Stockholms universitet

A Formal Proof of Karatsuba's Algorithm in Agda

Gustav Sahlin

Supervisors: Anders Mörtberg, Max Zeuner
Examiner: Lars Arvestad
Submission date: 2022-05-22

Kandidatuppsats i datalogi
Bachelor Thesis in Computer Science

Abstract

A way to ensure the correctness of an algorithm is to use formalized methods and verify it with a proof assistant. One of these proof assistants is Agda, a dependently typed programming language. This thesis aim is to prove in Agda that Karatsuba's algorithm for polynomials is equal to an ordinary polynomial multiplication algorithm. The polynomials are represented with lists and integer coefficients. The first part of the thesis is a brief introduction to Agda. The second part presents an implementation of the polynomial structure. A comparison is made between implementations of Karatsuba's algorithm in Agda and the functional programming language Haskell. In the final part, a proof of correctness is implemented. Due to lack of time, a few lemmas could not be formalized but are assumed in Agda and proven on paper in the thesis.

Sammanfattning

Ett sätt att verifieras en algoritm är att använda formella metoder med en bevisassistent. En av dessa bevisassistenter är Agda, ett programmeringsspråk med beroende typer. Syftet med denna uppsats är att bevisa i Agda att Karatsubas algoritm utför multiplikation av polynom korrekt. Polynom implementeras med hjälp av listor och heltalskoefficienter. Den första delen av uppsatsen är en kort introduktion till Agda. I den andra delen presenteras en implementering av polynomstrukturen. En jämförelse görs mellan implementeringar av Karatsubas algoritm i Agda och det funktionella programmeringsspråket Haskell. I den sista delen implementeras ett korrekthetsbevis. På grund av tidsbrist kunde några lemman inte formaliseras. De antas i Agda och bevisas på papper i uppsatsen.

Contents

1 Introduction

One crucial question in computer science is: How do we know that the computer programs we write will do what they are supposed to? The tool generally used in the industry for this problem is software testing. We test our programs with different inputs to see if they produce the correct output. If we manage to produce an error with our test, we change the program's code to make it pass the test. How thoroughly the testing is performed depends on how important the program is and how significant the consequences of a failure would be. After the software testing phase, we can be more or less sure that the program will produce the correct output. There is one problem with this, though. We can almost never be *certain* that the output will be 100% correct because we can not test all cases for non-trivial programs. Usually, the input space is infinite. To be certain about something, we need to prove it.

The Curry-Howard correspondence states that computer programs and intuitionistic logic proofs if you look at them the right way, are the same thing. Using this correspondence, an alternative to software testing is to do a proof of correctness. With proof, we can be sure about what a program will produce. One way of doing this is with formalized proofs in dependently typed programming languages. This thesis aims to prove the Karatsuba Algorithm for polynomial multiplication in Agda, a dependently typed programming language. The notion of dependent types is used in a mathematical field called type theory, a foundation of mathematics different from set theory. Agda is based on an improved version of Per Martin-Löf's Type Theory (MLTT).

2 The Agda Proof Assistant

Because Agda is based on type theory [5], it has certain properties that make it suitable for proving mathematical statements. One can prove that Agda is total, meaning that it will terminate [2]. There will be no runtime errors or nonterminating programs. Because of this fact and that Agda uses dependent types, it is suitable to be used as a proof assistant. Note that it is not possible to give a complete overview of Agda or type theory in this thesis. In this section, Agda will be presented so that it is possible to follow the proof of the thesis, and some notes will be provided on aspects of type theory.

2.1 An example of Agda code

Agda has a syntax that is similar to the functional programming language Haskell. The function length that returns the length of a list can be written in Haskell as:

```
length : [ a ] -> Int
length [] = 0
length (x : xs) = 1 + (length xs)
```

The same function in Agda can be written:

```
length : List A → ℕ
length []        =  zero
length (x :: xs)  =  suc (length xs)
```

The length functions are almost the same. Notable differences are that Agda uses Unicode characters. In Agda we write `List` instead of using brackets. The polymorphic type "a" in Haskell is in Agda written as "`A`". Agda is using ℕ instead of integers and it is common to use `suc` instead of "`1 +`". `suc` is short for successor. It is a function that returns the successor of a given number.

2.2 Creating numbers

To understand how Agda works, let's look at the natural numbers. We can define them as:

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

ℕ is a data type with two constructors, zero and suc. zero takes no argument, and suc takes one argument. This makes zero an element of the type ℕ. suc when provided an element (number) represents that element's successor. This is an inductive definition. These are extensively used in Agda. The type ℕ is called an inductively defined type [2]. One can see the constructors as rules, and with the two rules of ℕ, the only way we can describe elements of the type is:

```
zero
suc zero
suc (suc zero)
...
```

The element zero represents 0. The element suc zero represents 1, and so on.

