vi en introduktion til rekursionsligninger, en analyse af eksisterende arbejde inden for området mht. certificering af tidskompleksitet, introducerer signaturer og præsenterer en initiel, udvidet og til sidst, en endelig problemformulering som resten af rapporten vil forsøge at svare på.

Kapitel 2 er primært en introduktion til Agda, et afhængigt typed programmeringssprog, og dens centrale elementer, så som induktive familier og GADTs. Efter Agda introduktionen så fortsætter vi med at vise hvordan man konstruerer beviser i Agda, med et eksempel på kommutativitet for parvis addition, og derefter et bevis for den associative egenskab af naturligt tal multiplikation ved brug af den induktive metode. Vi viser også hvordan `rewrite` kan bruges til at gøre beviser meget kortere ved at kun give Agda beviser, og så lade den håndtere omskrivningen.

Kapitel 3 går i detaljer med implementeringen af kompleksitets signaturer, ved brug af `Timed` monaden, og forklarer hvordan det virker at *løfte* værdier og funktioner til `Timed` varianter. Vi giver et eksempel på hvordan append, (`_++_`), kan annoteres med kompleksitets information for at lave en ny version af den, (`_++′_`). Derefter så beviser vi at kompleksiteten af den annoterede append funktion er $\Theta(n)$. Og så beskriver vi en generaliseret proces for at lave et helt bibliotek af annoterede funktioner, ved at annotere hver eneste funktion i dens signatur, og ved at bruge stamfunktioner til at bevise kompleksiteten af sammensatte funktioner.

Til sidst der viser vi hvordan vi brugte denne fremgangsmåde til at lave et matrice bibliotek, og dertilhørende beviser, med en matrice multiplikation i $\mathcal{O}(n^4)$.

I kapitel 4 der introducrer vi Cocke-Younger-Kasami (CYK) algoritmen, og præsenterer en plan for at fremstille en certificeret version af denne, som bruger en forenklet tilgang til at udregne transitiv aflukning, i stedet for at reducere den transitive aflukning til multiplikation. Efter det så diskuterer vi vores første implementering af CYK og viser hvordan vores tilgang til tids certificering gør det svært at bruge på lister af ubestemte længder. Vores anden implementering bliver derefter præsenteret, hvor at vi bruger elementer der består af *bit strings* i stedet for lister. Til det der viser vi så vores beviser for CYKs kompleksitet, hvilket

viser sig at være $\mathcal{O}(n^5)$.

I kapitel 5 der diskuterer vi konsekvenserne af vores fremgangsmåde og diskuterer vores resultater i lys af både vores udvidede, samt vores endelige problemformuleringer og de kriteria vi brugte tilbage i kapitel 1 til at analysere andres arbejde. Derefter der konkluderer vi på vores resultater og vores problemformuleringer. Og til sidst der presenterer vi nogle idéer for fremtidig arbejde der kun være interessante af følge op på.

Contents

Contents **7**

List of Figures **9**

1 Introduction **11**
- 1.1 Recurrence relations . . . 12
- 1.2 Analysing the run time complexity of higher order functions . . . 14
- 1.3 Analysis of existing work . . . 15
- 1.4 Introducing signatures . . . 20
- 1.5 Final problem statement . . . 21

2 An Agda centric introduction to dependent types **23**
- 2.1 Introducing the Agda programming language . . . 23
- 2.2 Structure of proofs in Agda . . . 25

3 Certified complexity of matrix multiplication **31**
- 3.1 Implementing complexity signatures in Agda . . . 31
- 3.2 Matrix library . . . 35

4 A complexity certified CYK implementation **43**
- 4.1 An introduction to CYK . . . 43
- 4.2 Implementation of complexity certified CYK . . . 45
- 4.3 Proving the time complexity of the CYK algorithm . . . 49

5 Conclusion **51**
- 5.1 Results . . . 51
- 5.2 Discussion . . . 53
- 5.3 Future work . . . 56

Bibliography **57**

A Annotation library **59**

B Annotation library for vectors **61**

C Matrix library **63**

D Matrix proofs **69**

E Old CYK library **75**

F New CYK library **79**

G CYK proofs **83**

List of Figures

1.1 An example of a higher order function, map 15
1.2 An example of map applying lgd to a vector 15
1.3 The definition of the length function in TiML[6] 16
1.4 The monotone function for the recurrence relation. 17
1.5 An unannotated length function and an annotated one, using Danielsson's approach. 18

2.1 Inductive definition of natural numbers in Agda 24
2.2 Definition of the Vec type 24
2.3 Structurally recursive fib in Agda 25
2.4 Agda Pair Definition 25
2.5 Definition of Pair addition 26
2.6 Proof that Pair addition is commutative written out in steps 27
2.7 Proof that Pair addition is commutative written using rewrite 27
2.8 Base case for the proof of associativity of multiplication 28
2.9 The inductive case for the proof of associativity of multiplication 29
2.10 Multiplication associativity rewrite proof 29

3.1 lookup′ function, locates element in vector, from an index 35
3.2 matrixToColumn′ function, extracts column from matrix 37
3.3 MatrixMultiplicationBase function, requires parameterization with dot function 37
3.4 Matrix dot product, _•′_, lifted dot product _•′′_, and multiplication, _⊗_. . . . 38
3.5 lookup-cost-lem1 proving the exact time complexity of lookup′ 38
3.6 lookup-cost-proof proving the $\mathcal{O}$ time complexity of lookup′ 39
3.7 matrixToColumn′-cost-proof proving the complexity of column extraction from a matrix 39
3.8 •′-cost-proof and •′-≤-cost-proof, proving the Θ complexity of •′ and converting the proof to $\mathcal{O}$ complexity 40

3.9 matrix-multiply-cost-lem1, which from a proof of the complexity of computing the dot product, proves the complexity of computing a row 41
3.10 matrix-multiply-cost-proof base proof for parameterization with dot function and proof 42
3.11 ⊗-cost-proof proving the time complexity of $m_1 \otimes m_2$ 42

4.1 Matrix computation order 45
4.2 Definition of Rule data type 46
4.3 Definition of Language record 46
4.4 Definition of Element as a list of nonterminals 46
4.5 _∪_ function for lists 46
4.6 _•_ for matrices of lists 47
4.7 _⊗_ parameterized version of MatrixMultiplicationBase for matrices of lists 47
4.8 Definition of *bit string* Element 48
4.9 _∪_ function, union of two *bit strings* 48
4.10 _•′_ function, dot product function for vectors of Elements 48
4.11 _⊗′_ parameterized version of MatrixMultiplicationBase for matrices of Elements 48
4.12 ∪′-cost-proof proving the time complexity of $l \cup' r$ 49
4.13 •′-cost-proof proving the time complexity of $x \bullet' y$ 50
4.14 ⊗-cost-proof proving the time complexity of $m_1 \otimes' m_2$ 50

5.1 Complexity annotated matrix multiplication. 54
5.2 Normal matrix multiplication 54

Chapter 1

Introduction

Program certification is the process of formally proving that a program satisfies some property. The type checkers found in many programming languages provide one such example of certification, that a program will not have type errors at runtime. But this is not the limits of what program certification can do, but merely the beginnings. In a dependently typed programming language, such as Agda, the possibilities are greatly expanded. Previous work has contributed with the certification of the correctness of a suite of algorithms related to the parsing and recognition of context-free languages through the CYK algorithm, showing that the class of context-free languages is contained in P, by Firsov [1], and merge sort by Copello et al. [2] . While these papers certify the termination and correctness of the algorithms they examine, performance is a lesser concern and to a certain extent is left unexamined which is arguably failing to certify one of the most important characteristics. Especially with CYK which importantly features a $\mathcal{O}(n^3)$ time complexity.

It has been shown by Danielsson [3] that it is indeed possible to certify other characteristics of a program, such as time complexity, which indicates an interesting gap in the field. Certified algorithms exist without certification of their time or space complexity, or with a comparison of their run-time performance to non-certified, industry standard, counterparts.

The advantages of program certification are well understood, in that programs serve as proofs of their own correctness, but conversely the time and space complexity of these programs are less explored and seem like an interesting avenue for further research.

Based on the analysis we did in the preliminary report [4], we will in this section specify a problem statement that will be the foundation for the work done in this report, with the following *initial* problem statement:

How do we certify the run-time complexity of an algorithm using functional programming and a proof assistant?

This question will be further specified after an outline of recurrence relations, higher order functions and existing work.

1.1 Recurrence relations

Recurrence relations are an important tool that are useful in defining the complexity of a recursive algorithm. Because of this, in this section we will cover their concept, and explain how they are used when we are talking about the time complexity of recursive algorithms. We will also show how to turn a open-form recurrence relation into the closed-form, using an induction proof.

1.1.1 Introduction to recurrence relations

Recurrence relations are a way to succinctly express some part of a sequence, based on the preceding elements. In more specific terms, for the Fibonacci sequence we can determine the next number in the sequence, based on the two previous ones, as such:

$$f_n = f_{n-1} + f_{n-2} \quad \text{where} \quad n \in \mathbb{N}$$

In the case of Fibonacci we also require two initial conditions to be able to start the sequence. The two initial conditions are defined as such:

$$f_0 = 0$$
$$f_1 = 1$$

With the recurrence relation for Fibonacci numbers and its two initial conditions, we can calculate any number in the Fibonacci sequence.

As shown above, recurrence relations can tell us how the value of a function grows, based on its input. For example, a function lgd which returns the length of the list it is given, can be defined using a recursive function definition. This definition can be seen here:

$$lgd\ [] = 0$$
$$lgd\ (x :: xs) = 1 + lgd\ xs$$

The computational complexity of lgd can be defined by the recurrence relation T_{lgd}. The definition of T_{lgd} can be seen here:

$$T_{lgd}(0) = 0$$
$$T_{lgd}(n+1) = 1 + T_{lgd}(n)$$

Note the structural similarities between lgd and T_{lgd}.

One of the results of using recurrence relations to define the complexity of functions is that with some further work they can be used to determine the growth rate of a function. This can be done in four major ways, to achieve two different results. Using the substitution method, the recursion-tree method or the master theorem we can determine the asymptotic bounds of the function[5, p. 66]. Or, using an induction proof we can turn the recurrence relation from an open to a closed form. A recurrence relation is in a closed form when it is no longer recursively defined. This will allow us to directly read off its exact complexity and from that determine its asymptotic bound. In short, when doing the conversion from the open to the closed form we aim to replace any occurrences of the function calling itself on the right hand side of the definition.

As an example we will cover how to achieve the closed form using an induction proof, below. However, the substitution method, recursion-tree method and the master theorem are not covered as they are outside the scope of this work.

1.1.2 Proving closed forms of recurrences

We will now show that from the definition of a complexity function, given in open form by recurrence relations, can prove that the function can be given in a closed form, via a proof by induction:

We aim to show that $T_{lgd}(n) = n$. We will do this by using induction on n. To begin with we have the base case that:

$$T_{lgd}(0) = 0$$

If the length of the list is zero, then the complexity is also zero. The inductive case of T_{lgd} is:

$$T_{lgd}(n+1) = 1 + T_{lgd}(n)$$

We are going to assume that it holds for n and from that, try to prove that it holds for $n + 1$. As per the inductive hypothesis we have that:

$$T_{lgd}(n) = n$$

We can then substitute n in the place of $T_{lgd}(n)$ in the inductive step:

$$T_{lgd}(n+1) = 1 + n$$

And then lastly, we have that

$$1 + n = n + 1 \qquad \text{by commutativity}$$

This gives us:

$$T_{lgd}(n+1) = n + 1$$

To summarise, the open form recurrence for the length function is:

$$\begin{aligned} T_{lgd}(0) &= 0 \\ T_{lgd}(n+1) &= 1 + T_{lgd}(n) \end{aligned}$$

And from the example above we can now also show that the closed form is:

$$T_{lgd}(n) = n$$

1.2 Analysing the run time complexity of higher order functions

In this section we will explore why it can be more difficult to determine the run time complexity of higher order functions, as opposed to first order functions. Higher order functions are functions that as one of their arguments take another function or themselves return another function. Common examples of this are functions such as `map` and `fold`. First order functions are any functions that do *not* either take a function as one of their arguments, or do not return another function.

It can be difficult to accurately determine the exact run time complexity of a higher order function, as its run time can in general depend on the run time of the function it receives as an argument. While it is possible to determine how the function taken as an argument contributes to the total complexity, this is hardly useful when attempting to certify the run time complexity of the higher-order function independently of its arguments.

In fig. 1.1 is a common higher order function, `map`. `map` takes a function and a list and applies the function to every element of the input list and in the end it returns a new list.

```
map : ∀ {A B : Set} → (A → B) → List A → List B
map f []       = []
map f (x :: xs) = f x :: map f xs
```

Figure 1.1: An example of a higher order function, `map`

The complexity of `map`, on its own, is clearly $\mathcal{O}(n)$, where n is the number of elements in the list. However, if we have to account for f, the function that `map` takes as an argument, the complexity of the whole is now dependent on the complexity of f. Furthermore, imagine the list we give `map` is a list of lists and f is another higher order function. It becomes necessary to do an in depth complexity analysis of f before we can give any guarantees for the complexity of `map`.

However, the above example only accounts for higher order functions that can take any arbitrary function as an argument. Sometimes when higher order functions are used we can still give guarantees about their complexity. Higher order functions are often used in a specialised manner, where we know exactly what function they take as an argument and what kind of input they will be dealing with.

```
maplength : {A : Set} → {n m : ℕ} → Vec (Vec A n) m → Vec ℕ m
maplength [] = []
maplength (x :: xs) = map lgd (x :: xs)
```

Figure 1.2: An example of map applying lgd to a vector

The example code in fig. 1.2 shows a function `maplength` that defines a specialised version of `map` that always applies `lgd` to the vector it is given. Because of this constraint it is now very simple to analyse the complexity of map, as we already know the complexity of lgd, which is then multiplied by the size of the input vector.

1.3 Analysis of existing work

This section will present three papers of related work and discuss the differences of the individual approaches to inform an expanded problem statement. For each of these papers

we aim to understand their methodology, specifically in how they use a rich type system to determine the complexities of functions. In the end of the section we will compare the three papers, by evaluating what they might have in common, where they differentiate and what drawbacks each of them have.

1.3.1 TiML: A Functional Language for Practical Complexity Analysis with Invariants

Wang et al. create a functional language named TiML (Timed ML)[6]. TiML is an ML-like language, where it is possible to annotate the time-complexity of functions directly in the type. This is helped by use of indexed types where the programmer can annotate their types with a flexible value indicating size. The annotation is done manually by the programmer; the type checker will then generate recurrence relations for the annotated functions. The verification conditions of these recurrence relations are then discharged by an SMT-solver.

```
fun length_int [a] {n : Nat} (l : list a {n})
   return nat {n} using $(2 * n) =
    case l of
          [] => 0
    | _ :: xs => 1 + length xs
```

Figure 1.3: The definition of the length function in TiML[6]

In fig. 1.3 an example of the `length_int` function, defined in TiML, is shown. Compared to its counterpart in Standard ML, it is only slightly more verbose, mainly because of the requirement that the arguments must be explicitly defined. Another addition to this example, compared to Standard ML, is the time annotation. The time annotation is done by using the function `using` to specify the time bounds of the function. For `length_int` the time bounds is specified as `$(2 * n)`. The `$` unary operator converts a natural number into a measure of time.

