Λ ◦ λ:
A Functional Library for Lattice Cryptography

Eric Crockett*  Chris Peikert†

November 24, 2015

Abstract

This work describes the design and implementation of Λ ◦ λ, a general-purpose software library for lattice cryptography, written in the functional and strongly typed language Haskell. In comparison with several prior implementations of lattice-based cryptographic schemes, Λ ◦ λ has several novel and distinguishing features, which include:

• Generality and modularity: Λ ◦ λ defines simple but general interfaces for the lattice cryptography “toolbox,” allowing for a wide variety of cryptographic schemes to be expressed very naturally and concisely. For example, we implement an advanced fully homomorphic encryption (FHE) scheme in as few as 2–5 lines of code per feature, via code that very closely matches the scheme’s mathematical definition.

• Parallelism: Λ ◦ λ automatically exploits multi-core parallelism, achieving nearly linear speedups per core. It also allows for the use of other parallel “backends” (e.g., based on GPUs or other specialized hardware), with no changes to application code.

• Theory affinity: Λ ◦ λ is designed from the ground-up around the specialized ring representations, fast algorithms, and worst-case hardness proofs that have been developed for the Ring-LWE problem and its cryptographic applications. In particular, Λ ◦ λ implements fast algorithms for sampling from theory-recommended error distributions over arbitrary cyclotomic rings, and provides tools for maintaining tight control of error growth in cryptographic schemes.

• Advanced features: Λ ◦ λ exposes the rich hierarchy of cyclotomic rings to cryptographic applications. We use this to give the first-ever implementation of a set of FHE operations collectively known as “ring switching,” and also describe a more efficient variant that we call “ring tunneling.”

Finally, we document a variety of perspectives, objects, and algorithms related to practical and theoretically sound usage of Ring-LWE in cyclotomic rings, which we believe will serve as a useful reference for future implementations.

* School of Computer Science, Georgia Institute of Technology.
† Department of Computer Science and Engineering, University of Michigan. Much of this work was done while the author was at the Georgia Institute of Technology. This material is based upon work supported by the National Science Foundation under CAREER Award CCF-1054495, by DARPA under agreement number FA8750-11-C-0096, by the Alfred P. Sloan Foundation, and by a Google Research Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, DARPA or the U.S. Government, the Sloan Foundation, or Google. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.
Contents

1 Introduction ................................................. 4
  1.1 Introducing $\Lambda \circ \lambda$ .................................. 4
  1.2 Why Haskell? ............................................. 5
  1.3 Overview and Paper Organization ......................... 6
  1.4 Limitations and Future Work .............................. 7

2 Integer and Modular Arithmetic ............................. 8
  2.1 Representing $\mathbb{Z}$ and $\mathbb{Z}_q$ ......................... 8
  2.2 Reduce and Lift ........................................... 9
  2.3 Rescale .................................................. 9
  2.4 CTRans .................................................. 10
  2.5 Gadgets .................................................. 11
  2.6 Type-Level Arithmetic for Cyclotomic Indices .......... 12

3 Cyclotomic Rings ............................................ 13
  3.1 Mathematical Background .................................. 13
  3.2 Safe Interface: cyc ....................................... 15
  3.3 Unsafe Interface: ucyc .................................... 17
  3.4 ucyc Implementation ...................................... 18
    3.4.1 Representations ...................................... 18
    3.4.2 Arithmetic Operations ................................ 18
    3.4.3 Promoting from Base Ring to Cyclotomics .......... 19

4 Homomorphic Encryption in $\Lambda \circ \lambda$ ............... 20
  4.1 Keys, Plaintexts, and Ciphertexts ......................... 21
  4.2 Encryption and Decryption ................................ 22
  4.3 Homomorphic Addition and Multiplication ............... 23
  4.4 Modulus Switching ....................................... 24
  4.5 Key Switching and Linearization ........................ 24
  4.6 Ring Tunneling .......................................... 26

A Haskell Background ......................................... 32
  A.1 Types .................................................. 32
  A.2 Type Classes ............................................ 33

B More on Type-Level Cyclotomic Indices ...................... 34
  B.1 Promoting Factored Naturals .............................. 34
  B.2 Applying the Promotions ................................ 35

C Sparse Decompositions and Haskell Framework .............. 35
  C.1 Sparse Decompositions ................................... 36
  C.2 Haskell Framework ....................................... 36
## D Tensor Interface and Implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1 Mathematical Background</td>
<td>38</td>
</tr>
<tr>
<td>D.1.1 Cyclotomic Rings and Powerful Bases</td>
<td>38</td>
</tr>
<tr>
<td>D.1.2 (Tweaked) Trace, Dual Ideal, and Decoding Bases</td>
<td>41</td>
</tr>
<tr>
<td>D.1.3 Chinese Remainder Bases</td>
<td>43</td>
</tr>
<tr>
<td>D.2 Single-Index Transforms</td>
<td>44</td>
</tr>
<tr>
<td>D.2.1 Prime-Power Factorization</td>
<td>44</td>
</tr>
<tr>
<td>D.2.2 Embedding Scalars</td>
<td>44</td>
</tr>
<tr>
<td>D.2.3 Converting Between Powerful and Decoding Bases</td>
<td>44</td>
</tr>
<tr>
<td>D.2.4 Multiplication by $g_m$</td>
<td>45</td>
</tr>
<tr>
<td>D.2.5 Chinese Remainder and Discrete Fourier Transforms</td>
<td>46</td>
</tr>
<tr>
<td>D.2.6 Generating (Tweaked) Gaussians in the Decoding Basis</td>
<td>46</td>
</tr>
<tr>
<td>D.3 Two-Index Transforms and Values</td>
<td>48</td>
</tr>
<tr>
<td>D.3.1 Prime-Power Factorization</td>
<td>48</td>
</tr>
<tr>
<td>D.3.2 Coefficients in Relative Bases</td>
<td>48</td>
</tr>
<tr>
<td>D.3.3 Embed Transforms</td>
<td>48</td>
</tr>
<tr>
<td>D.3.4 Twace Transforms</td>
<td>49</td>
</tr>
<tr>
<td>D.4 CRT Sets</td>
<td>50</td>
</tr>
<tr>
<td>D.4.1 Mathematical Background</td>
<td>50</td>
</tr>
<tr>
<td>D.4.2 Computing CRT Sets</td>
<td>51</td>
</tr>
</tbody>
</table>

## E Tensor Product of Rings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.1 Tensor Product of Rings</td>
<td>53</td>
</tr>
</tbody>
</table>
1 Introduction

Lattice-based cryptography has seen enormous growth over the past decade, due to attractive features like apparent resistance to quantum attacks; good efficiency and parallelism, especially via the use of algebraically structured lattices arising from rings (e.g., \cite{HPS98,Mic02,LPRT10}); and versatile cryptographic constructions like identity-based, attribute-based, and fully homomorphic encryption (e.g., \cite{GPV08,Gen09,BGV12,GSW13,GVW13,BGG14}).

