CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# Asymptotically Faster Bignum Multiplication in a Proven Correct Arithmetic Library

Master's thesis in Computer Science - Algorithms, Languages, and Logic

OLLE LINDEMAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

# Asymptotically Faster Bignum Multiplication in a Proven Correct Arithmetic Library

OLLE LINDEMAN



Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Asymptotically Faster Bignum Multiplication
in a Proven Correct Arithmetic Library OLLE LINDEMAN



Supervisor: Magnus Myreen, Department
Examiner: Andreas Abel, Department

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Asymptotically Faster Bignum Multiplication
in a Proven Correct Arithmetic Library
OLLE LINDEMAN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

Arithmetic functions, such as +, -, *, div and mod on arbitrary precision integers (not only machine integers), are used in many important computer programs such as cryptographic software and computer algebra systems. Formal guarantees of the correctness of these systems heavily rely on the correctness of the underlying arithmetic functions.

Cryptography algorithms such as RSA and Diffie–Hellman require efficient operations over large numbers. Arbitrary precision arithmetic, also called bignum arithmetic, is arithmetic of large numbers that exceeds the size of machine words.

In this thesis, we describe and implement two asymptotically fast bignum multiplication algorithms, namely the Karatsuba and Toom-3 algorithms, using the interactive theorem prover HOL4. Further, we take steps towards integrating the Karatsuba algorithm into CakeML's verified bignum library by specifying an implementation which satisfies parts of the required format by the integration infrastructure.

Keywords: Bignum arithmetic, formal methods, karatsuba, toom-3

# Acknowledgements

I would like to thank my supervisor Magnus Myreen for all the support, great ideas, and interesting discussions during this thesis project. Your help has been invaluable, and thanks for always being available for questions, be it high-level text problems or technical proof-related problems. Also, I would like to thank my examiner Andreas Abel for new perspectives and constructive feedback on the work presented in this thesis. Finally, I end with one of my favorite quotes from Yogi Berra, which is as true in life as within the field of formal methods:

"If you don't know where you're going, you might not get there."

Olle Lindeman, Gothenburg, August 2017

# Contents

# 1

# Introduction

Arithmetic functions (such as +, -, *, div and mod) on arbitrary precision integers, not only machine integers, are used in many important computer programs such as cryptographic software and computer algebra systems. Formal guarantees of the correctness of these systems heavily rely on the correctness of the underlying arithmetic functions [1].

Cryptography algorithms such as RSA and Diffie–Hellman require efficient operations over large natural numbers. Arbitrary precision arithmetic, also called bignum arithmetic, is arithmetic of large numbers that exceeds the size of machine words. These algorithms treat machine words in the same way as humans treat digits when doing calculations by hand. Many of the simpler algorithms are familiar from school, e.g. a variant of long multiplication is used when multiplying large numbers (complexity $O(n^2)$). However, when the inputs become very large there is a collection of algorithms that perform better than the basic algorithms. For example, good bignum libraries switch to fast Karatsuba and Toom-3 algorithms for multiplication when the inputs pass a certain size [7].

CakeML is a functional programming language, based on a substantial subset of Standard ML, with a proven correct compiler and runtime. That means that the compiler has been proven to transform CakeML programs into semantically equivalent machine code. The semantics and the compiler algorithm of CakeML are specified in the higher-order logic of the HOL4 theorem prover [13].

This thesis project describes the implementation and verification of the Karatsuba[5] and Toom-3[3] algorithms, two asymptotically fast multiplication algorithms (faster than $O(n^2)$), using the interactive theorem prover HOL4. Further, it takes steps towards integrating the Karatsuba algorithm into CakeML's verified bignum library by specifying an implementation which satisfies parts of the required format by the integration infrastructure.

## 1.1 Related Work

Verification of bignum arithmetic libraries is an active research field. This thesis work is based on the verified library described in Myreen and Curello [8], where the

authors use the HOL4 theorem prover and a proof producing compiler and decompiler to produce verified x86-64 code for arbitrary precision arithmetic functions. Recently this verified bignum library has been generalized and integrated into the verified CakeML compiler, which targets several machine languages (not just 64-bit x86).

Affeldt [1] verifies arithmetic functions written in assembly using the Coq proof-assistant. The paper includes a fast implementation of modular multiplication called *Montgomery multiplication*. Myreen and Gordon [9] show the possibility of using Hoare logic directly to manually verify the correctness of an optimized machine-code implementation of Montgomery multiplication. Neither of the above consider asymptotically fast algorithms for regular multiplication of large numbers (faster than $O(n^2)$).

Rieu-Helft et al. [10] presents a fully verified bignum library, developed using the Why3 program verifier. They base their implementation on the GNU Multi-Precision library (GMP), which is a widely used, safety-critical, library for arbitrary-precision arithmetic. But, for multiplication the authors only verify the so called *schoolbook* algorithm ($O(n^2)$), which GMP uses for smaller numbers.

## 1.2 Contributions

This thesis presents the implementation and verification of two asymptotically fast algorithms for bignum multiplication, namely the Karatsuba algorithm and the Toom-3 algorithm. Further it describes steps towards integrating the Karatsuba algorithm into the verified CakeML compiler's bignum library.

The main contribution of this thesis is the work towards an extension of the bignum library from Myreen and Curello [8] with an asymptotically fast multiplication algorithm for large numbers. To the best of our knowledge, the Karatsuba and Toom-3 algorithms have not before been verified in interactive theorem provers, nor integrated into a verified compiler. Karatsuba has a complexity of $O(n^{1.585})$ and Toom-3's complexity is $O(n^{1.465})$ [7].

## 1.3 Method

The method for completing an integration of an algorithm into CakeML's bignum library is described in the three steps below. This thesis implements step 1 and 2.

1. First, we define the algorithm as a function in logic (HOL4). We then prove that this function correctly implements integer multiplication. The function operates over lists of machine words. The top-level correctness theorem we prove for each of the algorithms is easy to state using the function `i2mw`, which

converts an integer into signed list of machine words. Let **mwi_alg_mul** be an implementation of a multiplication algorithm, then the correctness statement relates this function to multiplication over the integers ($\times$).

$$\forall i\ j.\ \texttt{mwi\_alg\_mul}\ (\texttt{i2mw}\ i)\ (\texttt{i2mw}\ j) = \texttt{i2mw}\ (i \times j)$$

2. In order to integrate the algorithm into the CakeML bignum library, we implement an imperative version suitable for the existing infrastructure in the CakeML compiler (see Section 5.1). We then prove that this implementation correctly realizes the function defined and verified in step 1. This implementation can be constructed by stepwise refinement, i.e. multiple implementations, that construct a chain of correctness proofs from the top-level to the final implementation, via intermediate implementations. These intermediate functions may comply to only a subset of the constraints the CakeML integration puts on the format, and are thus easier to verify.

3. With an implementation from the previous step, that fulfills the required format, there exists a proof-producing translation from programs in the specific format into programs of an intermediate language of the CakeML compiler. The verified implementation of the algorithm can thus be integrated into the CakeML compiler, in the compiler phase which implements bignum arithmetic.

## 1.4 Thesis Outline

*Chapter 2* presents the relevant background knowledge necessary to understand the content of this thesis. The chapter starts with an introduction to bignum arithmetic where also the *schoolbook* multiplication algorithm is described. Further it presents the CakeML-project and how the integration process works for the bignum library. The chapter ends with an description of the interactive theorem prover HOL4 and how bignums are represented within the system.

*Chapter 3* describes the Karatsuba algorithm and *Chapter 4* the Toom-3 algorithm. Both chapters follow the structure of first defining the algorithm, then continue with its specification in HOL, and finally define the verification process of the implementation.

*Chapter 5* describes the development of an intermediate implementation of the Karatsuba algorithm as a step towards integration into the CakeML's bignum library. Here we present what format the integration infrastructure requires and specify an implementation that comply to parts of it. The implementation is proven to correspond to the functional implementation for previous chapter. We conclude with a description of progress made in developing an implementation that strictly follows the format and thus is ready to be integrated into CakeML's bignum library.

*Chapter 6* concludes with a summary of the thesis work, a discussion of the difficulties encountered during the thesis project and directions of future work.

# 2

# Background

This chapter presents necessary background knowledge for understanding the work presented in this thesis. It starts with an introduction to bignum arithmetic, with a focus on the multiplication operation (Section 2.1). We also describe the basic schoolbook $O(n^2)$ algorithm for bignum multiplication. In Section 2.2 we present an overview of CakeML and its bignum library. After this we introduce the HOL4 interactive theorem prover and how bignums are represented within the system (Section 2.3).

## 2.1 Bignum Arithmetic

Bignum arithmetic, also called arbitrary precision arithmetic, is arithmetic of large numbers that exceeds the size of machine words. This means that the digits of precision is limited only by the available memory of the computer, in contrast to fixed-precision arithmetic where the precision is limited by the arithmetic logical unit's (AUL) hardware, usually between 8 to 64 bits. Bignum arithmetic is used in many important computer systems such as cryptographic software and computer algebra systems. Many cryptographic algorithms such as RSA and Diffie-Hellman require efficient operations over large integers [12].

Bignum integer multiplication is the fundamental arithmetic operation of computing the product of two integers, where the numbers are represented using multiple computer words [12]. The most basic bignum multiplication algorithm is the standard *long multiplication*, typically taught in primary school. To multiply two numbers, $x$ and $y$, start by calculating the product of $x$ times the least significant digit of $y$. It then continues with each of the higher order digits of $y$, where all partial products are appropriate shifted, and finally summed. In Figure 2.1, an example of such a calculations is presented.

The time-complexity of this classical schoolbook algorithm grows with $O(n^2)$, where $n$ is the length of the operands. For multiplication of large numbers there is a collection of algorithms that perform better than this basic algorithm. This thesis focus on two *divide and conquer* algorithms, namely the Karatsuba algorithm and its generalization, the Toom-3 algorithm. Divide and conquer refers to a class of

$$\begin{array}{r} 3\,8\,4 \\ \times\quad 5\,6 \\ \hline 2\,3\,0\,4 \\ 1\,9\,2\,0\;\; \\ \hline 2\,1\,5\,0\,4 \end{array}$$

**Figure 2.1:** An example of standard schoolbook long-multiplication.

algorithms which solves a problem by dividing it into several subproblems, which are solved recursively and the solutions to these subproblems are combined into a solution of the original problem [6]. The Karatsuba has a complexity of $O(n^{1.585})$ and Toom-3's complexity is $O(n^{1.465})$ [7].

Good bignum libraries switch to these asymptotically faster algorithms when the inputs passes a certain size, for example the GMP library uses seven different multiplication algorithms at different input sizes [14]. The selection of an optimal cut-off length, that separates the use of the standard schoolbook algorithm to the use of an asymptotically faster algorithm, is a difficult problem, since it depends on many variables such as the specific architecture or hardware of the host system, and has not been addressed sufficiently in the literature [4].

Another factor that can influence the performance of a bignum library is the concrete representation of bignums. In Section 2.3.1 we describe how bignums is represented in HOL, the main tool used in this thesis, and how we abstract the specifics of the system architecture in the specification and verification of the algorithms. Also we describe how we let the cut-off length be an argument to the top-level functional implementations of the algorithms, which lets us prove their termination and correctness without specifying a specific cut-off length.

## 2.2 CakeML

CakeML is a functional programming language, based on a substantial subset of Standard ML, with a proven correct compiler and runtime. That means that the compiler has been proven to transform CakeML programs into semantically equivalent machine code. The semantics and compiler algorithm of CakeML are specified in the higher-order logic of the HOL4 theorem prover [13].

In this thesis we are interested in the bignum library of CakeML's verified bignum library. It is also specified and proven using HOL4 (which is presented in the following section). The bignum algorithms are verified using the method described in Section 1.3, with first a high-level implementation that is proven to correctly implement integer multiplication. Next an imperative version is implemented, which is proven to be equivalent to the previous specified function. Finally, the implemented functions are integrated to CakeML by a proof-producing translation from functions of a specific format into wordLang programs (wordLang is an intermediate language

in the CakeML's compiler phases). The generated wordLang programs are attached to the compiled CakeML programs as part of the compiler phase which implements bignum arithmetic.