2.3 Defining functions

We can define functions that do something with this data type concerning natural numbers, i.e., properties of the natural numbers. To get a predecessor of a number, we can define a function pred that takes an element of our defined typed ℕ and returns an element of type ℕ [2]:

```
pred : ℕ → ℕ
pred zero = zero
pred (suc n) = n
```

Further, we can define addition and multiplication recursively as:

```
_+_ : ℕ → ℕ → ℕ               _*_ : ℕ → ℕ → ℕ
zero + n = n                   zero    * n  =  zero
(suc m) + n = suc (m + n)     (suc m) * n  =  n + (m * n)
```

For addition, this means that if _+_ is given a zero and a number, it will return the number. Multiplication is defined in terms of addition.

2.4 Equality and reasoning

To be able to reason about types, we will need equality. To prove things, we want to be able to show that two terms of a type are equal [11]. Because the _=_ is already used for definitions, we will use the triple bar (_≡_) to represent this equality. _≡_ takes two types (the same terms), and the constructor represents proof that they are equal. It is a data type defined as:

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

One important property that equality has is congruence:

```
cong : ∀ {A B : Set} (f : A → B) {x y : A}
  → x ≡ y
    ---------
  → f x ≡ f y
cong f refl  =  refl
```

If two terms are equal and a function is applied, the result is still equal. When proving properties, we will reason in a chain. To connect the chain, the "≡⟨_⟩" operator will be used. It is needed because not everything is reflexive. Inside the brackets, one can make justifications for equality. An example of this is the term +-identity^r. It reads out as "right identity" and means that any number m added with 0 on the right-hand side will be equal to m:

```
+-identityʳ : ∀ (m : ℕ) → m + zero ≡ m
+-identityʳ zero = refl
+-identityʳ (suc m) =
  begin
    suc m + zero
  ≡⟨⟩
    suc (m + zero)
  ≡⟨ cong suc (+-identityʳ m) ⟩
    suc m
    ∎
```

Above is a recursive definition. We have a base case and an inductive case. The base case is reflexive because 0 + 0 is equal to 0. In the inductive case, we provide evidence for the proposition in the lines between `begin` and the black box ∎, and each line is connected with the ≡⟨_⟩ operator. If ≡⟨_⟩ is empty, the lines above and below are reflexive. In this case, we first do a reflexive step for clarity, then perform a recursive call inside the brackets of the ≡⟨_⟩ operator. We explicitly need to wrap the justification with congruence on `suc` for the equality to hold. The black box marks the end of the proof. The type of `+-identityʳ` is dependent. Dependent types are needed to express statements containing "for all", and "exists". A dependent type is a family of types that are indexed by objects of another type [2]. In the case of `+-identityʳ`, `m + zero ≡ m` is indexed by `m`.

2.5 Propositions as types

In type theory and in Agda, we can view propositions as types. The notion of a type is used to describe mathematical objects. Types have inhabitants. If we compare this to set theory, then a type would correspond to a set, and an inhabitant corresponds to a member of a set. The example in 2.4 where we prove `+-identityʳ` is a proposition. We managed to finish the proof, meaning that the terms are equal. We have verified the proposition and showed that the type `+-identityʳ` has an inhabitant.

3 Karatsuba Algorithm for Polynomials

Until the 20th century, the fastest way of multiplying two numbers was through the sometimes called "school multiplication" or "long multiplication". We multiply each digit in the first number to each digit in the second number, with the correct shifting, and then add the result. A math student in Russia changed this.

Karatsuba's Algorithm (KA) is a recursive multiplication algorithm. Anatoly Karatsuba discovered it in 1960. When attending a seminar held by Andrey Kolmogorov [6], Kolmogorov conjectured that the multiplication of two n-digit numbers requires $\mathcal{O}(n^2)$ operations. Karatsuba managed within a week to find a divide-and-conquer algorithm with better performance ($\mathcal{O}(n^{log_2(3)})$ [7]), thus, disproving the conjecture.

