Department of Mathematics
TUM School of Computation, Information and Technology
Technical University of Munich

A Verified Functional Implementation of the Schönhage-Strassen-Algorithm

Jakob Schulz

Thesis for the attainment of the academic degree

Master of Science

at the TUM School of Computation, Information and Technology of the Technical University of Munich

Supervisor:
Prof. Tobias Nipkow

Advisor:
Emin Karayel

Submitted:
Munich, 31st July 2023

Zusammenfassung

Der Schönhage-Strassen-Algorithmus multipliziert zwei ganze Zahlen der Länge n in $\mathcal{O}(n \log n \log \log n)$ Schritten auf einer mehrbändigen Turing-Maschine. Zentrales Ziel dieser Arbeit ist es, eine verifizierte Implementierung sowie verifizierte Laufzeitschranken anzugeben.

Als Repräsentation der Zahlen wählen wir boolsche Listen, auf denen zunächst grundlegende Algorithmen zur Addition, Subtraktion und klassischen Multiplikation angegeben werden, sowie eine verifizierte Implementierung der Karatsuba-Multiplikation mit Laufzeit $\mathcal{O}\left(n^{\log_2 3}\right)$.

Der bereits vorhandene AFP-Eintrag zu Zahlentheoretischen Transformationen [AK22] wird angepasst, um für bestimmte Restklassenringe anwendbar zu sein, die im Schönhage-Strassen-Algorithmus benötigt werden.

Schließlich wird eine Implementierung des Schönhage-Strassen-Algorithmus mit der erwähnten Laufzeit von $\mathcal{O}(n \log n \log \log n)$ basierend auf dem Original-Paper [SS71] gegeben.

Abstract

The Schönhage-Strassen-Algorithm multiplies two integers of length n in $\mathcal{O}(n \log n \log \log n)$ steps on a multitape Turing machine. The main goal of this thesis is to give a verified implementation of the algorithm as well as verified runtime bounds.

Integers are represented as LSBF (least significant bit first) boolean lists, on which simple algorithms for addition, subtraction and grid multiplication are given as well as a verified implementation of the Karatsuba-Multiplication with runtime $\mathcal{O}\left(n^{\log_2 3}\right)$.

The already existing AFP-entry for Number Theoretic Transforms [AK22] is adapted to a more general setting, which allows its application to certain quotient rings used in the Schönhage-Strassen-Algorithm.

After some final preparations, an implementation of the Schönhage-Strassen-Algorithm with runtime $\mathcal{O}(n \log n \log \log n)$ based on the original paper [SS71] is given.

Contents

1 Introduction

The main goal of this thesis is to verify the result of the original paper from Schönhage and Strassen [SS71], which states that it is possible to multiply integers (or equivalently natural numbers) of length n in $O(n \log n \log \log n)$ steps using a multitape Turing machine. The verification is done in Isabelle/HOL [NPW22].

We will work explicitly on binary representations rather than on Isabelle's built-in *nat*-type directly. In particular, we will use the type *nat-lsbf* = *bool list* with the least significant bits coming first in the list (LSBF-representation). Chapter 3 goes through the implementation of basic arithmetic operations on this representation as well as conversion functions from and to *nat* and some other auxiliary functions needed.

After these preparations, a verified implementation of the multiplication algorithm by Karatsuba and Ofman [KO62] with runtime $O\left(n^{\log_2 3}\right)$ is given in Chapter 4. This algorithm will be needed as a subroutine of the Schönhage-Strassen-Algorithm.

Next, the necessary theory about Fast Number Theoretic Transforms (FNTTs) is discussed in Chapter 5. The implementation is based on the work of Ammer and Kreuzer [AK22], who describe FNTTs in the context of fields. We adapt the theory to the more general setting of rings with adequate primitive roots of unity. The theoretical background is largely based on the lecture "Computeralgebra" held by Kemper [Kem21] in the winter semester 2021/2022 at TUM.

Finally, a verified implementation of the Schönhage-Strassen-Algorithm with runtime $O(n \log n \log \log n)$ is developed in Chapter 6, following the original paper from Schönhage and Strassen [SS71].

Since the Isabelle code contains formal proofs for our statements, there is no need to reproduce them in the same level of detail here. Instead, we will omit proofs of trivial statements,[1] but give informal versions of the proofs for more complicated ones. In both cases, we will give references to the corresponding Isabelle proofs.[2]

An overview of all Isabelle theory files is given in Table A.2. Moreover, the appendix contains an overview over some specific notations in Table A.1.

1.1 Method for Runtime Verification

Since there is no built-in measurement of runtime in Isabelle, there are two common methods for formalizing the runtime of some function $f : \overline{\alpha} \longrightarrow \beta$:

1. Define an independent runtime function $T_f : \overline{\alpha} \longrightarrow \mathbb{N}$.
2. Define a function $f_{\text{tm}} : \overline{\alpha} \longrightarrow (\beta \times \mathbb{N})$, where the runtime is returned in the second entry.

The second method can be implemented concisely using the time monad defined in Nipkow [Nip17], which can manage the runtime component automatically. For more complicated functions, this method is cleaner, less error-prone and we will hence use it for our runtime verification.

Also, we will always formally (i.e. in Isabelle code) prove that f_{tm} indeed calculates f in its first component, i.e. that $val\ (f_{\text{tm}}\ x) = f\ x$, but will generally not talk about that in this thesis. Moreover, note that, in theory, it is possible to just define e.g.

fun *f-tm* **where** *f-tm x* = *return* (*f x*)

resulting in a runtime of 0. However, we will avoid such insensible things and will write our functions in a way such that the runtime of each used function is accounted for. Additionally, we will always replace

[1] As is done in normal mathematical textbooks as well.

[2] For lemmas, this will mostly be done by writing the name of the corresponding Isabelle lemma next to the number/name of the lemma, see e.g. Lemma 5.12.

the equation symbol ”=” in each definition of a function f_{tm} by the symbol ”=*1*” defined in [Nip17] in order to account for any constant time that might be needed for processing the function call itself. For a simple example, see Figure 3.1.

As a last few comments:

1. Some functions (like the *rev* function) are defined in a way such that the resulting runtime is suboptimal, but have code equations with better runtime. Hence, we will mostly consider the equations returned by **code-thms** for runtime verification.
2. For functions on *nat*, we will also consider the output of **code-thms**, which results in a worse runtime than they may have after code generation (e.g. the addition $m + n$ of some m, n of type *nat* will be linear in m and n rather than logarithmic). We will treat the resulting runtime as a conservative bound, and use the type *nat-lsbf* (which explicitly represents numbers in binary) whenever that is insufficient.
3. Although runtime bounds given in Landau notation are easier to read, doing proofs is often easier with explicit bounds. In the Isabelle code, we will therefore mostly give explicit bounds, while the informal proofs will still mostly work with Landau notation.

1.2 The *estimation* Tactic

For many runtime proofs, the *estimation* tactic defined in `Estimation_Method` will be useful. It's idea is the following: assume we want to prove some inequality like

$$f(n) + f(2n) + 1 \le 10n^2 + 2 \quad (\forall n \in \mathbb{N}) \tag{1.1}$$

and we have a bound of the form

$$f(n) \le n^2 \quad (\forall n \in \mathbb{N}). \tag{1.2}$$

As a first step to show (1.1), an informal proof would apply the inequality (1.2) to the left-hand side and then continue by proving

$$n^2 + (2n)^2 + 1 \le 10n^2 + 2. \tag{1.3}$$

In Isabelle, this could be done using an equation chain. However, sometimes it would be easier if one would be able to just state ”use the inequality (1.2) and do simplifications to show (1.1)” directly. This is the main idea behind the *estimation* tactic, which can be applied to the goal (1.1) using (1.2) as *estimate* argument and leaves the goal (1.3), which can be shown using *simp*. In Isabelle code:

lemma *f-le*: $f\ n \le n^2$
proof ... **qed**

lemma $f\ n + f\ (2 * n) + 1 \le 10 * n^2 + 1$

1. $f\ n + f\ (2 * n) + 1 \le 10 * n^2 + 1$

apply (*estimation estimate*: *f-le*)

1. $n^2 + (2 * n)^2 + 1 \le 10 * n^2 + 1$

by *simp*

Applying an estimate to subterms of course requires that the surrounding operators satisfy monotonicity properties. The *estimation* tactic knows some common ones, mostly ones of operations on *nat*, but sometimes, additional goals may be leftover and need to be handled as well.

2 Preliminaries

2.1 Sums in Monoids

In Isabelle, sums of the form $\sum_{i \in I} a_i$ can be expressed by different means:

(a) For the operator $(+)$ on a type class satisfying the locale *monoid-add*, one can use *sum-list* (*map f xs*) (syntactic sugar: $\sum i \leftarrow xs.\ f\ i$) as well as *sum f I* (syntactic sugar: $\sum i \in I.\ f\ i$).

(b) For the operator $(\oplus)$ in the locale *abelian-monoid*, one can use *finsum f I* (syntactic sugar: $\bigoplus i \in I.\ f\ i$).

Since we will work in the context *cring*, we need to write sums using the $\oplus$ operator. For convenience, we want to define a notation similar to *sum-list*. For that, we define the function *monoid-sum-list*:

```
context abelian-monoid
begin

fun monoid-sum-list :: ['c ⇒ 'a, 'c list] ⇒ 'a where
  monoid-sum-list f [] = 0
| monoid-sum-list f (x # xs) = f x ⊕ monoid-sum-list f xs

end
```

as well as syntactic sugar by adapting the code from *finsum*:

```
syntax
  -monoid-sum-list :: index ⇒ idt ⇒ 'c list ⇒ 'c ⇒ 'a
    ((3⨁ --←--. -) [1000, 0, 51, 10] 10)
translations
  ⨁ Gi←xs. b ⇌ CONST abelian-monoid.monoid-sum-list G (λi. b) xs
```

Note that this definition yields the unconditional simplification rule *monoid-sum-list.simps*(*2*), compared to the slightly more complicated lemma *finsum-insert* (see Figure 2.1).

	monoid-sum-list.simps(*2*)	*finsum-insert*
Carrier assumptions		$f \in F \rightarrow carrier\ G$ $f\ a \in carrier\ G$
Other assumptions		*finite F* $a \notin F$
Statement	$(\bigoplus i \leftarrow (x \# xs).\ f\ i)$ $= f\ x \oplus (\bigoplus i \leftarrow xs.\ f\ i)$	$(\bigoplus i \in (insert\ a\ F).\ f\ i)$ $= f\ a \oplus (\bigoplus i \in F.\ f\ i)$

Figure 2.1 Comparison between insertion rules for *monoid-sum-list* resp. *finsum*.

Moreover, we get a congruence rule *monoid-sum-list-cong* similar to *finsum-cong*, but without the assumption that the summands have to be in the carrier (see Figure 2.2).

	monoid-sum-list-cong	*finsum-cong*[*OF refl*]
Carrier assumptions		$f \in A \to carrier\ G$
Other assumptions	$\bigwedge i.\ i \in set\ xs \Longrightarrow f\, i = g\, i$	$\bigwedge i.\ i \in A \Longrightarrow f\, i = g\, i$
Statement	$(\bigoplus i \leftarrow xs.\ f\, i) = (\bigoplus i \leftarrow xs.\ g\ i)$	$(\bigoplus i \in A.\ f\, i) = (\bigoplus i \in A.\ g\ i)$

Figure 2.2 Comparison between congruence rules for *monoid-sum-list* resp. *finsum.*

For simplicity, we will from now on write all sums using the symbol $\sum$. Standard lemmas about sums in the context of monoids are proven in `Preliminaries/Monoid_Sums`. One such lemma is the geometric sum lemma, formulated in the context of commutative rings:

Lemma 2.1 [*geo-monoid-sum-list*]. *Let R be a commutative ring and $x \in R$. Then, we have*

$$(1 - x) \cdot \sum_{i=0}^{r-1} x^i = 1 - x^r$$

Proof. The proof is done by induction on r. If $r = 0$, both sides equal 0. Otherwise, with $r = s + 1$, we have

$$(1 - x) \cdot \sum_{i=0}^{(s+1)-1} x^i = (1 - x) \cdot \left(x^s + \sum_{i=0}^{s-1} x^i \right) \overset{\text{IH}}{=} (1 - x) \cdot x^s + (1 - x^s) = 1 - x^{s+1}.$$

□

3 Binary Representations

In order to obtain sufficiently efficient algorithms, we need to represent natural numbers with logarithmic size. An obvious candidate would be the datatype *num* defined in `HOL.Num`:

datatype *num* = *One* | *Bit0 num* | *Bit1 num*

However, we will represent natural numbers as *bool list*s for the following reasons:

1. We can use any list function (like *rev*) without redefining it.
2. *num* represents the strictly positive natural numbers. Allowing 0 makes many algorithms simpler.
3. The representation of natural numbers using *num* is unique. However it is sometimes more convenient to allow leading/trailing zeros in order to control the length of representations.

Hence, we define:

type-synonym *nat-lsbf* = *bool list*

The conversion functions from and to *nat* can then be defined by

fun *to-nat* :: *nat-lsbf* ⇒ *nat* **where**
to-nat [] = *0*
| *to-nat* (*x*#*xs*) = (*eval-bool*[1] *x*) + *2* ∗ *to-nat xs*

fun *from-nat* :: *nat* ⇒ *nat-lsbf* **where**
from-nat 0 = []
| *from-nat x* = (*if x mod 2 = 0 then False else True*)#(*from-nat* (*x div 2*))

This defines a least significant bit first (LSBF) encoding, i.e. *from-nat 2* = [*False*, *True*]. Note that the encoding is not unique, since e.g. [*False*, *True*] and [*False*, *True*, *False*, *False*] both represent the same number. We will allow this non-uniqueness, since it makes some algorithms easier. The representation is correct in the sense that *to-nat* (*from-nat x*) = *x*.[2]

In `Binary_Representations/Binary_Representations`, some useful lemmas are shown in the section `nat/int in lsbf and conversions`. The most important lemmas are:

lemma *to-nat-app*: *to-nat* (*xs* @ *ys*) = *to-nat xs* + (*2* ^ *length xs*) ∗ *to-nat ys*
lemma *to-nat-length-bound*: *to-nat xs* < *2* ^ *length xs*
lemma *to-nat-length-lower-bound*: *to-nat* (*xs* @ [*True*]) ≥ *2* ^ *length xs*
lemma *to-nat-drop-take*: *to-nat xs* = *to-nat* (*take k xs*) + *2* ^ *k* ∗ *to-nat* (*drop k xs*)

The lemma *to-nat-drop-take* also implies the following two lemmas that will be used often:

lemma *to-nat-take*: *to-nat* (*take k xs*) = *to-nat xs mod 2* ^ *k*
lemma *to-nat-drop*: *to-nat* (*drop k xs*) = *to-nat xs div 2* ^ *k*

3.1 Addition

In order to define addition on *nat-lsbf*, we first define some auxiliary functions:

- *bit-add-carry* adds three single bits and returns the sum in two bits. It is defined exhaustively for any input combination.

[1] *eval-bool* has the same semantics as *of-bool*, but is defined in a slightly different way and on a non-generic type. This slightly simplifies some proofs later.
[2] See the Isabelle lemma *to-nat-from-nat*.

- *inc-nat* increments a number by one:

  ```
  fun inc-nat :: nat-lsbf ⇒ nat-lsbf where
  inc-nat [] = [True]
  | inc-nat (False # xs) = True # xs
  | inc-nat (True # xs) = False # (inc-nat xs)
  ```

- *add-carry* adds two numbers and a carry bit:

  ```
  fun add-carry :: bool ⇒ nat-lsbf ⇒ nat-lsbf ⇒ nat-lsbf where
  add-carry False [] y = y
  | add-carry False x [] = x
  | add-carry True [] y = inc-nat y
  | add-carry True x [] = inc-nat x
  | add-carry c (x#xs) (y#ys) = (let (a, b) = bit-add-carry c x y in a#(add-carry b xs ys))
  ```

Finally, we define

```
definition add-nat :: nat-lsbf ⇒ nat-lsbf ⇒ nat-lsbf where
add-nat x y = add-carry False x y
```

Lemma 3.1 [*add-nat-correct*]. *The addition defined by add-nat is correct, i.e. to-nat (add-nat x y) = to-nat x + to-nat y.*

For our runtime proofs, we will also need lemmas about the length of the results. The most important lemmas here are the following:

lemma *length-inc-nat-lower*: *length (inc-nat xs) ≥ length xs*
lemma *length-inc-nat-upper*: *length (inc-nat xs) ≤ length xs + 1*
lemma *length-inc-nat-iff*: *length (inc-nat xs) = length xs ⟷ (∃ ys zs. xs = ys @ False # zs)*
lemma *inc-nat-last-bit-True*: *length (inc-nat xs) = Suc (length xs) ⟹ ∃ zs. inc-nat xs = zs @ [True]*

corollary *length-add-nat-lower*: *length (add-nat xs ys) ≥ max (length xs) (length ys)*
corollary *length-add-nat-upper*: *length (add-nat xs ys) ≤ max (length xs) (length ys) + 1*
corollary *add-nat-last-bit-True*: *length (add-nat xs ys) = max (length xs) (length ys) + 1 ⟹ ∃ zs. add-nat xs ys = zs @ [True]*

We will not describe all runtime proofs in detail. However, we will look at one specific runtime formalization as a typical example, namely that of the *inc-nat* function (see Figure 3.1).

inc-nat	*inc-nat-tm*
inc-nat [] = [True]	*inc-nat-tm [] =1 return [True]*
inc-nat (False # xs) = True # xs	*inc-nat-tm (False # xs) =1 return (True # xs)*
inc-nat (True # xs) = False # (inc-nat xs)	*inc-nat-tm (True # xs) =1 do {*
	r ← inc-nat-tm xs;
	return (False # r)
	}

Figure 3.1 Comparison of the *inc-nat* function with its time monad version

The correctness of the monad version can be shown by induction and *simp*:

```
lemma val-inc-nat-tm[simp]: val (inc-nat-tm xs) = inc-nat xs
  by (induction xs rule: inc-nat-tm.induct) simp-all
```

The runtime can now be shown to be linearly bounded:

```
lemma time-inc-nat-tm-le: time (inc-nat-tm xs) ≤ length xs + 1
  by (induction xs rule: inc-nat-tm.induct) simp-all
```

In general, if we have a runtime monad version *f-tm* of some function *f*, the correctness proof (which we will always name *val-f-tm*) can be done with very little manual work. The runtime bound, however, of course will require more effort if f is more complicated.