## 2.3 HOL4: Interactive Theorem Prover

The implementation and corresponding correctness proofs of the CakeML compiler and its bignum library is entirely conducted within the HOL4 interactive theorem prover. Naturally, the work presented in this thesis is also carried out using the same system. Below is a brief overview of HOL4 and a description of how bignums are represented in HOL.

The HOL4 interactive theorem prover is a ML-based proof assistant for higher-order logic [11]. In HOL4, the user interactively proves theorems by steering the system with *proof-tactics*. Proof-tactics are functions which divide the current proof goal into one or more subgoals, along with a *validation* function. This function justifies the decomposition of the goal, by being able to produce a proof of the original goal, given proofs of the constructed subgoals. This allows the user to decompose a proof into smaller, more manageable pieces. All proofs must pass the logical core of HOL4, which is a ML-module implementing the basic inference rules of higher-order logic. In this way, HOL4 prevents false statements from being proved.

### 2.3.1 Abstraction of bignums in HOL

Bignum arithmetic algorithms operate over lists of machine words, with the least significant word first. A machine word is conveniently modelled in HOL as a finite Cartesian product of booleans, which we write as $\texttt{bool}^\alpha$, where $\alpha$ is the width of a machine word. This is convenient since, by using a variable for the width, we do not tie the specification of our algorithm to any specific architecture. From the included `wordsTheory` in HOL4, we have mappings from natural numbers to machine words (`n2w`) and back (`w2n`).

$$\texttt{n2w} : \mathbb{N} \rightarrow \texttt{bool}^\alpha$$
$$\texttt{w2n} : \texttt{bool}^\alpha \rightarrow \mathbb{N}$$

These mappings are important building blocks in formulating theorems about the correctness of arithmetic operations. The following theorems describe the mappings relationship to each other.

$$\vdash \texttt{n2w}\ (\texttt{w2n}\ w) = w$$
$$\vdash \texttt{w2n}\ (\texttt{n2w}\ n) = n\ \texttt{mod}\ 2^\alpha$$

As mentioned above, the bignum arithmetic algorithms operate on lists of these machine words, i.e. lists of type $\texttt{bool}^\alpha$ `list`. Functions for converting a natural

number into a list of multiple words (**n2mw**) and vice versa (**mw2n**) are defined by:

$$
\begin{array}{l}
\texttt{n2mw}\ n\ \texttt{=} \\
\mathsf{if}\ n = 0\ \mathsf{then}\ [] \\
\mathsf{else}\ \texttt{n2w}\ (n\ \texttt{mod}\ 2^\alpha)\texttt{::n2mw}\ (n\ \texttt{div}\ 2^\alpha)
\end{array}
$$

$$
\begin{array}{l}
\texttt{mw2n}\ [] = 0 \\
\texttt{mw2n}\ (x\texttt{::}xs) = \texttt{w2n}\ x + 2^\alpha \times \texttt{mw2n}\ xs,
\end{array}
$$

where :: is list cons. The above definitions only cover natural numbers. To represent integers we form a pair of a sign and a list of machine words. The functions for translating integers to this representation (**i2mw**) and back (**mw2i**) is given below.

$$
\begin{array}{l}
\texttt{i2mw}\ i = (i < 0\texttt{,n2mw}\ (\texttt{abs}\ i)) \\
\texttt{mw2i}\ (\texttt{F},xs) = \texttt{mw2n}\ xs \\
\texttt{mw2i}\ (\texttt{T},xs) = \texttt{-mw2n}\ xs
\end{array}
$$

The multiplication algorithms implemented in this thesis will operate on the above specified representation of integers. With the type variable $\alpha$ we abstract the specifics of the architecture, and as mentioned in the end of Section 2.1, we also want to defer the decision of a cut-off length, i.e. at which input size to use the divide-and-conquer algorithms instead of the schoolbook algorithm. We do this by adding the cut-off length as an argument to the top-level functions of the implemented algorithms. Thus, in Chapter 3 and 4 we present the functional implementations of the algorithms, which will operate over lists for machine words (as defined above) and have an additional argument for the cut-off.

### 2.3.2 Useful helper-functions

The specification and verification of the algorithms presented in the following chapters build on the previous work of Myreen and Curello [8]. Here we introduce the most important functions and theorems used, also we describe a few functions defined specifically for this thesis.

The first function we describe (**mw_addv**) computes the sum of two bignums and a carry bit, represented as lists of machine words and a boolean. The function operates in a restricted setting described by the following theorem.

$$
\begin{array}{l}
\vdash \texttt{length}\ ys \leq \texttt{length}\ xs \Rightarrow \\
\quad (\texttt{mw2n}\ (\texttt{mw\_addv}\ xs\ ys\ c) = \texttt{mw2n}\ xs + \texttt{mw2n}\ ys + \texttt{b2n}\ c)
\end{array}
$$

From the theorem we see that it operates with two lists $xs$, $ys$, and a boolean $c$, where **length** $ys$ is required to be smaller or equal than **length** $xs$, which simplifies its

implementation. This function works well in the setting of long-multiplication, but for the algorithms considered in this thesis we cannot give these kinds of guarantees. Therefore we define a helper function **mwn_add** which operates in a less restricted setting.

**mwn_add** $xs$ $ys$ =
if **length** $ys \leq$ **length** $xs$ then **mw_addv** $xs$ $ys$ **F** else **mw_addv** $ys$ $xs$ **F**

$\vdash$ **mw2n** (**mwn_add** $xs$ $ys$) = **mw2n** $xs$ + **mw2n** $ys$

For multiplication we have **mw_mul** already defined which also has restrictions on its arguments. Therefore we also define a helper-function (**mwn_mul**) for this operation. Here $0w$ is the machine word representing zero.

$\vdash$ (**length** $ys$ = **length** $zs$) $\Rightarrow$
(**mw2n** (**mw_mul** $xs$ $ys$ $zs$) = **mw2n** $xs$ $\times$ **mw2n** $ys$ + **mw2n** $zs$)

**mwn_mul** $xs$ $ys$ = **mw_mul** $xs$ $ys$ (**map** ($\lambda\, x.\ 0w$) $ys$)
$\vdash$ **mw2n** (**mwn_mul** $xs$ $ys$) = **mw2n** $xs$ $\times$ **mw2n** $ys$

Finally we mention **mw_fix** which removes leading zeros form a list of machine words. The function **front** drops the last element of a list.

**mw_fix** $xs$ =
if $xs$ = [] then []
else if **last** $xs$ = $0w$ then **mw_fix** (**front** $xs$)
else $xs$

In the following chapters (Chapter 3 and 4) we present the Karatsuba and Toom-3 algorithms, two bignum multiplication algorithms that are asymptotically faster than the schoolbook algorithm presented in in the beginning of this chapter (faster than $O(n^2)$). We describe their functional implementation in HOL and the corresponding proofs of termination and correctness. This constitute the first step in the verification method described in Section 1.3.

# 3

# The Karatsuba algorithm

The Karatsuba algorithm, discovered by Anatoly Karatsuba in 1960, was the first multiplication algorithm that was asymptotically faster than the standard quadratic algorithm. It reduces a multiplication of two large numbers to three multiplications of smaller numbers with some extra additions and shifts [5, 7]. This section presents the algorithm's definition and the following subsections describe the algorithm's specification (Section 3.2) and verification (Section 3.3) in HOL.

Lets consider two numbers $x$ and $y$ represented as $n$-digit strings in some base $B$, and let $m$ be a positive integer such that $m < n$. Then the two numbers can be written as

$$x = X_1 B^m + X_0, \quad y = Y_1 B^m + Y_0,$$

where $X_1 = (x_{n-1}, ..., x_m)_B$ is the "most significant part" of $x$ and $X_0 = (x_{m-1}, ..., x_0)_B$ is the "least signification part"; similarly $Y_1 = (y_{n-1}, ..., y_m)_B$ and $Y_0 = (y_{m-1}, ..., y_0)_B$. Then the product is

$$\begin{aligned} x \times y &= (X_1 B^m + X_0)(Y_1 B^m + Y_0) \\ &= z_2 B^{2m} + z_1 B^m + z_0, \end{aligned}$$

where

$$\begin{aligned} z_0 &= X_0 Y_0, \\ z_1 &= X_1 Y_0 + X_0 Y_1 \\ z_2 &= X_1 Y_1. \end{aligned}$$

The above formulation requires four multiplications, but Karatsuba observed that $xy$ can be calculated using only three multiplications with the extra cost of a few basic arithmetic operations. This is achieved by calculating $z_1$ using $z_0$ and $z_2$:

$$\begin{aligned} z_1 &= (X_1 + X_0)(Y_1 + Y_0) - z_2 - z_0 \\ &= X_1 Y_1 + X_1 Y_0 + X_0 Y_1 + X_0 Y_0 - z_2 - z_0 \\ &= X_1 Y_0 + X_0 Y_1 \end{aligned}$$

To summarize the Karatsuba algorithm, let $x = x_1 B^m + x_0$ and $y = y_1 B^m + y_0$ then

recursively compute:

$$\begin{aligned} z_0 &= X_0 Y_0 \\ z_1 &= (X_0 + X_1)(Y_0 + Y_1) \\ z_2 &= X_1 Y_1, \end{aligned}$$

and return $z_2 B^{2m} + (z_1 - z_0 - z_2)B^m + z_0$.

## 3.1 Complexity Analysis

A basic step of the Karatsuba algorithm with inputs of size $n$, performs a few basic arithmetic operations and then makes three recursive calls with input size of $n/2$. With the results from the recursive calls, it again performs a few operations to recombine the sub-solutions into the correct product. Let $T(n)$ be the time needed by the algorithm for a problem of size $n$, we get the following recurrence relation:

$$T(n) = 3T(\frac{n}{2}) + cn + d,$$

for some constants $c$ and $d$, which represent the time needed for the additions, subtractions and shifts in dividing and reassembling the subproblems. When $n$ increases, the cost of these basic arithmetic operations becomes negligible since they take time proportional to $n$. The *Master Theorem* [2] gives the asymptotic bound $T(n) = \Theta(n^{\log_2 3})$.

It follows, for sufficiently large $n$, that the Karatsuba algorithm is faster than the basic schoolbook algorithm. But for small $n$, the additional basic arithmetic operations that are performed during the basic step of the Karatsuba algorithm may make it run slower. Therefore, good bignum libraries use different multiplication algorithms depending on the size of the inputs. As described in Section 2.1, we defer the specification of this *cut-off* point by having it as an argument to the implementation.

## 3.2 Specification in HOL

When specifying the Karatsuba algorithm in HOL, we rely on the already defined bignum arithmetic operations present in the CakeML bignum library, and a few helper methods described in Section 2.2.

We start by defining **`mw_ktb_mul`** which implements the Karatsuba algorithm with the help of two other functions **`mw_ktb_mul_init`** and **`mw_ktb_mul_final`**.