3.1 Example for polynomials

KA can be implemented for polynomials as well [1]. Polynomial multiplication works the same way as multiplication of numbers, but without the "carry-over". KA uses the fact that a polynomial p with $n = degree(p) + 1$ (using floor division) can be written as:

$$p = p_2 + x^{\frac{n}{2}} p_1.$$

For example if $p = 1 + 2x + 3x^2 + 4x^3$, p can be written as:

$$p = 1 + 2x + 3x^2 + 4x^3 = (1 + 2x) + (3 + 4x)x^{\frac{4}{2}}$$

If we multiply two polynomials p and $q = (5 + 6x) + (7 + 8x)x^{\frac{4}{2}}$ they can be written as:

$$((1 + 2x) + (3 + 4x)x^2) \cdot ((5 + 6x) + (7 + 8x)x^2)$$

Let p and q be general polynomials. Let a and c be the higher degree parts and b and d the lower degree parts and we let $n = degree(min(p, q)) + 1$. We get the general case:

$$p \cdot q = (a \cdot x^{\frac{n}{2}} + b)(c \cdot x^{\frac{n}{2}} + d)$$

After multiplication we get:

$$a \cdot c \cdot x^{2 \cdot \frac{n}{2}} + (a \cdot d + b \cdot c)x^{\frac{n}{2}} + b \cdot d$$

Here we can make the observation that $(a \cdot d + b \cdot c)$ can be rewritten as:

$$(a \cdot d + b \cdot c) = (a + b)(c + d) - ac - bd$$

The whole expression can be written as:

$$a \cdot c \cdot x^{2 \cdot \frac{n}{2}} + ((a + b)(c + d) - ac - bd)x^{\frac{n}{2}} + b \cdot d$$

The equation above means that we only need to make three multiplications compared to ordinary multiplications four. But, we only make the multiplication if one of the polynomials degree is less than or equal to 2. Otherwise, we recursively call Karatsuba again. KA implemented in Haskell and Agda can be found in section 3.3.

3.2 Polynomial representation

KA performs multiplication for polynomials over any ring. We will restrict this in the thesis and only use integer coefficients. We will use lists to represent polynomials. For example the polynomial $(1 + 2x + 3x^2)$, is represented by the list `[1,2,3]`, functionally written as:

```
1 :: 2 :: 3 :: []
```

A polynomial can in general can be written as:

$$(a_0 x^0 + a_1 x^1 + a_2 x^2 + \ldots + a_i x^i)$$

In Agda we will let it correspond to:

```
a_0 :: a_1 :: a_2 :: ... :: a_i :: []
```

Apart from the pure representation, we will need operations on the lists corresponding to operations on polynomials. The most obvious one which we will show equal to KA is multiplication:

```
_*p_ : List ℤ → List ℤ → List ℤ
_*p_ [] ys = []
_*p_ xs [] = []
_*p_ (x :: xs) ys = (map (x *_) ys) +p ( +0 :: xs *p ys)
```

The map function can be seen as scalar multiplication of the list ys. We add this with the recursive call of xs *p ys, raised by one degree by putting +0 in front of the list. Addition is defined as:

```
_+p_ : List ℤ → List ℤ → List ℤ
xs +p [] = xs
[] +p ys = ys
(x :: xs) +p (y :: ys) = x + y :: (xs +p ys)
```

Subtracion -p is defined as:

```
negPoly : List ℤ → List ℤ
negPoly [] = []
negPoly (x :: xs) = (-i x) :: negPoly xs

_-p_ : List ℤ → List ℤ → List ℤ
_-p_ xs ys = xs +p (negPoly ys)
```

shiftRight m corresponds the operation of multiplying a polynomial with x^m:

```
shiftRight : ℕ → List ℤ → List ℤ
shiftRight zero xs = xs
shiftRight (suc n) xs = +0 :: shiftRight n xs
```

The function lengthens a list by adding n starting zeroes to it. "shiftRight n []" represents a list of n zeroes.

3.3 Comparison of algorithms

Agda is implemented in the functional language Haskell, and as mentioned earlier, the syntax is very similar [3]. Therefore it can be a good idea to first implement algorithms in Haskell and then translate them to Agda and prove them there. Below is KA implemented in Agda:

```
karatsuba' : ℕ → List ℤ → List ℤ → List ℤ
karatsuba' zero xs ys = xs *p ys
karatsuba' (suc n) xs ys with
        ((((length xs) / 2) ⊓ (length ys / 2)) ≤? 2)
...    | (yes _) = (xs *p ys)
...    | (no _) = ((shiftRight (2 *ℕ m) ac) +p
        (shiftRight m ad_plus_bc)) +p bd
       where
           m = ((length xs / 2) ⊓ (length ys / 2))
           b = take m xs
           a = drop m xs
           d  = take m ys
           c = drop m ys
           ac = karatsuba' n a c
           bd = karatsuba' n b d
           ad_plus_bc =
           ((karatsuba' n (a +p b) (c +p d) -p ac) -p bd)
```