The past few years have seen a movement toward the practical implementation of lattice-based schemes, with an impressive array of results. To date, each such implementation has been specialized to a particular cryptographic primitive (and sometimes even to a specific computational platform), e.g., collision-resistant hashing (using SIMD instruction sets) \cite{LMPR08}, digital signatures \cite{GLP2012,DDLL13}, fully homomorphic encryption (FHE) \cite{NLV11,HS} (using GPUs and FPGAs \cite{WHC12,CGRS14}), and key-establishment protocols \cite{BCNS15,ADPS15}. However, these systems share little common ground in their interfaces and implementations, and it is not easy to adapt them to the many other kinds of lattice-based constructions.

1.1 Introducing $\Lambda \circ \lambda$

This work describes the design and implementation of $\Lambda \circ \lambda$, a general-purpose software library for lattice-based cryptography, written in the functional, strongly typed programming language Haskell.\footnote{The name $\Lambda \circ \lambda$ refers to the combination of lattices and functional programming, which are often signified by $\Lambda$ and $\lambda$, respectively. The recommended pronunciation is “L O L.”} As with prior implementations, our main focus is on cryptosystems defined over cyclotomic rings, because they lie at the heart of efficient lattice-based cryptography (see, e.g., \cite{HPS98,Mic02,LPRT10,LPRT13}). However, $\Lambda \circ \lambda$ has several novel properties that distinguish it in scope and functionality from prior implementations, as we now discuss.

Generality, modularity, and concision: $\Lambda \circ \lambda$ defines a collection of simple but general interfaces and implementations for the lattice cryptography "toolbox," i.e., the handful of core operations that are shared across a wide variety of modern cryptographic constructions, from basic encryption and authentication primitives to advanced homomorphic and attribute-based systems. This allows cryptographic schemes to be expressed rather easily and naturally in $\Lambda \circ \lambda$, via code that closely mirrors their mathematical definitions. For example, we implement a full-featured FHE scheme in as few as 2–5 lines of code per feature.

In addition, $\Lambda \circ \lambda$ supports arbitrary cyclotomic rings. By contrast, most prior implementations are limited to the narrow subclass of two-power cyclotomics (which are algorithmically the simplest case). In $\Lambda \circ \lambda$, all cyclotomic rings are on “equal footing,” i.e., it is easy to implement cryptographic schemes generically, and then instantiate them to work in any satisfactory cyclotomic. We point out that many advanced techniques in ring-based cryptography, such as “plaintext packing” and homomorphic SIMD operations \cite{SV10,SV11}, inherently require non-two-power cyclotomics when using characteristic-two plaintext spaces like $\mathbb{F}_{2^k}$.

Performance and parallelism: in our preliminary experiments, $\Lambda \circ \lambda$ delivers performance in the same league as that of specialized implementations in lower-level languages like C/C++, for comparable cryptographic applications. In addition, $\Lambda \circ \lambda$ automatically exploits multi-core parallelism, providing near-linear speedups in the number of cores. Other “backends,” e.g., based on specialized hardware like GPUs, can also be implemented and easily plugged in without requiring any changes to application code.

\footnote{$\Lambda \circ \lambda$ is available on Hackage, the Haskell community’s central repository, and may be installed via cabal install lol. The latest version is also available at \url{https://github.com/cpeikert/lol}}

1
Theory affinity: \( \Lambda \circ \lambda \) is designed from the ground-up around the specialized ring representations, fast algorithms, and worst-case hardness proofs developed in \([\text{LPR10}, \text{LPR13}]\) for the design and analysis of Ring-LWE-based cryptosystems (in arbitrary cyclotomic rings). To our knowledge, \( \Lambda \circ \lambda \) is the first implementation of these techniques, which include:

- fast and modular algorithms for converting among the three most useful representations of ring elements, namely, those corresponding to the powerful, decoding, and Chinese Remainder Theorem (CRT) bases;
- fast algorithms for sampling from “theory-recommended” error distributions over rings—i.e., those for which the Ring-LWE problem enjoys provable worst-case hardness—for use in encryption and related operations;
- proper use of the powerful- and decoding-basis representations to maintain tight control of error growth under various cryptographic operations, and for the best error tolerance in decryption.

We especially emphasize the importance of using appropriate error distributions for Ring-LWE, because ad-hoc instantiations that are not supported by worst-case hardness proofs can turn out to be completely insecure (see, e.g., \([\text{ELOS15}, \text{CLS15}]\)).

Advanced features: \( \Lambda \circ \lambda \) exposes the rich hierarchy of cyclotomic rings, by making subring and extension-ring relationships accessible to cryptographic applications. Building on this, \( \Lambda \circ \lambda \) also provides the first implementation of a set of homomorphic operations collectively known as ring-switching \([\text{BGV12}, \text{GHPS12}, \text{AP13}]\). Ring-switching enables the homomorphic evaluation of certain structured linear transforms, which has applications to, e.g., asymptotically efficient “bootstrapping” algorithms for FHE \([\text{AP13}]\). In more detail:

- We document and implement a variety of important objects, linear transforms, and fast algorithms related to subring and extension-ring relations on cyclotomics. In particular, we describe simple linear-time algorithms for the core embed and “tweaked” trace operations in the three main bases of interest (powerful, decoding, and CRT), and for computing the relative analogues of these bases for cyclotomic extension rings.
- We describe and implement a more efficient variant of ring-switching, which we call ring tunneling. While the prior technique from \([\text{AP13}]\) “hops” from one ring to another through a common extension ring to evaluate a linear function, our new approach “tunnels” through a common subring, which makes it more efficient. In addition, we show how the evaluated linear function can be integrated into the accompanying key-switching step, thus unifying two operations into a simpler and even more efficient one.

1.2 Why Haskell?

Haskell has several properties that make it an excellent match for our goals. These include:

1. Elegant, functional syntax: Haskell’s syntax is very mathematical, which yields a close match between the definitions of lattice operations and their implementations in code.