**Definition 3.2.1**

```
mw_ktb_mul cutoff xs ys =
(let (xl,yl) = (length xs,length ys)
 in
  if cutoff < 12 ∨ xl < cutoff ∨ yl < cutoff then
   mw_fix (mwn_mul xs ys)
  else
   (let (m,x0,x1,y0,y1,x0x1,y0y1) = mw_ktb_mul_init xs ys
    in
    let z0 = mw_ktb_mul cutoff x0 y0 in
    let z1 = mw_ktb_mul cutoff x0x1 y0y1 in
    let z2 = mw_ktb_mul cutoff x1 y1
    in
     mw_ktb_mul_final m z0 z1 z2))
```

We structure the implementation in this way to make the verification process of the later imperative implementation in Section 5 simpler. One of the scenarios where we switch to the school book algorithm (**mwn_mul**) is when the cut-off is smaller than 12, which serves as a practical since it removes the need of handling edge cases and also simplifies the verification of memory requirements of the imperative implementation, see Section 5.2.1. The init-function does all the necessary work before the recursive calls and is defined as:

```
mw_ktb_mul_init xs ys =
(let m = max (length xs) (length ys) div 2 in
 let (x0,x1) = (take m xs,drop m xs) in
 let (y0,y1) = (take m ys,drop m ys)
 in
   (m,x0,x1,y0,y1,mwn_add x0 x1,mwn_add y0 y1))
```

Here we see the same steps as in the algorithm definition above, both $xs$ and $ys$ are split into two smaller parts based on $m$. We select $m$ as half of the maximum length of $xs$ and $ys$. By doing this, the most significant part of the split (i.e. $X_1$ or $Y_1$) can in some cases have a length of zero, e.g. when $xs$ is much larger than $ys$ such that **length** $ys \leq m$. Both **take** and **drop** handle cases with too large or small input gracefully.

$$\forall\, l\; n.\, \texttt{length}\; l \;\leq\; n \;\Longrightarrow\; \texttt{take}\; n\; l \;=\; l$$
$$\forall\, l\; n.\, \texttt{length}\; l \;\leq\; n \;\Longrightarrow\; \texttt{drop}\; n\; l \;=\; []$$

The other helper function **mw_ktb_mul_final** finalizes a basic step of the Karatsuba algorithm by a series of shifts, subtractions and additions. The shifts are implemented by prepending a multiword with a list of zero-words ($0w$). We use

**replicate** to construct such lists, which is a function that takes an integer $n$ and a value $v$ and returns a list of length $n$ with $v$ as the value of every element.

```
mw_ktb_mul_final m z0 z1 z2 =
(let p2 = replicate (2 × m) 0w ++ z2 in
 let p1 = replicate m 0w ++ mw_subv z1 (mwn_add z0 z2)
 in
  mw_fix (mwn_add z0 (mwn_add p1 p2)))
```

Here, **mw_subv** implements bignum subtraction in the same restricted setting as **mw_addv** defined in Section 2.3.2. To conclude the specification of the algorithm in HOL, we define the function **mwi_ktb_mul** which computes the resulting sign.

```
mwi_ktb_mul cutoff (s,xs) (t,ys) =
if (xs = []) ∨ (ys = []) then (F,[])
else (s ≠ t,mw_ktb_mul cutoff xs ys)
```

## 3.3 Algorithm verification

The top-level correctness theorem we want to prove can easily be stated using **i2mw** which converts an integer into a signed list of machine words. The theorem relates **mwi_ktb_mul** to multiplication ($\times$) over the integers.

**Theorem 3.3.1** ***mwi_ktb_mul*** *correctly implements integer multiplication.*

$$\vdash \textbf{mwi\_ktb\_mul}\ c\ (\textbf{i2mw}\ i)\ (\textbf{i2mw}\ j) = \textbf{i2mw}\ (i \times j)$$

Since **mwi_ktb_mul** is a thin wrapper around **mw_ktb_mul**, its correctness proof follows rather easily from the corresponding correctness proof of **mw_ktb_mul**.

**Theorem 3.3.2** ***mw_ktb_mul*** *correctly implements unsigned integer multiplication.*

$$\vdash \textbf{mw2n}\ (\textbf{mw\_ktb\_mul}\ cutoff\ xs\ ys) = \textbf{mw2n}\ xs \times \textbf{mw2n}\ ys$$

Here we only need to reason about natural numbers, which makes the proofs simpler and less repetitive. To complete the proof we have to first prove the termination of the recursive algorithm and after that prove the correctness.

### 3.3.1 Proof of termination

First, we prove that our implementation of the Karatsuba algorithm terminates. This is done by providing HOL4 with a metric that is shown to decrease for each

recursive call. In this case we use **length** $xs$ **++ length** $ys$ as metric and prove that the recursive calls for $z_0$, $z_1$, and $z_2$ are called with smaller arguments. We state a lemma for each of the three recursive call-points. The lemma for $z_0$ and $z_1$ are given below, the lemma for $z_2$ is similar to that for $z_0$. Note that we only need to care about **mw_ktb_mul_init**, since it handles the transformation of all the arguments for the recursive calls.

**Lemma 3.3.3**

$$\begin{aligned}\vdash\ & l > 2 \wedge \textbf{length}\ xs \geq l \wedge \textbf{length}\ ys \geq l \wedge \\ & ((m, x_0, x_1, y_0, y_1, x0x1, y0y1) = \textbf{mw\_ktb\_mul\_init}\ xs\ ys) \Rightarrow \\ & \textbf{length}\ x_0 + \textbf{length}\ y_0 < \textbf{length}\ xs + \textbf{length}\ ys\end{aligned}$$

**Lemma 3.3.4**

$$\begin{aligned}\vdash\ & l > 4 \wedge \textbf{length}\ xs \geq l \wedge \textbf{length}\ ys \geq l \wedge \\ & ((m, x_0, x_1, y_0, y_1, x0x1, y0y1) = \textbf{mw\_ktb\_mul\_init}\ xs\ ys) \Rightarrow \\ & \textbf{length}\ x0x1 + \textbf{length}\ y0y1 < \textbf{length}\ xs + \textbf{length}\ ys\end{aligned}$$

The lemmas for $z_0$ and $z_2$ are trivial to prove given the following helpful fact:

**Lemma 3.3.5**

$$\begin{aligned}\vdash\ & (\textbf{mw\_ktb\_mul\_init}\ xs\ ys = (m, x_0, x_1, y_0, y_1, x0x1, y0y1)) \Rightarrow \\ & (xs = x_0\ \textbf{++}\ x_1) \wedge (ys = y_0\ \textbf{++}\ y_1)\end{aligned}$$

The proof for $z_1$ is more involved since $x0x1$ and $y0y1$ are sums produced by **mwn_add**. To address this, let us first state a lemma that gives an upper bound on the length of the result from **mwn_add**.

**Lemma 3.3.6** *The result of* **mwn_add** *has a length less or equal to that of its largest argument plus 1.*

$$\vdash \textbf{length}\ (\textbf{mwn\_add}\ xs\ ys) \leq \textbf{max}\ (\textbf{length}\ xs)\ (\textbf{length}\ ys) + 1$$

With this lemma we can prove Lemma 3.3.4 by proving the simplified statement below. This is proven by using Lemma 3.3.5 and considering each case of **max** (**lenght** $x_0$) (**lenght** $x_1$) and for **max** (**lenght** $y_0$) (**lenght** $y_1$).

$$\begin{aligned}\vdash\ & \textbf{max}\ (\textbf{length}\ x_0)\ (\textbf{length}\ x_1) + \textbf{max}\ (\textbf{length}\ y_0)\ (\textbf{length}\ y_1) + 2 \leq \\ & \textbf{length}\ xs + \textbf{length}\ ys\end{aligned}$$

With the above lemmas stated and proved, the termination proof for **mw_ktb_mul** is finalized using the lemmas at each recursion to show that the metric, **length** $xs$ + **length** $ys$, decreases in each call. Next we continue with the correctness theorem which is proved by induction.

### 3.3.2 Proof of correctness

The correctness of the base case of **mw_ktb_mul** simply follows from the correctness theorem of **mwn_mul**. The remaining part of the proof concerns the case when the length of both $xs$ and $ys$ are larger than the cut-off. We decompose the proof by proving one correctness theorem for **mw_ktb_mul_init** and one for **mw_ktb_mul_final**.

**Theorem 3.3.7**

$$\begin{aligned}\vdash\ & (\textbf{mw\_ktb\_mul\_init}\ xs\ ys = (m, x_0, x_1, y_0, y_1, x0x1, y0y1)) \Rightarrow \\ & (\textbf{mw2n}\ xs = \textbf{mw2n}\ x_0 + 2^{\alpha\times m} \times \textbf{mw2n}\ x_1) \wedge \\ & (\textbf{mw2n}\ ys = \textbf{mw2n}\ y_0 + 2^{\alpha\times m} \times \textbf{mw2n}\ y_1)\end{aligned}$$

**Theorem 3.3.8**

$$\begin{aligned}\vdash\ & \textbf{mw2n}\ (\textbf{mwn\_add}\ xs\ zs) \leq \textbf{mw2n}\ ys \wedge \\ & \textbf{length}\ (\textbf{mwn\_add}\ xs\ zs) \leq \textbf{length}\ ys \Rightarrow \\ & (\textbf{mw2n}\ (\textbf{mw\_ktb\_mul\_final}\ d\ xs\ ys\ zs) = \\ & \textbf{mw2n}\ xs + \\ & 2^{\alpha\times d} \times (\textbf{mw2n}\ ys - \textbf{mw2n}\ xs - \textbf{mw2n}\ zs) + \\ & 2^{\alpha\times(2\times d)} \times \textbf{mw2n}\ zs)\end{aligned}$$

The first theorem describes how the value of $xs$ and $ys$ is related to $x_0$, $x_1$, $y_0$, and $y_1$. The second theorem states that the final-function does the correct recomposition of the recursive calculated products. Together with the induction hypothesis the correctness of the presented implementation is proved.

# 4

# The Toom-3 Algorithm

The Toom-3 algorithm is a specific instance of the Toom-Cook algorithm, named after Andrei Toom and Stephen Cook [3]. The algorithm works by splitting the operands into $k$ parts, and compute the product by performing operations on these smaller parts. Toom-3 is the specific instance where $k$ is 3. Here we start with presenting the Toom-3 algorithm in detail, then we continue with the functional specification of the algorithm in HOL under Section 4.1, and we end with the verification of the implementation.

The Toom-3 algorithm treats its operands as polynomials $U(x)$ and $V(x)$ of degree 2 by partitioning the numbers into 3 parts, which are used as coefficients.

$$U(x) = u_0 + u_1 x + u_2 x^2$$
$$V(x) = v_0 + v_1 x + v_2 x^2$$

Let $B$ be the base used to represent the bignums and $m$ the splitting size used, then we obtain the original operands with $U(B^m)$ and $V(B^m)$. The product $W(x) = U(x)V(x)$ is a polynomial of degree 4 with the coefficients:

$$w_0 = u_0 v_0$$
$$w_1 = u_0 v_1 + u_1 v_0$$
$$w_2 = u_0 v_2 + u_1 v_1 + u_2 v_0$$
$$w_3 = u_1 v_2 + u_2 v_1$$
$$w_4 = u_2 v_2$$

We note that calculating these coefficients would require 9 multiplications. The Toom-3 algorithm does not calculate the coefficients in this way, instead it utilizes the fact that a polynomial of degree $d$ can be fully characterized from an evaluation of $d+1$ points. In the specific case of Toom-3 we need to evaluate 5 points, $W(x_i) = U(x_i)V(x_i), i = 0, ..., 4$. The selection of evaluation points affects the performance of the algorithm, in this thesis only non-negative points were selected due to simpler

proofs.

$$\begin{aligned}
W(0) &= U(0)V(0) = u_0 v_0 \\
W(1) &= U(1)V(1) = (u_0 + u_1 + u_2)(v_0 + v_1 + v_2) \\
W(2) &= U(2)V(2) = (u_0 + 2u_1 + 4u_2)(v_0 + 2v_1 + 4v_2) \\
W(3) &= U(3)V(3) = (u_0 + 3u_1 + 9u_2)(v_0 + 3v_1 + 9v_2) \\
W(\infty) &= \lim_{x \to \infty} \frac{U(x)V(x)}{x^4} = u_2 v_2
\end{aligned}$$

These gives us five evaluations of the polynomial:

$$W(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4,$$

which can be used to form the following system of equations:

$$\begin{aligned}
W(0) &= w_0 \\
W(1) &= w_0 + w_1 + w_2 + w_3 + w_4 \\
W(2) &= w_0 + 2w_1 + 4w_2 + 8w_3 + 16w_4 \\
W(3) &= w_0 + 3w_1 + 9w_2 + 27w_3 + 81w_4 \\
W(\infty) &= w_4
\end{aligned}$$

The coefficients are obtained by solving the equation system.

$$\begin{aligned}
w_0 &= W(0) \\
w_1 &= \frac{-11}{6} W(0) + 3W(1) - \frac{3}{2} W(2) + \frac{1}{3} W(3) - 6W(\infty) \\
w_2 &= W(0) - \frac{5}{2} W(1) + 2W(2) - \frac{1}{2} W(3) + 11W(\infty) \\
w_3 &= \frac{-1}{6} W(0) + \frac{1}{2} W(1) - \frac{1}{2} W(2) + \frac{1}{6} W(3) - 6W(\infty) \\
w_4 &= W(\infty)
\end{aligned}$$

Finally we evaluate $W(B^m)$ to obtain the product $U(B^m)V(B^m)$, this concludes the multiplication algorithm. Instead of 9 multiplication, the Toom-3 only uses five, with some additional cheap operation such as multiplication, division and addition of small constants.