In Haskell KA can be written as:

```
karatsuba :: [Int] -> [Int] -> [Int]
karatsuba xs ys = if length xs <= 2 || length ys <= 2
    then mulPoly xs ys
    else ((shiftRight (toInteger (2 * m)) ac) ‘addPoly‘
    (shiftRight (toInteger m) ad_plus_bc)) ‘addPoly‘  bd
        where
        m = min (div (length xs) 2) (div (length ys) 2)
        b = take m xs
        a = drop m xs
        d = take m ys
        c = drop m ys
        ac = karatsuba a c
        bd = karatsuba b d
        a_plus_b = addPoly a b
        c_plus_d = addPoly c d
        ad_plus_bc = ((karatsuba a_plus_b c_plus_d)
        ‘subPoly‘ ac) ‘subPoly‘ bd
```

The inductive step in the Agda has a "`with`" clause checking the length of the lists. Depending on the lengths it performs cases "yes" or "no". It is the same operation as the "if-then-else" clause in Haskell where "yes" corresponds to "then" and "no" to "else". We are also using another operator, _≤?_. The operator makes it possible to use the outcome of a proposition. This is not needed in the definition of KA but will be used and explained in the proof.

There is one distinct difference between the algorithms in Haskell and Agda. In the definition in Agda there is a "`n : ℕ`" included and a base case performing `_*p_` if `n = zero`. We are using a helper function to initiate `n`. Agda needs the `n` for the algorithm to work. The reason is that without it, Agda is not sure if the recursion in the inductive case will make the algorithm terminate. We have to add a simpler recursion that the termination checker can handle. This recursion should not reach the case `n = 0` so that it intervenes with KA. What we do in this case is to initiate `n` to be the longest list length. We pass on the predecessor of `n` on each call. This recursion is slower than splitting list lengths in half, so this case will never be the one terminating KA, but it makes Agda accept the function.

3.4 QuickCheck

Before starting the proof, checking if the proposition is likely to hold is important. For this, QuickCheck was used. QuickCheck is a Haskell library that does property-based testing. Property-based testing tests function properties (meaning many cases) instead of just some cases, as is done in regular testing. It generates extremely thorough testing of functions. One way we can do this is if the function we are testing have some inverse property (`reverse (reverse xs) == xs`) or if two functions do the same thing (`mulPoly xs ys == karatsuba xs ys`). We use variables in our testing functions instead of single test cases, and QuickCheck generates a chosen amount of random test cases. For example, we can write:

```
prop :: [Int] -> [Int] -> Bool
prop a b =   (karatsuba a b) == (mulPoly a b)
```

We can then run "`quickCheck (withMaxSuccess 10000 prop)`" to get 10000 randomly generated test cases. It was done (successfully) in Haskell on `karatsuba` and `mulPoly` to determine if we can hope for a proof.

3.5 Problems with list representation

The list representation complicates some things.

3.5.1 Trailing zeroes

Lists with only zeroes and lists with trailing zeroes are one side effect of the polynomial representation. The list `[1,2,0]` represents the same polynomial as `[1,2]`, but Agda will not interpret them as the same. There is not a one-to-one but a many-to-one correspondence between the list representation and ordinary polynomials.

In the specific case of the KA, polynomial subtraction (`-p`) is performed. In particular the operation `xs -p xs`. The operation will generate a list of zeroes added to some other list. We need to ensure Agda that the list of zeroes is not longer than the list we will add it with. If it is not, we get trailing zeroes of arbitrary length, and this is something we want to avoid. In the cases where we have instances of zero lists in the Karatsuba proof, these will be shorter than the other polynomials, but there are a lot of comparisons needed to prove this.

3.5.2 Dependence on length

Several functions depend on the polynomial length and make the proof more complicated. The function taking a polynomial and splitting it into two parts, for instance, needs evidence that the index we are splitting on is smaller or equal to the list length.

3.6 Alternative representations

There are other alternatives to represent polynomials. In [8], a proof of Karatsuba for polynomials in Agda has been given. Polynomials are defined as record types in the paper. The record type group values together [4], so apart from the list itself, proofs can be included in a record. The proof ensures the last entry of the list is non-zero [8]. The representation does not consider lists that end with a zero. By doing this, the author avoids the problems with trailing zeroes.

[9] also implements a proof for Karatsuba for Polynomials, but in another dependently typed programming language called Coq. In Coq, there is a library called MatComp that does the same representation with records as in [8]. The MathComp library has its roots in the Four Color Theorem proof. Due to some technicalities in [9], the polynomials are represented as lists without any proof. These are then proved equal to the corresponding polynomials in the MatComp library.