1.3.2 Computer checked recurrence extraction for functional programs

Hudson attempts automatic extraction of recurrence relations[7]. The method works by defining a function f in a simple lambda calculus defined in Agda, this lambda calculus is called the *source language*. Then translating f to a *complexity language*, that is similar

to the source language, but is also annotated with information about complexity. In the complexity language recurrence relations are extracted by interpreting types as preorders and terms as monotone maps between preorders. This gives an approximation of the upper bounds of the complexity of the algorithm.

The cost is calculated during the translation from the source language in to the complexity language. It is built directly in to the types of the complexity language, and has no presence in the source language.

Hudson automatically extracts recurrence relations from a small lambda calculus, however, parsing the resulting recurrences is not trivial.

Reusing our previous example of a length function, now implemented in the λ-calculus source language, looks like this:

```
lam (listrec (var i0) z (suc (force (var (iS (iS i0))))))
```

This is then converted into a monotone function for the recurrence relation. For the sake of readability, the included proof of monotonicity are replaced by … in the example seen in fig. 1.4, as it can be several hundred lines long.

```
monotone
(λ x → 0 ,
    monotone
    (λ p₁ →
        (lrec p₁ ( 1, 0) (λ x xs x₃ → S (fst x₃) , S (snd x₃)))
    (λ a b c → … ))
(λ x y z₁ → … )
```

Figure 1.4: The monotone function for the recurrence relation.

The recurrence relation can be hard to see and requires some knowledge of how `lrec` works. The function has three arguments. **(p1)** is the first argument of the function and it is an input list. The second argument **(1, 0)** is the base case value for the empty list. Its third argument is a function that is given three arguments, the head of the list, the tail of the list, and the result of the inner call. The return value of the function is a pair where the first element is the runtime cost, and the second element is the result of the function.

Knowing this, the recurrence relation now becomes visible in the underlined parts. Or in a more traditional formatting:

$$\begin{aligned} T([]) &= 1 \\ T(x :: xs) &= 1 + T(xs) \end{aligned}$$

Do note, however, the argument of T is a list, instead the size of the input.

1.3.3 Lightweight semiformal time complexity analysis for purely functional data structures

Danielsson introduces a technique, and a library that implements it, that allows the user to reason about the runtime complexity of a function directly in its type[3]. Reasoning about simple recursive algorithms is often trivial, but to create efficient algorithms in functional programming languages with immutability, laziness is often required and this makes analysing complexity a much harder job.

This reasoning is done by a Thunk monad which is annotated with the number of computational steps required to produce its value. Thunk is used the accumulate the overall cost of a function, based on the cost of its subfunctions.

The annotation is done by the programmer by inserting a ✓for every function clause, lambda abstraction, etc.

An example of this annotation can be seen in fig. 1.5.

```
lgd : {A : Set} → {n : ℕ} → List A → ℕ
lgd [ ] = 0
lgd (_ :: xs) = 1 + lgd xs

lgd’ : {A : Set} → {n : ℕ} → Vec A n → Thunk n ℕ
lgd’ [] = return 0
lgd’ (_ :: xs) = lgd’ xs »= λ y → ✓ return (suc y)
```

Figure 1.5: An unannotated length function and an annotated one, using Danielsson’s approach.

1.3.4 Comparison

In this section we will compare and summarize the three papers and evaluate three characteristics of how their contributions are constructed, the three characteristics that will be analysed are as follows;

- How are the type systems utilized?
- What cost model is used?

- What analysis method is used?

Wang et al. state that the ability of the SMT solver to discharge the verification conditions may break easily if the syntax of the conditions are not entirely correct.

The index language for their types is restricted to natural numbers. It cannot specify, for example, `map` with an argument function whose running time depends on the sizes of individual list elements. This limits the kind of higher order functions that can be defined to ones where the run time of the argument function, of the higher order function, is constant across the function domain.

Danielsson creates a library that allows for a programmer to annotate code and receive a certain guarantee for its performance. It is clear that one of the drawbacks of this annotation method is that some of the original structure of an algorithm becomes somewhat obscured. This is because the annotation changes the return type of the annotated function, as well as inserting multiple different key words in the middle of it.

The main limitation of this library is that any algorithms the user wishes to analyse must be manually annotated with ✓and `return` for the complexity to be correctly calculated. This process may be tedious and prone to errors, as leaving out even a single ✓would change the end result. Any change that changes the runtime, even by a difference of one, will have an different type signature, which will then break any code the library is used for

Hudson creates an annotated complexity language expression from the source expression and while that expression may be equivalent to the recurrence relation of the function the source is based on, it takes familiarity with how the whole process works to see this resemblance.

Another significant limitation to the practicality of Hudson's approach is the fact that it works not with Agda as a whole, but with a source language, that is functionally only a subset of Agda. This source language is a simple lambda calculus, implemented in Agda. This also limits the readability of both the source and complexity languages, as the simplicity of the languages makes implementation of certain algorithms more cumbersome than in Agda proper.

In summary we will now evaluate these three papers based on the three different criteria we judge to be valuable in creating an objective analysis of where the three papers differ.

With Wang et al. and Danielsson the type system is used to carry information about the complexity of the function that is involved, whereas Hudson utilizes a lambda calculus and uses the translation to a complexity language to reason about that aspect.

W.r.t. the cost model, Wang et al. and Danielsson both count beta reductions, Danielsson by annotating every call and abstraction manually, and Wang et al. with annotation of the types rather than computations. Hudson counts rewrites from left to right.

The complexity language from Hudson and TiML from Wang et al. both use recurrence relations to reason about the complexity of the constructed functions. Where Danielsson instead lets the annotation and the type system of the language determine the exact complexity of the function. None of the studied contributions manage to reason about time complexity without some significant alterations of the functions they analyse. With Wang et al. and Danielsson the cost is included within the syntax of the solution language, and with Hudson a translation to a complexity specific language is performed. As such, any of the reviewed solutions would require some alteration of the source algorithm to be performed. One overarching tendency observed in all of the studied work is that the syntax is altered in a way that is far from standard and harder to read, this leads to the following *expanded* problem statement:

How close can a solution that certifies complexity be to a source language like Agda, in terms of syntax?

1.4 Introducing signatures

In this section we will introduce the concept of signatures. A signature[8] is a triple consisting of:

- A set of function symbols S_{func}
- A set of relation symbols S_{rel}
- An arity function $ar : S_{func} \cup S_{rel} \to \mathbb{N}$, that describes the number of parameters of symbols in the two sets above.

There is no requirement that a symbol takes any arguments, thus constants exist as functions that simply take zero arguments. From a signature, a language of well-formed sentences then arises, where every symbol in the sentence has the correct number of arguments.

One such example would be unary numbers described by the signature:

$$\langle \{zero, suc\}, \{=, <\}, \{zero \mapsto 0, suc \mapsto 1, = \mapsto 2, < \mapsto 2\} \rangle$$

From this, the entirety of arithmetic can be described.

This concept can be reversed, such that we consider each area of mathematics to have a signature, the symbols needed to describe it. Because a class of related algorithms exhibit the property that they utilize the same signature.

For example, sorting algorithms:

$$\langle \{ ::, \mathrm{cond} \}, \{ \leq \}, ar \rangle$$

With those few constructs, every comparative sorting algorithm can be implemented, using only *list concatenation*, *comparison*, *condition* and a recursive construct. This means that the signature of a sorting algorithm, the few constructs shown above, is a good indicator of what will eventually have an effect on its computational complexity. In section 3.1.1 we will discuss this further, and go in to detail about exactly how this could be done.

1.5 Final problem statement

Based on the subject of signatures, introduced in section 1.4, we can further refine our problem statement:

Is it the case that certifying the time complexity of an algorithm, by only annotating functions, without heavily changing the readability or the syntax of the algorithm is a good approach?

In this revised problem statement, we keep the focus of trying to preserve the readability and syntax of the algorithms we annotate. However, since signatures proved to be an interesting subject with some useful possibilities, they have been incorporated in to the problem statement.

The rest of this report is structured as follows: Chapter 2 presents the dependently typed programming language Agda and shows how proofs of function properties can be created in that language. Chapter 3 lays out the implementation details of a complexity certified matrix library. Chapter 4 shows how our matrix library can be used to certify the complexity of the CYK algorithm. Chapter 5 concludes on the report and discusses future work.

Chapter 2

An Agda centric introduction to dependent types

In this chapter, we will provide an introduction to dependent types and some other related typing concepts. This will primarily be done using Agda, a dependently typed programming language. Then we will demonstrate how proofs of properties can be created in Agda, by proving the commutative nature of pairwise addition, and by inductively proving the associativity of multiplication. The `rewrite` functionality to shorten proofs will also be explained.

2.1 Introducing the Agda programming language

In this section Agda will be briefly introduced, in order to establish a common ground of understanding for more advanced usages later on in section 2.2.

Agda is a dependently typed functional language, developed by Ulf Norrell [9]. Syntactically, it is similar to Haskell, but the type system has been replaced.

With dependent types the process of type inference becomes undecidable, but, in return, a program in Agda has different, stronger, characteristics such as being able to capture λ-terms in the types of functions and perform computation at the type level.

Agda features generalized algebraic data types, otherwise known as GADTs in the form of inductive families of types. The difference between GADTs and the more common algebraic data types is the ability to explicitly specify the type of a type constructor, which allows stricter guarantees about their values a feature that is both useful in general and for writing proofs.

With dependent types, types can depend on terms. One possibility this unlocks is lists, with size, called Vectors in Agda parlance. One possible definition of such a Vector can be

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

Figure 2.1: Inductive definition of natural numbers in Agda

found in fig. 2.2.

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Figure 2.2: Definition of the Vec type

With this type, it becomes possible to annotate any function operating on vectors to specify the relation between the sizes of any arguments and its return type.

For example, the zip function, which merges two vectors containing elements of type A and type B respectively, into a vector of pairs of A and B, can now specify that its arguments have the same length and the vector it returns will match the length of its input. And if the type checker cannot statically prove this, it will fail at compile time.

This example also introduces another Agda feature, implicit arguments. Arguments surrounded by curly brackets, such as $\{n : \mathbb{N}\}$, causes the type checker to search for a value required for the types to match. These arguments are usually not specified when the function is called, but it occasionally happens that the type checker is not able to infer the value automatically.

With dependent types, types can depend on arbitrary programs. Since determining if an arbitrary program will halt is known to be hard, and knowing that your type checker will eventually terminate is desirable, Agda has some restrictions to facilitate termination checking. These restrictions are as follows: functions written recursively in Agda must be either primitively, or structurally, recursive. A structurally recursive call must be a subexpression of the argument to the call. An example of a structurally recursive function would be the function fib, seen in fig. 2.3. This Fibonacci sequence relies on the inductive definition of natural numbers previously introduced, and shows how Agda is able to guarantee termination in a case like this, because the recursive case relies on subexpressions,

where the natural number is smaller, and the inductive nature of the numbers ensures that it will reach zero.

```
fib : ℕ → ℕ
fib zero = zero
fib (suc zero) = suc zero
fib (suc (suc n)) = (fib n) + (fib (suc n))
```

Figure 2.3: Structurally recursive fib in Agda

Agda allows for product types in the shape of records, such as Pair seen defined in fig. 2.4, providing two named fields, fst and snd.

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

Figure 2.4: Agda Pair Definition

The underscores are an example of Agda's *infix* operators. In such an operator, each _ represents a parameter, which when the operator is invoked, are naturally placed between the remaining syntax. In this case, _,_ defines an operator, where two parameters separated by a comma are used to create a Pair.

A major feature of a dependently typed functional programming language is that it allows proving propositions in a manner that is similar to how mathematical proofs are traditionally produced, which is aided by its stronger type system.

2.2 Structure of proofs in Agda

In this section we will present how proofs are constructed in Agda. We will start by introducing the function $+^p$ that adds the elements of two Pairs defined in section 2.1 and then prove that it is commutative. We will then recreate the proof, using the rewrite keyword present in Agda, to produce a significantly shorter proof. After this we will then expand

on the complexity of proofs, by introducing induction and proving the associativity of multiplication.

We will start by defining a new operator, pairwise addition, which we presume to be commutative: `_+ᵖ_` this operator takes two parameters of the type `Pair ℕ ℕ`. The operation is then defined as adding both `fst` fields together as the resulting `fst` field, and both `snd` fields together as the resulting `snd` field. The definition of this operator can be seen in fig. 2.5.

```
_+ᵖ_ : Pair ℕ ℕ → Pair ℕ ℕ → Pair ℕ ℕ
(afst , asnd) +ᵖ (bfst , bsnd) = (afst + bfst) , (asnd + bsnd)
```

Figure 2.5: Definition of `Pair` addition

In a case like this it is straightforward to inspect the operation manually and conclude that it must be commutative, given that we know that `+` is commutative, and `+ᵖ` is an expansion of that operation on a `Pair`. We now present a formal proof of the commutativity property using Agda.

We start the proof with a proposition, that for all pairs of natural numbers, we show that adding the *a* pair to the *b* pair, is the same as adding the *b* pair to the *a* pair or, in Agda:

$$\forall\ (a\ b : \mathtt{Pair}\ \mathbb{N}\ \mathbb{N}) \to a +^{p} b \equiv b +^{p} a.$$

To prove this proposition, we must provide a step by step equivalence, that can use other proofs as justification to rewrite the original statement to the goal statement.

In this specific case, we start with:

$$(afst + bfst)\ ,\ (asnd + bsnd)$$

And provide a justification in the `≡⟨⟩` brackets. The proof itself can be seen in fig. 2.6. The justification used here is `cong`, short for congruence, which takes a function, and a piece of evidence, that together show the relation is preserved for any function and allows for the substitution of equivalent sub expressions. By using a proof that addition is commutative, here named `+-comm`, as evidence, we show that we can swap the arguments, `afst` and `bfst`. Repeating the process for `asnd` and `bsnd`, we have shown that `Pair` addition with the `+ᵖ` operator is commutative.

By utilizing the `rewrite` keyword it becomes possible to write proofs for propositions in a significantly less verbose form, since `rewrite` simply requires the propositions used as

```
+ᵖ₁-comm : ∀ (a b : Pair ℕ ℕ) → a +ᵖ b ≡ b +ᵖ a
+ᵖ₁-comm (afst , asnd) (bfst , bsnd) =
  begin
    (afst + bfst) , (asnd + bsnd)
  ≡⟨ cong (λ x → (x , (asnd + bsnd))) (+-comm afst bfst) ⟩
    (bfst + afst) , (asnd + bsnd)
  ≡⟨ cong (λ x → ((bfst + afst) , x)) (+-comm asnd bsnd) ⟩
    (bfst + afst) , (bsnd + asnd)
  ∎
```

Figure 2.6: Proof that Pair addition is commutative written out in steps

evidence in the long proof, in order to reason out a full proof. A rewrite proof for _+ᵖ_ can be seen in fig. 2.7 where the same pieces of evidence are utilized, in the same order, but in a more succinct fashion.

```
+ᵖ₂-comm : ∀ (a b : Pair ℕ ℕ) → a +ᵖ b ≡ b +ᵖ a
+ᵖ₂-comm (afst , asnd) (bfst , bsnd) rewrite +-comm afst bfst
| +-comm asnd bsnd = refl
```

Figure 2.7: Proof that Pair addition is commutative written using rewrite

Proofs of an inductive nature can also be created. Consider the proof for multiplication associativity. Associativity is the property that the order of operations is irrelevant, like so:

$$(n * m) * p = n * (m * p)$$

or, in Agda terms:

$$\forall\ (m\ n\ p : \mathbb{N}) \to (m * n) * p \equiv m * (n * p)$$

As with a traditional proof by induction, the first step to the method is the base case, where an argument is zero, as seen in fig. 2.8, in this case it is straightforward to show that the operations are the same, without providing any justification, and indeed a manual inspection yields that any calculation will result in zero. The inductive case is the proof that if the

```
*-assoc₁ : ∀ (m n p : ℕ) → (m * n) * p ≡ m * (n * p)
*-assoc₁ zero n p =
  begin
    (zero * n) * p
  ≡⟨⟩
    zero * (n * p)
  ∎
```

Figure 2.8: Base case for the proof of associativity of multiplication

property holds for one natural number, then it will hold for that number, plus one.