2. Purity (“no side effects”): By default, computations cannot mutate state or otherwise modify their environment, so invoking a function on the same input always produces the same output. This makes code easier to reason about and test, and is a natural fit for the kinds of mathematical operations used in lattice cryptography. “Effectful” computations (e.g., those performing input/output or using random numbers) are still possible, but must be embedded in a structure that precisely delineates what effects are allowed. This likewise enforces discipline and eases analysis, leading to more reliable code.
3. **Strong, static typing**: Haskell is statically typed, i.e., every expression has a type that can be checked for validity at compile time. This catches many common classes of programming errors very early on, making for safer code. Static typing can also yield faster programs, by eliminating the need for many runtime checks and enabling other type-specific optimizations. Finally, Haskell’s type system lets the programmer express rich *constraints* on types, ensuring that only legal and meaningful expressions typecheck. For example, \( \Lambda \circ \lambda \) uses such constraints to restrict certain operations to valid subrings or extension rings.

4. **Power and concision**: Haskell natively supports many powerful abstractions like higher-order functions, functors and monads, and embedded domain-specific languages (DSLs). These allow the programmer to express computations at a high level of abstraction and modularity. For example, we use these tools in \( \Lambda \circ \lambda \) to concisely express a variety of important linear transformations in terms of their “sparse decompositions,” and to automatically derive corresponding fast algorithms.

5. **Performance and parallelism**: Well-crafted Haskell programs tend to run more efficiently than those written in other high-level languages. In some cases, compiled Haskell code can even be as fast or faster than hand-tuned C code (see, e.g., [Ste08]). Haskell also has substantial library support for expressing data-parallel computations (e.g., [KCL + 10, CKL + 11]), especially the “embarrassingly parallel” ones that are abundant in lattice cryptography.

For the reader who is new to Haskell, in Appendix A we give a brief tutorial that provides sufficient background to understand the code fragments appearing in this paper.

1.3 **Overview and Paper Organization**

The components of \( \Lambda \circ \lambda \) are arranged in a few main layers of interfaces and implementations; the remainder of this paper dedicates a section to each one in turn. From the bottom up, they are:

**Integer layer (Section 2 and Appendix B)**: This layer contains interfaces and implementations for domains like the integers \( \mathbb{Z} \) and its quotient rings \( \mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z} \). This includes specialized operations like rescaling and “(bit) decomposition,” which are used across a wide variety of cryptographic schemes. This layer also contains tools for representing and operating on moduli and cyclotomic indices at the *type level*, which enable static (compile-time) verification that all operations are mathematically valid.

**Tensor layer (Appendices C and D)**: This layer’s main interface, called `Tensor`, encapsulates all the “back-end” linear transformations and special values needed for working efficiently in cyclotomic rings, building on the mathematical framework developed in [LPR13]. So far, \( \Lambda \circ \lambda \) provides two implementations of `Tensor`, one in C and the other in pure Haskell. The latter is built upon the `repa` library for parallel array operations [KCL + 10, LCKJ12], together with a custom domain-specific language for expressing “sparse decompositions” of linear transforms (see Appendix C). Because the tensor layer is completely hidden from typical cryptographic applications, we defer to Appendix D the details of its design and implementation, which include several supporting linear transforms and algorithms that have not previously appeared in the literature.

**Cyclotomic layer (Section 3)**: This layer defines data types and interfaces that represent cyclotomic rings and their cryptographically relevant operations (including functions that map between different rings). Our implementation is essentially a thin wrapper around `Tensor`, which automatically manages the internal representations of ring elements to make operations as efficient as possible.

---

\(^3\)A common joke about Haskell code is “if it compiles, it must be correct.”
Cryptography layer (Section 4): This layer consists of any implementations of cryptographic schemes that rely upon the lower layers. As a detailed example, we describe an advanced Ring-LWE-based FHE scheme that unifies and refines a broad collection of features from a long series of works \cite{LPR10, BV11a, BV11b, BGV12, GHS12, GHPS12, AP13}. We also demonstrate how its implementation in $\Lambda \circ \lambda$ very closely and concisely matches its mathematical definition.

1.4 Limitations and Future Work

Security. While $\Lambda \circ \lambda$ has many attractive functionality and safety features, we stress that it is still an early-stage research prototype, and is not yet recommended for production purposes—especially in scenarios where security is vital. Potential security issues include, but may not be limited to:

- Most functions in $\Lambda \circ \lambda$ are not constant time, and may therefore leak secret information via timing or other side channels. (Protecting lattice cryptography from side-channel attacks is an important area for further research.)
- While $\Lambda \circ \lambda$ implements a fast algorithm for sampling from theory-recommended error distributions, the current implementation is somewhat naïve in terms of precision. By default, we use double-precision floating-point arithmetic to approximate a sample from a continuous Gaussian, before rounding. We have not yet analyzed the associated security implications, if any. (We note that Ring-LWE is robust to small variations in the error distribution, as shown in, e.g., \cite[Section 5]{LPR10}.)

Discrete Gaussian sampling. Many lattice-based cryptosystems, such as digital signatures and identity-based or attribute-based encryption schemes following \cite{GPV08}, require sampling from a discrete Gaussian probability distribution over a given lattice coset, using an appropriate kind of “trapdoor.” Supporting this operation in $\Lambda \circ \lambda$ is left to future work, for the following reasons. While it is straightforward to give a clean interface for discrete Gaussian sampling (similar to the Decompose class described in Section 2.5), providing a secure and practical implementation is very subtle, especially for arbitrary cyclotomic rings: one needs to account for the non-orthogonality of the standard bases, use practically efficient algorithms, and ensure sufficient fidelity to the desired distribution using only finite precision. Although there has been good progress in addressing these issues (see, e.g., \cite{DN12, LPR13, DP15b, DP15a}), obtaining a complete practical solution still requires further research.

Language layer. Rich lattice-based cryptosystems, especially homomorphic encryption, involve a large number of tunable parameters and different routes to the user’s end goal. In current implementations, merely expressing a homomorphic computation requires expertise in the intricacies of homomorphic encryption itself, and of its particular implementation. For future work, we envision domain-specific languages (DSLs) that allow the programmer to express a desired plaintext computation at a reasonably high level above the “native instruction set” of the homomorphic encryption scheme. A specialized compiler would then translate the user’s description into a homomorphic computation (on ciphertexts) using the cryptosystem’s instruction set, and possibly even instantiate secure parameters for it. Because Haskell is an excellent host language for embedded DSLs, we believe that $\Lambda \circ \lambda$ will serve as a strong foundation for such tools.