To summarise, the Toom-3 algorithm can be described in five steps:

1. Splitting
2. Evaluation
3. Point-wise multiplication
4. Interpolation

5. Recomposition

The algorithm starts by splitting each operand into three partitions, that are used as coefficients to form the polynomials $U(x)$ and $V(x)$. The next step is to evaluate these polynomials at five points, in the thesis's implementation we evaluate at $0, 1, 2, 3, \infty$. With these evaluations we can compute the *point-wise multiplications* $W(x_i) = U(x_i)V(x_i)$ recursively. Now we have five evaluations of the $W(x)$ polynomial of degree 4, which allows us to form a system of equations with the coefficients $w_0, w_1, ..., w_4$ as unknowns. We obtain these coefficients by solving the equation system. With the polynomial $W(x)$ fully characterized, we can evaluate $W(B^m)$ which returns the wanted product $U(B^m)V(B^m)$.

## 4.1 Specification in HOL

We start by defining the function **mw_toom3_mul** which implements the algorithm for non-negative numbers. It performs the splitting, evaluation, point-wise multiplication, interpolation, and recomposition which are implemented in the functions: **mw_toom3_split**, **mw_toom3_eval**, **mw_toom3_mul** (recursive calls), **mw_toom3_coef**, and **mw_toom3_recomp**. The base case is implemented in the same way as for the Karatsuba algorithm.

**Definition 4.1.1**

```
mw_toom3_mul cutoff xs ys =
if α < 4 then []
else
 (let (xl, yl) = (length xs, length ys)
  in
    if cutoff ≤ 10 ∨ xl < cutoff ∨ yl < cutoff then
     mw_fix (mwn_mul xs ys)
    else
     (let m = max xl yl div 3 + 1 in
      let (x0, x1, x2) = mw_toom3_split m xs in
      let (y0, y1, y2) = mw_toom3_split m ys in
      let (p0, p1, p2, p3, pinf) = mw_toom3_eval x0 x1 x2 in
      let (q0, q1, q2, q3, qinf) = mw_toom3_eval y0 y1 y2 in
      let r0 = mw_toom3_mul cutoff p0 q0 in
      let r1 = mw_toom3_mul cutoff p1 q1 in
      let r2 = mw_toom3_mul cutoff p2 q2 in
      let r3 = mw_toom3_mul cutoff p3 q3 in
      let rinf = mw_toom3_mul cutoff pinf qinf in
      let (w0, w1, w2, w3, w4) =
          mw_toom3_coef (r0, r1, r2, r3, rinf)
      in
       mw_fix (mw_toom3_recomp (m, w0, w1, w2, w3, w4))))
```

The first step of the implementation is to select $m$ based on the length of the operands. In our implementation we select $m$ to be the maximum length divided by three plus one. This implies that **max** $xl\ yl < 3m$ which is an useful lemma in the verification.

The function **mw_toom3_split** performs the splitting step for both $xs$ and $ys$. The implementation uses **take** and **drop** to partition the operand, where $x_0$ is the least significant part, followed by $x_1$ and $x2$ (the same applies for the splitting of $ys$).

**Definition 4.1.2**

$$\begin{aligned}&\textbf{\textit{mw\_toom3\_split}}\ m\ xs\ \texttt{=}\\&(\mathsf{let}\ t\ =\ \textbf{take}\ m\ \mathsf{and}\ xs'\ =\ \textbf{\textit{drop}}\ m\ xs\ \mathsf{in}\ (t\ xs\texttt{,}t\ xs'\texttt{,}\textbf{\textit{drop}}\ m\ xs'))\end{aligned}$$

The next step is the evaluation of five points each for the polynomials constructed with the coefficients from previous step. The function **mw_toom3_eval** performs the evaluation given the coefficients of a polynomial. To avoid negative integers our implementation evaluates five non-negative points, namely $0, 1, 2, 3$ and $\infty$. The helper function **mw_toom3_eval_x** evaluates a polynomial $P(x) = p_0 + p_1x + p_2x^2$ given values of $p_0, p_1, p_2, x$ and $x^2$.

**Definition 4.1.3**

$$\begin{aligned}&\textbf{\textit{mw\_toom3\_eval}}\ v_0\ v_1\ v_2\ \texttt{=}\\&(\mathsf{let}\ t_1\ =\ \textbf{\textit{mwn\_add}}\ (\textbf{\textit{mwn\_add}}\ v_0\ v_2)\ v_1\ \mathsf{in}\\&\ \mathsf{let}\ t_2\ =\ \textbf{\textit{mw\_toom3\_eval\_x}}\ (v_0\texttt{,}v_1\texttt{,}v_2)\ (\textbf{\textit{n2mw}}\ 2)\ (\textbf{\textit{n2mw}}\ 4)\ \mathsf{in}\\&\ \mathsf{let}\ t_3\ =\ \textbf{\textit{mw\_toom3\_eval\_x}}\ (v_0\texttt{,}v_1\texttt{,}v_2)\ (\textbf{\textit{n2mw}}\ 3)\ (\textbf{\textit{n2mw}}\ 9)\\&\ \mathsf{in}\\&\ \ (v_0\texttt{,}t_1\texttt{,}t_2\texttt{,}t_3\texttt{,}v_2))\end{aligned}$$

**Definition 4.1.4**

$$\begin{aligned}&\textbf{\textit{mw\_toom3\_eval\_x}}\ (p_0\texttt{,}p_1\texttt{,}p_2)\ x_1\ x_2\ \texttt{=}\\&(\mathsf{let}\ r_1\ =\ \textbf{\textit{mwn\_mul}}\ x_1\ p_1\ \mathsf{in}\\&\ \mathsf{let}\ r_2\ =\ \textbf{\textit{mwn\_mul}}\ x_2\ p_2\\&\ \mathsf{in}\\&\ \ \textbf{\textit{mwn\_add}}\ p_0\ (\textbf{\textit{mwn\_add}}\ r_1\ r_2))\end{aligned}$$

We have now completed step 1 and 2 of the Toom-3 algorithm. The next step is point-wise multiplication which computes $W(x_i) = U(x_i)V(x_i)$. This is done by recursion, calling **mw_toom3_mul** for each of the five points as seen in Definition 4.1.1.

The interpolation, i.e. calculation of the coefficients of $W(x)$, is performed by **mw_toom3_coef**, according to the same procedure described in the start of this chapter. The final recomposition is conducted in **mw_toom3_recomp** which evaluates $W(x)$ at $B^m$. This is implemented as a series of shifts and additions, where the shifts are implemented using **replicate** as explained in Section 3.2.

**Definition 4.1.5**

$$
\begin{array}{l}
\texttt{mw\_toom3\_recomp}\ (m, w_0, w_1, w_2, w_3, w_4) = \\
(\mathsf{let}\ xy\ =\ \texttt{replicate}\ m\ 0w\ \texttt{++}\ w_4\ \mathsf{in} \\
\ \mathsf{let}\ xy\ =\ \texttt{replicate}\ m\ 0w\ \texttt{++}\ \texttt{mwn\_add}\ w_3\ xy\ \mathsf{in} \\
\ \mathsf{let}\ xy\ =\ \texttt{replicate}\ m\ 0w\ \texttt{++}\ \texttt{mwn\_add}\ w_2\ xy\ \mathsf{in} \\
\ \mathsf{let}\ xy\ =\ \texttt{replicate}\ m\ 0w\ \texttt{++}\ \texttt{mwn\_add}\ w_1\ xy \\
\ \mathsf{in} \\
\ \ \texttt{mwn\_add}\ w_0\ xy)
\end{array}
$$

We conclude the specification of the Toom-3 algorithm in HOL, as we did for the Karatsuba, with defining a top-level function that manages the sign of integer multiplication.

$$
\begin{array}{l}
\texttt{mwi\_toom3\_mul}\ \mathit{cutoff}\ (s, xs)\ (t, ys) = \\
\mathsf{if}\ (xs = []) \vee (ys = []) \ \mathsf{then}\ (\texttt{F}, [\,]) \\
\mathsf{else}\ (s \neq t, \texttt{mw\_toom3\_mul}\ \mathit{cutoff}\ xs\ ys)
\end{array}
$$

## 4.2 Algorithm verification

The top-level theorem we want to prove is the same kind of correctness theorem stated for the Karatsuba algorithm in Section 3.3. It relates **mwi_toom3_mul** to multiplication over the integers.

**Theorem 4.2.1** ***mwi_toom3_mul*** *correctly implements integer multiplication.*

$$
\begin{array}{l}
4 \leq \alpha \Rightarrow \\
(\texttt{mwi\_toom3\_mul}\ c\ (\texttt{i2mw}\ i)\ (\texttt{i2mw}\ j) = \texttt{i2mw}\ (i \times j))
\end{array}
$$

The theorem has an assumption which ensures that there is enough space in one machine word for the constants used in the evaluation. Since **mwi_toom3_mul** only handles the sign of the resulting product, the correctness of it follows easily from the correctness proof of **mw_toom3_mul**.

**Theorem 4.2.2** ***mw_toom3_mul*** *correctly implements unsigned integer multiplication.*

$$
\begin{array}{l}
4 \leq \alpha \Rightarrow \\
(\texttt{mw2n}\ (\texttt{mw\_toom3\_mul}\ \mathit{cutoff}\ xs\ ys) = \texttt{mw2n}\ xs \times \texttt{mw2n}\ ys)
\end{array}
$$

### 4.2.1 Proof of termination

We start by proving that the recursive function **mw_toom3_mul** terminates. This is done by providing HOL with a metric, and show that this metric decreases at each

recursive call. The metric used for our Toom-3 implementation is the length of the operands, i.e. **length** $xs$+**length** $ys$. In Definition 4.1.1 we have five recursive calls. Before these recursive calls are reached we encounter the functions **mw_toom3_split** and **mw_toom3_eval**. We start with proving a few lemmas about the length of these function's results. Then we use those to show that the provided metric is smaller in each recursive call.

The **mw_toom3_split** function partitions a given multiword into three smaller parts by using **take** and **drop** in turn, as shown in Definition 4.1.2. Lemma 4.2.3 provides an upper bound on the length of the resulting parts. Note that the lemma includes an assumption on the splitting variable $m$, which our implementation in Definition 4.1.1 fulfills, as noted previously.

**Lemma 4.2.3**

$$\begin{aligned}&\textbf{length}\ ls < 3 \times m \wedge ((l_1, l_2, l_3) = \textbf{mw\_toom3\_split}\ m\ ls) \Rightarrow \\ &\textbf{length}\ l_1 \leq m \wedge \textbf{length}\ l_2 \leq m \wedge \textbf{length}\ l_3 < m\end{aligned}$$

With the above lemma we have an upper bound on the length of the input to **mw_toom3_eval**. Now we continue to state and prove upper bounds on the result of this function:

**Lemma 4.2.4**

$$\begin{aligned}&((t_0, t_1, t_2, t_3, tinf) = \textbf{mw\_toom3\_eval}\ v_0\ v_1\ v_2) \Rightarrow \\ &(\textbf{length}\ t_0 = \textbf{length}\ v_0)\end{aligned}$$

**Lemma 4.2.5**

$$\begin{aligned}&\textbf{length}\ v_0 \leq l \wedge \textbf{length}\ v_1 \leq l \wedge \textbf{length}\ v_2 < l \wedge \\ &((t_0, t_1, t_2, t_3, tinf) = \textbf{mw\_toom3\_eval}\ v_0\ v_1\ v_2) \Rightarrow \\ &\textbf{length}\ t_1 \leq l + 2\end{aligned}$$

**Lemma 4.2.6**

$$\begin{aligned}&\textbf{length}\ v_0 \leq l \wedge \textbf{length}\ v_1 \leq l \wedge \textbf{length}\ v_2 < l \wedge \textbf{length}\ (\textbf{n2mw}\ 4) \leq d \wedge \\ &((t_0, t_1, t_2, t_3, tinf) = \textbf{mw\_toom3\_eval}\ v_0\ v_1\ v_2) \Rightarrow \\ &\textbf{length}\ t_2 \leq l + d + 2\end{aligned}$$

Lemma 4.2.4 states the length of the evaluation at value 0, which is easy to prove since the result is the first argument of the function. We have the same lemma for $W(\infty)$, which is proved in similar fashion.