Let us conclude this section with the following lemma.

Lemma 3.2 [*time-add-nat-tm-le*]. *Adding two numbers represented in nat-lsbf can be done in linear time. More explicitly: The runtime of adding xs and ys is at most 2 ∗ max (length xs) (length ys) + 3.*

3.2 Truncating and Filling

In this section, we will look at two functions that switch between different binary representations of the same natural number. The first function is the *truncate*-function, which deletes all trailing zeros:

fun *truncate-reversed* :: *bool list* ⇒ *bool list* **where**
truncate-reversed [] = []
| *truncate-reversed* (*x#xs*) = (*if x then x#xs else truncate-reversed xs*)

definition *truncate* :: *nat-lsbf* ⇒ *nat-lsbf* **where**
truncate xs = *rev* (*truncate-reversed* (*rev xs*))

We call some *x* of type *nat-lsbf* **truncated** if applying *truncate* to it has no effect:

abbreviation *truncated* **where** *truncated x* ≡ *truncate x* = *x*

The section `truncating and filling` of `Binary_Representations/Binary_Representations` now contains a large number of lemmas about properties of the *truncate* function that are intuitively clear, which is why we will omit most of them here. Just to give a few examples, we have:

lemma *truncate-length-ineq*: *length* (*truncate xs*) ≤ *length xs*
lemma *truncated-iff*: *truncated x* ⟷ (*x* = [] ∨ *last x* = *True*)
lemma *truncate-as-take*: ⋀*xs*. ∃ *n*. *truncate xs* = *take n xs*
lemma *to-nat-eq-imp-truncate-eq*: *to-nat xs* = *to-nat ys* ⟹ *truncate xs* = *truncate ys*
lemma *truncate-and-length-eq-imp-eq*:
  assumes *truncate xs* = *truncate ys length xs* = *length ys*
  shows *xs* = *ys*

Contrary to deleting trailing zeros, we may also append trailing zeros in order to get a different representation. This is done with the *fill* function:

definition *fill* **where** *fill n xs* = *xs* @ *replicate* (*n* − *length xs*) *False*

The *fill* function satisfies:

lemma *to-nat-fill*[*simp*]: *to-nat* (*fill n xs*) = *to-nat xs*
lemma *length-fill'*: *length* (*fill n xs*) = *max n* (*length xs*)
lemma *fill-take-com*: *fill k* (*take k xs*) = *take k* (*fill k xs*)

3.3 Comparison and Subtraction

In order to implement comparison on *nat-lsbf*, we define the following auxiliary functions:

- *compare-nat-same-length-reversed* takes two *bool lists* of the same length as argument and compares them lexicographically:

  fun *compare-nat-same-length-reversed* :: *bool list* ⇒ *bool list* ⇒ *bool* **where**
  compare-nat-same-length-reversed [] [] = *True*
  | *compare-nat-same-length-reversed* (*False#xs*) (*False#ys*) = *compare-nat-same-length-reversed xs ys*
  | *compare-nat-same-length-reversed* (*True#xs*) (*False#ys*) = *False*
  | *compare-nat-same-length-reversed* (*False#xs*) (*True#ys*) = *True*
  | *compare-nat-same-length-reversed* (*True#xs*) (*True#ys*) = *compare-nat-same-length-reversed xs ys*
  | *compare-nat-same-length-reversed* - - = *undefined*

- If we have two numbers in the LSBF encoding with equal length, we can reverse them and then compare them with the lexicographic order:

  fun *compare-nat-same-length* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *bool* **where**
  compare-nat-same-length xs ys = *compare-nat-same-length-reversed* (*rev xs*) (*rev ys*)

- In order to compare any two numbers in LSBF encoding, we append zeros to the shorter number, making both numbers have equal length:

  definition *make-same-length* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* × *nat-lsbf* **where**
  make-same-length xs ys = (*let n* = *max* (*length xs*) (*length ys*) *in* ((*fill n xs*), (*fill n ys*)))

Finally, we define a comparison function on *nat-lsbf*:

definition *compare-nat* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *bool* **where**
compare-nat xs ys = (*let* (*fill-xs*, *fill-ys*) = *make-same-length xs ys in compare-nat-same-length fill-xs fill-ys*)

Lemma 3.3 [*compare-nat-correct*]. *The definition of compare-nat is correct, i.e. compare-nat xs ys* = (*to-nat xs* ≤ *to-nat ys*).

Lemma 3.4 [*time-compare-nat-tm-le*]. *Comparing two numbers of type nat-lsbf can be done in linear time.*

Now, we are ready to define subtraction on *nat-lsbf*:

definition *subtract-nat* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**
subtract-nat xs ys = (*if compare-nat xs ys then* [] *else*
let (*fill-xs*, *fill-ys*) = *make-same-length xs ys in*
inc-nat (*butlast* (*add-nat fill-xs* (*map Not fill-ys*))))

Lemma 3.5 [*subtract-nat-correct*]. *The definition of subtract-nat is correct, i.e. subtract-nat xs ys* = (*to-nat xs*) − (*to-nat ys*).

If the validity of Lemma 3.5 is not clear to the reader, there is also an informal proof in Appendix A.

Lemma 3.6 [*time-subtract-nat-tm-le*]. *Subtracting two numbers of type nat-lsbf can be done in linear time.*

3.4 Multiplying/Dividing by Powers of 2

Dividing some number represented in *nat-lsbf* by a power 2^k can simply be done by dropping the first k bits.[3] In order to multiply by a power 2^k, we can just shift the number k digits to the right:

definition *shift-right* :: *nat* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**
shift-right n xs = (*replicate n False*) @ *xs*

Lemma 3.7 [*to-nat-shift-right*]. *Applying shift-right n to xs effectively multiplies xs by* 2^n, *i.e. to-nat* (*shift-right n xs*) = *2 ^ n* ∗ *to-nat xs.*

Lemma 3.8 [*time-drop-tm*, *time-shift-right-tm*]. *Multiplying or dividing a number of type nat-lsbf by a power of* 2 *can be done in linear time.*

3.5 Subdividing Lists

For the Karatsuba-Algorithm, we will need to split the binary representation of a number in two halves. For that, we define the following functions:

[3]See the Isabelle lemma *to-nat-drop*.

fun *split-at* :: *nat* ⇒ *'a list* ⇒ *'a list* × *'a list* **where**
  split-at m xs = (*take m xs*, *drop m xs*)

definition *split* :: *nat-lsbf* ⇒ *nat-lsbf* × *nat-lsbf* **where**
  split xs = (*let n* = *length xs div* (*2*::*nat*) *in split-at n xs*)

For the implementation of the Schönhage-Strassen-Algorithm, we also need to split binary representations into multiple blocks of the same size. Therefore, we define a function *subdivide* that, given some $n > 0$ and some list *xs*, splits *xs* into blocks of size *n*:

fun *subdivide* :: *nat* ⇒ *'a list* ⇒ *'a list list* **where**
subdivide 0 xs = *undefined*
| *subdivide n* [] = []
| *subdivide n xs* = *take n xs* # *subdivide n* (*drop n xs*)

For example, we have:

subdivide 2 [*0*..<*6*] = [[*0*, *1*], [*2*, *3*], [*4*, *5*]]
subdivide 3 [*0*..<*6*] = [[*0*, *1*, *2*], [*3*, *4*, *5*]]

The number represented by the subdivided list is related to the numbers represented by the blocks as follows:

Lemma 3.9 [*to-nat-subdivide*]. *Assume* $n > 0$ *and length xs* = $n \cdot k$. *Then,*

$$\textit{to-nat xs} = \sum_{i=0}^{k-1} \textit{to-nat}\ (\textit{subdivide n xs}\ !\ i) \cdot 2^{i \cdot n}.$$

3.6 The *bitsize* Function

We define a function *bitsize*, which calculates the number of bits needed in order to represent some number of type *nat* in *nat-lsbf*.

fun *bitsize* :: *nat* ⇒ *nat* **where**
bitsize 0 = *0*
| *bitsize n* = *1* + *bitsize* (*n div 2*)

Lemma 3.10 [*bitsize-eq*]. *The function bitsize is correct, i.e. bitsize n* = *length* (*from-nat n*).

Other lemmas can be found in `Binary_Representations/Binary_Representations` in the section `the bitsize function` and include:

lemma *bitsize-length*: *bitsize n* ≤ *k* ⟷ *n* < *2* ^ *k*
lemma *bitsize-mono*: *n1* ≤ *n2* ⟹ *bitsize n1* ≤ *bitsize n2*

3.6.1 The *next-power-of-2* Function

Using the *bitsize* function, we can define another auxiliary function that, when applied to some $n \in \mathbb{N}$, returns the smallest power 2^k s.t. $n \leq 2^k$:

fun *is-power-of-2* :: *nat* ⇒ *bool* **where**
is-power-of-2 0 = *False*
| *is-power-of-2* (*Suc 0*) = *True*
| *is-power-of-2 n* = ((*n mod 2* = *0*) ∧ *is-power-of-2* (*n div 2*))

fun *next-power-of-2* :: *nat* ⇒ *nat* **where**
next-power-of-2 n = (*if is-power-of-2 n then n else 2* ^ (*bit-size n*))

3.7 Grid Multiplication

The ”usual” multiplication algorithm, also called **grid multiplication**, multiplies two numbers $x, y \in \mathbb{N}$ given in binary representation by consecutively adding copies of y for every 1 appearing in the representation of x (see Figure 3.2).

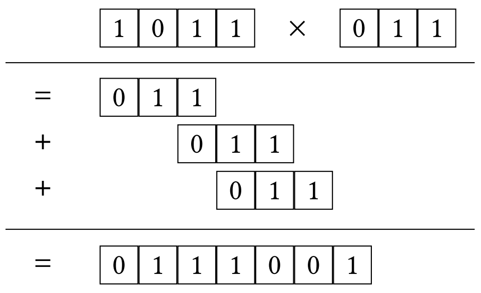


Figure 3.2 Visualization of grid multiplication.

Using our type *nat-lsbf*, this procedure can be implemented as follows:

```
fun grid-mul-nat :: nat-lsbf ⇒ nat-lsbf ⇒ nat-lsbf where
grid-mul-nat [] - = []
| grid-mul-nat (False#xs) y = False # (grid-mul-nat xs y)
| grid-mul-nat (True#xs) y = add-nat (False # (grid-mul-nat xs y)) y
```

Lemma 3.11 [*grid-mul-nat-correct*]. *The function grid-mul-nat is correct, i.e. to-nat (grid-mul-nat x y) = to-nat x ∗ to-nat y.*

Lemma 3.12 [*time-grid-mul-nat-tm-le*]. *The implementation of grid-mul-nat has quadratic runtime. More precisely: If xs has length n and ys has length m, then the runtime of grid-mul-nat xs ys is in* $O(n \cdot \max\{n, m\})$.

4 The Karatsuba-Algorithm

The Karatsuba-Algorithm [KO62] is a simple recursive multiplication algorithm that achieves a runtime of $\mathcal{O}\left(n^{\log_2 3}\right)$ bit operations and hence is an improvement compared to grid multiplication. Let us first give an informal explanation of the algorithm. Assume $x, y \in \mathbb{N}$ both have length 2^{k+1} in binary representation. Divide both into smaller blocks x_0, x_1, y_0, y_1 of length 2^k s.t. $x = x_0 + x_1 \cdot 2^k$ and $y = y_0 + y_1 \cdot 2^k$. Then, we have

$$x \cdot y = (x_0 + x_1 \cdot 2^k) \cdot (y_0 + y_1 \cdot 2^k) = x_0 \cdot y_0 + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^k + x_1 \cdot y_1 \cdot 2^{2k}.$$

Since multiplications by powers of 2 and addition can both be done in linear time, the time needed in order to evaluate the right hand side (RHS) of the equation is essentially the time needed for the 4 multiplications of the smaller blocks. The main ingredient of the Karatsuba-Algorithm is the observation that the term $x_1 \cdot y_0 + x_0 \cdot y_1$ can be replaced by a term that makes use of the results of $x_0 \cdot y_0$ and $x_1 \cdot y_1$, needing only 1 additional multiplication of small blocks:

$$x_1 \cdot y_0 + x_0 \cdot y_1 = x_0 \cdot y_0 + x_1 \cdot y_1 - (x_0 - x_1) \cdot (y_0 - y_1)$$

Since e.g. $x_0 - x_1$ might be negative, however, and negative numbers can not be represented in the type *nat-lsbf*, we need to be careful when calculating this term. The function *subtract-nat* defined in Section 3.3 returns 0 if the result in $\mathbb{Z}$ would be negative (which is consistent with the definition of subtraction on the type *nat*).

We begin by defining an auxiliary function that, given some numbers a, b, calculates $|a - b|$:

definition *abs-diff* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**
abs-diff x y = *add-nat* (*subtract-nat x y*) (*subtract-nat y x*)

Using this function, we will first calculate $|x_0 - x_1| \cdot |y_0 - y_1|$ and then, depending on the sign of $(x_0 - x_1) \cdot (y_0 - y_1)$, add to or subtract from $x_0 \cdot y_0 + x_1 \cdot y_1$:

$$\begin{aligned} & x_0 \cdot y_0 + x_1 \cdot y_1 - (x_0 - x_1) \cdot (y_0 - y_1) \\ = & \begin{cases} x_0 \cdot y_0 + x_1 \cdot y_1 - |x_0 - x_1| \cdot |y_0 - y_1| & \text{if } (x_0 \le x_1) \longleftrightarrow (y_0 \le y_1) \\ x_0 \cdot y_0 + x_1 \cdot y_1 + |x_0 - x_1| \cdot |y_0 - y_1| & \text{else} \end{cases} \end{aligned}$$

Example 4.1. *Consider the LSBF representations $x = 0111$ and $y = 1101$ of length 4. We start by building the smaller blocks:*

$$x = \underbrace{\boxed{1}\boxed{0}}_{x_0}\underbrace{\boxed{1}\boxed{1}}_{x_1} \qquad y = \underbrace{\boxed{1}\boxed{1}}_{y_0}\underbrace{\boxed{0}\boxed{1}}_{y_1}$$

Next, we calculate the absolute differences:

$$|x_0 - x_1| = \boxed{0}\boxed{1} \qquad |y_0 - y_1| = \boxed{1}\boxed{0}$$

Now, we perform the three necessary multiplications recursively or simply by grid multiplication:

$$x_0 \cdot y_0 = \boxed{1}\boxed{1}$$

$$x_1 \cdot y_1 = \boxed{0}\boxed{1}\boxed{1}$$

$$|x_0 - x_1| \cdot |y_0 - y_1| = \boxed{0}\boxed{1}$$

Since $x_0 \leq x_1$, but $y_0 > y_1$, we need to add the three results in order to obtain $x_0 \cdot y_0 + x_1 \cdot y_1 - (x_0 - x_1) \cdot (y_0 - y_1)$:

$$\begin{array}{ccccc} & \boxed{1} & \boxed{1} & & \\ + & \boxed{0} & \boxed{1} & \boxed{1} & \\ + & \boxed{0} & \boxed{1} & & \\ \hline = & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} \end{array}$$

Finally, we can combine this result with the results $x_0 \cdot y_0$ and $x_1 \cdot y_1$ to the term $x_0 \cdot y_0 + (x_0 \cdot y_0 + x_1 \cdot y_1 - (x_0 - x_1) \cdot (y_0 - y_1)) \cdot 2^2 + x_1 \cdot y_1 \cdot 2^4 = x \cdot y$:

$$\begin{array}{ccccccccc} & \boxed{1} & \boxed{1} & & & & & & \\ + & & & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} & & \\ + & & & & & \boxed{0} & \boxed{1} & \boxed{1} & \\ \hline = & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{1} \end{array}$$

The assumption that the length of the input numbers x, y is some power 2^k is needed to assure that not only x and y are of the same length, but also the input numbers of each recursive call have matching lengths. Our first version of the algorithm now looks like this:

```
fun karatsuba-on-power-of-2-length :: nat ⇒ nat-lsbf ⇒ nat-lsbf ⇒ nat-lsbf where
karatsuba-on-power-of-2-length k x y =
(if k ≤ karatsuba-lower-bound
then grid-mul-nat x y
else let
   (x0, x1) = split x;
   (y0, y1) = split y;
   k-div-2 = (k div 2);
   prod0 = karatsuba-on-power-of-2-length k-div-2 x0 y0;
   prod1 = karatsuba-on-power-of-2-length k-div-2 x1 y1;
   prod2 = karatsuba-on-power-of-2-length k-div-2
     (fill k-div-2 (abs-diff x0 x1))
     (fill k-div-2 (abs-diff y0 y1));
   add01 = add-nat prod0 prod1;
   r = (if (compare-nat x1 x0) = (compare-nat y1 y0)
      then subtract-nat add01 prod2
      else add-nat add01 prod2)
  in
  add-nat
   (add-nat prod0 (shift-right k-div-2 r))
   (shift-right k prod1) )
```

The first argument k is the length of the two numbers x and y. *karatsuba-lower-bound* is any constant in $\mathbb{N}_{\geq 1}$. We get the following correctness result:

Lemma 4.2 [*karatsuba-on-power-of-2-length-correct*]. *Assume $k = 2^l$, length $x = k$ and length $y = k$. Then, to-nat (karatsuba-on-power-of-2-length k x y) = to-nat x $*$ to-nat y.*

Now, if $x, y \in \mathbb{N}$ are given in binary representation without any assumption on their length, we can just append trailing zeros to satisfy the assumptions of Lemma 4.2. Using our *next-power-of-2* function from section 3.6.1 , we define:

```
fun karatsuba-mul-nat :: nat-lsbf ⇒ nat-lsbf ⇒ nat-lsbf where
karatsuba-mul-nat x y = (let k = next-power-of-2 (max (length x) (length y)) in
  karatsuba-on-power-of-2-length k (fill k x) (fill k y))
```

The correctness theorem is now an easy corollary from Lemma 4.2:

Lemma 4.3 [*karatsuba-mul-nat-correct*]. *The function karatsuba-mul-nat is correct, i.e.*

$$\textit{to-nat}\ (\textit{karatsuba-mul-nat}\ x\ y) = \textit{to-nat}\ x * \textit{to-nat}\ y.$$

The runtime proof is a bit more complicated:

Lemma 4.4 [*time-karatsuba-mul-nat-tm-le, time-karatsuba-mul-nat-bound-bigo*]. *Given binary representations xs and ys, karatsuba-mul-nat xs ys multiplies xs and ys in a runtime of* $O\left(m^{\log_2 3}\right)$*, where* $m = \max\{\textit{length xs}, \textit{length ys}\}$.