Applications. For future work, we envision implementations of a wide variety of other lattice-based cryptosystems in $\Lambda \circ \lambda$. Apart from digital signatures and identity/attribute-based encryption (which use discrete Gaussian sampling), other primitives that can be implemented using $\Lambda \circ \lambda$’s existing interfaces include:
standard Ring-LWE-based \cite{LPR10,LPR13} and NTRU-style encryption \cite{HPS98,SS11}, encryption with security under chosen-ciphertext attacks (e.g., from \cite{MP12}), and pseudorandom functions (PRFs) \cite{BPR12,BLMR13,BP14}. Using the above-mentioned language layer, we also plan to implement a simple and natural homomorphic evaluation of lattice-based PRFs, in the same spirit as prior homomorphic evaluations of AES \cite{GHS12,CLT14}.

**Whole-program optimization.** Currently, \(\Lambda \circ \lambda\)’s implementation of cyclotomic rings uses a heuristic to decide which representation will be most efficient for upcoming operations (e.g., it prefers the Chinese remainder representation, which is best for the most commonly invoked operations). It also allows the client to provide “advice” about the representation, which can prevent duplicate or unnecessary transformations in certain contexts. However, the heuristic is limited because it lacks information about the surrounding computation, and client-specified advice is error prone and requires the application programmer to have specialized knowledge. For future work, we envision a two-phase process that first “lazily” builds a description of the entire computation from the client code, then analyzes it to determine the best representation for each intermediate value before actually executing. Such a two-phase design is typical for embedded languages, and is easy to implement in Haskell.

## 2 Integer and Modular Arithmetic

At its core, lattice-based cryptography is built around arithmetic in the ring of integers \(\mathbb{Z}\) and quotient rings \(\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}\) of integers modulo \(q\), i.e., the cosets \(x + q\mathbb{Z}\) with the usual addition and multiplication operations. In addition to these standard ring operations, a variety of specialized operations are also widely used, e.g., “lifting” a coset in \(\mathbb{Z}_q\) to its smallest representative in \(\mathbb{Z}\), “rescaling” one quotient ring \(\mathbb{Z}_q\) to another, and “decomposing” a \(\mathbb{Z}_q\)-element as a vector of small \(\mathbb{Z}\)-elements.

Here we recall the relevant mathematical background for all these domains and operations, and describe how they are represented and implemented in \(\Lambda \circ \lambda\). This will provide a foundation for the next section, where we show how all these operations are very easily “promoted” from base rings (like \(\mathbb{Z}\) and \(\mathbb{Z}_q\)) to cyclotomic rings, to support ring-based cryptosystems. (Similar promotions can also easily be done to support plain-LWE/SIS systems, but we elect not to do so in our library, mainly because those systems are not as practically efficient.)

### 2.1 Representing \(\mathbb{Z}\) and \(\mathbb{Z}_q\)

We use fixed-precision primitive Haskell types like \texttt{Int} and \texttt{Int64} to represent the integers \(\mathbb{Z}\), and define our own specialized types like \texttt{ZqBasic q z} to represent \(\mathbb{Z}_q\). Here the \(q\) parameter is a “phantom” type that represents the value of the modulus \(q\), while \(z\) is an integer type (like \texttt{Int64}) specifying the underlying representation of the integer residues modulo \(q\).

This approach has multiple advantages: by defining \texttt{ZqBasic q z} as an instance of \texttt{Ring}, we can use \((+)\) and \((*)\) as usual without needing to write any explicit modular reductions. More importantly, at compile time the type system disallows operations on incompatible types—e.g., attempting to add a \texttt{ZqBasic q1 z} to a \texttt{ZqBasic q2 z} for distinct \(q1\), \(q2\)—with no runtime overhead. Finally, we implement \texttt{ZqBasic q z} as a \texttt{newtype} for \(z\), which means that these types have identical representations, and there is no runtime overhead associated with converting between them.
CRT/RNS representation. Some applications, like homomorphic encryption, can require moduli \( q \) that are too large for standard fixed-precision integer types. Many languages have support for unbounded integers (e.g., Haskell’s \( \text{Integer} \) type), but the operations are relatively slow. Moreover, the values have varying sizes, which means they cannot be stored efficiently in “unboxed” form in arrays. A standard solution is to use the Chinese Remainder Theorem (CRT), also known as Residue Number System (RNS), representation: choose \( q \) to be the product of several pairwise coprime (and sufficiently small) \( q_1, \ldots, q_t \), so that we have a ring isomorphism between \( \mathbb{Z}_q \) and the product ring \( \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_t} \), where addition and multiplication are both component-wise.

In Haskell, using the CRT representation—and more generally, working in product rings—is very natural using the generic pair type \( \mathbb{H}L\mathbb{I} \): we simply adopt the convention that whenever types \( a \) and \( b \) respectively represent rings \( A \) and \( B \), the pair type \( (a, b) \) represents the product ring \( A \times B \). This just requires defining the obvious instances of \( \text{Additive} \) and \( \text{Ring} \) for \( (a, b) \)—which in fact has already been done for us by the numeric prelude. Products of more than two rings are immediately supported by nesting pairs, e.g., \( ((a, b), c) \), or by using higher-arity tuples like \( (a, b, c) \). A final nice feature is that a pair (or tuple) has fixed representation size as long as all its components do, so arrays of pairs can be stored directly in “unboxed” form, without requiring an extra layer of indirection.

2.2 Reduce and Lift

Two basic, widely used operations are reducing a \( \mathbb{Z} \)-element to its residue class in \( \mathbb{Z}_q \), and lifting a \( \mathbb{Z}_q \)-element to its smallest integer representative, i.e., in \( \mathbb{Z} \cap [-\frac{q}{2}, \frac{q}{2}) \). These operations are examples of the natural homomorphism, and canonical representative map, for arbitrary quotient groups. Therefore, we define class \( \text{Additive} \ a \ \text{Additive} \ b \ \Rightarrow \text{Reduce} \ a \ b \) to represent that \( b \) is a quotient group of \( a \), and class \( \text{Reduce} \ a \ b \ \Rightarrow \text{Lift} \ b \ a \) for computing canonical representatives.\(^4\) These classes respectively introduce the functions

\[
\text{reduce} :: \text{Reduce} \ a \ b \Rightarrow a \rightarrow b \\
\text{lift} :: \text{Lift} \ b \ a \Rightarrow b \rightarrow a
\]

where \( \text{reduce} \circ \text{lift} \) should be the identity function.