In the Agda version of the inductive case, as seen in fig. 2.9, the initial equation is:

$$\mathtt{suc}\ m * n * p$$

Which is shown without additional evidence to be the same as:

$$(n + (m * n)) * p$$

Agda accepts this due equivalent to the definition of multiplication of natural numbers.

This follows from the fact that adding one to m would be equivalent to adding $n * p$ to the original equation, but ideally we would want to show that fully as such:

$$suc\ m * n * p = n * p + (m * n) * p$$

To achieve this, we use the proof for multiplication distribution as justification which states:

$$\forall\ (m\ n\ p\ :\ \mathbb{N}) \rightarrow (m + n) * p \equiv m *p + n * p$$

That facilitates rewriting the equation to:

$$n * p + (m * n) * p$$

As the final step of the proof, a congruence for plus is used as justification to show that $n * p + (m * n) * p$ can be rewritten to:

$$(n * p) + (m * (n * p))$$

Which is what we wanted to show.

```
*-assoc₁ (suc m) n p =
  begin
    suc m * n * p
  ≡⟨⟩
    (n + (m * n)) * p
  ≡⟨ *-distrib-+ n (m * n) p ⟩
     n * p + (m * n) * p
  ≡⟨ cong (_+_ (n * p)) ( *-assoc₁ m n p) ⟩
    (n * p) + (m * (n * p))
  □
```

Figure 2.9: The inductive case for the proof of associativity of multiplication

As with the previous example, the proof for multiplication associativity can also be significantly abbreviated by using the rewrite as seen in fig. 2.10. The base case is simple enough that it is reflexively equivalent, and therefore does not require rewriting at all. The inductive case uses the same pattern as shown in the last example, where the justifications used between the steps are provided and nothing else.

```
*-assoc₂ : ∀ (m n p : ℕ) → (m * n) * p ≡ m * (n * p)
*-assoc₂ zero n p = refl
*-assoc₂ (suc m) n p rewrite *-distrib-+ n (m * n) p
| *-assoc₂ m n p = refl
```

Figure 2.10: Multiplication associativity rewrite proof

To summarize, Agda allows for the creation of proofs to propositions by providing clear and well founded justifications and evidence for each step in the process of deduction. Often relying on previous proofs creates a foundation of logic that can be gradually expanded with more proofs. The rewrite keyword provides an efficient way to construct proofs that rely on other proofs.

Chapter 3

Certified complexity of matrix multiplication

In this chapter we will describe the implementation of *complexity signatures* as described in section 1.4. We will start by describing the mechanism itself. We will then expand on the functionality by creating a series of utility functions to ease the conversion of a function to a complexity annotated one. Following that, we will use this mechanism to convert vector append, _++_ , to a complexity annotated version, _++′_ and prove its complexity. And finally we will present key examples from the library of complexity annotated matrix functions and their associated proofs.

3.1 Implementing complexity signatures in Agda

The main goal behind complexity signatures is embedding computational cost into the code, while operating on what is recognizably the same code. We do this by taking the signature of a function (see: section 1.4) and replacing every function it calls with a complexity annotated variant, to also turn this function into a complexity annotated variant. Starting with type constructors, this can then be done bottom up to eventually complexity annotate an entire codebase.

We indicate that a function is complexity annotated with a prime (′).

Once we have done this, we can then count up the total cost for evaluating a function, which we can then formally reason about, and in a dependently typed language such as Agda, formally prove.

3.1.1 Complexity signature mechanism

The underlying mechanism behind complexity signatures is changing a given value, a, to be associated with its own cost, by putting it in a pair (a, n), where n is the time it has taken to produce this value. Functions operating on those values then sum the time taken to produce its parameters and increment it by some cost, based on how expensive the function itself is.

For this, we introduce the monadic type `Timed`, representing a value a of type A produced in n time.

```
data Timed (A : Set) : Set where
  time : ℕ → A → Timed A
```

With our goal of the code remaining visually similar, it is impractical to constantly construct and deconstruct instances of `Timed` in the code. We therefore introduce a few pieces of abstraction.

First we introduce the prime operator (`_′`). It converts a value into one produced in zero time. The implementation calls `time` and works as the conventional monad function `return`.

```
_′ : {A : Set} → A → Timed A
x ′ = time 0 x
```

We also define the bind operator (`_»=_`).

```
_»=_ : {A B : Set} → Timed A → (A → Timed B) → Timed B
_»=_ (time n x) fn = fn x ′′′ n
```

Converting single values with `_′` works for simple problems, but not for more complex programs. For this we introduce the `lift` function. It converts a normal function, into a function that takes timed arguments and produces its result in the time of its argument plus one, $T_{lift} = T_A + T_B + 1$. Thereby counting the number of function calls.

```
lift : {A B C : Set} → (A → B → Timed C) → (Timed A → Timed B → Timed C)
lift = lift″ ∘ lift′
```

This lift function itself consists of two other functions, `lift″` and `lift′`

The second of those functions, `lift′` serves approximately the same purpose as the prime operator, but operates on functions converting them into functions returning a value produced in $\mathcal{O}(1)$. It does not modify the arguments.

The first function, `lift″` converts a function which does not take timed arguments, but returns one, into a function both taking and returning timed types. This is particularly

helpful when implementing functions which need to deconstruct their arguments, as they otherwise have to deconstruct the `Timed` monad first.

The complete `lift` and `lift′` is usually only used for type constructors, such as cons (_::_).

But in some cases, such as integer addition, which is formally defined inductively in Agda to make writing proofs about its properties easier, we do directly lift it as a constant time operation since it compiles to the Haskell datatype Integer[10] where addition it a constant time operation in most cases.

3.1.2 Complexity signature

A signature for a library of matrix functions would have the signature:

$$\langle \varnothing, \{\oplus\ \ominus\ \otimes\ \circledast\ \ /\ |\ ^{t}\ \times\}, \ ar\rangle$$

With functions for mathematical operations on pairs of matrices, from left to right, *addition / subtraction*, *multiplication*, as well as *scalar multiplication*, *row- and columnwise attachment*, *transposition* and *creation*. For brevity we are not defining any comparisons on matrices, and thus the relation set of relation symbols is the empty set.

A *complexity* variant of this signature would then be:

$$\langle \varnothing, \{\oplus'\ \ominus'\ \otimes'\ \circledast'\ \ /'\ |'\ ^{t'}\ \times'\}, \ ar\rangle$$

3.1.3 An example with complexity signatures

With these elements defined, we can now introduce an example of a complexity annotated function. For this example, we will be creating a complexity annotated append function (_++′_).

First step is to redefine the constructor for the `Vec` type, _::_, to a complexity annotated variant.

```
_::″_ : {A : Set} → {n : ℕ} → Timed A → Timed (Vec A n) →
        Timed (Vec A (suc n))
_::″_ = lift _::_
```

With this defined, we can implement the actual append function. Ideally we would rewrite the append function, replacing _::_ with _::″_ and have everything work:

```
(x :: xs) ++′ y = x ::″ (xs ++′ y)
```

But this does not type check, as x is of the type A, not `Timed` A. This is solved by converting x to a value produced in 0 time with the _′ operator.

```
_++′_ : {A : Set} → {n m : ℕ} → Vec A n → Vec A m →
          Timed (Vec A (n + m))
[] ++′ y = y ′
(x ∷ xs) ++′ y = x ′ ∷″ (xs ++′ y)
```

After this, we might want to also create a version of `_++′_` which takes complexity annotated arguments too. In order to do this, we reuse the `lift` function.

```
_++″_ = lift _++′_
```

With the code written, we also want a proof of its complexity. Our immediate intuition is that the complexity is equivalent to the length of the first vector. We express this in Agda with the type:

```
++′-cost-proof : ∀ {A n m} (x : Vec A n) → (y : Vec A m) →
          cost (x ++′ y) ≡ n
```

Which we then show via a proof by induction. The base case for $n \equiv 0$ is trivially proven.

```
++′-cost-proof [] y = refl
```

For the inductive case, the proof is more involved. We assume the induction hypothesis that:

$$T_{++'}(n) = n$$

We start with the complexity of the function in a format that Agda accepts. Using our `lift″-proof` as justification to rewrite it to a new form which proves that the complexity of two lifted functions ($x\,'$ and x_1 ++′ y) can be combined.

```
++′-cost-proof {A} (x ∷ x₁) y =
  begin
    cost((lift _∷_) (x ′) (x₁ ++′ y))
  ≡⟨ lift″-proof _∷_ (x ′) (x₁ ++′ y) ⟩
    suc (cost (time 0 x)) + cost (x₁ ++′ y)
```

And then finally, invoking the proof recursively

```
≡⟨⟩
  cong suc (++′-cost-proof x₁ y)
```

This example makes a structure clear; complexity annotated libraries and certification of their complexity can be constructed in a process that can be generalized as:

1. Identify the simplest base functions of a set of functions in a given signature
2. Convert a base function to a `Timed` complexity annotated function with an adequate conversion, such as _′ or `lift`

3. Prove that the complexity of the base functions stays within some bound
4. Use the base functions as elements in the construction of composite complexity annotated functions.
5. Prove that these composite functions stay within some bound, using the proofs from the base functions as justification.
6. Repeat the process of constructing and proving composite complexity annotated functions with existing functions until the entire signature is covered.

3.2 Matrix library

This section will detail the implementation of our complexity certified matrix library giving key examples of functions and proofs. The entirety of the library and proofs can be found in appendix C and appendix D respectively.

A list of functions that will be used throughout this section can be found in table 3.1.

3.2.1 Matrix functions

Along with the already presented _++′_ we construct a Timed version of lookup, seen in fig. 3.1, based on the standard implementation of lookup. Where the _′ turns a regular value into a timed tuple, _″ takes a timed value and increments its recorded complexity by one. lookup′ does this for every element in its search space.

```
lookup′ : ∀ {n} → {A : Set} → Vec A n → Fin.Fin n → Timed A
lookup′ (x :: xs) Fin.zero = (x ′) ″
lookup′ (x :: xs) (Fin.suc i) = (lookup′ xs i) ″
```

Figure 3.1: lookup′ function, locates element in vector, from an index

lookup′ is used as a composite element to construct more advanced functions. For example matrixToColumn′, seen in fig. 3.2, which extracts a column from a matrix in order to manipulate that column for operations like dot product (·).

The whole library is constructed like this, iteratively building and proving more complex functions by using already made functions and proofs as composite elements in new functions. The most complex function created and proved in the library is matrix multiplication. This function exists as a base function, MatrixMultiplicationBase, seen in fig. 3.3, which is parameterized with a “dot product” argument. This way, defining matrix

tabulate′ : (Fin.Fin n → Timed A) → Timed (Vec A n)	Creates a Timed vector of length n by calling the first argument n times with a value from 0 to $n-1$.
matrixToRow′ : ($n \times m$) → Fin.Fin n → Timed (Vec A m)	Sister function to matrixToColumn′. Takes a matrix of type A, n rows and m columns, a finite number that is at most n and extracts a single row at that index in the matrix, as a vector of type A and length m.
lookup′ : Vec A n → Fin.Fin n → Timed A	Takes a vector of length n, and a finite number that is at most n and returns the element at that index in the vector.
matrixToColumn′ : ($A \triangleright n \times m$) → Fin.Fin m → Timed (Vec A n)	Takes a matrix of type A, n rows and m columns, a finite number that is at most m and extracts a single column at that index in the matrix, as a vector of type A and length n.
MatrixMultiplicationBase : (A : Set) → (dot : {k : ℕ} → Timed (Vec A k) → Timed (Vec A k) → Timed A) → {n m j : ℕ} → $A \triangleright n \times m$ → $A \triangleright m \times j$ → Timed ($A \triangleright n \times j$)	Base implementation of matrix multiplication, parameterized by a dot function, which it applies to two matrices of appropriate type and size.
_•′_ : {n : ℕ} → Vec ℤ n → Vec ℤ n → Timed ℤ	Takes two vectors with integer elements and performs the dot operation on them.
_•″_ : {n : ℕ} → Timed (Vec ℤ n) → Timed (Vec ℤ n) → Timed ℤ	Version of _•′_ also taking timed parameters.
_⊗_ : {n m j : ℕ} -> $n \times m$ → $m \times j$ → Timed($n \times j$)	Version of MatrixMultiplicationBase parameterized with _•″_, facilitates matrix multiplication for matrices with elements of integers.

Table 3.1: Matrix functions

```
matrixToColumn′ : {A : Set} → {n m : ℕ} → (A ▷ n × m) →
                            Fin.Fin m → Timed (Vec A n)
matrixToColumn′ [] i = []′
matrixToColumn′ (mat :: mats) i = (lookup′ mat i) ::″
                            matrixToColumn′ mats i
```

Figure 3.2: `matrixToColumn′` function, extracts column from matrix

multiplication for any arbitrary types only requires defining the dot product for said type. Two variants were created, one for multiplication of integers and one for us in the CYK algorithm, operating on symbols where the dot product is instead defined based on working with sets, which will be covered in section 4.1.

```
MatrixMultiplicationBase : (A : Set) → (dot : {k : ℕ}
  →  Timed (Vec A k)
  → Timed (Vec A k) -> Timed A)
  → {n m j : ℕ} → A ▷ n × m → A ▷ m × j → Timed (A ▷ n × j)
MatrixMultiplicationBase A dot left right =
  tabulate′ λ n →
    tabulate′ λ m →
      dot (matrixToRow′ left n) (matrixToColumn′ right m)
```

Figure 3.3: `MatrixMultiplicationBase` function, requires parameterization with dot function

This is then combined with a `dot` product implementation to create a specific implementation of matrix multiplication. In our case for the general matrix library, the dot product works on integers, this being `_•″_`, seen in fig. 3.4.

The iterative nature of these functions means that the value carried within the `Timed` monad in each of them is representative of the time complexity they exhibit.

3.2.2 Matrix complexity lemmas and their proofs

This section will present and explain complexity proofs and essential lemmas from each of the complexity certified matrix functions presented in section 3.2.

```
_•′_ : {n : ℕ} → Vec ℤ n → Vec ℤ n → Timed ℤ
[] •′ [] = 0ℤ′
(x ∷ xs) •′ (y ∷ ys) = (x ′ *″ y ′) +″ (xs •′ ys)

_•″_ : {n : ℕ} → Timed (Vec ℤ n) → Timed (Vec ℤ n) → Timed ℤ
_•″_ = lift″ _•′_

_⊗_ : {n m j : ℕ} -> n × m → m × j → Timed(n × j)
_⊗_ = MatrixMultiplicationBase ℤ _•″_
```

Figure 3.4: Matrix dot product, `_•′_`, lifted dot product `_•″_`, and multiplication, `_⊗_`.