Proof. The proof proceeds in the following steps:

1. Show a runtime bound for the auxiliary function *karatsuba-on-power-of-2-length*:
   (i) Define a recursive function $h : \mathbb{N} \to \mathbb{N}$ according to the structure of *karatsuba-on-power-of-2-length*:

$$h(k) := \begin{cases} O\left(k^2\right) & \text{if } k \leq \textit{karatsuba-lower-bound} \\ O\left(k\right) + 3 \cdot h(k/2) & \text{else.} \end{cases}$$

   (ii) Show that h indeed is a runtime bound.[1]
   (iii) Show that $h \in O\left(k^{\log_2 3}\right)$. This is done by defining a variant $\textit{h-real} : \mathbb{N} \to \mathbb{R}$, for which the *master-theorem* tactic of the Akra-Bazzi theory [Ebe15] can be applied.[2]
2. Use that bound to obtain a statement about the runtime of *karatsuba-mul-nat*:
   (i) Define a function g[3] $: \mathbb{N} \to \mathbb{N}$ that upper-bounds the runtime of *karatsuba-mul-nat*. In particular, $g(m)$ should bound the runtime of our Karatsuba implementation on inputs with maximum length m. Due to our conservative assumptions on the runtime of functions on *nat* and according to the definition of *karatsuba-mul-nat*, this function is chosen as

$$g(m) := O\left(k\right) + h(k),$$

   where k is the result of *next-power-of-2 m* (see section 3.6.1).
   (ii) Show that g indeed is a runtime bound of *karatsuba-mul-nat*.[4]
   (iii) Show that $g \in O\left(m^{\log_2 3}\right)$. An informal[5] proof is as follows: Since k is the *smallest* power of 2 which is larger than m, we have $k \leq 2 \cdot m$, i.e. $k =: k(m) \in O\left(m\right)$. Moreover, by 1. (iii) we have $O\left(k\right) + h(k) = O\left(k^{\log_2 3}\right)$. Lemma 4.5 now shows that indeed $g(m) \in O\left(m^{\log_2 3}\right)$.

□

Lemma 4.5 [*powr-bigo-linear-index-transformation*]. *Assume* $i : \mathbb{N} \to \mathbb{N}$ *is linearly bounded, i.e.* $i(n) \in O\left(n\right)$*, and* $f \in O\left(n^p\right)$ *with* $p > 0$ *and* $f : \mathbb{N} \to \mathbb{R}$*. Then,* $f(i(n)) \in O\left(n^p\right)$.

Proof. By assumption, there exist constants $c_1, c_2 > 0$ and $N_1, N_2 \in \mathbb{N}$ s.t.

$$i(n) \leq c_1 \cdot n, \qquad (n \geq N_1) \qquad (4.1)$$

$$|f(n)| \leq c_2 \cdot n^p. \qquad (n \geq N_2) \qquad (4.2)$$

Hence, if $n \geq N_1$ and $i(n) \geq N_2$, we have

$$|f(i(n))| \overset{(4.2)}{\leq} c_1 \cdot i(n)^p \overset{(4.1)}{\leq} \underbrace{c_1 \cdot c_2^p}_{=:c} \cdot n^p.$$

[1]This is done in the Isabelle lemma *time-karatsuba-on-power-of-2-length-tm-le-h*.
[2]See the Isabelle lemma *h-real-bigo*.
[3]This function is called *time-karatsuba-mul-nat-bound* in the Isabelle code.
[4]This is done in the Isabelle lemma *time-karatsuba-mul-nat-tm-le*.
[5]Formally, this is done in the Isabelle lemma *time-karatsuba-mul-nat-bound-bigo*.

Since there are only finitely many $j \leq N_2$, we can define

$$c_f := \max_{j \leq N_2} |f(j)|.$$

Hence, for all $n \in \mathbb{N}$ with $i(n) \leq N_2$, we have

$$|f(i(n))| \leq c_f,$$

and so we can conclude that for all $n \geq N_1$, we have

$$|f(i(n))| \leq c_f + c \cdot n^p.$$

Since $\mathcal{O}\left(c_f + c \cdot n^p\right) = \mathcal{O}\left(n^p\right)$, we are done. $\square$

5 Number Theoretic Transforms

In this chapter, let $(R, +, \cdot, 0, 1)$ be a commutative ring, and $R^\times := \{ x \in R \mid \exists y \in R : x \cdot y = 1 \}$ be the unit group of R.

5.1 Number Theoretic Transforms

Definition 5.1 [*root-of-unity-def, primitive-root-def*]. *Let $n > 0$. An element $\mu \in R$ is called*

(a) ***(n-th) root of unity*** *if $\mu^n = 1$*

(b) ***(n-th) primitive root*** *if $\mu^n = 1$ and $\mu^i \neq 1$ for all $i \in \{1..{<}n\}$.*

In a context where n is fixed, we define the ***group of roots of unity*** *$R^1 := \{ \mu \in R \mid \mu \text{ is an } n\text{-th root of unity} \}$.*

Lemma 5.2 [*roots-of-unity-group-is-group*]. *Let $n > 0$. Then, $(R^1, \cdot, 1)$ is a group. In particular, if μ is a root of unity, so is μ^i for any $i \in \mathbb{Z}$.*

Proof. $1^n = 1$, so $1 \in R^1$. Moreover, for $x, y \in R^1$, we have $(x \cdot y)^n = x^n \cdot y^n = 1$, so $x \cdot y \in R^1$. Hence, R^1 is a submonoid of the multiplicative monoid R. Now, for any $x \in R^1$, we have $x \cdot x^{n-1} = x^n = 1$, i.e. x has an inverse in R^1. □

Lemma 5.3 [*primitive-root-inv, primitive-root-recursion*]. *Assume μ is an n-th primitive root. Then:*

(a) μ^{-1} is an n-th primitive root.

(b) If $n = 2k$, then μ^2 is a k-th primitive root.

Proof. (a) According to Lemma 5.2, μ^{-1} is a root of unity. Now, let $i \in \{1..{<}n\}$ and assume $\left(\mu^{-1}\right)^i = 1$. Then, also $1 = \mu^n \cdot \mu^{-i} = \mu^{n-i}$. But since $n - i \in \{1..{<}n\}$, this contradicts the assumption that μ is an n-th primitive root.

(b) Obviously, we have $\left(\mu^2\right)^k = \mu^n = 1$. Moreover, let $i \in \{1..{<}k\}$. Then, $\left(\mu^2\right)^i = \mu^{2i}$, and since $2i \in \{1..{<}n\}$, it follows that $\mu^{2i} \neq 1$.

□

Definition 5.4 [*NTT-def, cyclic-convolution-def*]. *Let $\mu \in R$, and $a = (a_0, \ldots, a_{n-1}) \in R^n$. The* ***Number Theoretic Transform (NTT)*** *of a w.r.t. μ is defined as*

$$\mathrm{NTT}_\mu(a)_i := \sum_{j=0}^{n-1} a_j \cdot \left(\mu^i\right)^j \qquad (i \in \{0, \ldots, n-1\}).$$

For another vector $b = (b_0, \ldots, b_{n-1}) \in R^n$, we define the ***cyclic convolution*** *$a \star b \in R^n$ by*

$$(a \star b)_i := \sum_{j=0}^{n-1} \sum_{\substack{k=0 \\ j+k \equiv_n i}}^{n-1} a_j \cdot b_k = \sum_{j=0}^{n-1} a_j \cdot b_{(i-j) \bmod n} \qquad (i \in \{0, \ldots, n-1\}).$$

As it turns out, the classic convolution rule for discrete fourier transforms carries over to NTTs, as long as the NTT is done w.r.t a root of unity.

Lemma 5.5 [*root-of-unity-power-sum-product*]. *Assume μ is an n-th root of unity. Then*

$$\left(\sum_{i=0}^{n-1} a_i \cdot \mu^i\right) \cdot \left(\sum_{j=0}^{n-1} b_j \cdot \mu^j\right) = \sum_{k=0}^{n-1} \sum_{i=0}^{n-1} a_i \cdot b_{(n+k-i) \bmod n} \cdot \mu^k.$$

Proof. Because μ is an n-th root of unity, we have $\mu^i = \mu^j$ for all $i, j \in \mathbb{Z}$ with $i \equiv_n j$. Moreover, the map $j \mapsto (n + j - i) \bmod n$ defines a permutation of $\{0, \dots, n-1\}$ for any $i \in \{0, \dots, n-1\}$. Thus:

$$\begin{aligned}
\left(\sum_{i=0}^{n-1} a_i \cdot \mu^i\right) \cdot \left(\sum_{j=0}^{n-1} b_j \cdot \mu^j\right) &= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i \cdot b_j \cdot \mu^{i+j} \\
&= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i \cdot b_{(n+j-i) \bmod n} \cdot \mu^{i+(n+j-i) \bmod n} && \text{(index permutation)} \\
&= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i \cdot b_{(n+j-i) \bmod n} \cdot \mu^{j} && (i + (n+j-i) \bmod n \equiv_n j.)
\end{aligned}$$

□

Theorem 5.6 (Convolution rule) [*convolution-rule*]. *Let $a = (a_0, \dots, a_{n-1}), b = (b_0, \dots, b_{n-1}) \in R^n$ and assume μ is an n-th root of unity. Then, for all $i \in \{0, \dots, n-1\}$, we have*

$$\mathrm{NTT}_\mu(a)_i \cdot \mathrm{NTT}_\mu(b)_i = NTT_\mu(a \star b)_i.$$

Proof. By Lemma 5.2, μ^i is also an n-th root of unity. Hence, we have

$$\begin{aligned}
\mathrm{NTT}_\mu(a)_i \cdot \mathrm{NTT}_\mu(b)_i &= \left(\sum_{j=0}^{n-1} a_j \cdot \left(\mu^i\right)^j\right) \cdot \left(\sum_{k=0}^{n-1} b_k \cdot \left(\mu^i\right)^k\right) \\
&= \sum_{k=0}^{n-1}\sum_{j=0}^{n-1} a_j \cdot b_{(n+k-j) \bmod n} \cdot \left(\mu^i\right)^k && \text{(Lemma 5.5)} \\
&= \sum_{k=0}^{n-1} (a \star b)_k \cdot \left(\mu^i\right)^k \\
&= \mathrm{NTT}_\mu(a \star b)_i
\end{aligned}$$

□

Other results like the inversion rule, however, need additional assumptions. This motivates our next definition.

Definition 5.7. *Let μ be an n-th primitive root.*

*(a) μ is called **good** if $\forall i \in \{1..<n\} : \sum_{j=0}^{n-1} \left(\mu^i\right)^j = 0$.*

*(b) If $n = 2k$, μ is said to satisfy the **halfway property** if $\mu^k = -1$.*

Lemma 5.8 [*inv-good, inv-halfway-property*]. *Let μ be an n-th primitive root.*

(a) If μ is good, so is μ^{-1}.

(b) If μ satisfies the halfway property, so does μ^{-1}.

Proof. (a) Assume μ is good and let $i \in \{1..<n\}$. Then $n - i \in \{1..<n\}$ and hence, since $\mu^n = 1$,

$$0 = \sum_{j=0}^{n-1} \left(\mu^{n-i}\right)^j = \sum_{j=0}^{n-1} \left(\mu^{-i}\right)^j.$$

(b) If $\mu^k = -1$, then $\left(\mu^{-1}\right)^k = \left(\mu^k\right)^{-1} = (-1)^{-1} = -1$.

□

Before we can state the inversion rule, we still need one more thing. Note that there exists a unique ring homomorphism $\varphi : \mathbb{Z} \to R$, given by $\varphi(0) := 0$, $\varphi(n+1) := \varphi(n) + 1$ and $\varphi(-n) := -\varphi(n)$ $(n \geq 0)$. For simplicity, we will just write $n \in R$ instead of $\varphi(n) \in R$.

Theorem 5.9 (Inversion rule) [*inversion-rule*]. *Let μ be a good n-th primitive root and $a = (a_0, \ldots, a_{n-1}) \in R^n$. Then*

$$\mathrm{NTT}_{\mu^{-1}}\left(\mathrm{NTT}_\mu(a)\right) = n \cdot a,$$

where $n \cdot a = (n \cdot a_0, \ldots, n \cdot a_{n-1})$.

Proof. Let $i \in \{0, \ldots, n-1\}$. Then,

$$\begin{aligned}\mathrm{NTT}_{\mu^{-1}}\left(\mathrm{NTT}_\mu(a)\right)_i &= \sum_{j=0}^{n-1} \mathrm{NTT}_\mu(a)_j \cdot \left(\mu^{-i}\right)^j \\ &= \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} a_k \cdot \left(\mu^j\right)^k \cdot \left(\mu^{-i}\right)^j \\ &= \sum_{k=0}^{n-1}\left(a_k \cdot \sum_{j=0}^{n-1}\left(\mu^{k-i}\right)^j\right).\end{aligned}$$

In order to continue, let $k \in \{0, \ldots, n-1\}$ and consider three cases:
Case 1: $k = i$. Then, $k - i = 0$, and hence

$$\sum_{j=0}^{n-1}\left(\mu^{k-i}\right)^j = \sum_{j=0}^{n-1} 1 = n.$$

Case 2: $k > i$. Then $k - i \in \{1..{<}n\}$, and hence, since μ was assumed to be good,

$$\sum_{j=0}^{n-1}\left(\mu^{k-i}\right)^j = 0.$$

Case 3: $k < i$. Then $i - k \in \{1..{<}n\}$. Moreover, μ^{-1} is good because of Lemma 5.8 (a). Hence,

$$\sum_{j=0}^{n-1}\left(\mu^{k-i}\right)^j = \sum_{j=0}^{n-1}\left(\left(\mu^{-1}\right)^{i-k}\right)^j = 0.$$

So, we can continue our calculation as follows:

$$\begin{aligned}\sum_{k=0}^{n-1}\left(a_k \cdot \sum_{j=0}^{n-1}\left(\mu^{k-i}\right)^j\right) &= \sum_{k=0}^{n-1}\left(a_k \cdot n \cdot \delta_{i,k}\right) \\ &= n \cdot a_i.\end{aligned}$$

□

The following lemma now gives sufficient conditions which assure μ is a good primitive root in all situations that we will consider. Note that, in particular, the lemma implies that if R is a field, any primitive root is good.

Lemma 5.10 [*sufficiently-good*]. *Let μ be an n-th primitive root and assume*
(a) R is an integral domain or
(b) $n = 2^k$ for some $k > 0$ and μ satisfies the halfway property.
Then, μ is good.

Proof. Let $i \in \{1..{<}n\}$. We show $\sum_{j=0}^{n-1}\left(\mu^i\right)^j = 0$ in either case.

(a) Note that, since μ is a primitive root, $\mu^i \neq 1$, i.e. $1 - \mu^i \neq 0$. Since $\mu^n = 1$, we have $\left(\mu^i\right)^n = 1$, and hence

$$0 = 1 - \left(\mu^i\right)^n = \left(1 - \mu^i\right) \cdot \left(\sum_{j=0}^{n-1}\left(\mu^i\right)^j\right)$$

using Lemma 2.1 Since R is an integral domain and $1 - \mu^i \neq 0$, this shows $\sum_{j=0}^{n-1}\left(\mu^i\right)^j = 0$.