Instances of these classes are straightforward. We define an instance \( \text{Reduce} \ z \ (\mathbb{Z}q\text{Basic} \ q \ z) \) for any suitable integer type \( z \) and \( q \) representing a modulus that fits within the precision of \( z \), and a corresponding instance for \( \text{Lift} \). For product groups (pairs) used for CRT representation, we define the obvious instance \( \text{Reduce} \ a \ (b1, b2) \) whenever we have instances \( \text{Reduce} \ a \ b1 \) and \( \text{Reduce} \ a \ b2 \). However, we do not have (nor do we need) a corresponding \( \text{Lift} \) instance, because there is no sufficiently generic algorithm to combine canonical representatives from two quotient groups.

2.3 Rescale

Another operation commonly used in lattice cryptography is rescaling (sometimes also called rounding) \( \mathbb{Z}_q \) to a different modulus. Mathematically, the rescaling operation \( \lfloor \cdot \rceil_q : \mathbb{Z}_q \rightarrow \mathbb{Z}_{q'} \) is defined as

\[
\lfloor x + q\mathbb{Z} \rfloor_{q'} := \lfloor \frac{q'}{q} \cdot (x + q\mathbb{Z}) \rfloor = \left\lfloor \frac{q'}{q} \cdot x \right\rfloor + q'\mathbb{Z} \in \mathbb{Z}_{q'},
\]

\(^4\)Precision issues prevent us from merging \( \text{Lift} \) and \( \text{Reduce} \) into one class. For example, we can reduce an \( \text{Int} \) into \( \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \) if both components can be represented by \( \text{Int} \), but lifting may cause overflow.
where \( \lfloor \cdot \rfloor \) denotes rounding to the nearest integer. (Notice that the choice of representative \( x \in \mathbb{Z} \) has no effect on the result.) In terms of the additive groups, this operation is at least an “approximate” homomorphism: \( \lfloor x + y \rfloor_{q'} \approx \lfloor x \rfloor_{q'} + \lfloor y \rfloor_{q'} \), with equality when \( q | q' \). We represent the rescaling operation via

\[
\text{class } \text{Additive } a, \text{ Additive } b \Rightarrow \text{Rescale } a \ b, \text{ which introduces the function }
\]

\[
\text{rescale :: Rescale } a \ b \Rightarrow a \to b
\]

**Instances.** A straightforward instance, whose implementation just follows the mathematical definition, is \text{Rescale} \ (\mathbb{Z}q_{\text{Basic}} q_{1} z) (\mathbb{Z}q_{\text{Basic}} q_{2} z) \) for any integer type \( z \) and types \( q_{1}, q_{R} \) representing moduli that fit within the precision of \( z \).

More interesting are the instances involving product groups (pairs) used for CRT representation. A naïve implementation would apply Equation (2.1) to the canonical representative of \( x + q \mathbb{Z} \), but for large \( q \) this would require unbounded-integer arithmetic. Instead, following ideas from [GHS12], here we describe algorithms that avoid this drawback.

To “scale up” \( x \in \mathbb{Z}_{q_{1}} \) to \( \mathbb{Z}_{q_{1} q_{2}} \approx \mathbb{Z}_{q_{1}} \times \mathbb{Z}_{q_{2}} \) where \( q_{1} \) and \( q_{2} \) are coprime, i.e., to multiply by \( q_{2} \), simply output \( \langle x \cdot q_{2} \mod q_{1}, 0 \rangle \). This translates easily into code that implements the instance \text{Rescale} \ a \ (a, b). Notice, though, that the algorithm uses the value of the modulus \( q_{2} \) associated with \( b \). We therefore require \( b \) to be an instance of \text{class Mod}, which exposes the modulus value associated with the instance type. The instance \text{Rescale} \ b \ (a, b) \) works symmetrically.

To “scale down” \( x = (x_{1}, x_{2}) \in \mathbb{Z}_{q_{1}} \times \mathbb{Z}_{q_{2}} \approx \mathbb{Z}_{q_{1} q_{2}} \) to \( \mathbb{Z}_{q_{1}} \), we essentially need to divide by \( q_{2} \), discarding the (signed) remainder. To do this,

1. Compute the canonical representative \( \bar{x}_{2} \in \mathbb{Z} \) of \( x_{2} \).
   
   (Observe that \( \langle x'_{1} = x_{1} - (\bar{x}_{2} \mod q_{1}), 0 \rangle \in \mathbb{Z}_{q_{1}} \times \mathbb{Z}_{q_{2}} \) is the multiple of \( q_{2} \) closest to \( x = (x_{1}, x_{2}) \).)

2. Divide by \( q_{2} \), outputting \( q_{2}^{-1} \cdot x'_{1} \in \mathbb{Z}_{q_{1}} \).

The above easily translates into code that implements the instance \text{Rescale} \ (a, b) \ a, using the \text{Lift} and \text{Reduce} classes described above. The instance \text{Rescale} \ (a, b) \ b \) works symmetrically.

### 2.4 CRTrans

Fast multiplication in cyclotomic rings is enabled by converting ring elements to the Chinese remainder representation, using the Chinese Remainder Transform (CRT) over the base ring. This is an invertible linear transform akin to the Discrete Fourier Transform (over \( \mathbb{C} \)) or the Number Theoretic Transform (over appropriate \( \mathbb{Z}_{q} \)), which has a fast algorithm corresponding to its “sparse decomposition” (see Appendix D.2.5 and [LPR13, Section 3] for further details).

Applying the CRT and its inverse requires knowledge of certain roots of unity, and the inverse of a certain integer, in the base ring. For this purpose, we define \text{class} \ Ring \ r \Rightarrow \text{CRTrans} \ r, \text{ which exposes the necessary information for a base ring} \ r:

\[
\text{type CRTInfo } r = (\text{Int} \to r, r)
\]

\[
crtInfo :: \text{CRTrans } r \Rightarrow \text{Int} \to \text{Maybe } (\text{CRTInfo } r)
\]

The function \text{crtInfo}, given an integer \( m \), outputs the information required to apply and invert the index-\( m \) CRT over \( r \) (note that because the CRT may not exist for certain \( m \), the output type is a \text{Maybe}). The information consists of two components: (1) a function that, given an integer exponent \( i \), returns the \( i \)th
power of a certain principal \( m \)th root of unity \( \omega_m \) in \( r \), and (2) the multiplicative inverse of \( \hat{m} \) in \( r \), where \( \hat{m} = m/2 \) if \( m \) is even, else \( \hat{m} = m/2 \).