From Definition 4.1.3 we note that two additions is needed for $t1$. From Lemma 3.3.6 we have than the length of an addition is upper bounded by the maximum length of the operands plus 1. Two additions thus give the bound stated in Lemma 4.2.5.

For the last two evaluation points 2 and 3, we first state a lemma for the helper function **mw_toom3_eval_x**.

**Lemma 4.2.7**

$$\texttt{length}\ p_0 \leq l \wedge \texttt{length}\ p_1 \leq l \wedge \texttt{length}\ p_2 \leq l \wedge \texttt{length}\ x_1 \leq d \wedge \texttt{length}\ x_2 \leq d \Rightarrow \\ \texttt{length}\ (\texttt{mw\_toom3\_eval\_x}\ (p_0\texttt{,}p_1\texttt{,}p_2)\ x_1\ x_2) \leq l\ \texttt{+}\ d\ \texttt{+}\ 2$$

The proof of Lemma 4.2.6 (and similar for evaluation point 3) follows easily for the above stated lemma.

With the length of $p_0, ..., p_{inf}, q_0, ..., q_{inf}$ upper bounded by the lemmas on **mw_toom3_eval**, we continue to prove that each recursive call in **mw_toom3_mul** is called with smaller arguments. For this we first need to show that our selection of splitting length $m$, always is large enough to partition the operands. This is stated in the following lemma:

**Lemma 4.2.8**

$$xl < 3 \times (\texttt{max}\ xl\ yl\ \texttt{div}\ 3\ \texttt{+}\ 1) \\ yl < 3 \times (\texttt{max}\ xl\ yl\ \texttt{div}\ 3\ \texttt{+}\ 1)$$

Now we have all the necessary parts to conclude the termination proof of our Toom-3 implementation. The above Lemma 4.2.8 is used to fulfill the assumption in Lemma 4.2.3 which bounds the length of the results from **mw_toom3_split**. From this we continue with applying the length lemmas for each value of **mw_toom3_eval**, and with this we can prove the termination of our implementation, by showing that the length of the arguments for each recursive call is decreasing.

With the termination proof done we continue to prove the correctness theorem stated in Theorem 4.2.2.

### 4.2.2 Proof of correctness

We start with proving a theorem for **mw_toom3_split**, which states the relationship between the inputed number and its splitted parts. Here we see that it correctly corresponds to a polynomial constructed with the split parts as coefficients, evaluated at $B^m$ (where $B = 2^\alpha$ below).

**Theorem 4.2.9**

$$(\texttt{mw\_toom3\_split}\ m\ ls\ \texttt{=}\ (l_0\texttt{,}l_1\texttt{,}l_2)) \Rightarrow \\ (\texttt{mw2n}\ ls\ \texttt{=} \\ \texttt{mw2n}\ l_0\ \texttt{+}\ 2^{\alpha \times m} \times \texttt{mw2n}\ l_1\ \texttt{+} \\ 2^{\alpha \times (m\ \texttt{+}\ m)} \times \texttt{mw2n}\ l_2)$$

The next function is **mw_toom3_eval** which is used to evaluate five points of the two constructed polynomials. The points to evaluate is the ones stated above and the correctness theorem is as follows:

**Theorem 4.2.10**

$$
\begin{aligned}
&(\texttt{mw\_toom3\_eval}\ v_0\ v_1\ v_2 = (t_0, t_1, t_2, t_3, tinf)) \Rightarrow \\
&(\texttt{mw2n}\ t_0 = \texttt{mw2n}\ v_0) \wedge \\
&(\texttt{mw2n}\ t_1 = \texttt{mw2n}\ v_0 + \texttt{mw2n}\ v_1 + \texttt{mw2n}\ v_2) \wedge \\
&(\texttt{mw2n}\ t_2 = \texttt{mw2n}\ v_0 + 2 \times \texttt{mw2n}\ v_1 + 4 \times \texttt{mw2n}\ v_2) \wedge \\
&(\texttt{mw2n}\ t_3 = \texttt{mw2n}\ v_0 + 3 \times \texttt{mw2n}\ v_1 + 9 \times \texttt{mw2n}\ v_2) \wedge \\
&(\texttt{mw2n}\ tinf = \texttt{mw2n}\ v_2)
\end{aligned}
$$

From the definition of `mw_toom3_eval` the proofs of $t0$ and $tinf$ are trivial. For $t1$ we use the previous stated theorem for `mwn_add`. Now only the more complicated terms $t2$ and $t3$ are left to prove. Since both use `mw_toom3_eval_x` to calculate the result, we prove a lemma for that helper function first.

**Lemma 4.2.11**

$$
\begin{aligned}
&\texttt{mw2n}\ (\texttt{mw\_toom3\_eval\_x}\ (a, b, c)\ x_1\ x_2) = \\
&\texttt{mw2n}\ a + \texttt{mw2n}\ x_1 \times \texttt{mw2n}\ b + \texttt{mw2n}\ x_2 \times \texttt{mw2n}\ c
\end{aligned}
$$

This is exactly what we want to prove for $t_2$ and $t_3$ in Theorem 4.2.10, hence when the above lemma is proven we are also done with the theorem for `mw_toom3_eval`. The correctness of Lemma 4.2.11 follows easily from the definition of the function (Definition 4.1.4) and the correctness theorems of `mwn_add` and `mwn_mul`.

Currently we have stated and proven theorems for the splitting and evaluation parts of our Toom-3 implementation. The three remaining parts are: point-wise multiplication, interpolation, and recomposition. The point-wise multiplication is the recursive calls to `mw_toom3_mul` and will be proven with induction on the top-level correctness theorem (Theorem 4.2.2), which we defer to later. Now we continue with theorems for `mw_toom3_coef` and `mw_toom3_recomp`.

With the theorem for `mw_toom3_coef` we want to prove that our implementation returns the correct coefficients for $W(x)$ given five evaluations of the two constructed polynomials $V(x)$ and $U(x)$. To formulate this we need assumptions on the arguments of the function. With our previously proven theorems for `mw_toom3_split` and `mw_toom3_eval`, we can state the needed assumptions as follows:

**Theorem 4.2.12**

$$
\begin{aligned}
&(\texttt{mw\_toom3\_split}\ m\ xs = (x_0, x_1, x_2)) \wedge \\
&(\texttt{mw\_toom3\_split}\ m\ ys = (y_0, y_1, y_2)) \wedge \\
&(\texttt{mw\_toom3\_eval}\ x_0\ x_1\ x_2 = (p_0, p_1, p_2, p_3, pinf)) \wedge \\
&(\texttt{mw\_toom3\_eval}\ y_0\ y_1\ y_2 = (q_0, q_1, q_2, q_3, qinf)) \wedge \\
&(\texttt{mw2n}\ (\texttt{mw\_toom3\_mul}\ cutoff\ p_0\ q_0) = \texttt{mw2n}\ p_0 \times \texttt{mw2n}\ q_0) \wedge \\
&(\texttt{mw2n}\ (\texttt{mw\_toom3\_mul}\ cutoff\ p_1\ q_1) = \texttt{mw2n}\ p_1 \times \texttt{mw2n}\ q_1) \wedge \\
&(\texttt{mw2n}\ (\texttt{mw\_toom3\_mul}\ cutoff\ p_2\ q_2) = \texttt{mw2n}\ p_2 \times \texttt{mw2n}\ q_2) \wedge \\
&(\texttt{mw2n}\ (\texttt{mw\_toom3\_mul}\ cutoff\ p_3\ q_3) = \texttt{mw2n}\ p_3 \times \texttt{mw2n}\ q_3) \wedge \\
&(\texttt{mw2n}\ (\texttt{mw\_toom3\_mul}\ cutoff\ pinf\ qinf) = \\
&\ \texttt{mw2n}\ pinf \times \texttt{mw2n}\ qinf) \wedge \\
&(\texttt{mw\_toom3\_coef} \\
&\quad (\texttt{mw\_toom3\_mul}\ cutoff\ p_0\ q_0, \texttt{mw\_toom3\_mul}\ cutoff\ p_1\ q_1, \\
&\quad\ \texttt{mw\_toom3\_mul}\ cutoff\ p_2\ q_2, \texttt{mw\_toom3\_mul}\ cutoff\ p_3\ q_3, \\
&\quad\ \texttt{mw\_toom3\_mul}\ cutoff\ pinf\ qinf) = \\
&\ (w_0, w_1, w_2, w_3, w_4)) \Rightarrow \\
&(\texttt{mw2n}\ w_0 = \texttt{mw2n}\ x_0 \times \texttt{mw2n}\ y_0) \wedge \\
&(\texttt{mw2n}\ w_1 = \texttt{mw2n}\ x_0 \times \texttt{mw2n}\ y_1 + \texttt{mw2n}\ x_1 \times \texttt{mw2n}\ y_0) \wedge \\
&(\texttt{mw2n}\ w_2 = \\
&\ \texttt{mw2n}\ x_0 \times \texttt{mw2n}\ y_2 + \texttt{mw2n}\ x_1 \times \texttt{mw2n}\ y_1 + \\
&\ \texttt{mw2n}\ x_2 \times \texttt{mw2n}\ y_0) \wedge \\
&(\texttt{mw2n}\ w_3 = \texttt{mw2n}\ x_1 \times \texttt{mw2n}\ y_2 + \texttt{mw2n}\ x_2 \times \texttt{mw2n}\ y_1) \wedge \\
&(\texttt{mw2n}\ w_4 = \texttt{mw2n}\ x_2 \times \texttt{mw2n}\ y_2)
\end{aligned}
$$

The implementation of **mw_toom3_coef** is basically is a direct translation of the equations presented in the interpolation step in the description of the Toom-3 algorithm. The proof mostly rely on algebraic manipulation, with one crucial detail. Since the equations require at least one subtraction, we need to make sure that the difference is positive (since it is a requirement of the basic subtraction function already defined). Since we know that the resulting coefficients are non-negative, we can rearrange the equations to have only one subtraction as the last step, which solves the problem.

The final step of the algorithm is the recomposition, which basically is an evaluation of the $W(x)$ at $B^m$. The theorem we want to prove relates the result to the correct evaluation.

**Theorem 4.2.13**

$$
\begin{aligned}
&\texttt{mw2n}\ (\texttt{mw\_toom3\_recomp}\ (m, w_0, w_1, w_2, w_3, w_4)) = \\
&\texttt{mw2n}\ w_0 + \texttt{mw2n}\ w_1 \times 2^{\alpha \times m} + \\
&\texttt{mw2n}\ w_2 \times 2^{\alpha \times (2 \times m)} + \\
&\texttt{mw2n}\ w_3 \times 2^{\alpha \times (3 \times m)} + \\
&\texttt{mw2n}\ w_4 \times 2^{\alpha \times (4 \times m)}
\end{aligned}
$$

From Definition 4.1.5 we see that **mw_toom3_recomp** consists of a series of shifts and additions. The shifts are performed by a prepending a list of zero-words to

a multiword. The list of zeros is constructed by **replicate** which were defined in Section 3.2. The proof for the above stated theorem is similar to the proof of Theorem 3.3.8.

Now we have theorems for all the helper functions of **mw_toom3_mul** and thus we can return to the main correctness theorem of our Toom-3 implementation, Theorem 4.2.2.