3.2.2.1 `lookup′` lemma and proof

From the definition of `lookup` we know that at most one call is performed for each element in the vector `lookup` receives as an argument, because the finite number of the index is defined as being less than the length of the list. Intuitively, this means that the complexity of `lookup′` is $\mathcal{O}(n)$ which we prove with the help of `lookup-cost-lem1`, seen in fig. 3.5, which shows that the time complexity of `lookup′` on a list is equivalent to the successor of the complexity of `lookup′` on the tail of the list.

The full proof, `lookup-cost-proof`, seen in fig. 3.6, uses a `rewrite` of the lemma which it then invokes, recursively, until the length of the list is zero.

```
lookup-cost-lem1 : {n : ℕ} → {A : Set} → (x : A)
  → (xs : Vec A n) → (f : Fin.Fin n)
  → cost (lookup′ (x ∷ xs) (Fin.suc f)) ≡ suc (cost (lookup′ xs f))
lookup-cost-lem1 x xs f with lookup′ xs f
...                          | (time n y) = refl
```

Figure 3.5: `lookup-cost-lem1` proving the exact time complexity of `lookup′`

3.2.2.2 `matrixToColumn′` proof

This implementation of `matrixToColumn′` also uses the complexity annotated `Vec` constructor `_∷″_` and invokes itself recursively, for that reason we can extrapolate that the time complexity of the function must be $\mathcal{O}(n \cdot m)$ where n is the already proven complexity of `lookup′`,

```
lookup-cost-proof : {A : Set} → {n : ℕ} → (x : Vec A n)
  → (f : Fin.Fin n) → cost (lookup′ x f) ≤ n
lookup-cost-proof {A} {.(suc _)} (x ∷ x₁) Fin.zero = s≤s z≤n
lookup-cost-proof (x ∷ xs) (Fin.suc f)
        rewrite lookup-cost-lem1 x xs f =s≤s (lookup-cost-proof xs f)
```

Figure 3.6: `lookup-cost-proof` proving the $\mathcal{O}$ time complexity of `lookup′`

and *m* is the width of the input matrix. This is proven in `matrixToColumn′-cost-proof`, seen in fig. 3.7, which uses a rewrite `lift-cost-proof` on `_∷_` to prove that the inductively constructed `Vec` will have the combined cost of both of its operands. The proof also builds on the already established `lookup-cost-proof` as justification to prove recursively that the cost stays at or below the allowed bound.

```
matrixToColumn′-cost-proof : {n m : ℕ} → (x : n × m) →
    (i : Fin.Fin m)→ cost (matrixToColumn′ x i) ≤ n * suc m
matrixToColumn′-cost-proof {n} {m} [] i = z≤n
matrixToColumn′-cost-proof (x ∷ x₁) i rewrite
    lift-cost-proof _∷_ (lookup′ x i)
    (matrixToColumn′ x₁ i) = s≤s ( +-mono-≤ (lookup-cost-proof x i)
    (matrixToColumn′-cost-proof x₁ i))
```

Figure 3.7: `matrixToColumn′-cost-proof` proving the complexity of column extraction from a matrix

3.2.2.3 `_•′_` proof

The proof of the complexity of the matrix multiplication operator, `_⊗_`, is a more involved process that requires proofs of complexity for all the underlying functions, the base multiplication, dot operator, `_•′_`, and their associated lemmas.

Starting with `•′-cost-proof` for two vectors of equal length *n* we prove that the complexity of this function is exactly $\Theta(n \cdot 2)$, the base case for two empty vectors is reflexively equal, because $0 = 0 \cdot 2$. For the inductive case we expand the `_•_` function to the two internal functions used to define it, namely `_*_` and `_+_`, which both have a complexity of 1. As with `matrixToColumn′-cost-proof` we once again employ `lift-cost-proof` as

justification to separate the cost expression into two expressions, one for the heads of each vector, and one for the tails. Finally, the cost of the head expression is shown to be exactly 2, and the proof is recursively invoked to show the that hypothesis is true.

Because the other proofs are constructed around upper bound complexity and not exact complexity, a proof, `•′-≤-cost-proof`, seen in fig. 3.8, is created that succinctly proves that if the complexity of `_•′_` is $\Theta(n \cdot 2)$ then it is also $\mathcal{O}(n \cdot 2)$

```
•′-cost-proof : {n : ℕ} → (x y : Vec ℤ n) → cost (x •′ y) ≡ (n * 2)
•′-cost-proof [] [] = refl
•′-cost-proof {suc n} (x :: xs) (y :: ys) =
  begin
    cost ((time 0 x *″ time 0 y) +″ (xs •′ ys))
  ≡⟨⟩
    cost ((lift″ _+′_) (time 0 x *″ time 0 y) (xs •′ ys))
  ≡⟨ lift-cost-proof Int._+_ ((time 0 x *″ time 0 y)) ((xs •′ ys)) ⟩
    suc (cost (time 0 x *″ time 0 y)) + cost (xs •′ ys)
  ≡⟨⟩
    cong (_+_ 2) (•′-cost-proof xs ys)

•′-≤-cost-proof : {n : ℕ} → (x y : Vec ℤ n) → cost (x •′ y) ≤ (n * 2)
•′-≤-cost-proof x y rewrite •′-cost-proof x y = ≤-refl
```

Figure 3.8: `•′-cost-proof` and `•′-≤-cost-proof`, proving the Θ complexity of `•′` and converting the proof to $\mathcal{O}$ complexity

3.2.2.4 `MatrixMultiplicationBase` lemma and proof

From the parameterized nature of our matrix multiplication implementation suggests the creation of an equally parameterized proof. In Agda it is possible to supply proofs as arguments to other proofs, and `matrix-multiply-cost-lem1` utilizes this by requiring a dot operation, along with a proof of the complexity of the dot operation that is used directly to prove this lemma. The matrix multiplication function uses two nested `tabulate′` higher order functions, and this lemma certifies the complexity of the first `tabulate′` with the complexity of the internal functions, `matrixToRow`, `matrixToColumn` and `_dot_` that `tabulate′` takes as an argument.

The proven complexity of `tabulate′` is $\mathcal{O}(n \cdot q)$ where n is the number of times `tabulate′` calls the function it takes as a first argument, and q being the complexity of

the function it calls. That means that intuitively the complexity of the lemma should be $T_{tabulate} * (T_{dot} + T_{matrixToRow} + T_{matrixToColumn})$ and we prove that using the aforementioned proof of the complexity of tabulate′, tabulate′-cost-proof and a proof, •″-extended-cost-proof, that proves the complexity of the internal functions combined.

```
matrix-multiply-cost-lem1 : {n m j : ℕ} → {A : Set}
  → (_dot_ : {k : ℕ} → (x y : Vec A k) → Timed A)
  → (p : ℕ)
  → (dot-cost-proof : {k : ℕ} → (x y : Vec A k) → cost (x dot y) ≤ k * p)
  → (m₁ : A ▷ n × m) → (m₂ : A ▷ m × j) → (n₁ : Fin n )
  → cost ( tabulate′ λ m → matrixToRow′ m₁ n₁ ⟨ lift″ _dot_ ⟩
     matrixToColumn′ m₂ m) ≤ j * (m * p + n + m * suc j)
matrix-multiply-cost-lem1 _dot_ p dot-cost-proof m₁ m₂ n₁ =
  tabulate′-cost-proof
     (λ m → matrixToRow′ m₁ n₁ ⟨ lift″ _dot_ ⟩ matrixToColumn′ m₂ m)
     λ m → •″-extended-cost-proof _dot_ p dot-cost-proof m₁ m₂ n₁ m
```

Figure 3.9: matrix-multiply-cost-lem1, which from a proof of the complexity of computing the dot product, proves the complexity of computing a row

With the necessary lemma established, the full proof is then a matter of proving that the complexity of the second, outer, tabulate′ can be added on top of matrix-multiply--cost-lem1, seen in fig. 3.9, without invalidating the proof.

We prove that in matrix-multiply-cost-proof, seen in fig. 3.10, using a similar structure to matrix-multiply-cost-lem1, we use tabulate′-cost-proof to prove that the full complexity of matrix multiplication should be: $T_{tabulate1} * (T_{tabulate2} * (T_{dot} + T_{matrixToRow} + T_{matrixToColumn}))$

3.2.2.5 _⊗_ proof

The final proof, ⊗-cost-proof, seen in fig. 3.11, is a specialization of the generalized proof, matrix-multiply-cost-proof, where we supply the arguments to the base proof, the specialized dot operation, the matrices, and the dot cost proof, and that is sufficient. The complexity we prove for our matrix multiplication implementation is $\mathcal{O}(n \cdot (j \cdot (m \cdot 2 + n + m \cdot suc\, j)))$, which can be simplified to $\mathcal{O}(n \cdot j^2 \cdot m)$ or, specifically in the case of square matrices because then: $n = j = m$, we can simplify the complexity to $\mathcal{O}(n^4)$.

```
matrix-multiply-cost-proof : {A : Set} → {n m j : ℕ}
  → (_dot_ : {k : ℕ} → (x y : Vec A k) → Timed A)
  → (p : ℕ)
  → (dot-cost-proof : {k : ℕ} → (x y : Vec A k) → cost(x dot y) ≤ k * p)
  → (m₁ : A ▷ n × m)
  → (m₂ : A ▷ m × j)
  → cost (MatrixMultiplicationBase A (lift″ _dot_) m₁ m₂)
     ≤ n * (j * (m * p + n + m * suc j))
matrix-multiply-cost-proof {A} {n₁} _dot_ p dot-cost-proof m₁ m₂ =
  tabulate′-cost-proof
     (λ n →
       tabulate′ λ m →
         matrixToRow′ m₁ n ⟨ lift″ _dot_ ⟩ matrixToColumn′ m₂ m)
     λ n → matrix-multiply-cost-lem1 _dot_ p dot-cost-proof m₁ m₂ n
```

Figure 3.10: `matrix-multiply-cost-proof` base proof for parameterization with dot function and proof

```
⊗-cost-proof : {n m j : ℕ}
  → (m₁ : n × m)
  → (m₂ : m × j)
  → cost (m₁ ⊗ m₂) ≤ n * (j * (m * 2 + n + m * suc j))
⊗-cost-proof {n} m₁ m₂ = matrix-multiply-cost-proof
                                           _•′_
                                           2
                                           •′-≤-cost-proof
                                           m₁
                                           m₂
```

Figure 3.11: `⊗-cost-proof` proving the time complexity of $m_1 \otimes m_2$

Chapter 4

A complexity certified CYK implementation

In this chapter we will describe our approach to the CYK algorithm based on the paper "General Context-Free Recognition in Less than Cubic Time" by Valiant [11]. We will then explain the extension of our matrix library, with a certification of the time complexity of the CYK algorithm.

4.1 An introduction to CYK

The CYK algorithm [12, p. 290][11, 13] is a polynomial time, specifically $O(n^3)$, algorithm that checks if a provided string is a member of a context free grammar (abv. CFG) in Chomsky normal form (abv. CNF). This is important because any context free language can be represented by a CFG in CNF and thus the CYK algorithm shows that every context free language is a member of the class P.

A CNF grammar is specified by a quadruple $\langle N, T, R, S \rangle$, where N is a set of non-terminals $\{N_1, N_2, \dots, N_n\}$, T a set of terminals, R a set of grammar productions of the form:

- $N_i \rightarrow N_j N_k$
- $N_i \rightarrow t$, where $t \in T$
- $S \rightarrow \epsilon$, if the empty string is in the language.

And finally the starting non-terminal $S \in N$.

The algorithm computes the transitive closure of an upper triangular matrix on the input string w. If the transitive closure contains S in its upper right corner, w exists in the language.

The matrices consist of elements of subsets of N. The initial matrix consists of a diagonal where M_{ii} contains every non-terminal that has a rule of the form $A_j \rightarrow w_i$. A binary operator on two sets can then be defined:

$$M_1 \cdot M_2 = \{A_i \mid \exists A_j \in M_1, A_k \in M_2, \text{such that } (A_i \rightarrow A_j A_k \in R)\}$$

With this operator defined, multiplication of two matrices, x and y can then be defined as:

$$z_{ik} = \bigcup_{j=1}^{n} x_{ij} \cdot y_{jk}$$

Where all the subsets that the matrices consist off are compared with the defined $\cdot$ operation, finding the rules that could feasibly have constructed the symbols that are already present.

What we are interested in is multiplying a square matrix m with itself in order to get the *transitive closure* of that matrix. Defined as:

$$m^+ = m^1 \cup m^2 \cup, \ldots, \cup m^n$$

For an $n \times n$ sized matrix. Here m^i is defined as:

$$m^i = \bigcup_{j=1}^{i-1} m^j \cdot m^{i-j} \text{ and } m^1 = m$$

Every iteration will in turn have computed the next off-center diagonal, for a graphical example see fig. 4.1. Ultimately if the symbol found in the top right corner is the starting non-terminal S then the grammar recognizes the string w.

It is possible to reduce transitive closure to a single multiplication but for this report we will stick to the simple definition of transitive closure as described [11].

The rest of this chapter will further explore how we implemented this definition of matrix multiplication and the CYK algorithm in Agda, along with the proof of its time complexity.

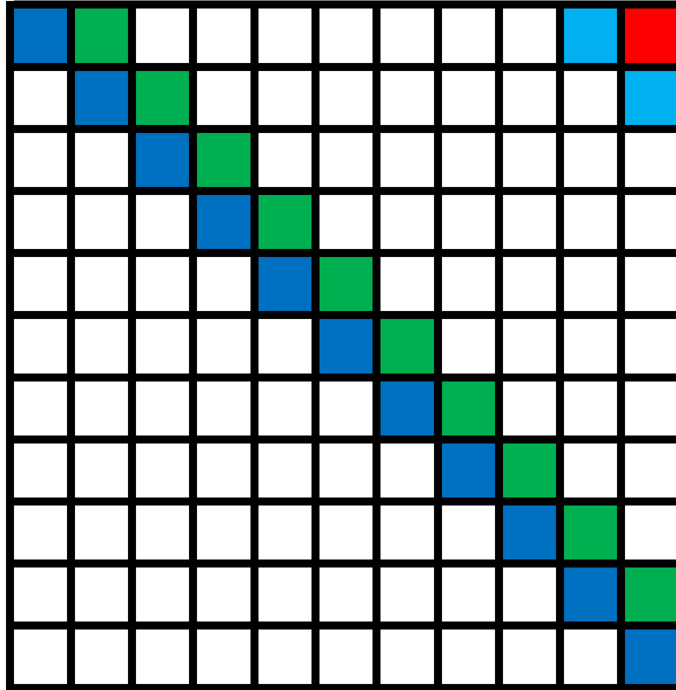

Figure 4.1: Matrix computation order

: Initial matrix   : 1st iteration
: $n-1$th iteration   : nth iteration

4.2 Implementation of complexity certified CYK

This section will explore the process of implementing a complexity certified version of CYK with our complexity library. We will present our initial, fully operational, implementation and explain why the limitations of our library made it difficult to certify. Then we will present our new implementation and explore the significant differences that made the certification process possible. The full source code for the CYK algorithm and the proofs are available in appendix F and appendix G respectively, additionally the first uncertified CYK implementation is available in appendix E.