We give nontrivial instances of \( \text{CRTTrans} \) for \( \text{ZqBasic q z} \) (representing \( \mathbb{Z}_q \)) for prime \( q \), and for \textbf{Complex Double} (representing \( \mathbb{C} \)). In addition, because we use tensors and cyclotomic rings over base rings like \( \mathbb{Z} \) and \( \mathbb{Q} \), we also need to define trivial instances of \( \text{CRTTrans} \) for \textbf{Int}, \textbf{Int64}, \textbf{Double}, etc., for which \textbf{crtInfo} always returns \textbf{Nothing}.

2.5 Gadgets

Many advanced lattice cryptosystems use special objects called \textit{gadgets} \cite{MP12}, which support certain operations as described below. For the purposes of this work, a gadget is a tuple over a quotient ring \( R_q = R/qR \), where \( R \) is a ring that admits a meaningful “geometry.” For concreteness, one can think of \( R \) as merely being the integers \( \mathbb{Z} \), but later on we generalize to cyclotomic rings.

Perhaps the simplest gadget is the powers-of-two vector \( g = (1, 2, 4, 8, \ldots, 2^{\ell-1}) \) over \( \mathbb{Z}_q \), where \( \ell = \lceil \log q \rceil \). There are many other ways of constructing gadgets, either “from scratch” or by combining gadgets. For example, one may use powers of integers other than two, mixed products, the Chinese Remainder Theorem, etc. The salient property of a gadget \( g \) is that it admits efficient algorithms for the following tasks:

1. Decomposition: given \( u \in R_q \), output a short vector \( x \) over \( R \) such that \( \langle g, x \rangle = g^t \cdot x \equiv u \pmod{q} \).

2. Error correction: given a “noisy encoding” of the gadget \( b^t = s \cdot g^t + e^t \pmod{q} \), where \( s \in R_q \) and \( e \) is a sufficiently short error vector over \( R \), output \( s \).

A key property is that decomposition and error-tolerant encoding relate in the following way (where the notation is as above, and \( \approx \) hides a short error vector over \( R \)):

\[
s \cdot u = (s \cdot g^t) \cdot x \approx b^t \cdot x \pmod{q}.
\]

We represent gadget vectors and their associated operations via the following classes:

```haskell
class Ring u => Gad get gad u where
gadget :: Tagged gad [u]
encode :: u -> Tagged gad [u]

class (Gad get gad u, Reduce r u) => Decompose gad u r where
decompose :: u -> Tagged gad [r]

class Gad get gad u => Correct gad u where
correct :: Tagged gad [u] -> u
```

The class \textbf{Gadget} \( gad \ u \) says that the ring \( u \) supports a gadget vector indexed by the type \( gad \); the gadget vector itself is given by the term \textit{gadget}. Note that its type is actually \textbf{Tagged} \( gad \ [u] \): this is a \textbf{newtype} for \( [u] \), with the additional type-level context \textbf{Tagged} \( gad \) indicating which gadget the vector represents (recall that there are many possible gadgets over a given ring). This tagging aids safety, by preventing the nonsensical mixing of values associated with different kinds of gadgets. In addition, Haskell provides

---

5A principal \( m \)th root of unity in \( r \) is an element \( \omega_m \) such that \( \omega_m^m = 1 \), and \( \omega_m^{m/t} - 1 \) is not a zero divisor for every prime \( t \) dividing \( m \). Along with the invertibility of \( \hat{m} \) in \( r \), these are sufficient conditions for the index-\( m \) CRT over \( r \) to be invertible.
generic ways of “promoting” ordinary operations to work within this extra context. (Formally, this is because
**Tagged** _gad_ is an instance of the **Functor** class.)

The class **Decompose** _gad_ _u_ _r_ says that a _u_-element can be decomposed into a vector of _r_-elements
(with respect to the gadget index by _gad_), via the **decompose** method. The class **Correct** _gad_ _u_ says that a
noisy encoding of a _u_-element (with respect to the gadget) can be corrected, via the **correct** method.

Note that we split the above functionality into three separate classes, both because their arguments are
slightly different (e.g., **Correct** has no need for the _r_ type), and because in some cases we have meaningful
instances for some classes but not others.

**Instances.** For our type **ZqBasic** _q_ _z_ representing _Z_q_, we give a straightforward instantiation of the “base-
b” gadget _g_ = (1, _b_, _b^2_, ...) and error correction and decomposition algorithms, for any positive integer _b_
(which is represented as a parameter to the gadget type). In addition, we implement the trivial gadget
_g_ = (1) ∈ _Z_1_q_, where the decomposition algorithm merely outputs the canonical _Z_-representative of its
_Z_q_-input. This gadget turns out to be useful for building nontrivial gadgets and algorithms for product rings,
as described next.

For the pair type (which, to recall, we use to represent product rings in CRT representation), we give
instances of **Gadget** and **Decompose** that work as follows. Suppose we have gadget vectors _g_1, _g_2 over
_R_q_1, _R_q_2, respectively. Then the gadget for the product ring _R_q_1 × _R_q_2 is essentially the concatenation
of _g_1 and _g_2, where we first attach 0 ∈ _R_q_2 components to the entries of _g_1, and similarly for _g_2. The
decomposition of _u_ ∈ _R_q_1 × _R_q_2 with respect to this gadget is the concatenation of the decompositions
of _u_1, _u_2. All this translates easily to the implementations

```haskell
gadget = (++ <$> (map (_,zero) <$> gadget) <*> (map (zero,) <$> gadget)

decompose (a,b) = (++ <$> decompose a <*> decompose b
```

In the definition of gadget, the two calls to map attach zero components to the entries of _g_1, _g_2, and (++)
appends the two lists. (The syntax <$>, <*> is standard applicative notation, which promotes normal functions
into the **Tagged** _gad_ context.)

### 2.6 Type-Level Arithmetic for Cyclotomic Indices

As discussed in the Section below, there is one cyclotomic ring for every positive integer index _m_. (The
index is also sometimes called the **conductor**.) The index _m_, and in particular its factorization, plays a major
role in the definitions of the ring operations. For example, the index- _m_ “Chinese remainder transform” is
similar to a mixed-radix FFT, where the radices are the prime divisors of _m_. In addition, cyclotomic rings
can sometimes be related to each other based on their indices. For example, the _m_-th cyclotomic can be seen
as a subring of the _m_’th cyclotomic if and only if _m_ | _m_’; the largest common subring of the _m_1-th and _m_2-th
cyclotomics is the _gcd(m_1, _m_2)_th cyclotomic, etc.