(b) Write $i = r \cdot 2^l$ with r odd. We show the claim by induction on l[1].
Case $l = 0$: We have

$$\begin{aligned}\sum_{j=0}^{n-1} (\mu^r)^j &= \sum_{j=0}^{2^{k-1}-1} (\mu^r)^j + \sum_{j=0}^{2^{k-1}-1} (\mu^r)^{2^{k-1}+j} \\ &= \sum_{j=0}^{2^{k-1}-1} \left((\mu^r)^j + (\mu^r)^{2^{k-1}+j}\right).\end{aligned}$$

Since μ satisfies the halfway property, i.e. $\mu^{2^{k-1}} = -1$, we have

$$(\mu^r)^{2^{k-1}+j} = \left(\mu^{2^{k-1}}\right)^r \cdot (\mu^r)^j = (-1)^r \cdot (\mu^r)^j \overset{r \text{ odd}}{=} -(\mu^r)^j .$$

In particular, the above sum evaluates to 0.
Case $l > 0$: Note that

$$\sum_{j=0}^{n-1} \left(\mu^{r \cdot 2^l}\right)^j = \sum_{j=0}^{2^{k-1}-1} \left((\mu^2)^{r \cdot 2^{l-1}}\right)^j + \sum_{j=0}^{2^{k-1}-1} \left((\mu^2)^{r \cdot 2^{l-1}}\right)^{2^{k-1}+j}$$

and

$$\left((\mu^2)^{r \cdot 2^{l-1}}\right)^{2^{k-1}+j} = \left(\mu^{2^k}\right)^{r \cdot 2^{l-1}} \cdot \left((\mu^2)^{r \cdot 2^{l-1}}\right)^j = \left((\mu^2)^{r \cdot 2^{l-1}}\right)^j$$

since $\mu^{2^k} = 1$, so

$$\sum_{j=0}^{n-1} \left(\mu^{r \cdot 2^l}\right)^j = 2 \sum_{j=0}^{2^{k-1}-1} \left((\mu^2)^{r \cdot 2^{l-1}}\right)^j .$$

Moreover, μ^2 is a 2^{k-1}-th primitive root by Lemma 5.3 (b). In order to use our induction hypothesis (concluding that the sum is 0), we need to show that $k - 1 > 0$, μ^2 satisfies the halfway property and that $r \cdot 2^{l-1} \in \{1..<2^{k-1}\}$.
First, assume for contradiction that $k - 1 = 0$. Then, $n = 2^k = 2$, and since $i = r \cdot 2^l < n$, it follows that $r \cdot 2^l < 2$. But $l > 0$ and r is odd, a contradiction.
So, indeed $k - 1 > 0$. Moreover,

$$(\mu^2)^{2^{k-2}} = \mu^{2^{k-1}} = -1,$$

i.e. μ^2 satisfies the halfway property. Finally, $1 \le r \cdot 2^{l-1}$ (since r is odd) and $r \cdot 2^{l-1} = \frac{1}{2} \cdot r \cdot 2^l < \frac{1}{2} \cdot n = 2^{k-1}$.

□

5.2 Fast Number Theoretic Transforms

If we calculate the NTT via its definition $\mathrm{NTT}_\mu(a)_i = \sum_{j=0}^{n-1} a_j \cdot (\mu^i)^j$, we need to iterate over a for every index $i < n$. This results in a quadratic runtime in n (measured in ring operations). The Fast Number Theoretic Transform (FNTT) is an algorithm that obtains a better runtime of $O(n \log n)$ ring operations and was already formalized for NTTs in fields by Ammer and Kreuzer [AK22]. We adapt the implementation to our needs, but will not prove runtime bounds in terms of ring operations. Instead, we will later show runtime bounds in terms of bit operations for concrete implementations of the FNTT in our rings of interest. First, let us show the following lemma that lies at the heart of the FNTT.

[1]Note that during the induction, n and μ will change, i.e. we cannot do a simple induction on l in the fixed context of our lemma. For formal details, we refer to the corresponding Isabelle lemma.

Lemma 5.11 [*NTT-recursion-1, NTT-recursion-2*]. *Assume $n = 2k$ is even. Let μ be an n-th primitive root that satisfies the halfway property (i.e. $\mu^k = -1$) and assume $a = (a_0, \dots, a_{n-1}) \in R^n$. Write $a_{even} := (a_0, a_2, \dots, a_{n-2})$ and $a_{odd} := (a_1, a_3, \dots, a_{n-1})$. Then, for any $j < k$, we have*

$$\begin{aligned} \mathrm{NTT}_\mu(a)_j &= \mathrm{NTT}_{\mu^2}(a_{even})_j + \mu^j \cdot \mathrm{NTT}_{\mu^2}(a_{odd})_j \\ \mathrm{NTT}_\mu(a)_{k+j} &= \mathrm{NTT}_{\mu^2}(a_{even})_j - \mu^j \cdot \mathrm{NTT}_{\mu^2}(a_{odd})_j \end{aligned}$$

Proof. Splitting even and odd terms, we obtain

$$\begin{aligned} \mathrm{NTT}_\mu(a)_j &= \sum_{i=0}^{n-1} a_i \cdot \mu^{ij} \\ &= \sum_{i=0}^{k-1} a_{2i} \cdot \mu^{2ij} + \sum_{i=0}^{k-1} a_{2i+1} \cdot \mu^{(2i+1)j} \\ &= \sum_{i=0}^{k-1} a_{2i} \cdot \left(\mu^2\right)^{ij} + \mu^j \cdot \sum_{i=0}^{k-1} a_{2i+1} \cdot \left(\mu^2\right)^{ij} \\ &= \mathrm{NTT}_{\mu^2}(a_{\mathrm{even}})_j + \mu^j \cdot \mathrm{NTT}_{\mu^2}(a_{\mathrm{odd}})_j \end{aligned}$$

and similarly

$$\begin{aligned} \mathrm{NTT}_\mu(a)_{k+j} &= \sum_{i=0}^{n-1} a_i \cdot \mu^{i(k+j)} \\ &= \sum_{i=0}^{k-1} a_{2i} \cdot \mu^{2i(k+j)} + \sum_{i=0}^{k-1} a_{2i+1} \cdot \mu^{(2i+1)(k+j)} \\ &= \sum_{i=0}^{k-1} a_{2i} \cdot \left(\mu^2\right)^{ij} \cdot \underbrace{\left(\mu^{2k}\right)^i}_{=1} + \underbrace{\mu^k}_{=-1} \cdot \mu^j \cdot \sum_{i=0}^{k-1} a_{2i+1} \cdot \left(\mu^2\right)^{ij} \cdot \underbrace{\left(\mu^{2k}\right)^i}_{=1} \\ &= \mathrm{NTT}_{\mu^2}(a_{\mathrm{even}})_j - \mu^j \cdot \mathrm{NTT}_{\mu^2}(a_{\mathrm{odd}})_j . \end{aligned}$$

□

Moreover, the Schönhage-Strassen-Algorithm relies on the following lemma:

Lemma 5.12 [*NTT-diffs*]. *In the situation of Lemma 5.11, we have*

$$\mathrm{NTT}_\mu(a)_j - \mathrm{NTT}_\mu(a)_{k+j} = 2 \cdot \mu^j \cdot \mathrm{NTT}_{\mu^2}(a_{odd})_j .$$

Proof. The equation follows by just inserting the equalities given in Lemma 5.11. □

Lemma 5.11 immediately yields a recursive procedure to calculate the NTT for vectors of a length which is a power of 2. We will write our first version similarly as in [AK22]:

```
fun FNTT :: 'a ⇒ 'a list ⇒ 'a list where
FNTT μ [] = []
| FNTT μ [x] = [x]
| FNTT μ [x, y] = [x ⊕ y, x ⊖ y]
| FNTT μ a = (let n = length a;
        nums1 = [a!i. i ← filter even [0..<n]];
        nums2 = [a!i. i ← filter odd [0..<n]];
        b = FNTT (μ [^] (2::nat)) nums1;
        c = FNTT (μ [^] (2::nat)) nums2;
        g = [b!i ⊕ (μ [^] i) ⊗ c!i. i ← [0..<(n div 2)]];
        h = [b!i ⊖ (μ [^] i) ⊗ c!i. i ← [0..<(n div 2)]]
      in g@h)
```

Theorem 5.13 [*FNTT-NTT*]. *Let $n = 2^k$, μ be an n-th primitive root with $\mu^{2^{k-1}} = -1$, and $a = (a_0, \dots, a_{n-1}) \in R^n$. Then, FNTT μ a = $\mathrm{NTT}_\mu(a)$.*

Proof. The proof is done via induction on μ and a with induction rule *FNTT.induct*. The recursive case then follows from Lemma 5.11. For details, we refer to the corresponding Isabelle proof. □

As described in [AK22], the list comprehensions need to be implemented carefully in order to get the desired runtime. Similarly as in [AK22] we therefore define a second algorithm *FNTT′* that makes use of the *evens-odds* function defined in `Number_Theoretic_Transform.Butterfly` in order to get rid of the first two list comprehensions. In a second step, we define a third algorithm *FNTT″* that rewrites the last two list comprehensions in a way so that the verification of our concrete implementation in the ring $\mathbb{Z}_{F_n}$ (given in section 6.2) becomes a bit easier. We end up with the following algorithm:

```
fun FNTT″ :: 'a ⇒ 'a list ⇒ 'a list where
FNTT″ μ [] = []
| FNTT″ μ [x] = [x]
| FNTT″ μ [x, y] = [x ⊕ y, x ⊖ y]
| FNTT″ μ a = (let n = length a;
                nums1 = evens-odds True a;
                nums2 = evens-odds False a;
                b = FNTT″ (μ [^] (2::nat)) nums1;
                c = FNTT″ (μ [^] (2::nat)) nums2;
                g = map2 (⊕) b (map2 (⊗) [μ [^] i. i ← [0..<(n div 2)]] c);
                h = map2 (λx y. x ⊖ y) b (map2 (⊗) [μ [^] i. i ← [0..<(n div 2)]] c)
             in g@h)
```

Agreement of *FNTT*, *FNTT′* and *FNTT″* is shown in the Isabelle lemmas *FNTT′-FNTT* and *FNTT″-FNTT′*.

6 The Schönhage-Strassen-Algorithm

6.1 Preliminaries

We will use the theory `HOL-Number_Theory.Residues`, which defines residue rings $\mathbb{Z}_n$ by using the carrier set $\{0..n-1\}$ with elements of type *int*.

6.1.1 Representations in $\mathbb{Z}_{2^k}$

For this subsection, fix some $k \in \mathbb{N}_{>0}$ and write $n = 2^k$.

We will represent the elements of $\mathbb{Z}_n$ by boolean lists of length k with the least significant bit coming first (LSBF). For the correctness proofs, we define the encoding/decoding functions *from-residue-ring* and *to-residue-ring*:[1]

definition *to-residue-ring* :: *nat-lsbf* $\Rightarrow$ *int* **where**
to-residue-ring xs = *int* (*to-nat xs*) *mod int n*

definition *from-residue-ring* :: *int* $\Rightarrow$ *nat-lsbf* **where**
from-residue-ring x = *fill k* (*from-nat* (*nat x*))

In order to get from any representation of a number in $\mathbb{N}$ to the representation of its residue modulo 2^k, we further define the function *reduce*, which just takes the k least significant bits (or appends trailing zeros if necessary). In total, we have the following conversion functions:[2]

$$\begin{tikzcd}
\mathbb{Z}_n \triangleq \textit{residue-ring } n \arrow[d, bend right, "\textit{from-residue-ring}"'] & & \mathbb{N} \triangleq \textit{nat} \arrow[ll, "\lambda x.\ \textit{int } x \textit{ mod } n"'] \arrow[d, bend left, "\textit{from-nat}"] \\
\begin{array}{c}\textit{nat-lsbf}\\ (\text{length } k)\end{array} \arrow[u, "\textit{to-residue-ring}"'] & & \textit{nat-lsbf} \arrow[ll, "\textit{reduce}"'] \arrow[u, "\textit{to-nat}"]
\end{tikzcd}$$

Addition can just be transferred from the *nat-lsbf* type using the *reduce* function:

definition *add-mod* **where**
add-mod x y = *reduce* (*add-nat x y*)

However, since $\mathbb{Z}_n$ is a ring, additive inverses always exist. Hence, we do not want to transfer the subtraction from *nat-lsbf*, but define it s.t. it matches the subtraction in $\mathbb{Z}_n$. Considering two elements $x, y \in \mathbb{Z}_n$ with $x \leq y$, we note that $x + 2^k \geq y$ in $\mathbb{N}$ and that $x \equiv x + 2^k \mod n$. So, the subtraction may be defined as follows:

definition *subtract-mod* **where**
subtract-mod xs ys =
  (*if compare-nat xs ys then*

[1]The application of *mod int n* in the definition of *to-residue-ring* is not necessary, but makes some lemmas easier.
[2]This is *not* a commutative diagram! (But it would be one without the *from* functions.)

```
    reduce (subtract-nat ((fill k xs) @ [True]) ys)
  else
    subtract-nat xs ys)
```

Lemma 6.1 [*time-add-mod-tm-le, time-subtract-mod-tm-le*]. *Addition and subtraction in $\mathbb{Z}_{2^k}$ can be done in linear time.*

6.1.2 Representations in $\mathbb{Z}_{F_k}$

For this subsection, fix some $k \in \mathbb{N}$ and let $n := F_k := 2^{2^k} + 1$.

Elements of $\mathbb{Z}_n$ will be represented by boolean lists of length 2^{k+1} in the LSBF encoding. This introduces a non-uniqueness different than the possibility of trailing zeros, since e.g. the binary encodings of 0 and n of length 2^{k+1} both represent the same number $0 \in \mathbb{Z}_n$. This non-uniqueness can be eliminated by yet another function *reduce*, that, given some list x in this non-unique representation, produces a binary encoding of $x \bmod n$. Moreover, we have conversion functions *from-residue-ring* and *to-residue-ring* from and to $\mathbb{Z}_{F_k}$ similar as with $\mathbb{Z}_{2^k}$. Finally, we also implement a function *from-nat-lsbf* that can be used to obtain a non-unique representation of length 2^{k+1} for the residue of any number in LSBF representation modulo n.

Here is an overview of the conversion functions:

$$\begin{tikzcd}
 & \mathbb{Z}_n \triangleq \textit{residue-ring } n \arrow[d, bend right, "\textit{from-residue-ring}"'] & \mathbb{N} \triangleq \textit{nat} \arrow[l, "\lambda x.\ int\ x\ mod\ n"] \arrow[d, bend left, "\textit{from-nat}"] \\
\begin{array}{c}\textit{nat-lsbf}\\ (\text{value} < n)\end{array} & \begin{array}{c}\textit{nat-lsbf}\\ (\text{length } 2^{k+1})\end{array} \arrow[l, "\textit{reduce}"'] \arrow[u, "\textit{to-residue-ring}"'] & \textit{nat-lsbf} \arrow[l, "\textit{from-nat-lsbf}"'] \arrow[u, "\textit{to-nat}"]
\end{tikzcd}$$

Both implementations of *reduce* and *from-nat-lsbf* make use of the fact that in $\mathbb{Z}_{F_k}$, we have $2^{2^k} \equiv -1$, and hence also $2^{2^{k+1}} \equiv 1$. The *reduce* function takes some number x of length 2^{k+1} as argument, which can be split into two numbers y, z of length 2^k each. Their values are related by $x = y + z \cdot 2^{2^k} \equiv y - z$, and a representant of x in $\{0..<n\}$ may hence be calculated as follows:

$$x \bmod n = \begin{cases} y - z & \text{if } y \geq z \\ (y + n) - z & \text{else.} \end{cases}$$

In order to implement the function *from-nat-lsbf*, we note that for any $l \in \mathbb{N}$ and $a_i \in \mathbb{N}$ $(i \in \{0, \ldots, l\})$:

$$\sum_{i=0}^{l} a_i \cdot 2^{i \cdot 2^{k+1}} \equiv \sum_{i=0}^{l} a_i$$

Hence, the implementation of these four functions can be done as follows:

```
fun to-residue-ring :: nat-lsbf ⇒ int where
to-residue-ring xs = int (to-nat xs) mod int n
fun from-residue-ring :: int ⇒ nat-lsbf where
from-residue-ring x = fill (2 ^ (k + 1)) (from-nat (nat x))

definition reduce :: nat-lsbf ⇒ nat-lsbf where
reduce xs = (let (ys, zs) = split xs in
  if compare-nat zs ys then
  subtract-nat ys zs
 else
  subtract-nat (add-nat (True # replicate (2 ^ k − 1) False @ [True]) ys) zs)
```

```
function from-nat-lsbf :: nat-lsbf ⇒ nat-lsbf where
from-nat-lsbf xs = (if length xs ≤ 2 ^ (k + 1) then fill (2 ^ (k + 1)) xs
  else from-nat-lsbf (add-nat (take (2 ^ (k + 1)) xs) (drop (2 ^ (k + 1)) xs)))
```

It can be seen easily that the *reduce* function has linear runtime (see Isabelle lemma *time-reduce-tm-le*). However, the runtime of *from-nat-lsbf* is not so obvious:

Lemma 6.2 [*time-from-nat-lsbf-tm-le*]. *Assume the length of xs is at most* $c \cdot 2^{k+1}$, *and that* $2^{k+1} \geq 4$ *(i.e.* $k \geq 1$*). Then, the runtime of from-nat-lsbf is in* $O\left(c^2 \cdot 2^{k+1}\right)$.

Proof. Write $e := 2^{k+1}$. Assume that *length xs* $\leq l$. We want to define a function g[3] $: \mathbb{N} \to \mathbb{N}$, where $g(l)$ is an upper bound for the runtime of *from-nat-lsbf xs*. Let $xs' :=$ *add-nat* (*take e xs*) (*drop e xs*) and consider the following cases:

Case 1: *length xs* $\leq e$**.** In that case, *from-nat-lsbf* only adds trailing zeros to *xs* in linear runtime, i.e. in $O(e)$.

Case 2: *length xs* $> 2 \cdot e$**.** Then, xs' has at least $e - 1$ bits less than *xs*. Since the calculation of xs' has a runtime of $O(e + l)$, we have $g(l) = O(e + l) + g(l - (e - 1))$.