The base case of **mw_toom3_mul** is trivial to prove with the correctness theorem already proven for **mwn_mul**. For the non base case, we use the induction hypothesis with the theorems for **mw_toom3_split** and **mw_toom3_eval** to fulfill the assumptions of Theorem 4.2.12 for **mw_toom3_coef**. With this we have proven that we have calculated the correct coefficients of $W(x)$. With Theorem 4.2.13 we know that **mw_toom3_recomp** correctly returns the product of the original operands. Hence, we have proven the correctness of our implementation of the Toom-3 algorithm.

# 5

# Towards a CakeML integration

In this chapter we take a step closer to integrating the Karatsuba algorithm into CakeML. This is done by another implementation of the algorithm, but this time in a more imperative style, i.e. we are moving more towards a more concrete low-level implementation.

The infrastructure for the CakeML-integration requires that the algorithm implementation follow a specific format, which puts constraints on for example memory usage. In Section 5.1 we describe the format required for the integration to be successful. Our implementation takes a step towards a CakeML integration by complying to parts of the format. In other terms, we specify an intermediate implementation according to the method described in Section 1.3.

Section 5.2 gives an overview of how our implementation manipulates the allocated memory, i.e. in what order and where the intermediate results are saved in memory. The following section (Section 5.3) presents the specification in HOL. Section 5.4 continues with how the implementation's termination and correctness is proven, and we conclude with a discussion of the remaining steps required for a successful CakeML-integration.

## 5.1 Constraints for the CakeML-integration

The integration into the CakeML's bignum library works by specifying an implementation of the algorithm in a specific format, which is then automatically integrated into CakeML via a proof-producing translation. In this section we describe this format by presenting parts of the schoolbook algorithm, already implemented in CakeML's bignum library.

**Definition 5.1.1**

```
mc_mul (l,r7,r9,r10,r12,xs,ys,zs) =
if r7 = 0w then (let r10 = r10 + r9 in (l,r10,xs,ys,zs))
else
 (let r7 = r7 − 1w in
  let r8 = EL (w2n r12) xs in
  let r12 = r12 + 1w in
  let r11 = 0w in
  let r1 = r11 in
  let (l,r1,r9,r10,ys,zs) =
      mc_mul_pass (l − 1,r1,r8,r9,r10,r11,ys,zs)
  in
  let r10 = r10 − r9
  in
   mc_mul (l − 1,r7,r9,r10,r12,xs,ys,zs))
```

**Theorem 5.1.1**

```
length (xs1 ++ xs) < 2^α ∧ length ys < 2^α ∧
(length zs = length ys) ∧ length (zs1 ++ zs ++ zs2) < 2^α ∧
length xs ≤ length zs2 ∧ ys ≠ [] ∧ 2 × length xs + length xs × length ys ≤ l ⇒
∃ zs3 l2.
 mc_mul_pre
  (l,n2w (length xs),n2w (length ys),n2w (length zs1),n2w (length xs1),
   xs1 ++ xs,ys,zs1 ++ zs ++ zs2) ∧
 (mc_mul
    (l,n2w (length xs),n2w (length ys),n2w (length zs1),n2w (length xs1),
     xs1 ++ xs,ys,zs1 ++ zs ++ zs2) =
  (l2,n2w (length (zs1 ++ mw_mul xs ys zs)),xs1 ++ xs,ys,
   zs1 ++ mw_mul xs ys zs ++ zs3)) ∧
 (length (zs1 ++ zs ++ zs2) =
  length (zs1 ++ mw_mul xs ys zs ++ zs3)) ∧
 l ≤ l2 + 2 × length xs + length xs × length ys
```

The implementation of `mc_mul` above gives an example of how a function in the required format looks like. For example we see that almost all of its variables have a name consisting of a "r" and a number, and are of the type $\alpha$`word`. This is one restriction enforced by the integration infrastructure, it resembles low-level programming with registers. The functions correctness theorem is given in Theorem 5.1.1, which relates its result to the higher-level function `mw_mul`. The format consists of the following constraints:

- Only allowed to work with *register variables*, which is of the type $\alpha$`word` and must have the name $ri$, where $i$ is the register index.
- The function needs to handle a *clock variable* $l$ which decrements in each recursive call.

- The function needs to return a *condition variable* which is set to false if any of the functions preconditions is violated.
- For functions operating on lists (array-like structures), only EL and LUPDATE is allowed for manipulating the list (only exception is in recursive calls, when we are allowed to use **take** and **drop**). Also, an upper bound on the length of the lists is needed.

For our implementation presented in the following sections, we focus on a subset of the requirements which are: the use of a *clock variable* and *condition variable*, also operate on a list of limited length. Hence, we are not limiting our implementation to *register variables* and we also allow a more freely manipulation of lists. This allows for an easier implementation, that works as an intermediate step towards an implementation that fully complies with the given format.

## 5.2 Memory manipulation

The top-level function for our implementation needs to operate on three arrays $xs, ys$, and $zs$, in order to satisfy the format required by the CakeML-integration. The arrays are represented as lists of machine words ( as usual), and both $xs$ and $ys$ is considered immutable. Therefore $zs$ is our *working memory* where all intermediate results are saved. In this section we present how our implementation utilizes these arrays.

In Figure 5.1 the initial state of $xs$,$ys$ and $zs$ for a multiplication is presented. We see that $xs$ and $ys$ contain $x$ and $y$ respectively, and remember that the bignum representation used in HOL have the least significant digit placed first in the list. So the significance increases from left to right in the list. We also see that $zs$ is initialized, but we do not know what it contains, which is represented as grey area in the rectangle.


**Figure 5.1:** Initial memory allocation before multiplication.

Now, the imperative implementation presented in this chapter contains the same parts as the implementation of the Karatsuba algorithm from the previous chapter. That is, it starts with a function that initializes the algorithm (**imp_ktb_init**), then three recursive calls (to **imp_ktb_mul**, then ends with a call to a function which finalizes the algorithm (**imp_ktb_final**). Below follows a description of how these functions manipulates the available memory.

The first function to store its result in $zs$ is the init-function ( **mw_ktb_mul_init** from Chapter 3 and **imp_ktb_mul_init** is the corresponding function in the imper-

ative implementation). It has the responsibility to compute $x0x1$ and $y0y1$ which we now need to store in $zs$. From the initial state given in Figure 5.1, the init-function stores $x0x1$ and $y0y1$ in the beginning of $zs$, as depicted in Figure 5.2. We see also that the operands have been split into its respective parts, and that $x0l$ and $y0l$ describe the length of $x0$ and $y0$ (we don't need $x1l$ and $y1l$ since for example $x1l = xl - x0l$). The arrows in the figure shows which parts of the memory is used to calculate $x0x1$ and $y0y1$.

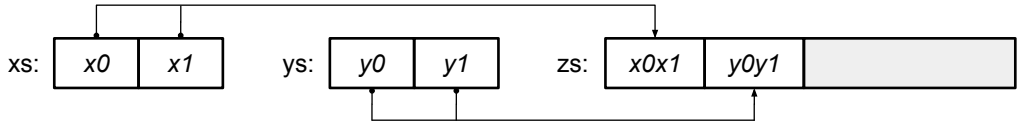


**Figure 5.2:** Memory state after call to init-function.

For the recursive calls we need not just to overwrite elements in $zs$, but also to change what memory $xs$ and $ys$ refers to. In Figure 5.3, we above the dotted line, what $xs$, $ys$, and $zs$ are before the recursive call. Below the line we see what they are during the recursive call. The arrows going from the pre-state to call-state show to what memory $xs$, $ys$, and $zs$ are referenced to. The first recursive call calculates $z_0 = x_0y_0$, and hence $xs$ is changed to only include $x0$ and $ys$ to only include $y0$. We also want to keep our previous result in $zs$ ($x0x1$ and $y0y1$) and thus change $zs$ to not overwrite it. In the bottom of Figure 5.4 the post-state of the recursive call is displayed, where $zs$ is pointing to its original memory location.

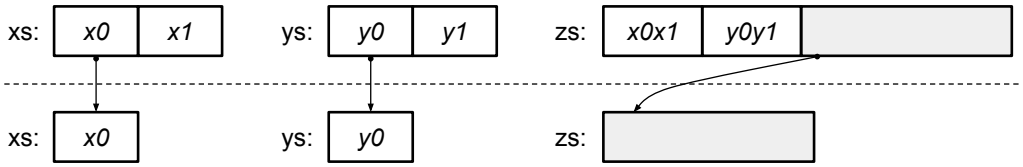


**Figure 5.3:** Memory reallocation before recursive call for $z0$.

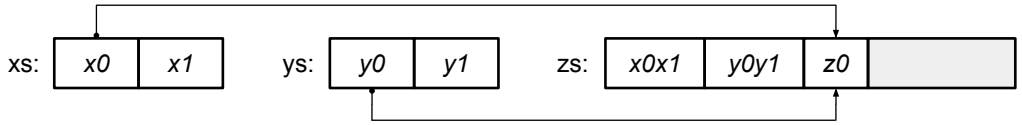


**Figure 5.4:** Memory state after recursive call for $z0$.

The next recursive call computes $z_2 = x_1y_1$ and we again change $xs$, $ys$, and $zs$ to point to the correct parts. Figure 5.5 presents the post state, after the recursive call of $z2$.

The final recursive call of the Karatsuba algorithm calculates $z_1 = (x_0+x_1)(y_0+y_1)$. We have $x0x1 = (x_0 + x_1)$ and $y0y1 = (y_0 + y_1)$ stored in the beginning of $zs$, and thus need to point $xs$ and $ys$ to their respective location in $zs$. This is displayed in Figure 5.6, and its post state is presented in Figure 5.7.

After the last recursive call we have $z0$, $z1$, and $z2$ stored in $zs$. The final step of the Karatsuba algorithm is to recombine these values into the wanted product.

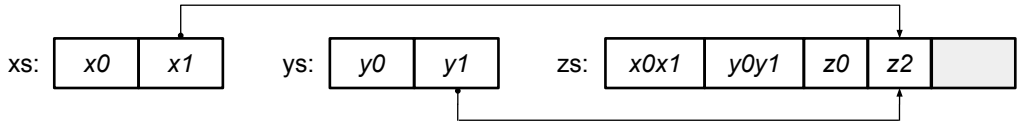


**Figure 5.5:** Memory state after recursive call for $z2$.

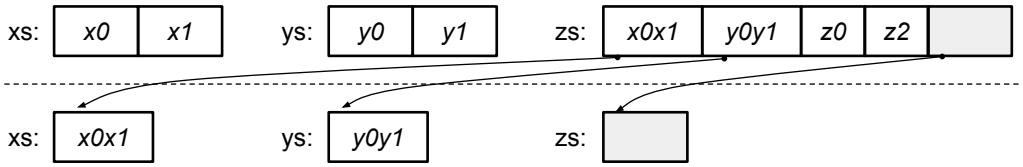


**Figure 5.6:** Memory state after recursive call for $z1$.

This is done by the **mw_ktb_mul_final** function, from Chapter 3, the corresponding function in this chapter is **imp_ktb_final**. In Figure 5.8 we present how it perform the needed operations within $zs$. It starts by calculating $z1 - z0 - z2$ in place of $z1$. Then, $z0 + B2mz2$ can be computed by simply moving $z0$ to the front of $zs$, and $z2$ to position $2m$, since $z0$ has a maximum length of $2m$ so the numbers do not overlap. To conclude the calculations, $B^m(z1 - z0 - z2)$ is added (denoted as $z1'$). Observe that some additional operations is needed, such as clearing elements in $zs$ to zero before moving elements, also the lengths in the figures are not representative.

With the memory usage defined, we now continue with deriving an upper bound on the size of $zs$ depending on the size of the operands. After that, we continue with the specification and verification of the implementation.

### 5.2.1 Memory requirements

The implementation presented in this chapter operates with limited working memory, as a step towards a CakeML-compatible implementation. Hence, we need to define how much memory needs to be allocated. We would like to formulate as tight of a limit as possible, but since this is an intermediate implementation, and our main goal is to prove its correctness (not efficiency) we provide a good-enough limit here.