As explained in section 4.1 the CYK algorithm assumes a set of rules that belong to a context free language. In Agda we define a datatype `Rule`, seen in fig. 4.2, to hold rules on the forms possible in a CNF grammar.

Then we define a `Language`, seen in fig. 4.3, that contains the set of `Rule`s in a vector, as well as the number of rules.

The first function we create for the algorithm is `_•_`, which calculates the product of two

```
data Rule (T N : Set) : Set where
  _⇒_ : N → T → Rule T N
  _⇒_o_ : N → N → N → Rule T N
```

Figure 4.2: Definition of Rule data type

```
record Language (T N : Set) : Set₁ where
  field
    _N≟_ : Decidable {A = N} _≡_
    _T≟_ : Decidable {A = T} _≡_
    №rules : ℕ
    rules : Vec (Rule T N) №rules
```

Figure 4.3: Definition of Language record

elements in the matrix by finding matching nonterminal symbols that match rules in the language. In Agda we define Element as being a set of nonterminal symbols, contained in a list, seen in fig. 4.4, and define the union, _∪ _, seen in fig. 4.5, as two Elements appended with duplicates removed.

```
Element : Set
Element = List.List Nonterminal
```

Figure 4.4: Definition of Element as a list of nonterminals

```
_∪_ : Element → Element → Element
lhs ∪ rhs = deduplicate _N≟_ (lhs L++ rhs)
```

Figure 4.5: _∪_ function for lists

Based on this definition of union, and the auxiliary function `⇒_∘_` that determines the possible rules in a row and column, we can define `_•_`, seen in fig. 4.6, and use that along with our library of complexity certified matrix operations presented in section 3.2 to define a new matrix multiplication `_⊗_` operator, seen in fig. 4.7.

```
_•_ : {n : ℕ} → Vec Element n → Vec Element n → Element
(row :: rows) • (_ :: column :: columns)
  = (⇒ row ∘ column) ∪ (rows • (column :: columns))
_ • _ = L[]
```

Figure 4.6: `_•_` for matrices of lists

```
_⊗_ : {n m j : ℕ} → (suc n) × (suc m) → (suc m) × (suc j)
  → Timed ((suc n) × (suc j))
_⊗_ = MatrixMultiplicationBase Element (lift _•_)
```

Figure 4.7: `_⊗_` parameterized version of `MatrixMultiplicationBase` for matrices of lists

While this implementation works and accurately computes the CYK algorithm when testing, it is not time annotated correctly and an examination revealed that complexity certification would be very difficult. This is because of the choice to make `Element` a list, which in turn makes the result of the union function a list of indeterminate size. While we know that the contents of each individual set of nonterminals can, at maximum, contain all nonterminals, convincing Agda of this fact is harder. This shows a clear restriction of design space. In the current state of our library it is hard to certify arbitrary implementations of the CYK algorithm. Unpredictability regarding the number of intermediate calculations greatly increases the difficulty of proving complexity.

With the observations from the old implementation a new complexity certifiable iteration, with fixed sized elements, was constructed. We redefined an `Element`, seen in fig. 4.8, as a vector of booleans, with the length of the number of nonterminals in the language, the value of each boolean is then based on whether or not the associated nonterminal is found to derive that `Element`. This construction is called a *bit string*. Importantly, this means that the union operator, `_∪_`, seen in fig. 4.9, will always result in `Element`s of the same size

```
Element : Set
Element = Vec Bool №nonterminals
```

Figure 4.8: Definition of *bit string* `Element`

```
_∪′_ : {n : ℕ} → Vec Bool n → Vec Bool n → Timed (Vec Bool n)
lhs ∪′ rhs = zipWith′ (lift′ _∨_) lhs rhs
```

Figure 4.9: `_∪_` function, union of two *bit strings*

because since the total number of nonterminals does not change and proving the complexity becomes trivial.

With the improved implementation we can then create new implementations of dot product, `_•_`, seen in fig. 4.10, and matrix multiplication, seen in fig. 4.11, `_⊗_`, in a way that is accurately time certified.

```
_•′_ : {n : ℕ} → Vec Element n → Vec Element n → Timed Element
(row :: row₁ :: rows) •′ (discard :: column :: columns)
  = (⇒ row ∘ column) ∪″ ((row₁ :: rows) •′ (column :: columns))
(x :: []) •′ (y :: []) = EmptyElement
([]) •′ ([]) = EmptyElement
```

Figure 4.10: `_•′_` function, dot product function for vectors of `Element`s

```
_⊗′_ : {n m j : ℕ} → (suc n) × (suc m) → (suc m) × (suc j)
  → Timed ((suc n) × (suc j))
_⊗′_ = MatrixMultiplicationBase Element _•″_
```

Figure 4.11: `_⊗′_` parameterized version of `MatrixMultiplicationBase` for matrices of `Element`s

4.3 Proving the time complexity of the CYK algorithm

Now that we have an implementation of the CYK algorithm, we will then prove its complexity.

We already have the proof matrix-multiply-cost-proof that along with a proof of the complexity of the dot function, proves the complexity of matrix multiplication.

Since complexity proofs depend on proofs of their constituent functions, we start by certifying the complexity of _∪′_ with ∪′-cost-proof, seen in fig. 4.12. This is done by simple recursion.

```
∪′-cost-proof : {n : ℕ} → (l r : Vec Bool n) → cost (l ∪′ r) ≤ n * 2
∪′-cost-proof l r with l | r
∪′-cost-proof l r | [] | [] = z≤n
∪′-cost-proof l r | l₁ :: ls | r₁ :: rs rewrite lift″-cost-proof _::′_
  (time 1 (l₁ ∨ r₁)) (zipWith′ (λ x y → time 1 (x ∨ y)) ls rs)
  = s≤s $ s≤s $ ∪′-cost-proof ls rs
```

Figure 4.12: ∪′-cost-proof proving the time complexity of $l \cup' r$

This proof is then used as justification to prove •′-cost-proof, seen in fig. 4.13, that the complexity of the individual operations is based on the size of the elements, and therefore the total number of rules. Even though the complexity bound of •′-cost-proof is similar to the matrix library proof for the equivalent operation, presented in section 3.2.2.3, and the proof in itself becomes structurally smaller, Agda can not guarantee termination and we have to insert the Terminating flag. The last proof, ⊗-cost-proof, seen in fig. 4.14, shows that the complexity bound of it is similar to that of the matrix multiplication defined in the matrix library, with the complexity of the two nested tabulate′s, matrixToRow′ and matrixToColumn′ functions added on top of the complexity of the _•′_ operator.

With the final proof we have shown that our CYK implementation has a time complexity of $\mathcal{O}(n \cdot j \cdot m \cdot (n^t \cdot 2 + n^t \cdot n^t \cdot (n^t + n^r)) + n + m \cdot suc\, j)$ which can be simplified to $\mathcal{O}(n \cdot j^2 \cdot m)$ or, specifically in this case because CYK matrices are square so, $j = n = m$, we can simplify to $\mathcal{O}(n^4)$. Because of our simple approach to transitive closure we never reduce recognition to a single multiplication and therefore we have to multiply the matrix n times in total to achieve recognition, this makes the final complexity $\mathcal{O}(n^5)$.

```
{-# TERMINATING #-}
•′-cost-proof : {n : ℕ} → (x y : Vec Element n)
  → cost (x •′ y) ≤ n * (№nonterminals * 2 + №nonterminals *
  №nonterminals * (№nonterminals + №rules))
•′-cost-proof [] [] = z≤n
•′-cost-proof (x ∷ []) (y ∷ []) = z≤n
•′-cost-proof (x ∷ xs@(x₁ ∷ x₂)) (y ∷ ys@(y₁ ∷ y₂))
  rewrite lift″-cost-proof _∪′_ (⇒ x ∘ y₁) ((x₁ ∷ x₂) •′ (y₁ ∷ y₂)) =
    +-mono³-≤
      (∪′-cost-proof (value (⇒ x ∘ y₁)) (value (xs •′ ys)))
      (cost≤rule-count³-lemma x y₁)
      (•′-cost-proof xs ys)
```

Figure 4.13: •′-cost-proof proving the time complexity of $x \bullet' y$

```
⊗-cost-proof : {n m j : ℕ} → (m₁ : n × m) → (m₂ : m × j)
  → cost (m₁ ⊗′ m₂) ≤ n * (j * (m * (№nonterminals * 2 +
  №nonterminals * №nonterminals * (№nonterminals + №rules)) +
  n + m * suc j))
⊗-cost-proof m₁ m₂ = matrix-multiply-cost-proof _•′_
  (№nonterminals * 2 + №nonterminals * №nonterminals
  * (№nonterminals + №rules)) •′-cost-proof m₁ m₂
```

Figure 4.14: ⊗-cost-proof proving the time complexity of $m_1 \otimes' m_2$

Chapter 5

Conclusion

In this final chapter we will close out the report with a discussion of the work done, a conclusion on our problem statement and a perspective towards promising avenues of research and work for the future.

5.1 Results

In this report we developed a process for the certification of time complexity with minimal annotation, by using the `Timed` monad, and constructing iteratively more advanced functions based on simpler certified functions. We used that process to create a complexity certified matrix library, with matrix multiplication certified to be computed in $\mathcal{O}(n^4)$ time. Additionally we created a complexity certified CYK algorithm in $\mathcal{O}(n^5)$ time.

We have shown that the our certified functions can be almost identical to normal functions written in Agda, and that the only mandatory alteration requirement is the need to encapsulate the return type in a `Timed` monad.

As we suggested in the *final* problem statement our approach is centered around the signature of a class of functions, e.g. sorting functions or matrix multiplication. This signature shows what functions will need to be annotated before it is possible to accurately track the computational complexity of a given function using our method. An annotated signature for a class of functions is called a complexity signature. The functions in this complexity signature will then keep track of how many steps it takes to calculate a value. This is similar to Danielsson's approach, but has been restructured to have a clearer syntax, by removing the need to use the `≻` function, and implementing the other monad signature function `return` as a postfix operator `_′` which is less visually intrusive.

This makes the annotated function syntactically similar to the original unannotated one.

To assist in annotating, a library with a number of helper functions was created. Instead of having to manually re-write simple non-recursive operators, the library can do it automatically using the `lift` functions. However, anything recursive or more complex will have to be re-written and annotated manually. This is because `lift` assumes that the function it is lifting has a complexity of $\mathcal{O}(1)$.

A part of the library has functions built-in that were designed specifically for working with vectors, and their timed equivalents. While this makes it easier to work with vectors, as opposed to other forms of lists, it is not impossible to annotate functions that use e.g. lists instead.

The annotation library was used in the annotation of a matrix library we had created in a previous report[4].

5.2 Discussion

In this section we will discuss the ramifications of our work, by comparing it to existing work, as well as unannotated Agda code, and present the limitations of our approach.

5.2.1 Criteria evaluation

In section 1.3 we introduced three analysis criteria, that we then used to evaluate three different papers that were of particular interest to us. The three criteria were as follows:

- How are type systems utilized?
- What cost model is used?
- What analysis model is used?

And we will now analyze our own contribution with the same criteria. In our type system, we use a monad and term annotation, much like Danielsson's approach. We utilize the `Timed` monad that contains the value of the underlying type, as well as a natural number telling us how many steps it took to calculate this value. These monads then propagate up through the call-tree of a function, getting combined along the way, and in the end we end up with a result containing, partly, the output of the untimed function, as well as the final time it took to compute.

The cost model relies on manual annotation of functions which would ideally count beta reductions.

Analysis is done manually by constructing Agda proofs for base annotated functions and using those proofs as justification in more complex proofs.

5.2.2 Comparing annotated and unannotated code

We presented the following *expanded* problem statement in section 1.3.4

How close can a solution that certifies complexity be to a source language like Agda, in terms of syntax?

In a comparison between two variations of matrix multiplication, one with complexity annotation, in fig. 5.1, and one without, in fig. 5.2, it becomes clear that the annotated version of the function is similar to the original in structure. In fact without the prime naming convention, that the annotated version of `foo` should be named `foo′`, the only real change to the structure of the code would be changing the return type to one encapsulated in a `Timed` monad.

To achieve certification of time complexity, however, a proof is required which is an extra requirement and one of the primary limitations of our approach.

```
_⊗_ : {n m j : ℕ} -> (suc n) × (suc m) → (suc m) × (suc j)
  → Timed((suc n) × (suc j))
left ⊗ right = tabulate′ λ n →
                tabulate′ λ m →
                 matrixToRow′ left n •″ matrixToColumn′ right m
```

Figure 5.1: Complexity annotated matrix multiplication.

```
_⊗_ : {n m j : ℕ} -> n × m → m × j → n × j
left ⊗ right = tabulate λ n →
                tabulate λ m →
                 matrixToRow left n • matrixToColumn right m
```

Figure 5.2: Normal matrix multiplication

5.2.3 On the n^5 complexity of our CYK implementation

We will in this section touch on the computational complexity discrepancy between our implementation and other established approaches.

The time complexity of the CYK algorithm we constructed is, as proved, $\mathcal{O}(n^5)$, which is far in excess of the expected complexity of a CYK algorithm. This is due to the $\mathcal{O}(n^2)$ complexity of our `matrixToColumn′` function and our naive implementation of transitive closure, where we matrix multiply n times, as opposed to once. Valiant has shown that as low as $\mathcal{O}(n^{2.81})$ is possible [11], and $\mathcal{O}(n^3)$ is considered the baseline. This shows that complexity certification could be useful, in that we know with certainty that our implementation is subpar and could use improvement.

5.2.4 General limitations of our approach

In section 1.5 we presented the following *final* problem statement, which we will discuss here.

Is it the case that certifying the time complexity of an algorithm, by only annotating functions, without heavily changing the readability or the syntax of the algorithm is a good approach?

The primary limitations of our approach are the burden of proof, lack of bound correctness and restricted design space.

Unlike the other approaches we have examined, our library requires each and every annotated function to have proofs created, because composite functions need proof of the complexity of each individual function they use as justification to prove their complexity. This is not a requirement for the other contributions we have examined. Wang et al. [6] use an SMT solver to check their bounds; Hudson [7] converts expressions written in a limited language to complexity expressions that check his bounds and Danielsson's [3] Thunk monad carries the complexity as a visible value which Agda calculates, which can then be inspected and proven to be below some bound. Specifically in the case of Danielsson it is arguable that the values calculated would be equivalent, as we both ideally count beta reductions, and contain the result in a monad, however, since we chose to keep the value hidden for readability, the `Timed` monad provides no visible bound on a function by itself, so while the time complexity values could be equivalent we need distinct proofs.

Additionally, the manual nature of proving complexity means that a complexity bounds correctness relies on the programmer to annotate correctly thus tightening the bound and proving the minimal complexity possible. A proof that a function has a complexity of $\mathcal{O}(n)$ is formulated such that it will also be accepted if the proof claimed $\mathcal{O}(n^2)$, because if $x \leq n$, then $x \leq n^2$. It is possible to use the library for certifying the Θ of a complexity bound, but not always practical if the runtime varies depending on the value of input parameters, not just their size. And once the lower bound has been lost, it can be hard or impossible to establish it again.