In _Λ ◦ λ_, a cyclotomic index _m_ is specified by an appropriate type _m_, and the data types representing
cyclotomic rings (and their underlying coefficient tensors) are parameterized by such an _m_. Based on this
parameter, _Λ ◦ λ_ **generically derives** algorithms for all the relevant operations in the corresponding cyclotomic.
In addition, for operations that involve more than one cyclotomic, _Λ ◦ λ_ expresses and **statically enforces** (at
compile time) the laws governing when these operations are well defined.

---

6For simplicity, here we have depicted _r_ as an additional parameter of the **Decompose** class. Our actual code adopts the more
idiomatic practice of using a **type family** DecompoF _u_, which is defined by each instance of **Decompose**.
We achieve the above properties using Haskell’s type system, with the help of the powerful data kinds extension \cite{YWC+12} and the singletons library \cite{EW12, ES14}. Essentially, these tools enable the “promotion” of ordinary values and functions from the data level to the type level. More specifically, they promote every value to a corresponding type, and promote every function to a corresponding type family, i.e., a function on the promoted types. We stress that all type-level computations are performed at compile time, yielding the dual benefits of static soundness guarantees and no runtime overhead.

Implementation. Concretely, \(\Lambda \circ \lambda\) defines a special data type \textbf{Factored} that represents positive integers by their factorizations, along with several functions on such values. Singletons then promotes all of this to the type level. This yields concrete “factored types” \(\textit{Fm}\) for various useful values of \(m\), e.g., \(F1, \ldots, F100, F128, F256, F512\), etc. In addition, it yields the following type families, where \(m1, m2\) are variables representing any factored types:

- \textit{FMul} \(m1 \cdot m2\) (synonym: \(m1 \times m2\)) and \textit{FDiv} \(m1 \div m2\) (synonym: \(m1 \div m2\)) respectively yield the factored types representing \(m1 \cdot m2\) and \(m1/m2\) (if it is an integer; else it yields a compile-time error);
- \textit{FGCD} \(m1 \cdot m2\) and \textit{FLCM} \(m1 \cdot m2\) respectively yield the factored types representing \(\gcd(m1, m2)\) and \(\lcm(m1, m2)\);
- \textit{FDivides} \(m1 \div m2\) yields the (promoted) boolean type \textbf{True} or \textbf{False}, depending on whether \(m1| m2\).

In addition, \(m1 \textbf{`Divides`} m2\) is a convenient synonym for the constraint \(m1 \sim \text{Divides} m1 \div m2\).

Finally, \(\Lambda \circ \lambda\) also provides several entailments representing number-theoretic laws that the compiler itself cannot derive from our data-level code. For example, transitivity of the “divides” relation is represented by the entailment

\[(k \textbf{`Divides`} l, l \textbf{`Divides`} m) :- (k \textbf{`Divides`} m)\]

which allows the programmer to satisfy the constraint \(k|m\) in any context where the constraints \(k|\ell\) and \(\ell|m\) are satisfied.

Further details on type-level indices and arithmetic, and how they are used to derive algorithms for cyclotomic ring operations, may be found in Appendix B.

3 Cyclotomic Rings

In this section we describe our interfaces and implementations for working in cyclotomic rings. The material is divided in two main parts: in Section 3.2 we describe our “safe” interface, which completely hides from clients the internal representations of ring elements. Then in Sections 3.3 and 3.4 we describe a lower-level “unsafe” interface and implementation that allows limited control over the internal representation, along with functions whose behavior depends on the representation.

3.1 Mathematical Background

To appreciate the material in this section, one only needs the following high-level background; see Section D.1 and \cite{LPR10, LPR13} for many more mathematical and computational details.
Cyclotomic rings. For a positive integer \( m \), the \( m \)th cyclotomic ring is \( R = \mathbb{Z}[\zeta_m] \), the ring extension of the integers \( \mathbb{Z} \) obtained by adjoining an element \( \zeta_m \) having multiplicative order \( m \). The ring \( R \) is contained in the \( m \)th cyclotomic number field \( K = \mathbb{Q}(\zeta_m) \). The minimal polynomial (over the rationals) of \( \zeta_m \) has degree \( n = \varphi(m) \), so \( \deg(K/\mathbb{Q}) = \deg(R/\mathbb{Z}) = n \). We endow \( K \) (and thus \( R \)) with a geometry via a certain function \( \sigma : K \rightarrow \mathbb{C}^n \) called the canonical embedding. E.g., we define the \( \ell_2 \) norm on \( K \) as \( \|x\|_2 = \|\sigma(x)\|_2 \), and use this to define Gaussian-like distributions over \( R \) and (discretizations of) \( K \).

For cryptographic purposes, there are two particularly important \( \mathbb{Z} \)-bases of \( R \): the powerful basis \( \overline{p}_m \) and the decoding basis \( \overline{d}_m \). There is also a special element \( g_m \in R \), which is used for managing error terms in cryptographic applications, as described later in Section 4.

The \( m \)th cyclotomic ring \( R = \mathbb{Z}[\zeta_m] \) can be seen as a subring of the \( m \)'th cyclotomic ring \( R' = \mathbb{Z}[\zeta_{m'}] \) if and only if \( m | m' \), and in such a case we can embed \( R \) into \( R' \) by identifying \( \zeta_m \) with \( \zeta_{m'/m} \). In the reverse direction, we can tween from \( R' \) to \( R \), which is a certain \( R \)-linear function that fixes \( R \) pointwise. (The name is short for “tweaked trace,” because the twace is the appropriate variant of the true trace function to our “tweaked” setting, described next.) The relative powerful basis \( \overline{p}_{m',m} \) is an \( R \)-basis of \( R' \) that is obtained by “factoring out” (in a formal sense) the powerful basis of \( R \) from that of \( R' \), and similarly for the relative decoding basis \( \overline{d}_{m',m} \).