4 The correctness proof

This thesis aims to show

```
ismul : (xs ys : List ℤ)
  → xs *p ys ≡ karatsuba xs ys
```

We will need properties for `*p` and `+p` for the proof. We will use identity, associativity, and commutativity for `+p`. For `*p` identity, associativity, commutativity, and distributivity over `+p` is proven. The same properties will hold for Karatsuba, but in the proof, we will go from `*p` to `karatsuba` because the definition of `*p` is simpler. When the proof is finished, we can be sure that these properties hold for Karatsuba as well.

4.1 Example lemmas

A lot of functions and lemmas were implemented for the proof. Proofs about properties of length, properties of shiftRight, properties of operations, and all these combined were made. All of these reside in a pretty extensive library to the proof [10]. This section will consist of a few examples. Two simple ones and one a little bit more complex.

4.1.1 Length of list relation to shiftRight

The following lemma proves that we can rewrite the length of `shiftRight m xs`, as the length of the list `xs` added to `m`.

```
shiftRight-list-len : ∀ (m : ℕ) (xs : List ℤ)
  → length (shiftRight m xs) ≡ length xs + m
shiftRight-list-len zero xs
        rewrite +-identityʳ (length xs) = refl
shiftRight-list-len (ℕ.suc m) xs =
  begin
    suc (length (shiftRight m xs))
  ≡⟨ cong ℕ.suc (shiftRight-list-len m xs) ⟩
    suc (length xs + m)
  ≡⟨ cong ℕ.suc (+-comm (length xs) m) ⟩
    suc (m + length xs)
  ≡⟨⟩
    suc m + length xs
  ≡⟨ +-comm (suc m) (length xs) ⟩
    length xs + suc m
    ∎
```

In the base case we prove that `length xs + m` is equal to `length xs`. The inductive case is straightforward with a recursive call and commutativity of _+_.

4.1.2 shiftRight inside shiftRight

If we have a `shiftRight` inside a `shiftRight` both applied to the same list, we can simplify:

```
shiftRight-shiftRight : ∀ (m n : ℕ) (xs  : List ℤ)
  → shiftRight m (shiftRight n xs) ≡ shiftRight (m + n) xs
shiftRight-shiftRight zero n xs = refl
shiftRight-shiftRight (suc m) n xs
        rewrite shiftRight-shiftRight m n xs = refl
```

The lemma gives us that we can write it as "`shiftRight (m + n) xs`". The base case is reflexive, and the inductive case is straightforward with a recursive call.

4.1.3 Distributivity of *p over shiftRight

Distributivity of `*p` over `shiftRight` is one of the more complex and important proofs. It is expressing

$$(a \cdot b)x^m = ax^m \cdot b.$$

In the list representation it is written as:

```
shiftRight m (xs *p ys) ≡ (shiftRight m xs) *p ys
```

The property does not work the same, though, because `shiftRight m []` is not empty but a list of zeroes of length `m`. The problem with lists of zeroes is discussed in 3.5.1. Given this, we need to have a condition on the polynomial lengths.

```
shiftRight-*p : ∀ (m : ℕ) (xs ys : List ℤ)
  → zero < length xs
  → zero < length ys
  →  shiftRight m (xs *p ys) ≡ (shiftRight m xs) *p ys
shiftRight-*p zero xs ys zero<lenXS zero<lenYS = refl
shiftRight-*p (ℕ.suc m) (x :: xs) (y :: ys) z<xs z<ys =
   begin
     +0 :: shiftRight m ((x :: xs) *p (y :: ys))
    ≡⟨ cong (+0 ::_)
    (shiftRight-*p m (x :: xs) (y :: ys) z<xs z<ys) ⟩
```

```
      +0 :: ((shiftRight m (x :: xs)) *p (y :: ys) )
    ≡⟨ cong (+0 ::_) (equality) ⟩
       (+0 :: (shiftRight m (x :: xs))) *p (y :: ys)
     ∎
 where
 equality : (shiftRight m (x :: xs) *p (y :: ys)) ≡
 (map (_*_ +0) ys +p (shiftRight m (x :: xs) *p (y :: ys)))

  equality =
   begin
    (shiftRight m (x :: xs) *p (y :: ys))

    ≡⟨ sym (addZeroes (length ys)
      (shiftRight m (x :: xs) *p (y :: ys))
      (<⇒≤ (length-lemma m y (x :: xs) ys z<xs ))) ⟩

      shiftRight (length ys) [] +p
      (shiftRight m (x :: xs) *p (y :: ys))

    ≡⟨ cong (_+p (shiftRight m (x :: xs) *p
      (y :: ys))) (sym (map-shiftRight-zero ys)) ⟩

      (map (_*_ +0) ys +p
      (shiftRight m (x :: xs) *p (y :: ys)))
         ∎
```