As our first implementation of the CYK algorithm revealed, some data structures are difficult to prove the complexity of. If the algorithm calls for lists of arbitrary length it could be difficult to use our library and general approach to provide a complexity certified version.

5.3 Future work

We theorize that the extensive manual proof burden could be lessened due to the observed formulaic nature of creating them, generating most proofs in an automated manner thus reducing the workload of certifying complexity with our method.

With regards to our implementation of the CYK algorithm it could be interesting to see if the least complex versions of the algorithm could have their complexity certified, currently we only handle complexity in terms of integers, so certifying matrix multiplication in $\mathcal{O}(n^{2.81})$ would require an extension of the library to handle complexity in floating point terms. Certifying matrix multiplication in $\mathcal{O}(n^3)$ however is more approachable, with column access in linear time, and reducing transitive closure to matrix multiplication [11] a modification of what we have implemented would be in cubed complexity.

Currently our method does not handle memoization well, which is an important concept in algorithm design.

By using the `lift′` function it is possible reuse arguments multiple times without paying for them multiple times, but since an expression contains it cost, naively, a bound value will pay for it multiple times.

Due to this, expressions which should be semantically equivalent, might not have the same computational cost when subexpressions get reused.

For more complex cases of memoization found in dynamic programming, it would be necessary to prove to Agda that a value has been computed ahead of time to correctly prove the computational cost.

We have not attempted to do so, but suspect it to be non-trivial. It would be interesting to try and provide an easier method for this, but this will ultimately run up against the halting problem.

If we look at our library in an abstract manner, what we have designed counts resource usage but it is not necessarily time, certifying any countable resource such as space with our approach would be feasible as well. It would require that the `cost` counted from a call be altered from counting time to counting space instead.

Another approach would be combining Danielson's cost directly in the type, with certain aspects of our solution.

Danielson's approach requires a lot of annotation in every function, but the resulting complexity does not require writing proofs. Nothing should prevent implementing `lift′` using Danielson's thunk library.

This approach might lead to a "best of both worlds" scenario of reduced annotation burden while not requiring any proofs.

Bibliography

[1] D. Firsov, "Certification of context-free grammar algorithms," Ph.D. dissertation, Tallinn University of Technology, 2016. [Online]. Available: https://digikogu.taltech.ee/et/item/91ad60ad-514c-470c-a00a-68548d7d39ba

[2] E. Copello, A. Tasistro, and B. Bianchi, "Case of (quite) painless dependently typed programming: Fully certified merge sort in agda," in *Programming Languages - 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings*, ser. Lecture Notes in Computer Science, F. M. Q. Pereira, Ed., vol. 8771. Springer, 2014, pp. 62–76. [Online]. Available: https://doi.org/10.1007/978-3-319-11863-5_5

[3] N. A. Danielsson, "Lightweight semiformal time complexity analysis for purely functional data structures," *SIGPLAN Not.*, vol. 43, no. 1, p. 133–144, Jan. 2008. [Online]. Available: https://doi.org/10.1145/1328897.1328457

[4] J. Elgaard, C. Møllnitz, and S. Q. Rannes, "A Survey of Dependently Typed Programming and Its Foundations," 9th semester project report, Department of Computer Science, Aalborg University, January 2020.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[6] P. Wang, D. Wang, and A. Chlipala, "TiML: a functional language for practical complexity analysis with invariants," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133903

[7] B. Hudson, "Computer-checked recurrence extraction for functional programs," Ph.D. dissertation, Wesleyan University, 2016.

[8] Various. (2020) Signature (logic). [Online]. Available: https://en.wikipedia.org/wiki/Signature_(logic)

[9] U. Norell, “Towards a practical programming language based on dependent type theory,” Ph.D. dissertation, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[10] T. A. Community. (2020) Built-ins —agda v2.6.1 documentation. [Online]. Available: https://agda.readthedocs.io/en/v2.6.1/language/built-ins.html#integers

[11] L. G. Valiant, “General context-free recognition in less than cubic time,” *J. Comput. Syst. Sci.*, vol. 10, no. 2, p. 308–315, Apr. 1975. [Online]. Available: https://doi.org/10.1016/S0022-0000(75)80046-8

[12] M. Sipser, *Introduction to the Theory of Computation*. Cengage Learning, 2013.

[13] D. Firsov and T. Uustalu, “Certified CYK parsing of context-free languages,” *J. Log. Algebr. Meth. Program.*, vol. 83, no. 5-6, pp. 459–468, 2014. [Online]. Available: https://doi.org/10.1016/j.jlamp.2014.09.002

Appendix A

Annotation library

```
module CostBase where

open import Data.Vec hiding (_»=_)
import Data.Fin as Fin
open import Data.Nat.Base
open import Function

data Timed (A : Set) : Set where
  time : ℕ → A → Timed A

infixl 30 _′

-- Turn a value into a value produced in zero time
_′ : {A : Set} → A → Timed A
x ′ = time 0 x

cost : {A : Set} → Timed A → ℕ
cost (time n _) = n


value : {A : Set} → Timed A → A
value (time _ x) = x

-- Increment the time by one unit
_″ : {A : Set} → Timed A → Timed A
time n x ″ = time (suc n) x
```

```
-- Increment the time by n time units
_‴_ : {A : Set} → Timed A → ℕ → Timed A
time n₁ x ‴ n₂ = time (n₁ + n₂) x

--Lifts a non-recursive function into a timed function,
-- that takes 1 time unit to execute
lift′ : {A B C : Set} → (A → B → C) → (A → B → Timed C)
lift′ fn = λ x y → (time 1 (fn x y))

lift″ : {A B C : Set} → (A → B → Timed C) →
                                (Timed A → Timed B → Timed C)
lift″ fn = λ { (time n a) (time m b) → fn a b ‴ (n + m) }

lift : {A B C : Set} → (A → B → C) → (Timed A → Timed B → Timed C)
lift = lift″ ∘ lift′

_»=_ : {A B : Set} → Timed A → (A → Timed B) → Timed B
_»=_ (time n x) fn = fn x ‴ n
```

Appendix B

Annotation library for vectors

```
module CostVec where

open import CostBase

open import Data.Nat.Base
import Data.Fin.Base as Fin

open import Data.Vec public hiding (_»=_)

-- Cons operation with a cost of 1
_::′_ : {A : Set} → {n : ℕ} → A → Vec A n → Timed (Vec A (suc n))
_::′_ = lift′ _::_

_::″_ : {A : Set} → {n : ℕ} → Timed A → Timed (Vec A n) → Timed (Vec A (suc n))
_::″_ = lift″ _::′_

-- Append operation with cost. should return element with cost of (length fst + cost snd)
_++′_ : {A : Set} → {n m : ℕ} → Vec A n → Vec A m → Timed (Vec A (n + m))
[] ++′ y = y ′
(x :: xs) ++′ y = x ′ ::″ (xs ++′ y)

_++″_ : {A : Set} → {n m : ℕ} → Timed (Vec A n) → Timed (Vec A m) → Timed (Vec A (n + m))
_++″_ = lift″ _++′_

-- Timed lookup, based completely on the lookup in the Agda std lib
lookup′ : ∀ {n} → {A : Set} → Vec A n → Fin.Fin n → Timed A
lookup′ (x :: xs) Fin.zero = (x ′) ″
```

```
lookup′ (x :: xs) (Fin.suc i) = (lookup′ xs i) ″

raiseVec : {A : Set} → {n : ℕ} → Vec (Timed A) n → Timed (Vec A n)
raiseVec [] = [] ′
raiseVec (time n x :: xs) = (raiseVec xs) »= λ y → time n (x :: y)
```

Appendix C

Matrix library

```
module Matrix where

open import Data.Vec
open import Data.Nat
import Data.Fin as Fin
open import Function

open import CostBase
open import CostVec

_▷_×_ : Set → ℕ → ℕ → Set
A ▷ n × m = Vec (Vec A m) n

-- Gets a row from a Matrix (Zero indexed) - Timed
matrixToRow′ : {A : Set} → {n m : ℕ} → (A ▷ n × m)
  → Fin.Fin n → Timed (Vec A m)
matrixToRow′ mat i = lookup′ mat i

-- Gets a Column from a Matrix (Zero indexed) - Timed
matrixToColumn′ : {A : Set} → {n m : ℕ} -> (A ▷ n × m)
  → Fin.Fin m → Timed (Vec A n)
matrixToColumn′ [] i = []′
matrixToColumn′ (mat :: mats) i = (lookup′ mat i) ::′′ matrixToColumn′ mats i


-- Timed tabulate, based completely on the tabulate in the Agda std lib
tabulate′ : ∀ {n} → {A : Set} → (Fin.Fin n → Timed A) → Timed (Vec A n)
```

```
tabulate′ f = raiseVec $ tabulate f

MatrixMultiplicationBase : (A : Set) → (dot : {k i j : ℕ} → Timed (Vec A k)
  → Timed (Vec A k) -> Timed A)
  → {n m j : ℕ} → A ▷ n × m → A ▷ m × j → Timed (A ▷ n × j)
MatrixMultiplicationBase A dot left right =
  tabulate′ λ n →
    tabulate′ λ m →
      dot {i = Fin.toℕ n} {j = Fin.toℕ m}
      (matrixToRow′ left n) (matrixToColumn′ right m)


module IntMatrix where

open import CostVec hiding (_⊛_)
open import Data.Nat.Base using (ℕ; suc; zero; _+⋎_) renaming (_+_ to _ℕ+_)
import Data.Fin as Fin
open import Data.Integer renaming (suc to succ)
open import Function
import Relation.Binary.PropositionalEquality as Eq
open Eq using (_≡_; refl; cong; sym)

-- Debug stuff for printing
open import Agda.Builtin.IO
open import Agda.Builtin.String
open import Agda.Builtin.Unit
open import CostBase
open import Matrix

_*″_ : Timed ℤ → Timed ℤ → Timed ℤ
_*″_ = lift″ (lift′ _*_)

_+′_ : ℤ → ℤ → Timed ℤ
_+′_ = lift′ _+_

_+″_ : Timed ℤ → Timed ℤ → Timed ℤ
_+″_ = lift″ _+′_

0ℤ′ : Timed ℤ
0ℤ′ = 0ℤ ′
```

```
-- Matrix, a nested vector.
-- @a the numeric type of the data contained in the nested vector.
_×_ : ℕ → ℕ → Set
n × m = ℤ ▷ n × m

-- Matrix, a nested vector.
-- @a the numeric type of the data contained in the nested vector.
-- Timed
_×′_ : ℕ → ℕ → Set
n ×′ m = Timed (Vec (Vec ℤ m) n)
-- Can convert a row to a 1 x M Matrix
-- this is used to append rows to a Matrix.
-- Makes individual rows and columns work as one dimensional matrixes
rowToMatrix : {m : ℕ} → Vec ℤ m → 1 × m
rowToMatrix [] = [] :: []
rowToMatrix x = x :: []

-- Can convert a row to a 1 x M Matrix
-- this is used to append rows to a Matrix.
-- Makes individual rows and columns work as one dimensional matrixes
-- Timed
rowToMatrix′ : {m : ℕ} → Timed (Vec ℤ m) → 1 ×′ m
rowToMatrix′ (time n []) = [] ::′ []
rowToMatrix′ (time n x) = x ′ ::″ [] ′

-- Can convert a column to an n × 1 matrix
columnToMatrix : {n : ℕ} → Vec ℤ n → n × 1
columnToMatrix [] = []
columnToMatrix (x :: xs) = (x :: []) :: columnToMatrix xs

-- Can convert a column to an n × 1 matrix
-- Timed
columnToMatrix′ : {n : ℕ} → Timed (Vec ℤ n) → n ×′ 1
columnToMatrix′ (time n []) = [] ′
columnToMatrix′ (time n (x :: xs)) = (x :: []) ′ ::″ columnToMatrix′ (xs ′)


-- Gets a row from a Matrix (Zero indexed)
matrixToRow : {n m : ℕ} → (n × m) → Fin.Fin n → Vec ℤ m
```

```
matrixToRow mat i = lookup mat i


-- Gets a Column from a Matrix (Zero indexed)
matrixToColumn : {n m : ℕ} → (n × m) → Fin.Fin m → Vec ℤ n
matrixToColumn [] i = []
matrixToColumn (mat :: mats) i = lookup mat i :: matrixToColumn mats i


-- Attaches two matrices rowwise
[_/_] : {n1 n2 m : ℕ} → n1 × m → n2 × m → (n1 ℕ+ n2) × m
[ first / second ] = first ++ second

-- Attaches two matrices rowwise
--Timed
[_/′_] : {n1 n2 m : ℕ} → n1 ×′ m → n2 ×′ m → (n1 ℕ+ n2) ×′ m
[ first /′ second ] = first ++″ second


-- Takes a natural number m and returns a 0 x m matrix
createEmpties : {m : ℕ} → m × 0
createEmpties {m = zero} = []
createEmpties {m = (suc k)} = [] :: createEmpties


-- Takes a natural number m and returns a 0 x m matrix
--Timed
createEmpties′ : {m : ℕ} → m ×′ 0
createEmpties′ {m = zero} = [] ′
createEmpties′ {m = (suc k)} = [] ′ ::″ createEmpties′

-- Matrix dot product
_•_ : {n : ℕ} → Vec ℤ n → Vec ℤ n → ℤ
[] • [] = 0ℤ
(x :: xs) • (y :: ys) = x * y + (xs • ys)

_•′_ : {n : ℕ} → Vec ℤ n → Vec ℤ n → Timed ℤ
[] •′ [] = 0ℤ′
(x :: xs) •′ (y :: ys) = (x ′ *″ y ′) +″ (xs •′ ys)

_•″_ : {n : ℕ} → Timed (Vec ℤ n) → Timed (Vec ℤ n) → Timed ℤ
```

```
_•′′_ = lift′′ _•′_

-- Matrix multiplication.
-- @n rows of the left matrix
-- @m columns of the left matrix, rows of the right matrix. (Must be equal)
-- @j columns of the right matrix
_⊗_ : {n m j : ℕ} → n × m → m × j → Timed(n × j)
_⊗_ = MatrixMultiplicationBase ℤ _•′′_



-- Attaches two matrices columnwise.

-- [_|_] : {n m1 m2 : ℕ} -> n × m1 -> n × m2 -> Timed (n × (m1 ℕ+ m2))
--[ first | second ] = zipWith′ (_++′_) first second

-- Transposes a matrix from n m -> m n
-- @a can be any type.
_ᵀ : {n m : ℕ} -> n × m -> m × n
_ᵀ [] = createEmpties
_ᵀ (x :: xs) = zipWith (_::_) x (xs ᵀ)

-- Helper function for zipping two matrices together.
zipWithM : {m n : ℕ} -> ( ℤ -> ℤ -> ℤ) -> m × n -> m × n -> m × n
zipWithM fn m1 m2 = zipWith (zipWith fn) m1 m2

_⊕_ : {n m : ℕ} -> n × m -> n × m -> n × m
_⊕_ [] [] = []
_⊕_ m1 m2 = zipWithM (_+_) m1 m2

-- Matrix subtraction
-- @a The negative type of a, note,
-- subtraction is only possible if the type involved can be negative.
_⊖_ : {n m : ℕ} -> n × m -> n × m -> n × m
_⊖_ m1 m2 = zipWithM (_-_) m1 m2
```