Case 3: *length xs* $\in \{e + 1, \ldots, 2 \cdot e\}$**.** Then, xs' has a length of at most $e + 1$, and the recursive call *from-nat-lsbf* xs' will produce some $xs'' =$ *add-nat* (*take e* xs') (*drop e* xs') with *length* $xs'' \leq e$. Both the calculation of xs' and xs'' need $O(e + l)$ time, and *from-nat-lsbf* xs'' returns after zero-padding the argument in linear time. Hence, the total runtime in this case is in $O(e + l)$.

In total, this gives rise to a recursive runtime bound of the form

$$g(l) := \begin{cases} O(e + l) & \text{if } l \leq 2 \cdot e \\ O(e + l) + g(l - (e - 1)) & \text{else,} \end{cases}$$

where the constants hidden in the O-notation are the same for each l. This recursive equation is solved explicitly in the Isabelle lemma *time-from-nat-lsbf-tm-bound-closed*:

$$g(x + l \cdot (e - 1)) = O\left(l \cdot (e + x) + \left(\sum\{0..<l\}\right) \cdot (e - 1)\right). \qquad (e + 2 \leq x \leq 2e, l \geq 0) \tag{6.1}$$

We now want to look at $g(c \cdot e)$, which gives us our desired runtime bound. Without loss of generality, assume $c \cdot e > 2 \cdot e$. Then, after writing

$$c \cdot e = x' + y' \cdot (e - 1)$$

with $y' < e - 1$, we can rewrite this term as

$$x' + y' \cdot (e - 1) = x + y \cdot (e - 1)$$

where

$$x = \begin{cases} x' + 2(e - 1) & \text{if } x' \leq 2 \\ x' + (e - 1) & \text{else} \end{cases}$$

was chosen s.t. $e + 2 \leq x' \leq 2e$. Finally, we can apply (6.1) and get

$$g(c \cdot e) = g(x' + y' \cdot (e - 1)) = O\left(y' \cdot (e + x') + \left(\sum\{0..<c\}\right) \cdot (e - 1)\right) = O\left(c^2 \cdot e\right).$$

□

[3]Named *time-from-nat-lsbf-tm-bound* in the Isabelle code

Addition can now be transferred from *nat-lsbf* using the *from-nat-lsbf* function, but since the result of *add-nat xs ys* may only have length 2^{k+1} or $2^{k+1}+1$ (if *xs* and *ys* both have length 2^{k+1}), we simplify the definition as follows:

definition *add-fermat* **where**
add-fermat xs ys = (*let zs* = *add-nat xs ys in if length zs* = *2 ^* (*k* + *1*) + *1 then inc-nat* (*butlast zs*) *else zs*)

Lemma 6.3 [*time-add-fermat-tm-le*]. *Addition in* $\mathbb{Z}_{F_k}$ *can be done in linear time.*

Before implementing subtraction we will first implement two other functions that we will also need later:

definition *multiply-with-power-of-2* :: *nat-lsbf* ⇒ *nat* ⇒ *nat-lsbf* **where**
multiply-with-power-of-2 xs m = *rotate-right m xs*

definition *divide-by-power-of-2* :: *nat-lsbf* ⇒ *nat* ⇒ *nat-lsbf* **where**
divide-by-power-of-2 xs m = *rotate-left m xs*

Here, *rotate-left* and *rotate-right* are list rotation functions, where *rotate-left* is an alternative implementation of the *rotate* function from `HOL.List`.

Lemma 6.4 [*multiply-with-power-of-2-correct*, *divide-by-power-of-2-correct*]. *Let xs be a representation of* $x \in \mathbb{Z}_{F_k}$ *and* $m \in \mathbb{N}$.

(a) multiply-with-power-of-2 xs m calculates a representation of $x \cdot 2^m$.

(b) divide-by-power-of-2 xs m calculates a representation of $x \cdot 2^{-m}$.

Proof. (a) Let *ys* be the first $2^{k+1} - m$ elements of *xs* and *zs* be the remaining m elements, representing $y, z \in \mathbb{N}$. Rotating *xs* by m elements results in the list *zs* @ *ys* representing $z + y \cdot 2^m$. Moreover, in $\mathbb{Z}_{F_k}$ we have

$$z + y \cdot 2^m \equiv_{F_k} z \cdot 2^{2^{k+1}} + y \cdot 2^m = \left(y + 2^{2^{k+1}-m}\right) \cdot 2^m = x \cdot 2^m.$$

(b) Let $y \in \mathbb{Z}_{F_k}$ be the value represented by *rotate-left m xs*. Since *rotate-right m* (*rotate-left m xs*) = *xs*, we get by (a):

$$y \cdot 2^m = x.$$

Since $2 \in \left(\mathbb{Z}_{F_k}\right)^\times$, this implies $y = x \cdot 2^{-m}$.

□

Lemma 6.5 [*time-multiply-with-power-of-2-tm-le*, *time-divide-by-power-of-2-tm-le*]. *Multiplying or dividing by powers of* 2 *in* $\mathbb{Z}_{F_k}$ *can be done in linear time. More precisely: Multiplying or dividing xs by* 2^m *can be done in* $O(\max\{m, \textit{length } xs\})$, *i.e.* $O\left(\max\{m, 2^k\}\right)$ *if length xs* $= 2^{k+1}$.

Using the fact that $2^{2^k} \equiv_{F_k} -1$, we can rewrite $x - y \equiv_{F_k} x + y \cdot 2^{2^k}$ for elements $x, y \in \mathbb{Z}_{F_k}$ and hence define:

definition *subtract-fermat* **where**
subtract-fermat xs ys = *add-fermat xs* (*multiply-with-power-of-2 ys* (*2 ^ k*))

Lemma 6.6 [*time-subtract-fermat-tm-le*]. *Subtraction in* $\mathbb{Z}_{F_k}$ *can be done in linear time.*

6.2 FNTTs in $\mathbb{Z}_{F_k}$

As in Section 6.1.2, fix some $k \in \mathbb{N}$ and let $n := F_k := 2^{2^k} + 1$. Recall our general FNTT algorithm in the context of a commutative ring R:

```
fun FNTT'' :: 'a ⇒ 'a list ⇒ 'a list where
FNTT'' μ [] = []
| FNTT'' μ [x] = [x]
| FNTT'' μ [x, y] = [x ⊕ y, x ⊖ y]
| FNTT'' μ a = (let n = length a;
          nums1 = evens-odds True a;
          nums2 = evens-odds False a;
          b = FNTT'' (μ [^] (2::nat)) nums1;
          c = FNTT'' (μ [^] (2::nat)) nums2;
          g = map2 (⊕) b (map2 (⊗) [μ [^] i. i ← [0..<(n div 2)]] c);
          h = map2 (λx y. x ⊖ y) b (map2 (⊗) [μ [^] i. i ← [0..<(n div 2)]] c)
        in g@h)
```

We want to give an implementation of this algorithm in $R = \mathbb{Z}_{F_k}$. The Schönhage-Strassen-Algorithm uses $\mu \in \{2, 4\}$. In the recursive calls of the FNTT, the primitive root will hence always be a power of 2. The multiplication by the powers of μ needed in order to calculate g and h can hence be implemented using the *multiply-with-power-of-2* function. Similarly, for $\mu = 2^{-1}$, the *divide-by-power-of-2* function can be used. Since these functions work on the exponent, we will also just keep track of the exponents of the primitive roots.

Moreover, we need to implement the calculation of g and h carefully in order to achieve a sufficient runtime. We therefore define the auxiliary function *fft-combine-b-c-aux* that combines the lists b and c elementwise by first multiplying resp. dividing (specified by the input function g) the entry in c and then adding resp. subtracting (specified by the input function f) the result from the entry in a (the parameter $(revs, e)$ accumulates the result):

```
fun fft-combine-b-c-aux :: (nat-lsbf ⇒ nat-lsbf ⇒ nat-lsbf) ⇒ (nat-lsbf ⇒ nat ⇒ nat-lsbf) ⇒ nat ⇒ nat-lsbf list
× nat ⇒ nat-lsbf list ⇒ nat-lsbf list ⇒ nat-lsbf list where
fft-combine-b-c-aux f g l (revs, e) [] [] = rev revs

| fft-combine-b-c-aux f g l (revs, e) (b # bs) (c # cs) =
   fft-combine-b-c-aux f g l ((f b (g c e)) # revs, (e + l) mod 2 ^ (k + 1)) bs cs
| fft-combine-b-c-aux f g l - - - = undefined
```

The concrete combination functions can then be defined as:

```
definition fft-combine-b-c-add where
fft-combine-b-c-add l bs cs = fft-combine-b-c-aux add-fermat multiply-with-power-of-2 l ([], 0) bs cs
definition fft-combine-b-c-subtract where
fft-combine-b-c-subtract l bs cs = fft-combine-b-c-aux subtract-fermat multiply-with-power-of-2 l ([], 0) bs cs

definition ifft-combine-b-c-add where
ifft-combine-b-c-add l bs cs = fft-combine-b-c-aux add-fermat divide-by-power-of-2 l ([], 0) bs cs
definition ifft-combine-b-c-subtract where
ifft-combine-b-c-subtract l bs cs = fft-combine-b-c-aux subtract-fermat divide-by-power-of-2 l ([], 0) bs cs
```

Ultimately, this leads to the following implementation of the *FNTT''* blueprint:

```
fun fft :: nat ⇒ nat-lsbf list ⇒ nat-lsbf list where
fft l [] = []
| fft l [x] = [x]
| fft l [x, y] = [add-fermat x y, subtract-fermat x y]
| fft l a = (let nums1 = evens-odds True a;
          nums2 = evens-odds False a;
          b = fft (2 * l) nums1;
          c = fft (2 * l) nums2;
          g = fft-combine-b-c-add l b c;
          h = fft-combine-b-c-subtract l b c
         in g@h)
```

The *ifft* function is defined similarly.

The correctness of this implementation follows from Lemma 5.13 after showing that the assumptions are satisfied:

Lemma 6.7 [*ord-2*]. *The order of* 2 *in the multiplicative group* $(\mathbb{Z}_{F_k})^\times$ *is* 2^{k+1}.

Proof. Since $2^{2^{k+1}} \equiv_{F_k} 1$, the order of 2 must be a divisor of 2^{k+1}, i.e. some power 2^i with $i \le k+1$. Assume for contradiction that $i \le k$. Then

$$1 = 1^{2^{k-i}} \equiv_{F_k} \left(2^{2^i}\right)^{2^{k-i}} = 2^{2^k} \equiv_{F_k} -1,$$

a contradiction. □

Lemma 6.8 [*two-powers-primitive-root*]. *Let* $k \in \mathbb{N}$. *Assume* $i \le k$ *and* $i + s = k + 1$. *Then,* 2^{2^i} *is a* 2^s*-th primitive root in* $\mathbb{Z}_{F_k}$.

Proof. We have

$$\left(2^{2^i}\right)^{2^s} = 2^{2^{i+s}} = 2^{2^{k+1}} \equiv_{F_k} 1.$$

Next, let $j \in \{1..<2^s\}$. Using the assumptions, we have $j \cdot 2^i \in \{1..<2^{k+1}\}$. By Lemma 6.7, this implies

$$\left(2^{2^i}\right)^j = 2^{j \cdot 2^i} \not\equiv_{F_k} 1.$$

□

Lemma 6.9 [*fft-correct, ifft-correct*]. *Let* $a' \in (\mathbb{Z}_{F_k})^n$ *where* $n = 2^l$ *for some* $l \in \mathbb{N}_{>0}$. *Assume a is a representation of* a', *i.e.* a' = *map to-residue-ring a. Moreover, assume* $i + l = k + 1$ *for some* $i \in \mathbb{N}$, *and let* $s := 2^i$.

(a) *fft s a correctly calculates a representation of* $\mathrm{NTT}_{2^s}(a')$.

(b) *ifft s a correctly calculates a representation of* $\mathrm{NTT}_{2^{-s}}(a')$.

Proof. (a) By induction, using the induction rule *fft.induct*, we get *map to-residue-ring a* = *FNTT''* 2^s a'.[4] The lemmas *FNTT''-FNTT'* and *FNTT'-FNTT* then show *FNTT''* 2^s a' = *FNTT* 2^s a'. By Lemma 6.8, 2^s is a 2^l-th primitive root. Moreover,

$$(2^s)^{2^{l-1}} = 2^{2^{i+l-1}} = 2^{2^k} \equiv_{F_k} -1.$$

Hence, the assumptions of Lemma 5.13 are satisfied and we conclude *FNTT* 2^s a' = $\mathrm{NTT}_{2^s}(a')$.

(b) The proof is similar to (a). In order to apply Lemma 5.13, use Lemma 5.3 (a) and Lemma 5.8 (b) and proceed as in (a).

□

In order to show a runtime bound for the *fft* function, we first need to bound the runtime of the auxiliary functions:

Lemma 6.10 [*time-evens-odds-tm-le*]. *The function evens-odds has linear runtime in the length of the list argument.*

Lemma 6.11 [*time-fft-combine-b-c-aux-tm-le*]. *Assume bs and cs are lists of the same length with entries of length* $e := 2^{k+1}$. *Let f be a function nat-lsbf ⇒ nat-lsbf ⇒ nat-lsbf and assume the runtime of f xs ys is in* $O\left(2^k + \textit{length xs} + \textit{length ys}\right)$ *(which is satisfied for* $f \in \{\textit{add-fermat}, \textit{subtract-fermat}\}$ *due to Lemmas 6.3 and 6.6). Moreover, let* $s < e$ *and let g be one of the two functions multiply-with-power-of-2-tm or divide-by-power-of-2-tm. Then, the runtime of fft-combine-b-c-aux f g l (revs, s) bs cs is in*

$$O\left(\textit{length revs} + (e + l) \cdot \textit{length bs}\right).$$

[4] See the Isabelle lemma *fft-correct'*.

Proof. Note that the property $s < e$ is preserved in recursive calls because of the application of *mod 2 ^ (k + 1)*. Now, each element b of the list bs is processed together with the corresponding element c of the list cs in the function call $f\ b\ (g\ c\ s)$, which (by assumptions and Lemma 6.5) takes time $O\left(2^k + 2 \cdot e + \max\{s, e\}\right) = O\left(e\right)$. Moreover, the calculation of $(s + l)$ *mod 2 ^ (k + 1)* takes time $O\left(s + l + e\right) = O\left(l + e\right)$ (due to our conservative estimates for functions on the *nat* type). In total, each element of the list bs contributes $O\left(e + l\right)$ to the runtime and one element to the accumulator.

After all elements are processed, the accumulator contains *length bs* + *length revs* elements which are reversed in time $O\left(\textit{length bs} + \textit{length revs}\right)$.

Thus, the total runtime is in

$$O\left(\textit{length bs} + \textit{length revs} + \textit{length bs} \cdot (e + l)\right) = O\left(\textit{length revs} + \textit{length bs} \cdot (e + l)\right).$$

□

Lemma 6.12 [*time-fft-tm-le, time-ifft-tm-le*]. *Assume $a \in \left(\mathbb{Z}_{F_k}\right)^{2^m}$. Then, fft l a resp. ifft l a have a runtime of $O\left(m \cdot 2^m \cdot 2^k + l \cdot 2^{2m}\right)$.*

Proof. Using Lemmas 6.3, 6.6, 6.10 and 6.11, the definition of *fft* gives rise to the following recursive runtime equation:

$$t(l, a) = \begin{cases} O\left(1\right) & \text{if } a = [] \\ O\left(1\right) & \text{if } a = [x] \\ O\left(2^k\right) & \text{if } a = [x,y] \\ O\left(3 \cdot \textit{length } a + 2 \cdot (l + 2^k) \cdot \frac{\textit{length } a}{2}\right) + t(2l, a_{\text{even}}) + t(2l, a_{\text{odd}}) & \text{else.} \end{cases}$$

Here, the summands in the last equation arise as follows:

- $O\left(3 \cdot \textit{length } a\right)$ is the time needed to split a into its even-/odd-indexed part and appending g and h in the last line.
- $O\left(2 \cdot (l + 2^k) \cdot \frac{\textit{length } a}{2}\right)$ is the time needed to combine the results of the recursive calls due to Lemma 6.11.
- The last two summands are the runtime of the recursive *fft* calculations.