The base case is reflexive. In the inductive case the first step is to do a recursive call. To finish the proof we need to show:

```
  +0 :: ((shiftRight m (x :: xs)) *p (y :: ys) )
≡⟨ cong (+0 ::_) (equality) ⟩
   (+0 :: (shiftRight m (x :: xs))) *p (y :: ys)
```

Here we us a proof inside the proof. We call it `equality` and we must prove:

```
equality : shiftRight m (x :: xs) *p (y :: ys) ≡
    map (_*_ +0) ys +p (shiftRight m (x :: xs) *p (y :: ys))
```

In "equality" we add a list of zeroes of `length ys` to `(shiftRight m (x :: xs)) *p (y :: ys)`. This can be done without consequences if the list of zeroes have shorter or equal length, which is true in this case. `ys` is of course shorter than `y :: ys` multiplied with something else of positive length. We finally rewrite "`shiftRight (length ys) []`" to "`map (_*_ +0) ys`" and we have thus proved `equality`.

4.2 Agda standard library

Agda has a standard library that contains tools for writing proofs. The library includes data types such as naturals, integers, relations, and properties about these. The library has been used extensively for relational proofs, where we compare lengths. One example is the constructor `m≤n⇒m⊓o≤n`. If `m` is less than or equal to `n`, then the minimum of `m` and `o` is less than or equal to `n`.

4.3 Lemmas not implemented in Agda

There was not enough time to implement all lemmas. We motivate these on paper.

4.3.1 Length related lemmas

The notion that for two polynomials p and q (deg stands for degree),

$$deg(p \cdot q) = deg(p) + deg(q) - 1, deg(p) > 0 < deg(q)$$

can be expressed as:

```
→ 0 < length xs
→ 0 < length ys
→ length (xs *p ys) = (length xs +p length ys) - 1
```

There was not enough time to prove this, and QuickCheck tests have been made instead. The test was

```
prop4 :: [Integer] -> [Integer] -> Property
prop4 xs ys = (xs /= []) && (ys /= []) ==>
    length (xs ‘mulPoly‘ ys) == length xs + length ys - 1
```

The command "`quickCheck (withMaxSuccess 100000 prop4)`" generated:

```
+++ OK, passed 100000 tests; 24789 discarded.
```

The discarded test cases were where the lists were empty. Evidently, a case QuickCheck tests thoroughly. Three other statements had to be assumed about length relations. Note that these are operating over naturals and that division round down. The first one is

```
m/2>2⇒5<m : ∀ (m : ℕ)
→ 2 < m / 2
→ 5 < m
```

It reads out: "If m divided by 2 is greater than 2, then m is greater than 5". The second statements assume:

```
drop-lemma : (xs ys : List ℤ)
→ 5 < (length xs)
→ 5 < (length ys)
→ 3 ≤ length (drop ((length xs / 2) ⊓ (length ys / 2)) xs))
```

The length of xs and ys is greater or equal to 6. We take the minimum of half of `xs` and half of `ys` as `m`. It follows that the length of `drop m xs` is smaller or equal to 3. The third statement does the same thing as `drop-lemma` but to `ys`.

4.3.2 Reducing multiplication

The rewrite

$$(a \cdot d + b \cdot c) = (a + b)(c + d) - ac - bd,$$

is the key step in KA and reduces the number of multiplications we have to do (because we have already multiplied ac and bd in KA). We shall call the right-hand side of the equality sign for "the single-multiplication case". It turns out the rewrite is not true with the list representation of this thesis, ie

```
(a *p d) +p (b *p c) ≢
(((a +p b) *p (c +p d)) -p (a *p c)) -p (b *p d)
```

Consider the following example:

```
xs = [1,2,3,4,5,6,7]
ys = [1,2,3,4,5,6,7]
a = [4,5,6,7]
b = [1,2,3]
d = [2,2,2]
c = [4,5,6,7]
```

where

```
m = min (div (length xs) 2) (div (length ys) 2)
b = take m xs
a = drop m xs
d = take m ys
c = drop m ys
```

We get:

```
(a *p d) +p (b *p c) = [8,26,56,68,64,42]
(((a +p b) *p (c +p d)) -p (a *p c)) -p (b *p d) =
[8,26,56,68,64,42,0]
```

From this example we can see that the variable `m =`
`min (div (length xs) 2) (div (length ys) 2)` will control how the lists `a,b,c,d` will be formed, and in this case, the higher degree parts `a` and `c` gets longer than the lower degree parts `b` and `d`. When performing `((a +p b) *p (c +p d))`, `b` and `d` will not affect the highest degree in the list. In the next step, when we subtract, `(a *p c)` will have the same coefficient on the highest degree, and a trailing zero will therefore be generated. This is not a problem because, in the whole expression, we can use:

```
((shiftRight m ((((a +p b) *p (c +p d)) -p (a *p c)) -p
(b *p d))))
+p (shiftRight (2 *ℕ m) (a *p c))
≡
(shiftRight m ((a *p d) +p (b *p c)))) +p
(shiftRight (2 *ℕ m) (a *p c)
```