Appendix D

Matrix proofs

```
module CostVec.Properties where

open import CostVec
open import CostBase
open import Data.Nat
open import Data.Nat.Properties
open import Function
import Data.Fin as Fin

open import Relation.Binary.PropositionalEquality
open Relation.Binary.PropositionalEquality.≡-Reasoning

raiseVec-proof-zero : {A : Set} → cost (raiseVec {A = A} []) ≡ 0
raiseVec-proof-zero = refl

raiseVec-proof-one : {A : Set} → (x : Timed A)
  → cost (raiseVec (x ∷ [])) ≡ cost x
raiseVec-proof-one (time n x) rewrite +-comm n 0 = refl

raiseVec-proof-comm : {A : Set} → {n : ℕ} → (x : Timed A)
  → (xs : Vec (Timed A) n)
  → cost (raiseVec (x ∷ xs)) ≡ cost x + cost (raiseVec xs)
raiseVec-proof-comm (time n _) xs with raiseVec xs
... | (time n₁ _) rewrite +-comm n n₁ = refl

raiseVec-proof : {A : Set} → {n : ℕ} → (list : Vec (Timed A) n)
  → sum (map cost list) ≡ cost (raiseVec list)
```

```
raiseVec-proof [] = refl
raiseVec-proof ((time n x) :: xs) rewrite
  raiseVec-proof-comm (time n x) xs =
  begin
    n + foldr (λ v → ℕ) _+_ 0 (map cost xs)
  ≡⟨⟩
    n + sum (map cost xs)
  ≡⟨ cong (λ x → n + x ) (raiseVec-proof xs) ⟩
    n + cost (raiseVec xs)
  □

lookup-cost-lem1 : {n : ℕ} → {A : Set} → (x : A)
  → (xs : Vec A n) → (f : Fin.Fin n)
  → cost (lookup′ (x :: xs) (Fin.suc f)) ≡ suc (cost (lookup′ xs f))
lookup-cost-lem1 x xs f with lookup′ xs f
...                              | (time n y) = refl

lookup-cost-proof : {A : Set} → {n : ℕ} → (x : Vec A n)
  → (f : Fin.Fin n) → cost (lookup′ x f) ≤ n
lookup-cost-proof {A} {.(suc _)} (x :: x₁) Fin.zero = s≤s z≤n
lookup-cost-proof (x :: xs) (Fin.suc f) rewrite
  lookup-cost-lem1 x xs f =
  s≤s (lookup-cost-proof xs f)


module Matrix.Properties where
open import Data.Nat
open import Data.Nat.Properties
open import Data.Integer as Int using (ℤ)
open import CostVec
import Data.Vec
open import Data.Vec.Properties
open import CostVec.Properties
open import CostBase
open import CostBase.Properties
open import Function
open import IntMatrix
open import Matrix
import Data.Fin as Fin
```

```
open import Relation.Binary.PropositionalEquality
open Relation.Binary.PropositionalEquality.≡-Reasoning


m≤m : {n : ℕ} → n ≤ n
m≤m {zero} = z≤n
m≤m {suc n} = s≤s m≤m

≤-n-inj : {n q : ℕ} → q ≤ n + q
≤-n-inj {n} {zero} = z≤n
≤-n-inj {n} {suc q} rewrite +-suc n q = s≤s ≤-n-inj

≤-+-injˡ : {n m : ℕ } → (q : ℕ) → n ≤ m → n + q ≤ m + q
≤-+-injˡ {zero} {m} q prf = ≤-n-inj
≤-+-injˡ {suc n} {suc m} q (s≤s prf) = s≤s (≤-+-injˡ q prf)

≤-+-injʳ : {n m : ℕ } → (q : ℕ) → n ≤ m → q + n ≤ q + m
≤-+-injʳ {n} {m} q prf rewrite +-comm q n | +-comm q m = ≤-+-injˡ q prf

matrixToRow′-cost-proof : {n m : ℕ} → (x : n × m) → (i : Fin.Fin n)
  → cost ( matrixToRow′ x i) ≤ n
matrixToRow′-cost-proof x f = lookup-cost-proof x f

matrixToColumn′-cost-proof : {n m : ℕ} → (x : n × m) → (i : Fin.Fin m)
  → cost (matrixToColumn′ x i) ≤ n * suc m
matrixToColumn′-cost-proof {n} {m} [] i = z≤n
matrixToColumn′-cost-proof (x :: x₁) i rewrite
  lift-cost-proof _::_ (lookup′ x i)
  (matrixToColumn′ x₁ i) = s≤s ( +-mono-≤ (lookup-cost-proof x i)
  (matrixToColumn′-cost-proof x₁ i))

suc-inter-proof : (n : ℕ) → suc (n + 1 * n) ≡ n + 1 * suc n
suc-inter-proof n rewrite +-suc n (n + zero)= refl

•′-cost-proof : {n : ℕ} → (x y : Vec ℤ n) → cost (x •′ y) ≡ (n * 2)
•′-cost-proof [] [] = refl
•′-cost-proof {suc n} (x :: xs) (y :: ys) =
  begin
    cost ((time 0 x *″ time 0 y) +″ (xs •′ ys))
  ≡⟨⟩
```

```
    cost ((lift″ _+′_) (time 0 x *″ time 0 y) (xs •′ ys))
  ≡⟨ lift-cost-proof Int._+_ ((time 0 x *″ time 0 y)) ((xs •′ ys)) ⟩
    suc (cost (time 0 x *″ time 0 y)) + cost (xs •′ ys)
  ≡⟨⟩
    cong (_+_ 2) (•′-cost-proof xs ys)

sum-proof-one : (m : ℕ) → sum (m ∷ []) ≡ m
sum-proof-one m = +-comm m (foldr (λ v → ℕ) _+_ 0 [])

sum-const-proof : {n : ℕ} → {A : Set} (m : ℕ) → (x : Vec A n ) →
         sum  (map (const m) x) ≡ n * m
sum-const-proof m [] = refl
sum-const-proof {n} m (x ∷ x₁) =
  begin
    sum (const m x ∷ map (const m) x₁)
  ≡⟨⟩
    sum (Data.Vec.[ const m x ] ++ map (const m) x₁)
  ≡⟨ sum-++-commute Data.Vec.[ const m x ] {map (const m) x₁} ⟩
    sum (Data.Vec.[ const m x ]) + (sum (map (const m) x₁))
  ≡⟨ cong (_+ (sum (map (const m) x₁))) $ sum-proof-one m ⟩
    m + (sum (map (const m) x₁))
  ≡⟨⟩
    cong (_+_ m) (sum-const-proof m x₁)


allx : {A : Set} → {a : A} → (const 2) a ≡ 2


    module Matrix.Properties2 where

open import Data.Nat
open import Data.Nat.Properties
open ≤-Reasoning
open import Data.Fin hiding (_≤_; _+_)
open import Relation.Binary.PropositionalEquality
open import Function


open import IntMatrix
open import Matrix
```

```
open import CostBase
open import CostVec

open import Matrix.Properties
open import CostBase.Properties
open import CostVec.Properties
open import Data.Vec.Properties

column-row-cost-proof : {n m j : ℕ} (m₁ : n × m) → (m₂ : m × j)
  → (n₁ : Fin n ) → (j₁ : Fin j)
  → cost(matrixToRow′ m₁ n₁) + cost(matrixToColumn′ m₂ j₁) ≤ n + (m * suc j)
column-row-cost-proof m₁ m₂ n₁ j₁ = +-mono-≤
  (matrixToRow′-cost-proof m₁ n₁)
  (matrixToColumn′-cost-proof m₂ j₁)

•″-extended-cost-proof : {n m j : ℕ} (m₁ : n × m) → (m₂ : m × j)
  → (n₁ : Fin n ) → (j₁ : Fin j)
  → cost(matrixToRow′ m₁ n₁ •″ matrixToColumn′ m₂ j₁) ≤ (m * 2) + (n + m * suc j)
•″-extended-cost-proof {n} {m} {j} m₁ m₂ n₁ j₁ rewrite
  lift″-cost-proof _•′_ (matrixToRow′ m₁ n₁) (matrixToColumn′ m₂ j₁) |
  •′-cost-proof (value (matrixToRow′ m₁ n₁)) (value (matrixToColumn′ m₂ j₁)) |
  +-assoc (m * 2) (cost (matrixToRow′ m₁ n₁)) (cost (matrixToColumn′ m₂ j₁))
  =  (+-monoʳ-≤ (m * 2) (column-row-cost-proof m₁ m₂ n₁ j₁))


tabulate′-cost-lem1 : {A : Set} {n q z : ℕ} → (fn : Fin z → Timed A)
  → (prf : (a : Fin z) → cost (fn a) ≤ q) → (x : Vec (Fin z) n)
  → sum (map cost (map fn x)) ≤ n * q
tabulate′-cost-lem1 fn prf [] = z≤n
tabulate′-cost-lem1 fn prf (x ∷ x₁) rewrite sum-++-commute (cost (fn x) ∷ [])
  {map cost (map fn (x₁))} | +-comm (cost (fn x)) 0 =
  +-mono-≤ (prf x) (tabulate′-cost-lem1 fn prf x₁)



tabulate′-cost-proof : {n q : ℕ} → {A : Set} → (fn : Fin n → Timed A)
  → (prf : (a : Fin n) → cost (fn a) ≤ q) → cost(tabulate′ fn) ≤ n * q
tabulate′-cost-proof {n} fn prf
  rewrite sym $ raiseVec-proof (tabulate fn)
  | tabulate-allFin fn
```

```
  with allFin n
... | x = tabulate′-cost-lem1 fn prf x


⊗-cost-lem1 : {n m j : ℕ} → (m₁ : n × m) → (m₂ : m × j) → (n₁ : Fin n )
  → cost ( tabulate′ λ m → matrixToRow′ m₁ n₁ •′′ matrixToColumn′ m₂ m)
  ≤ j * ((m * 2) + (n + m * suc j))
⊗-cost-lem1 m₁ m₂ n₁ = tabulate′-cost-proof
  (λ m → matrixToRow′ m₁ n₁ •′′ matrixToColumn′ m₂ m)
  λ m → •′′-extended-cost-proof m₁ m₂ n₁ m

⊗-cost-proof : {n m j : ℕ} → (m₁ : n × m) → (m₂ : m × j)
  → cost (m₁ ⊗ m₂) ≤ n * (j * ((m * 2) + (n + m * suc j)))
⊗-cost-proof {n₁} m₁ m₂ = tabulate′-cost-proof (λ n → tabulate′ λ m
  → matrixToRow′ m₁ n •′′ matrixToColumn′ m₂ m ) λ n → ⊗-cost-lem1 m₁ m₂ n
```

Appendix E

Old CYK library

```
module Language where

open import Data.Nat
open import Data.Vec
open import Data.Bool
open import Relation.Binary using (Decidable)
open import Relation.Binary.PropositionalEquality using (_≡_)
open import Data.String

infixr 10 _⇒_o_
infixr 9 _⇒_

data Rule (T N : Set) : Set where
  _⇒_   : N → T → Rule T N
  _⇒_o_ : N → N → N → Rule T N

record Language (T N : Set) : Set₁ where
  field
    _N≟_ : Decidable {A = N} _≡_
    _T≟_ : Decidable {A = T} _≡_
    shw : N → String
    №rules : ℕ
    rules : Vec (Rule T N) №rules


import Language
module CykMatrix
```

```
  {Terminal Nonterminal : Set}
  (Lang : Language.Language Terminal Nonterminal) where

open import Data.Nat hiding (_≟_)
import Data.Integer
import Data.Integer as ℤ
open import Data.List as List renaming (_::_ to _L::_ ; _++_ to _L++_ ;
        [] to L[] ; map to lmap ; drop to ldrop)
        hiding (lookup ; reverse ; tabulate)-standing in for sets
open import Data.Vec renaming (map to vmap ; fromList to vFromList ;
        zipWith to vZipWith)
open import Relation.Nullary
open import Relation.Binary using (Decidable)
open import Relation.Binary.PropositionalEquality using (_≡_; cong; refl)
open import Data.Product renaming (_×_ to _××_)
open import Data.Maybe
open import Data.Fin renaming (_≟_ to _F≟_) hiding (lift)
open import Data.Bool hiding (_≟_)
open import Function
open import Data.String hiding (toList)
open import Text.Printf
open import Debug.Trace

open import Matrix
open import CostBase


open Language.Language Lang

Element : Set
Element = List.List Nonterminal

_×_ : ℕ → ℕ → Set
_×_ = _▷_×_ Element

- Union of two sets of elements (_+_ action)
_∪_ : Element → Element → Element
lhs ∪ rhs = deduplicate _N≟_ (lhs L++ rhs)
```

```
allPairs' : Element → Element → List (Nonterminal ×× Nonterminal)
allPairs' L[] y = L[]
allPairs' (x L:: xs) y = lmap (_,_ x) y L++ allPairs' xs y

isRule : Nonterminal ×× Nonterminal → Maybe Nonterminal
isRule (l , r) = isRule' (rules ) l r
  where isRule' : {n : ℕ} → Vec (Language.Rule Terminal Nonterminal)
            n → Nonterminal → Nonterminal → Maybe Nonterminal
        isRule' [] a b = nothing
        isRule' ((c Language.⇒ b o a) :: rules₁) l r with a N≟ r | b N≟ l
        ... | true because _ | true because _ = just c
        ... | _ | _ = isRule' rules₁ l r
        isRule' ( _ :: rules₁) a b = isRule' rules₁ l r

- All combining production rules (_*_ action)
- {∀abc c | a ∈ lhs ∧ b ∈ rhs ∧ c ← a ∘ b ∈ Productions }
⇒_∘_ : Element → Element → Element
⇒ lhs ∘ rhs = List.mapMaybe isRule $ allPairs' lhs rhs


_•_ : {n : ℕ} → Vec Element n → Vec Element n → Element
(row :: rows) • (_ :: column :: columns)
  = (⇒ row ∘ column) ∪ (rows • (column :: columns))
_ • _ = L[]


_⊗_ : {n m j : ℕ} → (suc n) × (suc m) → (suc m) × (suc j)
  → Timed ((suc n) × (suc j))
_⊗_ = MatrixMultiplicationBase Element (lift _•_)

create : {n : ℕ} → Vec Terminal n → n × n
create string = tabulate λ x → tabulate λ y → create' string x y
  where terminalRules : {n : ℕ} → Vec (Language.Rule Terminal Nonterminal) n
            → Terminal → Element
        terminalRules [] t = List.[]
        terminalRules ((x Language.⇒ x₁) :: v) t with x₁ T≟ t
        ... | true because _ = x L:: terminalRules v t
        ... | _ = terminalRules v t
```

```
        terminalRules (_ :: v) t = terminalRules v t
        create' : {n : ℕ} → Vec Terminal n → Fin n → Fin n → Element
        create' {n} string x y with x F≟ y
        ... | true because _ = terminalRules rules $ lookup string y
        ... | _ = List.[]

_⊕_ : {n m : ℕ} → n × m → n × m → n × m
lhs ⊕ rhs = vZipWith (λ x y → vZipWith _∪_ x y) lhs rhs

-We only actually look at half the matrix.
zeroHalf : {n m : ℕ} → n × m → n × m
zeroHalf [] = []
zeroHalf {n} {m} (M :: M₁) = zeroAfter n M :: zeroHalf M₁
  where zeroAfter : {n : ℕ} → (m : ℕ) → Vec Element n → Vec Element n
        zeroAfter m [] = []
        zeroAfter zero (x :: x₁) = L[] :: zeroAfter zero x₁
        zeroAfter (suc m) (x :: x₁) = x :: zeroAfter m x₁
```