Rewriting this equation in terms of m yields the recursive equation

$$\tilde{t}(l, m) = \begin{cases} O\left(1\right) & \text{if } m = 0 \\ O\left(2^k\right) & \text{if } m = 1 \\ O\left(2 \cdot 2^m + (l + 2^k) \cdot 2^m\right) + 2 \cdot \tilde{t}(2l, m - 1) & \\ = O\left((l + 2^k) \cdot 2^m\right) + 2 \cdot \tilde{t}(2l, m - 1) & \text{else.} \end{cases}$$

In the Isabelle proof, the constants hidden in the O-notation are given explicitly. For an informal proof, we will just use any simple constants and define

$$g(l, m) := \begin{cases} 1 & \text{if } m = 0 \\ 2^k & \text{if } m = 1 \\ (l + 2^k) \cdot 2^m + 2 \cdot g(2l, m - 1) & \text{else.} \end{cases}$$

The equation for $m > 1$ can be solved as follows:

$$\begin{aligned}
g(l,m) &= 2^0 \cdot (2^0 \cdot l + 2^k) \cdot 2^m + 2^1 \cdot g(2^1 \cdot l, m-1) \\
&= 2^0 \cdot (2^0 \cdot l + 2^k) \cdot 2^m + 2^1 \cdot (2^1 \cdot l + 2^k) \cdot 2^{m-1} + 2^2 \cdot g(2^2 \cdot l, m-2) \\
&= \dots \\
&= \left(\sum_{i=0}^{m-1} 2^i \cdot (2^i \cdot l + 2^k) \cdot 2^{m-i} \right) + 2^{m-1} \cdot g(2^{m-1} \cdot l, 1) \\
&= \left(\sum_{i=0}^{m-1} (2^i \cdot l + 2^k) \cdot 2^m \right) + 2^{m-1} \cdot 2^k \\
&= \left(\sum_{i=0}^{m-1} 2^i \right) \cdot l \cdot 2^m + m \cdot 2^k \cdot 2^m + 2^{m-1} \cdot 2^k \\
&= (2^m - 1) \cdot l \cdot 2^m + m \cdot 2^k \cdot 2^m + 2^{m-1} \cdot 2^k.
\end{aligned}$$

This shows that $g(l,m) \in \mathcal{O}\left(m \cdot 2^k \cdot 2^m + l \cdot 2^{2m}\right)$. □

6.3 A Special Residue Problem

We need to solve the following problem in the Schönhage-Strassen-Algorithm: Given $\xi \in \mathbb{Z}_{F_n}$ and $\eta \in \mathbb{Z}_{2^{n+2}}$, find the unique $z \in \{0..<F_n \cdot 2^{n+2}\}$ s.t. $z \equiv_{F_n} \xi$ and $z \equiv_{2^{n+2}} \eta$. Schönhage and Strassen [SS71] give an explicit solution by

$$\delta :\equiv_{2^{n+2}} \eta - \xi \in \mathbb{Z}_{2^{n+2}}, \tag{6.2}$$

$$z := \xi + \delta \cdot (2^{2^n} + 1). \tag{6.3}$$

This can be implemented as follows:

definition *solve-special-residue-problem* **where**
solve-special-residue-problem n ξ η =
  (let δ = int-lsbf-mod.subtract-mod (n + 2) η (take (n + 2) ξ) in
  add-nat ξ (add-nat (replicate (2 ^ n) False @ δ) δ))

Lemma 6.13 [*solve-special-residue-problem-correct, special-residue-problem-unique-solution*]. *Let $n \geq 2$, $\xi \in \mathbb{Z}_{F_n}$ and $\eta \in \mathbb{Z}_{2^{n+2}}$. Then, z as given in (6.2) and (6.3) is the unique solution in $\{0..<F_n \cdot 2^{n+2}\}$ satisfying $z \equiv_{F_n} \xi$ and $z \equiv_{2^{n+2}} \eta$.*

Proof. Since $n \geq 2$, we have $2^n \geq n + 2$, and hence

$$z = \xi + \delta + \delta \cdot \underbrace{2^{2^n}}_{\equiv_{2^{n+2}} 0} \equiv_{2^{n+2}} \xi + \delta \equiv_{2^{n+2}} \eta.$$

$z \equiv_{F_n} \xi$ follows immediately from the definition of z. The uniqueness is a consequence of the chinese remainder theorem[5], since F_n and 2^{n+2} are coprime. □

Since all involved functions have linear runtime, we immediately get:

Lemma 6.14 [*time-solve-special-residue-problem-tm-le*]. *The function solve-special-residue-problem has linear runtime, i.e. $\mathcal{O}(2^n + \text{length } \eta + \text{length } \xi)$ or simply $\mathcal{O}(2^n)$ if length $\eta = n + 2$ and length $\xi = 2^{n+1}$.*

[5]See Isabelle lemma *chinese-remainder-very-simple-nat*.

6.4 The Schönhage-Strassen-Algorithm in $\mathbb{Z}_{F_m}$

In this section, we will describe our implementation of the Schönhage-Strassen-Algorithm for numbers in $\mathbb{Z}_{F_m}$ in Isabelle. We will implement the algorithm step by step, as in the original paper [SS71].

All functions from Sections 6.1.1, 6.1.2 and 6.2 are defined in locales *int-lsbf-mod* resp. *int-lsbf-fermat* fixing the resp. parameter k. In order to use these functions, we specify the qualified versions here, e.g. *int-lsbf-mod.reduce* $(n + 2)$ for the *reduce* function in the context of representations of $\mathbb{Z}_{2^{n+2}}$.

Assume a and b are representations of numbers in $\mathbb{Z}_{F_m}$. If m is small, i.e. $m < 3$, we just use the classic grid multiplication in order to multiply a and b and convert the result using the *int-lsbf-fermat.from-nat-lsbf* function.

For the rest of this section, let $m \geq 3$. Define

$$n := \begin{cases} \frac{m+1}{2} & \text{if } m \text{ is odd} \\ \frac{m+2}{2} & \text{if } m \text{ is even,} \end{cases}$$

i.e. $m = 2n - 1$ if m is odd and $m = 2n - 2$ otherwise. Moreover, define

$$n_{\text{oe}} := \begin{cases} n+1 & \text{if } m \text{ is odd} \\ n & \text{if } m \text{ is even.} \end{cases}$$

Note that $m + 1 = (n - 1) + n_{\text{oe}}$. The first step is to subdivide a and b into a list a' resp. b' containing blocks of size 2^{n-1}. Since they are representations of numbers in $\mathbb{Z}_{F_m}$, i.e. have length 2^{m+1}, there will be $2^{n_{\text{oe}}}$ such blocks (see Figure 6.1).

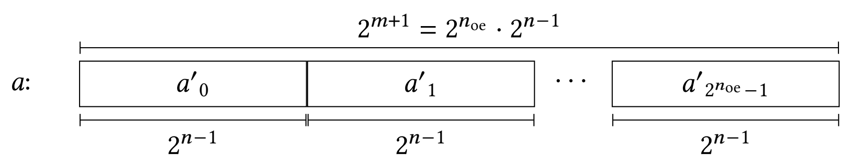

$2^{m+1} = 2^{n_{\text{oe}}} \cdot 2^{n-1}$
a:
a'_0
a'_1
$\cdots$
$a'_{2^{n_{\text{oe}}}-1}$
2^{n-1}
2^{n-1}
2^{n-1}

Figure 6.1 Constructing a' from a.

As in [SS71], we define for verification purposes:

$$c'_j := \sum_{\sigma=0}^{2^{n_{\text{oe}}}-1} a'_\sigma \cdot b'_{(2^{n_{\text{oe}}}+j-\sigma) \bmod 2^{n_{\text{oe}}}} \qquad (j \in \{0..{<}2^{n_{\text{oe}}}\}) \tag{6.4}$$

$$z'_j := c'_j - c'_{2^{n_{\text{oe}}-1}+j} + 2^{n_{\text{oe}}+2^n} \qquad \left(j \in \{0..{<}2^{n_{\text{oe}}-1}\}\right) \tag{6.5}$$

$$z'_j := 2^{n_{\text{oe}}+2^n} \qquad \left(j \in \{2^{n_{\text{oe}}-1}..{<}2^{n_{\text{oe}}}\}\right) \tag{6.6}$$

The trick of Schönhage and Strassen is now to rewrite the product $a \cdot b$ in terms of the ${z'}_j$[6]:

$$\begin{aligned}
a \cdot b &= \left(\sum_{\sigma=0}^{2^{n_{\mathrm{oe}}}-1} {a'}_\sigma \cdot 2^{\sigma \cdot 2^{n-1}} \right) \cdot \left(\sum_{\rho=0}^{2^{n_{\mathrm{oe}}}-1} {b'}_\rho \cdot 2^{\rho \cdot 2^{n-1}} \right) \\
&\equiv_{F_m} \sum_{j=0}^{2^{n_{\mathrm{oe}}}-1} \sum_{\sigma=0}^{2^{n_{\mathrm{oe}}}-1} {a'}_\sigma \cdot {b'}_{(2^{n_{\mathrm{oe}}}+j-\sigma) \bmod 2^{n_{\mathrm{oe}}}} \cdot 2^{j \cdot 2^{n-1}} && \text{(Lemmas 5.5, 6.8)} \\
&= \sum_{j=0}^{2^{n_{\mathrm{oe}}}-1} {c'}_j \cdot 2^{j \cdot 2^{n-1}} \\
&= \sum_{j=0}^{2^{n_{\mathrm{oe}}-1}-1} {c'}_j \cdot 2^{j \cdot 2^{n-1}} + \sum_{j=0}^{2^{n_{\mathrm{oe}}-1}-1} \left({c'}_{2^{n_{\mathrm{oe}}-1}+j} - 2^{n_{\mathrm{oe}}+2^n} + 2^{n_{\mathrm{oe}}+2^n}\right) \cdot 2^{(2^{n_{\mathrm{oe}}-1}+j) \cdot 2^{n-1}} \\
&= \sum_{j=0}^{2^{n_{\mathrm{oe}}-1}-1} {c'}_j \cdot 2^{j \cdot 2^{n-1}} + 2^{2^{n_{\mathrm{oe}}-1} \cdot 2^{n-1}} \cdot \sum_{j=0}^{2^{n_{\mathrm{oe}}-1}-1} \left({c'}_{2^{n_{\mathrm{oe}}-1}+j} - 2^{n_{\mathrm{oe}}+2^n}\right) \cdot 2^{j \cdot 2^{n-1}} \\
&\quad + \sum_{j=0}^{2^{n_{\mathrm{oe}}-1}-1} 2^{n_{\mathrm{oe}}+2^n} \cdot 2^{(2^{n_{\mathrm{oe}}-1}+j) \cdot 2^{n-1}} \\
&\equiv_{F_m} \sum_{j=0}^{2^{n_{\mathrm{oe}}-1}-1} \left({c'}_j - {c'}_{2^{n_{\mathrm{oe}}-1}+j} + 2^{n_{\mathrm{oe}}+2^n}\right) \cdot 2^{j \cdot 2^{n-1}} \\
&\quad + \sum_{j=0}^{2^{n_{\mathrm{oe}}-1}-1} 2^{n_{\mathrm{oe}}+2^n} \cdot 2^{(2^{n_{\mathrm{oe}}-1}+j) \cdot 2^{n-1}} && \left(2^{2^{n_{\mathrm{oe}}-1} \cdot 2^{n-1}} = 2^{2^m} \equiv_{F_m} -1\right) \\
&= \sum_{j=0}^{2^{n_{\mathrm{oe}}}-1} {z'}_j \cdot 2^{j \cdot 2^{n-1}}. && (6.7)
\end{aligned}$$

So, it suffices to calculate the ${z'}_j$ for $j \in \{0..{<}2^{n_{\mathrm{oe}}-1}\}$ in order to obtain the product $a \cdot b$ in $\mathbb{Z}_{F_m}$. The Schoenhage-Strassen-Algorithm proceeds as follows:

1. Calculate the residues of ${z'}_j$ modulo 2^{n+2}.
2. Calculate the residues of ${z'}_j$ modulo F_n.
3. Combine the results to reconstruct ${z'}_j \in \mathbb{Z}_{F_m}$.

6.4.1 Residues in $\mathbb{Z}_{2^{n+2}}$

In order to calculate the residues of ${z'}_j$ modulo 2^{n+2}, it suffices to consider the $n + 2$ least significant bits of each of the ${a'}_\sigma$ and ${b'}_\rho$ due to (6.4). However, calculating the residues of the ${z'}_j$ directly would still take $O\left(2^n \cdot M(n)\right)$ time for each j, where $M(n)$ is the time needed to multiply two numbers in $\mathbb{Z}_{n+2}$. In total, the calculation of all ${z'}_j$ would hence need $O\left(2^{2n} \cdot M(n)\right)$ time, which is not fast enough unless $M(n)$ would be in $O(n)$.

Schönhage and Strassen solve this problem by performing a single multiplication, from whose result the residues of the ${z'}_j$ can be read off. As a first step, the $n + 2$ least significant bits of each ${a'}_\sigma$, defined as α_σ, are padded with trailing zeros to get numbers of length $3n + 5$.[7] The concatenation of these segments, called u, then has length $2^{n_{\mathrm{oe}}} \cdot (3n+5)$. Similarly, β_ρ and v are defined for ${b'}_\rho$. An illustration of the process for u can be found in Figure 6.2.

[6]This equation is labelled *result0* in the Isabelle proof of *schoenhage-strassen-correct'*.

[7]It would suffice to pad the residues to length $2n + 4 + n_{\mathrm{oe}}$, but for simplicity (and since it does not matter for the asymptotic runtime) we just use $3n + 5$ in any case.

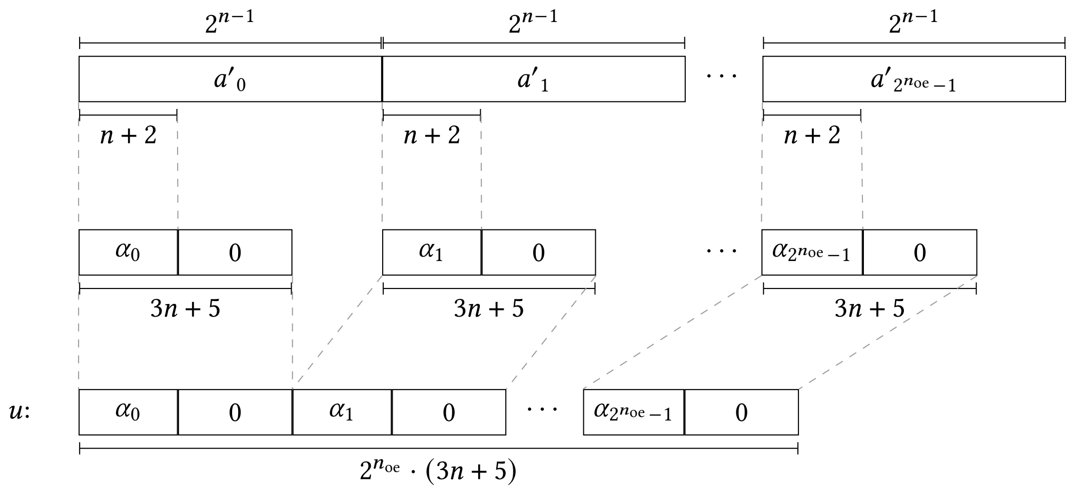


Figure 6.2 Constructing u from a'.

Now, we can multiply u and v roughly in time $O(M(n \cdot 2^n))$, e.g. in $O\left((n \cdot 2^n)^{\log_2 3}\right)$ when using Karatsuba-Multiplication (see Lemma 4.4), which will be fast enough. The result may vary in length, but we can just delete or append trailing zeros to ensure a length of $2^{n_{oe}+1} \cdot (3n+5)$.[8] Let us call the result with this length uv.

Subdividing uv into $2^{n_{oe}+1}$ blocks $\gamma_0, \gamma_1, \ldots, \gamma_{2^{n_{oe}+1}-1}$ of size $3n+5$ each, we have[9]

$$uv = \sum_{k=0}^{2^{n_{oe}+1}-1} \gamma_k \cdot 2^{k \cdot (3n+5)}$$

and

$$\begin{aligned} uv = u \cdot v &= \left(\sum_{i=0}^{2^{n_{oe}}-1} \alpha_i \cdot 2^{i \cdot (3n+5)} \right) \cdot \left(\sum_{j=0}^{2^{n_{oe}}-1} \beta_j \cdot 2^{j \cdot (3n+5)} \right) \\ &= \sum_{i=0}^{2^{n_{oe}}-1} \sum_{j=0}^{2^{n_{oe}}-1} \alpha_i \cdot \beta_j \cdot 2^{(i+j) \cdot (3n+5)} \\ &= \sum_{k=0}^{2^{n_{oe}+1}-1} \underbrace{\sum_{i=0}^{2^{n_{oe}}-1} \sum_{j=0}^{2^{n_{oe}}-1} \delta_{i+j,k} \cdot \alpha_i \cdot \beta_j}_{=:\gamma'_k} \cdot 2^{k \cdot (3n+5)}. \end{aligned}$$

Because $\gamma_k < 2^{3n+5}$ and $\gamma'_k < 2^{n_{oe}} \cdot 2^{n+2} \cdot 2^{n+2} \leq 2^{3n+5}$, the following lemma shows that $\gamma_k = \gamma'_k$ for $k \in \{0..<2^{n_{oe}+1}\}$.

Lemma 6.15 [*power-sum-nat-eq*]. *Let $x, c, n \in \mathbb{N}$, $a_i, b_i \in \mathbb{N}$ ($i \in \{0..<n\}$). Assume $x > 1$, $c > 0$ and $a_i, b_i < x^c$ for all $i \in \{0..<n\}$. Moreover, assume*

$$\sum_{i=0}^{n-1} a_i \cdot x^{i \cdot c} = \sum_{i=0}^{n-1} b_i \cdot x^{i \cdot c}.$$

Then, $a_i = b_i$ for all $i \in \{0..<n\}$.

[8] Of course, we need to formally prove this in Isabelle. This is done in the proof of *schoenhage-strassen-correct'* in the line *to-nat uv = to-nat u ∗ to-nat v*.

[9] See the equation with label *to-nat-γ* in the proof of *schoenhage-strassen-correct'*.