We know that the trailing zero will be consumed if we show:

```
length ((shiftRight m ((((a +p b) *p (c +p d)) -p (a *p c))
-p (b *p d)))) ≤ length (shiftRight (2 *ℕ m) (a *p c))
```

`(shiftRight (2 *ℕ m) (a *p c))` will not have any trailing zeroes because no subtraction has been done to it. We assume the length correspondence mentioned in 4.3.1:

```
→ 0 < length xs
→ 0 < length ys
→ length (xs *p ys) = (length xs +p length ys) - 1
```

Let:

```
p = (a *p d) +p (b *p c)
q = (((a +p b) *p (c +p d)) -p (a *p c)) -p (b *p d)
m = min (div (length xs) 2) (div (length ys) 2)
b = take m xs
a = drop m xs
d = take m ys
c = drop m ys
```

Without loss of generality we can assume (because of commutativty of `*p`):

```
length xs ≥ length ys
```

This implies:

```
m = length ys / 2
```

First assume `length ys` is even. This implies:

```
length c = length d = length b = m, length a ≥ m
```

Gives us lengths:

```
length p = length a + m - 1 = length xs - 1
length q = max(length xs - 1, length xs - 1, 2m-1) =
length xs - 1
```

Now for the case `length ys` is odd. This implies:

```
length d = length b = m
length a ≥ length c = m + 1
```

Gives us lengths:

```
length p = length xs - 1
length q = max(2m - 1, length a + m, length xs) = length xs
```

We use proven lemmas (see [10]) where _⊔_ is maximum:

```
shiftRight-list-len : ∀ (m : ℕ) (xs : List ℤ)
  → length (shiftRight m xs) ≡ length xs + m

x+y≡x⊔y : ∀ (xs ys : List ℤ)
  → length (xs +p ys) ≡ length xs ⊔ length ys
```

Then it follows, given the 4.3.1, that

```
length p + m ≤ 2m + length (a *p c)
length q + m ≤ 2m + length (a *p c)
```

i.e.

```
length p, length q ≤ m + length a + length c - 1
```

4.4 Karatsuba proof

We are going to perform the proof from *p to karatsuba'. For the whole proof, look at [10]. Below is the definition, the base case, and the inductive case, conditioned on the lengths of the lists.

```
ismul' : ∀ (n : ℕ) (xs ys : List ℤ)
  → xs *p ys ≡ karatsuba' n xs ys
ismul' zero xs ys = refl
ismul' (suc n) xs ys with
    (((length xs / 2) ⊓ (length ys / 2)) ≤? 2)
...                     | (yes _) = refl
...                     | (no ¬m≤2) =
```

We will do the proof with karatsuba' where the actual algorithm is; karatsuba without the apostrophe just passes on the lists and initiates the recursion mentioned in 3.3. Variables for the proof, the same as in the KA, are defined as:

```
m = ((length xs / 2) ⊓ (length ys / 2))
b = take m xs
a = drop m xs
d = take m ys
c =  drop m ys
ac = karatsuba' n a c
bd = karatsuba' n b d
ad_plus_bc = ((karatsuba' n (a +p b) (c +p d) -p ac) -p bd)
m>2 : 2 < m
m>2 = ≰⇒> ¬m≤2
```

We begin the proof.

The base case is reflexive by definition. If the condition of the "`with`" in the inductive case is true, i.e., one of the split lists has a length less than 3, we perform multiplication with *p. This case is, of course, reflexive. If the condition is false, we are provided with evidence (¬m≤2) that `m` (`length xs / 2) ⊓ (length ys / 2)`) is not less than 3. This evidence is needed later in several parts of the proof where we need to show that lists are not empty. We must rewrite `xs *p ys` to:

```
((shiftRight (2 *ℕ m) ac) +p
(shiftRight m ad_plus_bc)) +p bd
```

We start with the expression:

```
xs *p ys
```

First, we must split xs and ys into two parts each. For this, we use the function split-p. split-p splits a list zs into two parts on a given natural n, given that n ≤ length zs. We will provide split-p with m. m is as seen above the minimum length of length xs / 2 and length ys / 2. We prove the condition with cases in Agda. If the length of xs / 2 is the minimum, then m = xs / 2 < xs. In the other case, m = length ys / 2 is the minimum, then
m ≤ length xs / 2 < length xs. We will now have the expression:

```
(b +p shiftRight m a) *p (d +p shiftRight m c)
```