Appendix F

New CYK library

```
open import Language

module CykMatrix
  {Terminal Nonterminal : Set}
  (Lang : Language Terminal Nonterminal) where

open import Function
open import Data.Nat hiding (_≟_)
open import Data.Nat.Properties
open import Data.Product renaming (_×_ to _××_) hiding (map; zip)
open import Data.Maybe hiding (zipWith; zip; map)
open import Data.Fin renaming (_≟_ to _F≟_) hiding (lift; _≤_; _<_; _+_)
open import Data.Bool renaming (_≟_ to _B≟_) hiding (_≤_; _<_)
open import Relation.Nullary
open import Relation.Binary using (Decidable)
import Data.List
import Data.Integer as ℤ
import Data.Vec.Bounded as Vec≤

open import Matrix
open import CostBase
open import CostVec hiding (allPairs)
open Language.Language Lang

Element : Set
```

```
Element = Vec Bool №nonterminals

EmptyElement : Timed Element
EmptyElement = replicate false ′

_×_ : ℕ → ℕ → Set
_×_ = _▷_×_ Element


_∪[_]′ : Element → Nonterminal → Timed Element
set ∪[ element ]′ = set [ N→Fin element ]:=′ true

_∪[_]″ : Timed Element → Timed Nonterminal → Timed Element
_∪[_]″ = lift″ _∪[_]′

setIndexes’ : {n : ℕ} → Vec Bool n → n ≤ №nonterminals
  → ζ[ m ∈ ℕ ] Vec (Fin №nonterminals) m ×× m ≤ n
setIndexes’ [] P = 0 , [] , z≤n
setIndexes’ {suc n} (false :: set) (s≤s P) with setIndexes’ set (≤-step P)
... | count , vec , prf = count , vec , ≤-step prf
setIndexes’ {suc n} (true :: set) (s≤s P) with setIndexes’ set (≤-step P)
... | count , vec , prf = suc count , fromℕ< (s≤s P) :: vec , s≤s prf


setIndexes : Element → ζ[ n ∈ ℕ ] Vec (Fin №nonterminals)
  n ×× n ≤ №nonterminals
setIndexes e = setIndexes’ e ≤-refl


setAsList : Element → Data.List.List Nonterminal
setAsList set = toList $ map Fin→N $ proj₁ $ proj₂ $ setIndexes set

-- Union of two sets of elements (_+_ action)
_∪′_ : {n : ℕ} → Vec Bool n → Vec Bool n → Timed (Vec Bool n)
lhs ∪′ rhs = zipWith′ (lift′ _∨_)    lhs rhs


_∪″_ : Timed Element → Timed Element → Timed Element
_∪″_ = lift″ _∪′_

allPairs : {n m : ℕ} → {A B : Set} → Vec A n → Vec B m →
        Vec (A ×× B) (n * m)
```

```
allPairs x y = map _,_ x ⊛* y

allPairs' : Element ×× Element → ζ[ n ∈ ℕ ]
  (Vec (Nonterminal ×× Nonterminal)
  n ×× n ≤ (№nonterminals * №nonterminals) )
allPairs' (x , y) with setIndexes x | setIndexes y
... | xlength , xvec , xproof | ylength , yvec , yproof = xlength * ylength ,
        allPairs (map Fin→N xvec) (map Fin→N yvec) , *-mono-≤ xproof yproof

isRule' : {n : ℕ} → Vec (Language.Rule Terminal Nonterminal) n
  → Nonterminal → Nonterminal → Timed (Maybe Nonterminal)
isRule' [] l r = nothing'
isRule' (a ⇒ t :: xs) l r = isRule' xs l r''
isRule' (c ⇒ a ∘ b :: xs) l r with a N≟ l | b N≟ r
... | true because proof₁ | true because proof₂ = just c'
... | true because proof₁ | false because proof₂ = isRule' xs l r''
... | false because proof₁ | br = isRule' xs l r''

isRule : Nonterminal ×× Nonterminal → Timed (Maybe Nonterminal)
isRule (l , r) = isRule' (rules ) l r

appendIf' : Element → Maybe Nonterminal → Timed Element
appendIf' set (just c) = set ∪[ c ]'
appendIf' set nothing = set'

appendIf'' : Timed Element → Timed (Maybe Nonterminal) → Timed Element
appendIf'' = lift'' appendIf'

⇒∘-helper : {n : ℕ} → Vec (Nonterminal ×× Nonterminal) n → Timed Element
⇒∘-helper [] = EmptyElement
⇒∘-helper (x :: xs) = appendIf'' (⇒∘-helper xs) (isRule x)

⇒∘-helper² : {n : ℕ} → Vec≤.Vec≤ (Nonterminal ×× Nonterminal) n
  → Timed Element
⇒∘-helper² (vec Vec≤., bound) = ⇒∘-helper vec

- All combining production rules (_*_ action)
- {∀abc c | a ∈ lhs ∧ b ∈ rhs ∧ c ← a ∘ b ∈ Productions }
⇒_∘_ : Element → Element → Timed Element
⇒ lhs ∘ rhs with allPairs' (lhs , rhs)
```

```
... | n , vec , prf = ⇒∘-helper vec -allPairs' lhs rhs

_•′_ : {n : ℕ} → Vec Element n → Vec Element n → Timed Element
(row :: row₁ :: rows) •′ (discard :: column :: columns)
  = (⇒ row ∘ column) ∪″ ((row₁ :: rows) •′ (column :: columns))
(x :: []) •′ (y :: []) = EmptyElement
([]) •′ ([]) = EmptyElement

_•″_ : {n : ℕ} → Timed (Vec Element n) → Timed (Vec Element n) →
         Timed Element
_•″_ = lift″ _•′_

_⊗′_ : {n m j : ℕ} → (suc n) × (suc m) → (suc m) × (suc j)
  → Timed ((suc n) × (suc j))
_⊗′_ = MatrixMultiplicationBase Element _•″_

_⊗″_ : {n m j : ℕ} → Timed (suc n × suc m) → Timed (suc m × suc j)
  → Timed (suc n × suc j)
_⊗″_ = lift″ _⊗′_

create : {n : ℕ} → Vec Terminal n → Timed (n × n)
create string = tabulate′ λ x → tabulate′ λ y → create' string x y
  where terminalRules : {n : ℕ} → Vec (Language.Rule Terminal Nonterminal) n
            → Terminal → Timed Element
        terminalRules [] t = EmptyElement
        terminalRules ((x ⇒ x₁) :: v) t with x₁ T≟ t
        ... | true because _ = terminalRules v t ∪[ x ′ ]″
        ... | _              = terminalRules v t
        terminalRules (_ :: v) t = terminalRules v t
        create' : {n : ℕ} → Vec Terminal n → Fin n → Fin n → Timed Element
        create' {n} string x y with x F≟ y
        ... | true because _ = terminalRules rules $ lookup string y
        ... | _              = EmptyElement

_⊕′_ : {n m : ℕ} → n × m → n × m → Timed (n × m)
lhs ⊕′ rhs = zipWith′ (λ x y → zipWith′ _∪′_ x y) lhs rhs

_⊕″_ : {n m : ℕ} → Timed (n × m) → Timed (n × m) → Timed (n × m)
_⊕″_ = lift″ _⊕′_
```

Appendix G

CYK proofs

```
open import Language

module Matrix.CykProperties
  {T N : Set}
  (Lang : Language T N) where

open import CykMatrix Lang

open import Data.Fin hiding (_≤_; _+_)
open import Data.Nat
open import Data.Nat.Properties
open import Data.Bool hiding (_≤_; T)
open import Data.Maybe
import Data.Vec.Bounded as Vec≤
open Data.Nat.Properties.≤-Reasoning
open import Data.Product renaming (_×_ to _××_) hiding (map; zip)
open import Data.Product.Properties
-open import Pair renaming (Pair to _××_) hiding (map)
open import Relation.Nullary
open import Relation.Binary using (Decidable)
open import Relation.Binary.PropositionalEquality
open import Data.Vec.Properties
open import Function

open import CostBase
open import CostBase.Properties
```

```
open import CostVec
open import CostVec.Properties2
open import Matrix.Properties2
import Relation.Binary.PropositionalEquality using (cong)

open Language.Language Lang

1≤sucn : {n : ℕ} → 1 ≤ suc n
1≤sucn {zero} = s≤s z≤n
1≤sucn {suc n} = ≤-step 1≤sucn


∪[]′-cost-proof : (e : Element) → (n : N) →
        cost (e ∪[ n ]′) ≤ №nonterminals
∪[]′-cost-proof e n = []:=′-cost-proof e (N→Fin n) true

appendIf′-cost-proof : (e : Element) → (m : Maybe N)
  → cost(appendIf′ e m) ≤ №nonterminals
appendIf′-cost-proof e nothing = z≤n
appendIf′-cost-proof e (just x) = ∪[]′-cost-proof e x

isRule’-cost-proof : {n : ℕ} → (xs : Vec (Rule T N) n) → (l r : N)
  → cost(isRule’ xs l r) ≤ n
isRule’-cost-proof [] l r = z≤n
isRule’-cost-proof (x :: xs) l r with x
isRule’-cost-proof (x :: xs) l r | a ⇒ t with ″suc-cost-proof (isRule’ xs l r)
isRule’-cost-proof (x :: xs) l r | a ⇒ t | ≡ rewrite ≡
  = s≤s (isRule’-cost-proof xs l r)

isRule’-cost-proof (x :: xs) l r | c ⇒ a o b with a N≟ l | b N≟ r
isRule’-cost-proof (x :: xs) l r | c ⇒ a o b | true because proof₁
  | true because proof₂ = z≤n
isRule’-cost-proof (x :: xs) l r | c ⇒ a o b | false because proof₁
  | does₂ because proof₂ rewrite ″suc-cost-proof (isRule’ xs l r)
  = s≤s (isRule’-cost-proof xs l r)
isRule’-cost-proof (x :: xs) l r | c ⇒ a o b | true because proof₁
  | false because proof₂ rewrite ″suc-cost-proof (isRule’ xs l r)
  = s≤s (isRule’-cost-proof xs l r)

pemdas-lemma : (q r n : ℕ) → q + (r + n * (q + r)) ≡ q + r + n * (q + r)
pemdas-lemma q r n rewrite +-assoc q r (n * (q + r)) = refl
```

```
order-lemma : (i j k : ℕ) → i + j + k ≡ i + k + j
order-lemma i j k rewrite +-assoc i j k | +-comm j k | sym (+-assoc i k j)
  = refl

isRule-cost-proof : (pair : N ×× N) → cost (isRule pair) ≤ №rules
isRule-cost-proof (fst , snd) = isRule'-cost-proof rules fst snd

+-mono³-≤ : {n m o i j k : ℕ} → n ≤ i → m ≤ j → o ≤ k → n + m + o ≤ i + j + k
+-mono³-≤ p1 p2 p3 = +-mono-≤ (+-mono-≤ p1 p2) p3

⇒∘-helper-cost-proof : {n : ℕ} → (xs : Vec (N ×× N) n)
  → cost(⇒∘-helper xs) ≤ n * (№nonterminals + №rules)
⇒∘-helper-cost-proof [] = z≤n
⇒∘-helper-cost-proof (x :: xs)
  rewrite lift″-cost-proof appendIf′ (⇒∘-helper xs) (isRule x)
  | order-lemma ( cost (appendIf′ (value (⇒∘-helper xs))
  (value (isRule' rules (proj₁ x) (proj₂ x)))))
  (cost (⇒∘-helper xs)) (cost (isRule' rules (proj₁ x) (proj₂ x)))
  = +-mono³-≤
    (appendIf′-cost-proof (value (⇒∘-helper xs))
    (value (isRule' rules (proj₁ x) (proj₂ x))))
    (isRule'-cost-proof rules (proj₁ x) (proj₂ x))
    (⇒∘-helper-cost-proof xs)

*-mono²ˡ-≤ : ∀ {p i j n m k} → i ≤ n → j ≤ m → k ≤ i * j * p → k ≤ n * m * p
*-mono²ˡ-≤ {p} i≤n j≤m k≤ijp = ≤-trans k≤ijp
                              $ *-monoˡ-≤ p
                              $ *-mono-≤ i≤n j≤m

cost≤rule-count³-lemma : (l r : Element)
  → cost(⇒ l ∘ r) ≤ №nonterminals * №nonterminals *
                      (№nonterminals + №rules)
cost≤rule-count³-lemma l r with setIndexes l | setIndexes r
cost≤rule-count³-lemma l r | nl , ls , lproof | nr , rs , rproof with (concat
  (CostVec.map (λ f → CostVec.map f (CostVec.map Fin→N rs))
  (CostVec.map _,_ (CostVec.map Fin→N ls)
  )))) | *-mono-≤ lproof rproof
cost≤rule-count³-lemma l r | nl , ls , lproof | nr , rs , rproof | m | prf =
  *-mono²ˡ-≤ lproof rproof (⇒∘-helper-cost-proof m)
```

```
∪′-cost-proof : {n : ℕ} → (l r : Vec Bool n) → cost (l ∪′ r) ≤ n * 2
∪′-cost-proof l r with l | r
∪′-cost-proof l r | [] | [] = z≤n
∪′-cost-proof l r | l₁ ∷ ls | r₁ ∷ rs rewrite lift″-cost-proof _∷′_
  (time 1 (l₁ ∨ r₁))    (zipWith′ (λ x y → time 1 (x ∨ y)) ls rs)
  = s≤s $ s≤s $ ∪′-cost-proof ls rs

- This function is terminating,
- and the recursive call is trivially smaller.
{-# TERMINATING #-}
•′-cost-proof : {n : ℕ} → (x y : Vec Element n)
  → cost (x •′ y) ≤ n * (№nonterminals * 2 + №nonterminals * №nonterminals
  * (№nonterminals + №rules))
•′-cost-proof [] [] = z≤n
•′-cost-proof (x ∷ []) (y ∷ []) = z≤n
•′-cost-proof (x ∷ xs@(x₁ ∷ x₂)) (y ∷ ys@(y₁ ∷ y₂))
  rewrite lift″-cost-proof _∪′_ (⇒ x ∘ y₁) ((x₁ ∷ x₂) •′ (y₁ ∷ y₂)) =
    +-mono³-≤
      (∪′-cost-proof (value (⇒ x ∘ y₁)) (value (xs •′ ys)))
      (cost≤rule-count³-lemma x y₁)
      (•′-cost-proof xs ys)

⊗-cost-proof : {n m j : ℕ} → (m₁ : n × m) → (m₂ : m × j)
  → cost (m₁ ⊗′ m₂) ≤ n * (j * (m * (№nonterminals * 2 + №nonterminals
  * №nonterminals * (№nonterminals + №rules)) + n + m * suc j))
⊗-cost-proof m₁ m₂ = matrix-multiply-cost-proof _•′_
  (№nonterminals * 2 + №nonterminals * №nonterminals
  * (№nonterminals + №rules)) •′-cost-proof m₁ m₂
```