Proof. Under the assumptions, the j-th coefficient in the sums can be extracted:

$$a_j = \left\lfloor \frac{\sum_{i=0}^{n-1} a_i \cdot x^{i \cdot c}}{x^{j \cdot c}} \right\rfloor \bmod x^c = \left\lfloor \frac{\sum_{i=0}^{n-1} b_i \cdot x^{i \cdot c}}{x^{j \cdot c}} \right\rfloor \bmod x^c = b_j.$$

□

Now, note that we can rewrite c'_k in $\mathbb{Z}_{2^{n+2}}$ as follows (for $k \in \{0..<2^{n_{\text{oe}}}\}$):[10]

$$\begin{aligned}
c'_k &= \sum_{\sigma=0}^{2^{n_{\text{oe}}}-1} a'_\sigma \cdot b'_{(2^{n_{\text{oe}}}+k-\sigma) \bmod 2^{n_{\text{oe}}}} \\
&\equiv_{2^{n+2}} \sum_{\sigma=0}^{2^{n_{\text{oe}}}-1} \alpha_\sigma \cdot \beta_{(2^{n_{\text{oe}}}+k-\sigma) \bmod 2^{n_{\text{oe}}}} \\
&= \sum_{\sigma=0}^{2^{n_{\text{oe}}}-1} \sum_{\rho=0}^{2^{n_{\text{oe}}}-1} [\sigma + \rho \equiv_{2^{n_{\text{oe}}}} k] \cdot a'_\sigma \cdot b'_\rho \\
&= \sum_{\sigma=0}^{2^{n_{\text{oe}}}-1} \sum_{\rho=0}^{2^{n_{\text{oe}}}-1} ([\sigma + \rho = k] + [\sigma + \rho = k + 2^{n_{\text{oe}}}]) \cdot a'_\sigma \cdot b'_\rho \\
&= \gamma'_k + \gamma'_{2^{n_{\text{oe}}}+k} = \gamma_k + \gamma_{2^{n_{\text{oe}}}+k}.
\end{aligned}$$

Hence, for $j \in \{0..<2^{n_{\text{oe}}-1}\}$:

$$\begin{aligned}
z'_j &= c'_j - c'_{2^{n_{\text{oe}}-1}+j} + 2^{n_{\text{oe}}+2^n} \\
&\equiv_{2^{n+2}} (\gamma_j + \gamma_{2^{n_{\text{oe}}}+j}) - (\gamma_{2^{n_{\text{oe}}-1}+j} + \gamma_{2^{n_{\text{oe}}}+2^{n_{\text{oe}}-1}+j}) \\
&= (\gamma_j - \gamma_{2^{n_{\text{oe}}-1}+j}) + (\gamma_{2 \cdot 2^{n_{\text{oe}}-1}+j} - \gamma_{3 \cdot 2^{n_{\text{oe}}-1}+j})
\end{aligned}$$

Dividing $[\gamma_0, \ldots, \gamma_{2^{n_{\text{oe}}+1}-1}]$ into four blocks $\gamma^{(0)}, \gamma^{(1)}, \gamma^{(2)}, \gamma^{(3)}$ of length $2^{n_{\text{oe}}-1}$, i.e. $\gamma_j^{(i)} = \gamma_{i \cdot 2^{n_{\text{oe}}-1}+j}$, we can hence calculate (with additions/subtractions in $\mathbb{Z}_{n+2}$)

$$\eta_j := \left(\gamma_j^{(0)} - \gamma_j^{(1)}\right) + \left(\gamma_j^{(2)} - \gamma_j^{(3)}\right)$$

and conclude $\eta_j \equiv_{2^{n+2}} z'_j$.

Runtime

The construction of u resp. v can be done in linear runtime, but we will not discuss the details here. As noted earlier, their multiplication needs time $O\left((n \cdot 2^n)^{\log_2 3}\right)$. Splitting uv into the blocks $\gamma_0, \gamma_1, \ldots, \gamma_{2^{n_{\text{oe}}+1}-1}$ can again be done in linear time, i.e. in $O(n \cdot 2^n)$ since uv has length $2^{n_{\text{oe}}+1} \cdot (3n+5)$. Similarly, subdividing the list $[\gamma_0, \ldots, \gamma_{2^{n_{\text{oe}}+1}-1}]$ into the four blocks $\gamma^{(0)}, \gamma^{(1)}, \gamma^{(2)}, \gamma^{(3)}$ can be done in $O(n \cdot 2^n)$. Finally, since addition and subtraction in $\mathbb{Z}_{n+2}$ can be done in linear time by Lemma 6.1, calculating all η_j (for $j < 2^{n_{\text{oe}}-1}$) needs a runtime of $O(n \cdot 2^n)$.

6.4.2 Residues in $\mathbb{Z}_{F_n}$

Let $j \in \{0..<2^{n_{\text{oe}}}\}$. Since

$$c'_j = \sum_{\sigma=0}^{2^{n_{\text{oe}}}-1} a'_\sigma \cdot b'_{(2^{n_{\text{oe}}}+j-\sigma) \bmod 2^{n_{\text{oe}}}} \equiv_{F_n} (a' \star b')_j \tag{6.8}$$

[10]Corresponding Isabelle equation: γc in the proof of *schoenhage-strassen-correct'*.

with $a' = (a'_0, \ldots, a'_{2^{n_{\text{oe}}}-1}), b' = (a'_0, \ldots, a'_{2^{n_{\text{oe}}}-1}) \in \left(\mathbb{Z}_{F_n}\right)^{2^{n_{\text{oe}}}}$, we transform a' and b' using NTTs in the ring $\mathbb{Z}_{F_n}$, multiply them componentwise and use inverse NTTs to transform the result back. As in Schönhage and Strassen [SS71], we will use $\mu := 2^p$ as primitive root, where

$$p := \begin{cases} 1 & \text{if } m \text{ is odd} \\ 2 & \text{else.} \end{cases}$$

As a first step, we calculate

$$\hat{a} := \text{NTT}_\mu(a'), \qquad \hat{b} := \text{NTT}_\mu(b')$$

using the function *int-lsbf-fermat.fft*. Note that e.g. the entries of a' have length 2^{n-1}, and hence need to be padded to a length of 2^{n+1} first in order to be representations of numbers in $\mathbb{Z}_{F_n}$. The *fft* function produces the correct result due to Lemma 6.9 (a), because $p = 2^{p-1}$ (since $p \in \{1, 2\}$) and $(p-1) + n_{\text{oe}} = n + 1$.

The componentwise multiplication needs to be done in $\mathbb{Z}_{F_n}$. Hence, we can call the algorithm recursively to calculate

$$\hat{c}_j := \hat{a}_j \cdot \hat{b}_j \quad (j \in \{0..<2^{n_{\text{oe}}}\}).$$

However, this would lead to a total runtime worse than $O(n \log n \log\log n)$. Schönhage and Strassen point out that, since it is enough to obtain the differences $c'_j - c'_{2^{n_{\text{oe}}-1}+j}$ for $j \in \{0..<2^{n_{\text{oe}}-1}\}$, we only need to calculate $\hat{c}_j$ if j is odd. So, we recursively calculate

$$\hat{c}_{\text{odd}} := (\hat{c}_1, \hat{c}_3, \ldots, \hat{c}_{2^{n_{\text{oe}}}-1}) \in \left(\mathbb{Z}_{F_n}\right)^{2^{n_{\text{oe}}-1}}.$$

Next, the inverse NTT

$$c_{\text{diffs}} := \text{NTT}_{\mu^{-2}}(\hat{c}_{\text{odd}})$$

is calculated using the *int-lsbf-fermat.ifft* function, where $\mu^{-2} = 2^{-2p} = 2^{-2^p}$. This can be done due to Lemma 6.9 (b), since $p + (n_{\text{oe}} - 1) = n + 1$.

We will now show how the residues of the z'_j can be obtained from c_{diffs}. First, by Theorem 5.9 and Lemma 5.10, we have[11]

$$\text{NTT}_{\mu^{-1}}(\text{NTT}_\mu(c'_{\text{mod}})) = 2^{n_{\text{oe}}} \cdot c'_{\text{mod}}$$

where $c'_{\text{mod}} = (c'_0 \bmod F_n, \ldots, c'_{2^{n_{\text{oe}}}-1} \bmod F_n) \in \left(\mathbb{Z}_{F_n}\right)^{2^{n_{\text{oe}}}}$. Hence, for $j \in \{0..<2^{n_{\text{oe}}-1}\}$,

$$\begin{aligned} c'_j - c'_{2^{n_{\text{oe}}-1}+j} &\equiv_{F_n} 2^{-n_{\text{oe}}} \cdot \left(\text{NTT}_{\mu^{-1}}(\text{NTT}_\mu(c'_{\text{mod}}))_j - \text{NTT}_{\mu^{-1}}(\text{NTT}_\mu(c'_{\text{mod}}))_{2^{n_{\text{oe}}-1}+j}\right) \\ &= 2^{-n_{\text{oe}}} \cdot 2 \cdot \mu^{-j} \cdot \text{NTT}_{\mu^{-2}}(\text{NTT}_\mu(c'_{\text{mod}})_{\text{odd}})_j && \text{(Lemma 5.12)} \\ &= 2^{-n_{\text{oe}}} \cdot 2 \cdot \mu^{-j} \cdot \text{NTT}_{\mu^{-2}}(\text{NTT}_\mu(a' \star b')_{\text{odd}})_j && \text{((6.8))} \end{aligned}$$

By Theorem 5.6, we have $\text{NTT}_\mu(a' \star b')_i = \hat{a}_i \cdot \hat{b}_i = \hat{c}_i$ for all $i \in \{0..<2^{n_{\text{oe}}}\}$. In particular, $\text{NTT}_\mu(a' \star b')_{\text{odd}} = \hat{c}_{\text{odd}}$. Inserting this and the definition of c_{diffs}, we get

$$c'_j - c'_{2^{n_{\text{oe}}-1}+j} \equiv_{F_n} 2^{-n_{\text{oe}}} \cdot 2 \cdot \mu^{-j} \cdot (c_{\text{diffs}})_j$$

and thus

$$z'_j \equiv_{F_n} 2^{-n_{\text{oe}}} \cdot 2 \cdot \mu^{-j} \cdot (c_{\text{diffs}})_j + 2^{n_{\text{oe}}+2^n} =: \xi_j \in \mathbb{Z}_{F_n}.$$

Runtime

Calculating the NTTs $\hat{a}$ and $\hat{b}$ needs time $O\left(n_{\text{oe}} \cdot 2^{n_{\text{oe}}} \cdot 2^n + p \cdot 2^{2n_{\text{oe}}}\right) = O\left(n \cdot 2^{2n}\right)$ by Lemma 6.12. Moreover, we need to recursively calculate $\hat{c}_i$ for all odd $i \in \{0..<2^{n_{\text{oe}}}\}$, i.e. have $2^{n_{\text{oe}}-1}$ recursive calls for the multiplications in $\mathbb{Z}_{F_n}$.

Each ξ_j can be calculated by multiplying $(c_{\text{diffs}})_j$ with $2^{-n_{\text{oe}}+1-p\cdot j} \equiv_{F_n} 2^{2^{n+1}-(n_{\text{oe}}+p\cdot j-1)}$, which can be done in $O(2^n)$ using Lemma 6.5, and adding $2^{n_{\text{oe}}+2^n}$ to the result, which can be done in $O(n + 2^n) = O(2^n)$ by

[11]Equation *aux1* in the proof of *schoenhage-strassen-correct'*.

Lemma 6.3. Since there are $2^{n_{oe}-1}$ indices $j \in \{0..<2^{n_{oe}-1}\}$, the calculation of all ξ_j from c_{diffs} needs time $O\left(2^{2n}\right)$.

Thus, calculating all ξ_j from a' and b' needs a total time of $O\left(n \cdot 2^{2n}\right) + 2^{n_{oe}-1} \cdot T(n)$, where $T(n)$ is the recursive runtime for multiplication in $\mathbb{Z}_{F_n}$.

6.4.3 Combining the Residues and Constructing the Result

After calculating ξ_j and η_j for $j \in \{0..<2^{n_{oe}-1}\}$, we can use the function *solve-special-residue-problem* to obtain some $z_j \in \{0..<F_n \cdot 2^{n+2}\}$ which is the unique solution to the equation system

$$\begin{aligned} z_j &\equiv_{F_n} \xi_j \\ z_j &\equiv_{2^{n+2}} \eta_j. \end{aligned}$$

Since z'_j also solves this equation system, we therefore have $z_j = z'_j$.

Recalling (6.7), i.e.

$$a \cdot b \equiv_{F_m} \sum_{j=0}^{2^{n_{oe}}-1} z'_j \cdot 2^{j \cdot 2^{n-1}},$$

we are now ready to calculate the final result by inserting

$$z'_j = \begin{cases} z_j & \text{if } j \in \{0..<2^{n_{oe}-1}\} \\ 2^{n_{oe}+2^n} & \text{if } j \in \{2^{n_{oe}-1}..<2^{n_{oe}}\}. \end{cases}$$

In order to implement this efficiently, we define the function *combine-z*, which, given some $l \in \mathbb{N}_{>0}$ and a list $zs = [zs_0, \ldots, zs_{s-1}]$ of numbers each having length at least l, returns a representation of

$$\sum_{i=0}^{s-1} zs_i \cdot 2^{i \cdot l}.$$

Since $zs_0, \ldots, zs_{i-1}$ are the only numbers relevant for the $i \cdot l$ least significant bits of the result, we can store these $i \cdot l$ bits in an accumulator and iteratively calculate the next l bits of the result. As long as there are at least two entries zs_i, zs_{i+1} left, we proceed as follows:

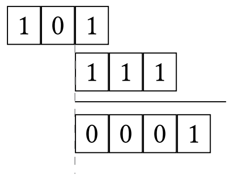
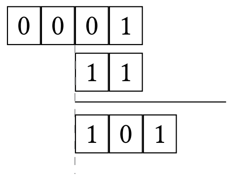
1. Append the least significant l bits of zs_i to the accumulator.
2. Add the remaining bits of zs_i to zs_{i+1} to obtain some number r.
3. Replace zs_{i+1} by r and continue with the list $[r, zs_{i+2}, \ldots, zs_{s-1}]$.

If there is only one element in the list left, we append it to the accumulator, too. Finally, when there are no more elements left, we can return the result from the accumulator. An illustration of the procedure with $l = 2$ can be found in Figure 6.3. As a further modification, we store the accumulator in reverse, so that adding the next bits to it can be done in constant time. Ultimately, we obtain the following algorithm:

```
fun combine-z-aux where
combine-z-aux l acc [] = concat (rev acc)
| combine-z-aux l acc [z] = combine-z-aux l (z # acc) []
| combine-z-aux l acc (z1 # z2 # zs) = (let
    (z1h, z1t) = split-at l z1 in
    combine-z-aux l (z1h # acc) ((add-nat z1t z2) # zs)
  )
```

```
definition combine-z :: nat ⇒ nat-lsbf list ⇒ nat-lsbf where
combine-z l zs = combine-z-aux l [] zs
```

We will only state the correctness and runtime lemmas without proof here.

accumulator	list	calculation
[]	[101, 111, 11]	
[10]	[0001, 11]	
[10, 00]	[101]	
[10, 00, 101]	[]	

Result: 1000101

Figure 6.3 Visualization of the algorithm used for the *combine-z*-function.

Lemma 6.16 [*combine-z-correct*]. *Let $l \in \mathbb{N}_{>0}$, and let $zs = [zs_0, \ldots, zs_{s-1}]$ be a list of numbers in LSBF representation, each of length at least l. Then, combine-z l zs returns a representation of*

$$\sum_{i=0}^{s-1} zs_i \cdot 2^{i \cdot l}.$$

Lemma 6.17 [*time-combine-z-tm-le*]. *Let $l \in \mathbb{N}_{>0}$, and let $zs = [zs_0, \ldots, zs_{s-1}]$ be a list of numbers in LSBF representation, each of length at most (!) L. Then, combine-z l zs runs in time $O\left((l + L) \cdot s\right)$.*

Runtime

Calculating each of the z_j from the ξ_j and η_j using *solve-special-residue-problem* takes time $O\left(2^n\right)$ by Lemma 6.14. Hence, calculating all z_j for $j \in \{0..{<}2^{n_{\text{oe}}-1}\}$ takes $O\left(2^{n_{\text{oe}}-1} \cdot 2^n\right) = O\left(2^{2n}\right)$ time.

Constructing the list $z' = [z'_0, \ldots, z'_{2^{n_{\text{oe}}}} - 1]$ from the z_j ($j \in \{0..{<}2^{n_{\text{oe}}-1}\}$) can be done in linear time, i.e. in $O\left(2^n\right)$.

Constructing the result from z', i.e. applying the *combine-z* function to it with block size $l = 2^{n-1}$, takes time $O\left(\left(2^{n-1} + (2^n + n + 4)\right) \cdot 2^{n_{\text{oe}}}\right) = O\left(2^{2n}\right)$ by Lemma 6.17, using that w.l.o.g. all entries of z' have length at most $2^n + n + 4$.[12]

In total, the construction of the result from the ξ_j and η_j thus takes time $O\left(2^{2n}\right)$.