The next step is to use distributivity of *p over +p. We use both right and left distribution and get:

```
((b *p d) +p ((b *p (shiftRight m c)) +p
((shiftRight m a) *p d))) +p
(shiftRight m a *p shiftRight m c)
```

Continuing, we will use a lemma to rewrite the last part of the expression and make:

```
  (shiftRight m a *p shiftRight m c)
≡⟨ shiftRight-two-m m a c 2<xs 2<ys ⟩
  ((shiftRight (2 *ℕ m) (a *p c)))
```

The lemma shiftRight-two-m is a special case of shiftRight-*p, explained in 4.1.3. shiftRight-two-m need proof that both list lengths are greater than zero. We can prove this because xs and ys are greater than 5. Otherwise, we wouldn't be in the "no" case. xs and ys are split with drop m and take m, creating a, b,c,d and m are greater than 2. For b and d created with drop, we are using the assumptions m/2>2⇒5<m and drop-lemma stated in 4.3.1. With some moving of elements, we now have the expression:

```
((b *p d) +p (((shiftRight m c) *p b) +p
((shiftRight m a) *p d))) +p
(shiftRight (2 *ℕ m) (a *p c))
```

Finally, we have to rewrite the middle expression:

```
((shiftRight m c) *p b) +p ((shiftRight m a) *p d)
```

We will invoke shiftRight-*p (explained in 4.1.3) twice to be able to write:

```
(shiftRight m ((a *p d) +p (b *p c)))
```

Now we perform the key step where the rewrite of `(a *p d) +p (b *p c)` to the single multiplication case happens. This rewrite is assumed in the Agda proof and is proved in 4.3.2. We now have the expression:

```
((b *p d) +p
(shiftRight m ((((a +p b) *p (c +p d)) -p (a *p c))
-p (b *p d)))) +p (shiftRight (2 *ℕ m) (a *p c))
```

At this stage, we make the recursive calls. We will invoke the inductive hypothesis given by definition on all polynomials bounded by the operator `_*p_`, and with some moving around of the expressions, we arrive at:

```
((shiftRight (2 *ℕ m) ac) +p
(shiftRight m ad_plus_bc)) +p bd
```
∎

5 Conclusion

KA was proved equal to `mulPoly` in Agda, given the assumptions in 4.3. The assumptions about `lengths`:s would probably have been relatively easy to prove. The rewrite of `(a *p d) +p (c *p d)` is trickier, but it would boil down to proving length relations so that we know the trailing zero will be consumed.

The polynomial representation with lists had it's biggest strength in being intuitive, but it made proving KA equal to `mulPoly` a slow process. It generates a lot of cases because we have to consider the length of the lists. One solution would be not to consider lists with trailing zeroes, as mentioned in 3.6. Another possibility when working with integers could be to check if there are trailing zeroes and remove them. It corresponds to adding them to the degree zero part of a polynomial. If a list only contains zeroes, all zeroes except the last one are removed.

An interesting property of KA and the polynomial representation is that even though trailing zeroes can occur when performing subtraction, the KA deals with these, and will always produce a list without trailing zeroes.

It would be interesting to implement KA in Cubical Agda, an extension of Agda allowing a polynomial representation that might be easier to handle.

References

[1] J. Abdeljaoued and H. Lombardi. *Méthodes Matricielles Introduction à la Complexité Algébrique*. Springer-Verlag, 2004.

[2] A. Bove and P. Dybjer. Dependent types at work. `http://www.cse.chalmers.se/~peterd/papers/DependentTypesAtWork.pdf`, 2009.

[3] Ulf Norell et al. Installation. `https://agda.readthedocs.io/en/v2.6.2.1/getting-started/installation.html`, 2021.

[4] Ulf Norell et al. What is agda. `https://agda.readthedocs.io/en/v2.6.2.1/getting-started/what-is-agda.html`, 2021.

[5] Ulf Norell et al. Agda. `https://wiki.portal.chalmers.se/agda/pmwiki.php`, 2022.

[6] Anatoly Alexeyevich Karatsuba. The complexity of computations. `http://www.ccas.ru/personal/karatsuba/divcen.pdf`, 1995.

[7] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1981.

[8] S. D. Meshveliani. *A Certified Program for the Karatsuba Method to Multiply Polynomials*. Pleiades Publishing, 2022.

[9] Anders Mörtberg. *Formalizing Refinements and Constructive Algebra in Type Theory*. University of Gothenburg, 2014.

[10] Gustav Sahlin. Karatsuba-for-polynomials-in-agda. `https://github.com/gustav5/Karatsuba-for-Polynomials-in-Agda`, 2022.

[11] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. July 2020.