The memory requirement is formulated on the format $A(\texttt{length}\ xs + \texttt{length}\ ys) \leq \texttt{length}\ zs$, where $A$ is a non-negative number (for brevity we will use the prefix $l$ for the length of variables, e.g. $yl = \texttt{length}\ ys$ and $x0x1l = \texttt{length}\ x0x1$).

From the above description of the memory manipulation of the implementation, we see that the recursive call for $z1$ have the largest requirement on the size of $zs$. This is because it is the last recursive call, which means that the front of $zs$ already is occupied with intermediate results such as $z0$ and $z2$. Also, $z1$ has the largest arguments of the recursive calls, with $z1 = (x0 + x1)(y0 + y1)$. Therefore, if the memory requirement for $z1$ can be satisfied, the requirements for $z0$ and $z2$ follows.

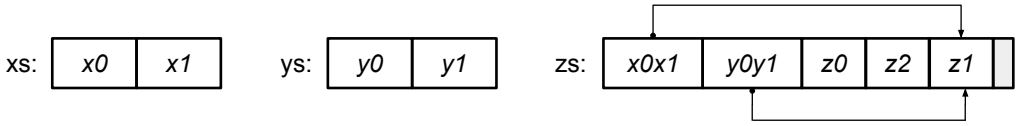


**Figure 5.7:** Memory state after recursive call for $z1$.

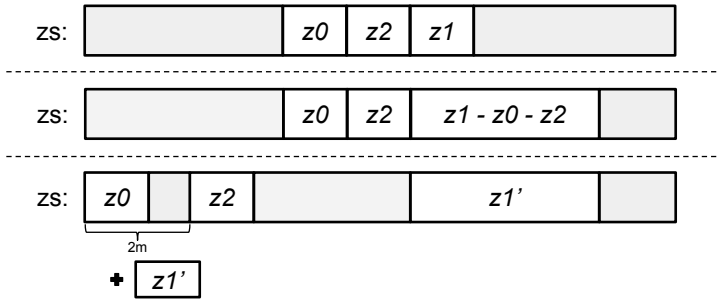


**Figure 5.8:** How final function calculates the final product.

From Figure 5.7 the following inequality need to hold:

$$A(x0x1l + y0y1l) \leq A(xl + yl) - (x0x1l + y0y1l + z0l + z2l) \tag{5.1}$$

On the left side we have the requirement for the recursive call for $z1$, which has the arguments $x0x1$ and $y0y1$. On the right side the assumption on $zs$ ($A(xl{+}yl) <= zs$) is present, with the already occupied memory for $x0x1$, $y0y1$, $z0$, and $z2$ subtracted. Now, $z0l = x_0y_0$ and have a maximum size of $x0l + y0l$, and similar statement can be said about $z2l$. For this we have that $z0l + z2l \leq x0l + y0l + x1l + y1l = xl + yl$, which gives the following from (5.1):

$$(A + 1)(x0x1l + y0y1l) \leq (A - 1)(xl + yl) \tag{5.2}$$

Let $x_{max} = \texttt{max}\ x0l\ x1l$ and $x_{min} = xl - x_{max}$, and the same for $y_{max}$ and $y_{min}$, e.g. if $x_{max} = x0l$ then $x_{min} = x1l$ and vice versa. With this notation we know the following facts:

$$\begin{aligned} xl &= x_{max} + x_{min}, & x0x1l &\leq x_{max} + 1 \\ yl &= y_{max} + y_{min}, & y0y1l &\leq y_{max} + 1 \end{aligned} \tag{5.3}$$

With the above, inequality 5.2 can be further simplified to:

$$2(x_{max} + y_{max}) + 2A + 2 \leq (A - 1)(x_{min} + y_{min}) \tag{5.4}$$

Now we want to specify an upper bound on $x_{max} + y_{max}$ and a lower bound on $x_{min} + y_{min}$ with a common term in order to get an useful statement. This term

is the splitting value $m =$ **max** $xl$ $yl$ **div** 2. Both $x_{max}$ and $y_{max}$ is bounded above by $m + 1$, since $m$ uses integer division (e.g. when $xl = 2n + 1$ we have $x0l = n$ and $x1l = n + 1$). For the lower bound on $x_{min} + y_{min}$, lets consider the worst-case scenario where $yl < xl$ and $yl = m$. That implies that $y_{min} = 0$ and $x_{min} = m$, which gives the lower bound $m <= x_{min} + y_{min}$. By applying these bounds to the inequality (5.4) we get:

$$\begin{aligned} 2(2m + 2) + 2A + 2 &\leq (A - 1)m \\ 2A + 6 &\leq (A - 5)m \end{aligned}$$

Which implies the following constraints on $m$ and $A$:

$$\frac{2A + 6}{A - 5} \leq m, \quad 5 < A \tag{5.5}$$

$$\frac{5m + 6}{m - 2} \leq m, \quad 2 < m \tag{5.6}$$

This presents a trade-off between the amount of allocated memory needed, and the cut-off point of the algorithm (since the only guarantee on $m$ is in relation with the cut-off point). For example, if $A = 6$ then $18 \leq m$, which means that the cut-off (e.g. end of recursion) needs to be at least $2 \times m = 36$.

That the memory requirement puts limits on the cut-off point of the implementation seems strange, but arises from the implementation detail that $x0x1$ and $y0y1$ have a risk of exceeding the size of a machine word, and thus result in a carry. In the worst-case scenario described above, we have only guarantees related to $m$ to work with, which gives the resulting requirements presented in 5.5. There exists alternative solutions to avoid the potential carry in $x0x1$ and $y0y1$, for example use a base smaller than the size of a machine word [4], or an alternative formulation of the Karatsuba algorithm that have $z_1 = (x_0 - x_1)(y_0 - y_1)$ (as GMP [14]). Those are possible options that could tighten the memory allocation needed, but could pose other problems such as the need to handle negative integers which the alternative formulation requires. For our implementation, we settle with the given algorithm instance given in Section 3, and for the constraints on memory we continue with $A = 10$ and $m = 6$ (cut-off = 12).

## 5.3 Specification in HOL

In this section we present an imperative styled version of the Karatsuba implementation from Chapter 3, which is a step towards a CakeML-integration. It is a step towards satisfying the required format by the CakeML-infrastructure by operating with a working-memory $zs$ where intermediate results are stored, using a clock-variable to prove its termination, and also have a conditional boolean which tracks that memory limits are kept.

The implementation's main function is called **imp_ktb_mul** and corresponds to **mw_ktb_mul**. It follows the same structure, with the functions **imp_ktb_init** and **imp_ktb_final** in place of **mw_ktb_mul_init** and **mw_ktb_mul_final**. Also, **imp_ktb_mul** takes three additional arguments: a counter $l$, a list for the result $zs$, and a boolean $c$. It also returns three additional results: a new value for $l$, length of result, $zs$ with product in front, and the boolean $c$ which is **true** if everything went ok, otherwise **false**. The definition of **imp_ktb_mul** is presented below.

**Definition 5.3.1**

```
imp_ktb_mul (cutoff,l,xs,ys,zs,c) =
(let (l0,xl,yl) = (l,length xs,length ys)
 in
  if cutoff < 12 ∨ xl < cutoff ∨ yl < cutoff then
   (let (l,rl,zs,c) = imp_mwn_mul (l,xs,ys,zs,c)
    in
     imp_mw_fix (l,rl,zs,c))
  else
   (let (l',m,x0l,y0l,x0x1l,y0y1l,zs',c') =
        imp_ktb_init (l,xs,ys,zs,c)
    in
    let l'' = min l' l0
    in
     if l'' = 0 then (l'',0,zs',F)
     else
      (let c'' = c' ∧ x0l ≤ length xs ∧ y0l ≤ length ys in
       let (l''',z0l,zs'',c''') =
            imp_ktb_mul
             (cutoff,l'' − 1,take x0l xs,take y0l ys,
              drop (x0x1l + y0y1l) zs',c'')
       in
       let l'''' = min l''' l0
       in
        if l'''' = 0 then (l'''',0,zs',F)
        else
         (let c'''' = c''' ∧ x0l ≤ length xs ∧ y0l ≤ length ys in
          let (l''''',z2l,zs''',c''''') =
               imp_ktb_mul
                (cutoff,l'''' − 1,drop x0l xs,drop y0l ys,
                 drop z0l zs'',c'''')
          in
          let l''''' = min l''''' l0
          in
           if l''''' = 0 then (l''''',0,zs',F)
           else
            (let (l'''''',z1l,zs'''',c'''''') =
                 imp_ktb_mul
                  (cutoff,l''''' − 1,take x0x1l zs',
                   take y0y1l (drop x0x1l zs'),drop z0l zs''',
                   c''''')
             in
             let l = min l'''''' l0 in
             let zs =
                  take (x0x1l + y0y1l) zs' ++ take z0l zs'' ++
                  take z2l zs''' ++ zs''''
             in
              imp_ktb_final
               (l,m,x0x1l + y0y1l,z0l,z2l,z1l,zs,
                c''''''))))))
```

Compared to **mw_ktb_mul** (Definition 3.2.1), **imp_ktb_mul** seems at first sight to be much more complicated. But, if we ignore the details we see that the same structure is there, with a few more statements around each recursive call that operates on $l$ and $c$.

The base case calculates the product with two newly defined helper functions, which operates on $zs$ in the same fashion as **imp_ktb_mul**. These helper functions are based on the functions previously defined (see Section 2.3.2), and calculate their results outside $zs$ and then put the result into the list. The proper low-level implementation would require the operation to be performed completely within the bounds of $zs$, but for this intermediate implementation we make this simplification. Definition 5.3.2 presents the implementation of **imp_mwn_mul**, where the multiplication is performed and then inserted into $zs$, also it checks that the result fits in $zs$.

**Definition 5.3.2**

$$
\begin{array}{l}
\textbf{\textit{imp\_mwn\_mul}}\ (l, xs, ys, zs, c) = \\
(\mathsf{let}\ xy\ =\ \textbf{\textit{mwn\_mul}}\ xs\ ys\ \mathsf{in} \\
\ \mathsf{let}\ c\ =\ c \wedge \textbf{\textit{length}}\ xy \leq \textbf{\textit{length}}\ zs \\
\ \mathsf{in} \\
\quad (l, \textbf{\textit{length}}\ xy, xy\ \textbf{\textit{++}}\ \textbf{\textit{drop}}\ (\textbf{\textit{length}}\ xy)\ zs, c))
\end{array}
$$

After **imp_mwn_mul** any unnecessary zero-words are removed from the tail of the list of machine words by **imp_mw_fix**, which is defined below in Definition 5.3.3.

**Definition 5.3.3**

$$
\begin{array}{l}
\textbf{\textit{imp\_mw\_fix}}\ (l, rl, zs, c) = \\
(\mathsf{let}\ c\ =\ c \wedge rl \leq \textbf{\textit{length}}\ zs\ \mathsf{in} \\
\ \mathsf{let}\ rs\ =\ \textbf{\textit{mw\_fix}}\ (\textbf{\textit{take}}\ rl\ zs) \\
\ \mathsf{in} \\
\quad (l, \textbf{\textit{length}}\ rs, rs\ \textbf{\textit{++}}\ \textbf{\textit{drop}}\ (\textbf{\textit{length}}\ rs)\ zs, c))
\end{array}
$$

With the base case covered, we continue with a basic step of the imperative implementation. It starts with a call to **imp_ktb_init** which corresponds to **mw_ktb_mul_init**, but this version stores $x0x1$ and $y0y1$ in $zs$ instead of returning them directly. Also note that we do not need the lengths for $x1$ and $y1$, since $\textbf{length}\ x0 = \textbf{length} xs - \textbf{length} x0$.