6.4.4 Implementation

Combining all previous considerations, we get the following implementation and correctness result:

```
function schoenhage-strassen :: nat ⇒ nat-lsbf ⇒ nat-lsbf ⇒ nat-lsbf where
schoenhage-strassen m a b =
 (if m < 3 then int-lsbf-fermat.from-nat-lsbf m (grid-mul-nat a b) else
 let
   n = (if odd m then (m + 1) div 2 else (m + 2) div 2);
   oe-n = (if odd m then n + 1 else n);
```

[12]For the entries $z'_j = 2^{n_{\text{oe}}+2^n}$, this is obvious. For the entries $z'_j = z_j \in \{0..{<}F_n \cdot 2^{n+2}\}$, this follows since $F_n \cdot 2^{n+2} < 2^{2^n+1} \cdot 2^{n+2} = 2^{2^n+n+3}$. Note that we could hence also assume w.l.o.g. that all entries of z' have length at most $2^n + n + 3$, however weakening this bound simplifies some Isabelle code.

```
  — residue mod 2^(n+2)
  a′ = subdivide (2 ^ (n − 1)) a;
  α = map (int-lsbf-mod.reduce (n + 2)) a′;
  u = concat (map (fill (3∗n + 5)) α);
  b′ = subdivide (2 ^ (n − 1)) b;
  β = map (int-lsbf-mod.reduce (n + 2)) b′;
  v = concat (map (fill (3∗n + 5)) β);
  uv = ensure-length ((3∗n + 5) ∗ 2 ^ (oe-n + 1)) (karatsuba-mul-nat u v);
  γ = subdivide (2 ^ (oe-n − 1)) (subdivide (3∗n + 5) uv);
  η = map4 (λx y z w.
      int-lsbf-mod.add-mod (n + 2)
      (int-lsbf-mod.subtract-mod (n + 2) (take (n + 2) x) (take (n + 2) y))
      (int-lsbf-mod.subtract-mod (n + 2) (take (n + 2) z) (take (n + 2) w))
    )
    (γ ! 0) (γ ! 1) (γ ! 2) (γ ! 3);

  — residue mod F_n
  prim-root-exponent = (if odd m then 1 else 2);
  a-dft = int-lsbf-fermat.fft n prim-root-exponent (map (fill (2 ^ (n + 1))) a′);
  b-dft = int-lsbf-fermat.fft n prim-root-exponent (map (fill (2 ^ (n + 1))) b′);
  c-dft-odds = map2 (schoenhage-strassen n) (evens-odds False a-dft) (evens-odds False b-dft);

  c-diffs = int-lsbf-fermat.ifft n (prim-root-exponent ∗ 2) c-dft-odds;

  ξ′ = map2 (λcj j. int-lsbf-fermat.add-fermat n
      (int-lsbf-fermat.multiply-with-power-of-2 cj (2 ^ (n + 1) − (oe-n + prim-root-exponent ∗ j − 1)))
      (int-lsbf-fermat.from-nat-lsbf n (replicate (oe-n + 2 ^ n) False @ [True])))
    c-diffs [0..<2 ^ (oe-n − 1)];
  ξ = map (int-lsbf-fermat.reduce n) ξ′;

  — Combine the residues and construct the result
  z = map2 (solve-special-residue-problem n) ξ η;
  z-filled = map (fill (2 ^ (n − 1))) z;
  z-consts = replicate (2 ^ (oe-n − 1)) (replicate (oe-n + 2 ^ n) False @ [True]);
  z-sum = combine-z (2 ^ (n − 1)) (z-filled @ z-consts);
  result = int-lsbf-fermat.from-nat-lsbf m z-sum

  — return the resulting sum
 in result)
```

Lemma 6.18 [*schoenhage-strassen-correct′*]. *Let a and b be representations of numbers in $\mathbb{Z}_{F_m}$. Then, schoenhage-strassen m a b calculates $a \cdot b \in \mathbb{Z}_{F_m}$.*

Our previous runtime considerations yield a recursive runtime equation of the form

$$T(m) = \mathcal{O}\left(n \cdot 2^{2n} + (n \cdot 2^n)^{\log_2 3}\right) + 2^{n_{\mathrm{oe}}-1} \cdot T(n)$$

where $T(k)$ is the runtime of *schoenhage-strassen* m a b with $a, b \in \mathbb{Z}_{F_m}$ and n, n_{oe} are defined as in the beginning of section 6.4. In the Isabelle code, the function f is a more verbose recursive runtime bound (with less simplifications). Inserting the definitions of n and n_{oe} (and doing some simplifications, e.g. using that $(n \cdot 2^n)^{\log_2 3} \in \mathcal{O}\left(2^{2n}\right)$[13]) yields another runtime bound f' given by

$$f'(m) = \begin{cases} c_0 & \text{if } m < 3 \\ c_1 \cdot m \cdot 2^m + c_2 + 2^{\left\lfloor \frac{m+1}{2} \right\rfloor} \cdot f'\left(\left\lfloor \frac{m+2}{2} \right\rfloor\right) & \text{if } m \geq 3 \end{cases}$$

[13]Isabelle lemma *kar-aux-lem*.

with some constants $c_0, c_1, c_2 > 0$.[14] Hence, the only thing left to do for our runtime verification is to find a closed bound for f'.

This is done similarly as in [SS71]: Defining $\gamma_0 := 2c_1 + c_2$, one can show[15]

$$f'(2n-2) \le \gamma_0 \cdot n \cdot 2^{2n-2} + 2^{n-1} \cdot f'(n)$$
$$f'(2n-1) \le \gamma_0 \cdot n \cdot 2^{2n-1} + 2^{n} \cdot f'(n)$$

(for $n \ge 3$) and then, using induction on k:[16]

$$f'(m) \le \gamma \cdot k \cdot 2^{k+m} \tag{6.9}$$

for $k \ge 1$, $m \le 2^k + 1$ and $\gamma := \max\{\gamma_0, f'(0), f'(1), f'(2), f'(3)\}$.

Thus, we can choose $k \approx \log_2(m)$ and get a runtime bound of approximately[17]

$$\gamma \cdot \log_2(m) \cdot m \cdot 2^m.$$

Writing $l := 2^{m+1}$ for the length of the input numbers a and b, we have $m \approx \log_2(l)$ and hence a runtime bound of $O\left(\log_2(\log_2(l)) \cdot \log_2(l) \cdot l\right)$.

6.5 The Schönhage-Strassen-Algorithm in ℕ

In order to multiply any two numbers $a, b \in \mathbb{N}$, we just choose an $m \in \mathbb{N}$ such that

1. m is large enough so that $a \cdot b < F_m$ holds in $\mathbb{N}$, i.e. we can calculate the product by multiplying a and b in $\mathbb{Z}_{F_m}$
2. m is small enough so that the multiplication in $\mathbb{Z}_{F_m}$ is fast enough for our desired runtime bound.

The following choice of m suffices:

definition *schoenhage-strassen-mul* **where**
schoenhage-strassen-mul a b = (let m = max (bitsize (length a)) (bitsize (length b)) + 1 in
int-lsbf-fermat.reduce m (schoenhage-strassen m (fill (2 ^ (m + 1)) a) (fill (2 ^ (m + 1)) b))
)

Theorem 6.19 [*schoenhage-strassen-mul-correct*]. *Let $a, b \in \mathbb{N}$ be given in LSBF representation. Then, schoenhage-strassen-mul a b correctly calculates a representation of $a \cdot b \in \mathbb{N}$.*

Proof. Let $m := \max\{bitsize\ (length\ a), bitsize\ (length\ b)\} + 1$.

For $c \in \{a, b\}$, we have $length\ c < 2^{bitsize\ (length\ c)}$, and hence $length\ c < 2^{m-1}$ by definition of m. In particular, $length\ c \le 2^{m+1}$, so the length of *fill* $(2$ ^ $(m+1))$ c is precisely 2^{m+1}, i.e. $a' :=$ *fill* $(2$ ^ $(m+1))$ a and $b' :=$ *fill* $(2$ ^ $(m+1))$ b satisfy the assumption of Lemma 6.18.

Hence, *schoenhage-strassen m a′ b′* calculates a representation of $a' \cdot b' \in \mathbb{Z}_{F_m}$.
Applying *int-lsbf-fermat.reduce m* yields a result c where $c \equiv_{F_m} a' \cdot b' \equiv_{F_m} a \cdot b \in \mathbb{N}$ and $c < F_m$. If we can also show that $\mathbb{N} \ni a \cdot b < F_m$, we can conclude $c = a \cdot b \in \mathbb{N}$.

For that, note that $a < 2^{length\ a} < 2^{2^{m-1}}$ using that $length\ a < 2^{m-1}$ (similarly for b), and thus

$$a \cdot b < 2^{2^{m-1}} \cdot 2^{2^{m-1}} = 2^{2^m} < F_m.$$

□

Theorem 6.20 [*time-schoenhage-strassen-mul-tm-le, schoenhage-strassen-bound-bigo*]. *The implementation schoenhage-strassen-mul performs integer multiplication in $O\left(n \cdot \log_2(n) \cdot \log_2(\log_2(n))\right)$.*

More precisely: given input numbers a, b in LSBF representation with length $a \le n$ and length $b \le n$, schoenhage-strassen-mul a b needs $O\left(n \cdot \log_2(n) \cdot \log_2(\log_2(n))\right)$ bit operations.

[14] The Isabelle lemmas *time-schoenhage-strassen-tm-le* and *f-le-f′* show that f' is indeed a runtime bound.
[15] Isabelle lemma *f′-oe-rec*.
[16] Isabelle lemmas *f′-le-aux1* and *f′-le-aux2*.
[17] See the upcoming Theorem 6.20.

Proof. The calculation of m can be done in linear time, i.e. in $\mathcal{O}(n)$. Applying *fill* $(2 \text{ ^ } (m + 1))$ can be done in $\mathcal{O}\left(2^{m+1}\right)$. The runtime of the main part of the algorithm, namely the multiplication in $\mathbb{Z}_{F_m}$, can be estimated by

$$\gamma \cdot \mathit{bitsize}\ m \cdot 2^{\mathit{bitsize}\ m+m}$$

using equation (6.9) for $k := \mathit{bitsize}\ m$. Finally, applying the *int-lsbf-fermat.reduce* function can be done in linear time, i.e. in $\mathcal{O}(2^m)$.

Note that, using the assumptions and the monotonicity of the *bitsize*-function, we have $m \le \mathit{bitsize}\ n+1$. Hence, we get a total runtime in

$$\mathcal{O}\left(n + 2^m + \mathit{bitsize}\ m \cdot 2^{\mathit{bitsize}\ m+m}\right) = \mathcal{O}\left(n + 2^{\mathit{bitsize}\ n} + \mathit{bitsize}\ (\mathit{bitsize}\ n + 1) \cdot 2^{\mathit{bitsize}\ (\mathit{bitsize}\ n + 1)} \cdot 2^{\mathit{bitsize}\ n}\right).$$

The *bitsize*-function can now be estimated by the log-function as follows:[18]

$$\mathit{bitsize}\ n \le \log_2(n) + 1$$

Thus, the runtime bound can be rewritten to

$$\mathcal{O}\left(n + 2^{\log_2(n)} + \log_2(\log_2(n)) \cdot 2^{\log_2(\log_2(n))} \cdot 2^{\log_2(n)}\right) = \mathcal{O}\left(n \cdot \log_2(n) \cdot \log_2(\log_2(n))\right).$$

□

[18] Isabelle lemma *bitsize-le-log*.

7 Conclusion

We have given an implementation of the Schönhage-Strassen-Multiplication in Isabelle, as well as a detailed runtime verification using time monads. Along the way, we also implemented and verified the runtime of primitive operations and the Karatsuba-Multiplication on natural numbers given in binary representation. Further, we gave a commutative ring version of Number Theoretic Transforms and implemented the *estimation* tactic.

As next steps, the following would be possible:

- Specialize our NTT version to the context of finite fields, obtaining some of the proofs done by Ammer and Kreuzer [AK22] as consequence of the more general proofs.
- Similarly, specialize the NTT to the context of complex numbers, obtaining some proofs done by Ballarin [Bal05].
- Restructure some Isabelle proofs. Most importantly, the proofs of *fft-carrier*, *ifft-carrier*, *fft-correct′* and *ifft-correct′* contain many duplicate statements, e.g. that the lengths of the recursive arguments are again powers of 2, that could be outsourced to an adequate locale.
- Improve automation for the time verification process.

A Appendix

$a \equiv_n b$	$a \equiv b \mod n$	
$\mathbb{N}$	$\{0, 1, 2, \dots\}$	
$\mathbb{N}_{>0}$	$\{1, 2, 3, \dots\}$	
$\{a..<b\}$	$\{\, x \in \mathbb{N} \mid a \le x < b \,\}$	
$[P]$	$\begin{cases} 1 & \text{if } P \text{ holds} \\ 0 & \text{else} \end{cases}$	
$\delta_{i,j}$	$[i = j]$	
$f : \overline{\alpha} \to \beta$	$f : \alpha_1 \to \alpha_2 \to \cdots \to \alpha_n \to \beta$	if f is a function taking n curried arguments
F_n	$2^{2^n} + 1$	
$\sum X$	$\sum_{x \in X} x$	
$a \star b$	cyclic convolution of a and b, see Definition 5.4	

Table A.1 Overview of used notation.

Proof of Lemma 3.5. For simplicity, assume that *xs* and *ys* have the same length (otherwise, the reader may replace them by *fill-xs* resp. *fill-ys* in the following paragraphs). If *to-nat xs* ≤ *to-nat ys*, the claim follows by correctness of *compare-nat*. So, assume *to-nat xs* > *to-nat ys*. Note that adding *ys* to its complement results in a list consisting only of *True* bits:

$$\begin{array}{llc} & \textit{ys} & \boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{1} \\ + & \textit{map Not ys} & \boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0} \\ \hline = & & \boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1} \end{array}$$

This implies *to-nat* (*map Not ys*) = *2* ^ (*length ys*) − *1* − *to-nat ys*. Moreover, since *to-nat xs* > *to-nat ys*, it can be seen that adding *xs* to *map Not ys* results in an overflow bit. This bit can be cut off by applying *butlast*, effectively decreasing the result by *2* ^ (*length ys*). In total, we get:

$$\begin{aligned} & \textit{to-nat}\ (\textit{inc-nat}\ (\textit{butlast}\ (\textit{add-nat xs}\ (\textit{map Not ys})))) \\ & = 1 + \textit{to-nat}\ (\textit{butlast}\ (\textit{add-nat xs}\ (\textit{map Not ys}))) \\ & = 1 + \textit{to-nat xs} + (2\ \hat{}\ (\textit{length ys}) - 1 - \textit{to-nat ys}) - 2\ \hat{}\ (\textit{length ys}) \\ & = \textit{to-nat xs} - \textit{to-nat ys}. \end{aligned}$$

□

File	Description	Sections	SLOC
`Estimation_Method.thy`	Proof tactic for applying inequalities (with focus on type *nat*) in a forward-manner	1.2	38
`Preliminaries/`			
`Preliminaries.thy`	General Preliminaries		263
`Monoid_Sums.thy`	Finite sums in the context of abelian monoids and commutative rings	2.1	558
`Sum_Lemmas.thy`	Lemmas about the existing *sum-list*-function		545
`Ring_Lemmas.thy`	Auxiliary lemmas in commutative rings		221
`Binary_Representations/`			
`Abstract_Representations.thy`	Abstraction of the properties of a representation		127
`Abstract_Representations_2.thy`	More general abstraction of representations		115
`Binary_Representations.thy`	Binary Representations using the *nat-lsbf*-type; basic arithmetic operations	3	1891
`Binary_Representations-Runtime.thy`	Runtime formalization		1059
`Runtime_Lemmas.thy`	Specific lemma explicitly solving a recursive runtime inequality		106
`Karatsuba/`			
`Karatsuba.thy`	Karatsuba-Multiplication on natural numbers	4	430
`Karatsuba-Runtime.thy`	Runtime formalization		669
`NTT_Rings/`			
`NTT_Rings.thy`	Theory for Number Theoretic Transforms (NTTs) in rings: primitive roots of unity, convolution rule, inversion rule	5.1	914
`FNTT_Rings.thy`	Theory for Fast Number Theoretic Transforms (FNTTs) in rings	5.2	739
`Schoenhage_Strassen/`			
`Z_mod_power_of_2.thy`	Representations of and operations on $\mathbb{Z}_{2^k}$	6.1.1	248
`Z_mod_power_of_2-Runtime.thy`	Runtime formalization		113
`Z_mod_Fermat.thy`	Representations of and operations on $\mathbb{Z}_{F_k}$ (including FNTTs)	6.1.2, 6.2	2160
`Z_mod_fermat-Runtime.thy`	Runtime formalization		1414
`Schoenhage_Strassen.thy`	The Schönhage-Strassen-Algorithm (after some final preparations)	6.3, 6.4, 6.5	2419
`Schoenhage_Strassen-Runtime.thy`	Runtime formalization		2808
`Runtime_Lemmas_Landau.thy`	Auxiliary lemmas about Landau-Notation for functions of type $nat \Rightarrow nat$		162

For simplicity, SLOC refers to the total number of lines in the source file, including comments and empty lines.

Table A.2 Overview of file contents.

List of Figures

List of Tables

Bibliography

[AK22] T. Ammer and K. Kreuzer. “Number Theoretic Transform”. In: *Archive of Formal Proofs* (2022). https://isa-afp.org/entries/Number_Theoretic_Transform.html, Formal proof development. ISSN: 2150-914x.

[Bal05] C. Ballarin. “Fast Fourier Transform”. In: *Archive of Formal Proofs* (2005). https://isa-afp.org/entries/FFT.html, Formal proof development. ISSN: 2150-914x.

[Ebe15] M. Eberl. “The Akra-Bazzi theorem and the Master theorem”. In: *Archive of Formal Proofs* (2015). https://isa-afp.org/entries/Akra_Bazzi.html, Formal proof development. ISSN: 2150-914x.

[KO62] A. Karatsuba and Y. Ofman. “Multiplication of many-digital numbers by automatic computers”. In: *Dokl. Akad. Nauk SSSR* 145 (2 1962). http://mi.mathnet.ru/dan26729, pp. 293–294.

[Kem21] G. Kemper. *Computeralgebra.* University Lecture at the Technical University of Munich. Winter term. 2021/2022.

[NPW22] T. Nipkow, L. Paulson, and M. Wenzel. *A Proof Assistant for Higher-Order Logic.* https://isabelle.in.tum.de/doc/tutorial.pdf. Springer, 2022.

[Nip17] T. Nipkow. “Verified Root-Balanced Trees”. In: *Asian Symposium on Programming Languages and Systems, APLAS 2017.* Ed. by B.-Y. E. Chang. Vol. 10695. LNCS. https://www21.in.tum.de/~nipkow/pubs/aplas17.pdf. Springer, 2017, pp. 255–272.

[SS71] A. Schönhage and V. Strassen. “Schnelle Multiplikation großer Zahlen”. In: *Computing* 7 (1971), pp. 281–292.