**Definition 5.3.4**

```
imp_ktb_init (l,xs,ys,zs,c) =
(let m = max (length xs) (length ys) div 2 in
 let (x0,x1) = (take m xs,drop m xs) in
 let (y0,y1) = (take m ys,drop m ys) in
 let x0x1 = mwn_add x0 x1 in
 let y0y1 = mwn_add y0 y1 in
 let c = c ∧ length x0x1 + length y0y1 ≤ length zs
 in
  (l,m,length x0,length y0,length x0x1,length y0y1,
   x0x1 ++ y0y1 ++ drop (length (x0x1 ++ y0y1)) zs,c))
```

After `imp_ktb_init` there is three recursive calls to calculate $z_0$, $z_2$, and $z_1$. For each recursive call, four steps is taken. First, we check if the clock-variable $l$ is 0, if so we stop the recursion and return an arbitrary value. This aids in proving the termination of the algorithm, but for the correctness $l$ need to have a large enough value for the implementation to terminate correctly. If $l$ is not zero the function continues with a check on the length of the arguments. This is to make sure that no out-of-bounds-error is present, if so $c$ will become false which is then returned. The two first steps verifies that it is ok to perform another recursive call, which is the third step. Here we note that we do not overwrite $zs$ in the return values, which would result in a loss of data. Instead we receive the resulting list with a temporary variable (in the case of the first recursive call, $zs'$) Finally, $l$ is set to the minimum of itself and its original value $l0$, this is an implementation-trick that makes the termination easier to prove.

The last function call in `imp_ktb_mul` is the call to `imp_ktb_final`, which given an initialized $zs$ according to Figure 5.8 performs the needed shifts, subtractions, and additions to calculate the wanted product. In Definition 5.3.5 the current implementation is presented. Note that this is, as the other helper functions, a simplified version that gives the correct result but perform its computation outside of $zs$.

**Definition 5.3.5**

```
imp_ktb_final (l,m,zp,z0l,z2l,z1l,zs,c) =
(let c = c ∧ zp + z0l + z1l + z2l ≤ length zs in
 let z0 = take z0l (drop zp zs) in
 let z2 = take z2l (drop (zp + z0l) zs) in
 let z1 = take z1l (drop (zp + z0l + z2l) zs) in
 let p2 = replicate (2 × m) 0w ++ z2 in
 let p1 = replicate m 0w ++ mw_subv z1 (mwn_add z0 z2) in
 let xy = mw_fix (mwn_add z0 (mwn_add p1 p2))
 in
  (l,length xy,xy ++ drop (length xy) zs,c))
```

This concludes the specification of `imp_ktb_mul` in HOL. As mentioned, it has the same structure as `mw_ktb_mul` but instead of handling intermediate results directly,

it stores them in and fetches them from $zs$. The implementation also has two additional arguments $l$, which works as a clock that decreases at each function call, and $c$, which checks that we stay within the length limitations of the bignum representations. In the next section a description of the verification of this implementation is given.

## 5.4 Verification in HOL

For the implementation presented in this chapter we do not want to prove that it correctly implements multiplication over the integers directly. Instead, we want to prove that it correctly implements the previously specified Karatsuba implementation from Section 3. In this way, we build a kind of proof-chain downwards to more machine-code like implementations. We state the top-level correctness theorem below, and then we continue with the proof of termination (Section 5.4.1), which is easier than previous proof due to the introduction of the clock $l$. After the termination is proven we give a description of how the correctness theorem is proven. Note that, as of writing, not every detail of the main correctness proof is fully verified in HOL, but hopefully this will be completed before the presentation.

The top-level correctness theorem we want to prove for **imp_ktb_mul** is given below. Instead of relating the function to multiplication over the integers, the theorem relates it to **mw_ktb_mul**. By proving the theorem we indirectly prove that **imp_ktb_mul** correctly implements multiplication, which follows from the already verified correctness theorem of **mw_ktb_mul** (Theorem 3.3.2).

**Theorem 5.4.1**

$$
\begin{aligned}
&10 \times (\textbf{length}\ xs\ \textbf{+}\ \textbf{length}\ ys) \leq \textbf{length}\ zs \wedge \textbf{length}\ xs\ \textbf{+}\ \textbf{length}\ ys < l \wedge (c \iff \textbf{T}) \Rightarrow \\
&\exists\, zs_1\ l_2. \\
&\quad (\textbf{imp\_ktb\_mul}\ (co\textbf{,}l\textbf{,}xs\textbf{,}ys\textbf{,}zs\textbf{,}c)\ \textbf{=} \\
&\quad\ (l_2\textbf{,}\textbf{length}\ (\textbf{mw\_ktb\_mul}\ co\ xs\ ys)\textbf{,}\textbf{mw\_ktb\_mul}\ co\ xs\ ys\ \textbf{++}\ zs_1\textbf{,} \\
&\quad\ \textbf{T})) \wedge (\textbf{length}\ (\textbf{mw\_ktb\_mul}\ co\ xs\ ys\ \textbf{++}\ zs_1)\ \textbf{=}\ \textbf{length}\ zs) \wedge \\
&\quad l \leq l_2\ \textbf{+}\ \textbf{length}\ xs\ \textbf{+}\ \textbf{length}\ ys
\end{aligned}
$$

The above theorem also states requirements on both the length of $zs$ and the size of $l$, which need to hold for each recursive call. In Section 5.2.1 we derived an upper bound on the memory usage, which will be useful in the proof procedure. Note also that the theorem makes no guarantees on the preservation of the content in $zs$, only that the resulting product is placed in the front. In the following section we will start to prove the termination of the given implementation, after this we continue with the proofs for the above state correctness theorem.

### 5.4.1 Termination proof

The termination of **imp_ktb_mul** is easier to prove than for the other implementations verified in previous chapters. This is mainly due to the introduction of the clock variable $l$. Our implementation, given in Definition 5.3.1, decreases the clock for each function call, and terminates if it reaches zero. Therefore, we can prove the termination by using $l$ as a metric that is proven to decrease in each recursive call.

The clock-trick makes the termination proof trivial, but it comes with a downside. The downside is that for correctness we now need to show that $l$ never reaches zero, so that there is enough time for the algorithm to terminate and return the correct result.

### 5.4.2 Correctness proof

With the termination proven, we now want to complete the proof for the correctness theorem stated above (Theorem 5.4.1). We begin with proving correctness theorems for the functions used by **imp_ktb_mul**. Currently, when writing this, the proof of the correctness theorem is not fully verified in HOL, but are a few steps left.

The correctness theorem for **imp_ktb_init** states how the functions result relates to the result of **mw_ktb_mul_init**. The theorem is given below.

**Theorem 5.4.2**

$$
\begin{aligned}
&\texttt{length}\ xs + \texttt{length}\ ys \leq \texttt{length}\ zs \wedge \\
&((m, x_0, x_1, y_0, y_1, x0x1, y0y1) = \texttt{mw\_ktb\_mul\_init}\ xs\ ys) \Rightarrow \\
&\exists\, zs_1. \\
&\quad (\texttt{imp\_ktb\_init}\ (l, xs, ys, zs, \texttt{T}) = \\
&\quad\ (l, m, \texttt{length}\ x_0, \texttt{length}\ y_0, \texttt{length}\ x0x1, \texttt{length}\ y0y1, x0x1 \mathbin{++} y0y1 \mathbin{++} zs_1, \texttt{T})) \wedge \\
&\quad (\texttt{length}\ (x0x1 \mathbin{++} y0y1 \mathbin{++} zs_1) = \texttt{length}\ zs)
\end{aligned}
$$

Since the implementation of **imp_ktb_init** is similar to the implementation of **mw_ktb_mul_init**, the correctness theorem follows from proving that there is enough space in $zs$ to accommodate $x0x1$ and $y0y1$. This holds by the requirement on $zs$ from the theorem, since $\texttt{length}\ (x0x1 \mathbin{++} y0y1) \leq \texttt{length}\ (ys \mathbin{++} xs)$.

Next we address the correctness of **imp_ktb_final** which is proven in a similar fashion. Theorem 5.4.3 states the function's relation to **mw_ktb_mul_final**.

**Theorem 5.4.3**

$$
\begin{array}{l}
(xy \texttt{ = mw\_ktb\_mul\_final } m\ z_0\ z_1\ z_2) \Rightarrow \\
\exists\, zs_2. \\
\quad (\texttt{imp\_ktb\_final} \\
\qquad (l\texttt{,}m\texttt{,length }zs_0\texttt{,length }z_0\texttt{,length }z_2\texttt{,length }z_1\texttt{,}zs_0 \texttt{ ++ } z_0 \texttt{ ++ } z_2 \texttt{ ++ } z_1 \texttt{ ++ } zs_1\texttt{,} \\
\qquad \texttt{T}) \texttt{ =} \\
\quad (l\texttt{,length }xy\texttt{,}xy \texttt{ ++ } zs_2\texttt{,T})) \wedge \\
\quad (\texttt{length }(xy \texttt{ ++ } zs_2) \texttt{ = length } (zs_0 \texttt{ ++ } z_0 \texttt{ ++ } z_2 \texttt{ ++ } z_1 \texttt{ ++ } zs_1))
\end{array}
$$

With the above stated theorems, we can now turn our attention to the top-level correctness. This is as of writing not completed in HOL. It turns out the most difficult part of the proof is regarding the requirement of space in $zs$. Parts of the derivation of the requirement (in Section 5.2.1) have been formalized into HOL, but some minor details is still due. When this is in place, the two above stated theorems of `imp_ktb_init` and `imp_ktb_final` with the induction hypothesis on `imp_ktb_mul` is enough to prove the correctness.

## 5.5 Further steps for a CakeML-integration

The implementation presented in this chapter fulfills parts of the format required for a CakeML integration of the algorithm, and works as a step towards a valid implementation.

Our initial aim of this thesis was to successfully integrate the Karatsuba algorithm into CakeML's bignum library. An implementation that satisfies all constraints listed in Section 5.1 have been specified, but only partially proven, with some significant challenges that were not overcome. Instead we implemented this intermediate implementation, that takes a concrete step towards a CakeML integration, which is left as future work.

# 6

# Conclusion

In this thesis, we have described and implemented two asymptotically fast bignum multiplication algorithms, namely the Karatsuba (Chapter 3) and Toom-3 algorithms (Chapter 4), using the interactive theorem prover HOL4. The implementations have been verified to correctly implement integer multiplication. Further, we have taken a step towards extending the CakeML's bignum library with the Karatsuba algorithm.

The integration into the CakeML's bignum library works by specifying an implementation of the algorithm in a specific format, which is then integrated into CakeML via a proof-producing translation. Our initial aim for this thesis was to specify such an implementation, but due to mainly lack of time, we have not managed to achieve this. Instead we have taken a step towards it, by specifying an implementation that satisfy parts of the required format. This was presented in Chapter 5, where we have also verified its termination and correctness.

# Bibliography


[1] Reynald Affeldt. On construction of a library of formally verified low-level arithmetic functions. *ISSE*, 9(2):59–77, 2013.

[2] Jon Louis Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, September 1980.

[3] Stephen A. Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966.

[4] Shamil Dzhatdoyev and Marco T. Morazan. On the implementation of bignum multiplication. Presented at Trends in Functional Programming (TFP), Canterbury, UK, 2017.

[5] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595, 1962.

[6] Jon Kleinberg and Éva Tardos. *Algorithm design*. Pearson/Addison-Wesley, Boston, Mass, 2006.

[7] Donald E Knuth. *The art of computer programming, volume 2: (2nd ed.): Seminumerical algorithms*. Addison-Wesley Reading, 1981.

[8] Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2013.

[9] Magnus O. Myreen and Michael J. C. Gordon. Verification of machine code implementations of arithmetic functions for cryptography. Presented as a rough diamond in the short-paper category of: Schneider, K., Brandt, J. (eds.) Theorem Proving in Higher Order Logics, Emerging Trends Proceedings (TPHOLs, Poster Session). University of Kaiserslautern, 2007.

[10] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. How to Get an Efficient yet Verified Arbitrary-Precision Integer Library. working paper or preprint, April 2017.

[11] Konrad Slind and Michael Norrish. A brief overview of HOL4. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5170 LNCS:28–32, 2008.

[12] Berk Sundar. Multiprecision Multiplication. *Encyclopedia of Cryptography and Security*, 1, 2011.

[13] Yong K. Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A New Verified Compiler Backend for CakeML. 2016.

[14] Granlund Torbjorn and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library. December 2016.