Ring-LWE and (tweaked) error distributions. Ring-LWE is a family of computational problems that was defined and analyzed in [LPR10, LPR13]. Those works deal with a form of Ring-LWE involving a special (fractional) ideal \( R' \) that is dual to \( R \). More specifically, the problem relates to “noisy” products

\[
   b_i = a_i \cdot s + e_i \mod q R' \,
\]

where \( a_i \in R/qR \), \( s \in R'/qR' \) (so \( a_i \cdot s \in R'/qR' \)), and \( e_i \) is drawn from some error distribution \( \psi \), which is a parameter of the problem. In one of the worst-case hardness theorems for Ring-LWE, the distribution \( \psi \) corresponds to a spherical Gaussian \( D_r \) of parameter \( r = \alpha q \approx n^{1/4} \) or more in the canonical embedding.\(^7\)

Such spherical distributions also behave very well in cryptographic applications.

For practical purposes, it is convenient to use a form of Ring-LWE that does not involve \( R' \). As suggested in [AP13], this can be done with no loss in security or efficiency by working with an equivalent “tweaked” form of the problem, which is obtained by multiplying the noisy products \( b_i \) by a certain factor \( t = t_m \in R_m \) for which \( t \cdot R' = R \). Doing so yields new noisy products

\[
   b'_i = t \cdot b_i = a_i \cdot (t \cdot s) + (t \cdot e_i) \mod q R,
\]

where both \( a_i \) and \( s' = t \cdot s \) reside in \( R/qR \), and the error term \( t \cdot e_i \) comes from the “tweaked” distribution \( t \cdot \psi \). Note that when \( \psi \) corresponds to a spherical Gaussian (in the canonical embedding), its tweaked form \( t \cdot \psi \) may be very far from spherical, but this is not a problem: the tweaked form of Ring-LWE is entirely equivalent to the above one involving \( R' \), because the tweak is reversible. (In Section 4 we show how to properly manage error terms from the tweaked distribution in cryptographic applications.)

Finally, we remark that the decoding basis of \( R \) is merely the “tweaked” decoding basis of \( R' \) (as defined in [LPR13]), so all the efficient algorithms described in [LPR13] involving \( R' \) and its decoding basis—e.g., for sampling from spherical Gaussians, converting between bases of \( R' \), etc.—carry over without any modification to the tweaked setting.

\(^7\)Moreover, no subexponential (in \( n \)) attacks are known when \( r \geq 1 \) and \( q = \text{poly}(n) \).
3.2 Safe Interface: Cyc

The data type $\text{Cyc} \ t \ m \ r$ and its associated operations (see Figure 1) represents the $m$th cyclotomic ring over a base ring $r$—typically, one of $\mathbb{Q}$, $\mathbb{Z}$, or $\mathbb{Z}_q$—backed by an underlying $\text{Tensor}$ type $t$ (see Section D for details on Tensor). The functions and instances associated with $\text{Cyc}$ represent the cryptographically relevant operations and values associated with cyclotomic rings. In particular, the interface surrounding $\text{Cyc}$ is “safe,” in that it completely hides the internal representations of ring elements from the client, and it can never produce a runtime error (assuming all other types $t$, $m$, $r$, etc. satisfy their requisite contracts). Therefore, we recommend that cryptographic applications use $\text{Cyc}$ wherever possible.

**Instances.** As one might expect, $\text{Cyc} \ t \ m \ r$ is an instance of $\text{Eq}$, $\text{Additive}$, $\text{Ring}$, etc., for any appropriate choices of $t$, $m$, and $r$. Therefore, we can use the standard operators ($=\,\), ($+$), ($*$), etc., along with any polymorphic functions that rely upon them. In addition, we naturally “promote” instances of $\text{Reduce}$, $\text{Gadget}$, $\text{Decompose}$, and others from the base ring $r$ to $\text{Cyc} \ t \ m \ r$. For example, we have $\text{Reduce}\ (\text{Cyc} \ t \ m \ z)\ (\text{Cyc} \ t \ m \ zq)$ whenever we have $\text{Reduce}\ z\ zq$. These promoted instances are implemented very generically and concisely, as described below in Section 3.4.3.

**Functions.** We now describe the main functions that operate on $\text{Cyc}$ data (see Figure 1), starting with those that involve a single index $m$.

- **$\text{scalarCyc}$** embeds a scalar element from the base ring $r$ into the $m$th cyclotomic ring over $r$.
- **$\text{mulG}$, $\text{divG}$** respectively multiply and divide by the special element $g_m$ in the $m$th cyclotomic ring. These operations are commonly used in applications, and have particularly efficient algorithms in our implementations, which is why we expose them as special functions (rather than, say, just exposing a value representing $g_m$). Note that because the input may not always be divisible by $g_m$, the output type of $\text{divG}$ is a $\text{Maybe}$.
- **$\text{adviseB}$** for $B = \text{Pow}$, $\text{Dec}$, $\text{CRT}$ returns an equivalent ring element that might be represented in, respectively, the powerful, decoding, or Chinese Remainder Theorem basis (if it exists). This has no externally visible effect on the results of any operations, but it can serve as a useful optimization hint: if one needs to compute $v \ast w_1$, $v \ast w_2$, etc., then advising that $v$ be in CRT representation can speed up these operations by avoiding duplicate CRT conversions across the operations.

The following functions relate to sampling error terms from theory-recommend probability distributions:

- **$\text{tGaussian}$** samples an element of $K$ from the “tweaked” spherical Gaussian distribution $t \cdot D_r$, given $v = r^2$. (See Section 3.1 above for background on, and the relevance of, tweaked Gaussians. The input is $v = r^2$ because that is more convenient for implementation.) Because the output is random, its type must be monadic: $\text{rand} (\text{Cyc} \ t \ m \ r)$ for $\text{MonadRandom}$ $\text{rnd}$.
- **$\text{errorRounded}$** is a discretized version of $\text{tGaussian}$, which generates a sample from the latter and rounds each decoding-basis coefficient to the nearest integer, thereby producing an output in $R$.

---

8 However, we do not promote $\text{Lift}$ and $\text{Rescale}$ instances, because lifting and rescaling are basis dependent, and applications often need to perform them in a specified basis. Instead, we define $\text{liftCyc}$ and $\text{rescaleCyc}$ functions, which take arguments that indicate the desired basis.
data Basis = Pow | Dec -- powerful and decoding

scalarCyc :: (Fact m, Celt t r) => r -> Cyc t m r
mulG :: (Fact m, Celt t r) => Cyc t m r -> Cyc t m r
divG :: (Fact m, Celt t r) => Cyc t m r -> Maybe (Cyc t m r)
advisePow, adviseDec, adviseCRT
:: (Fact m, Celt t r) => Cyc t m r -> Cyc t m r

-- for sampling error terms
tGaussian :: (OrdFloat q, ToRational v, MonadRandom rnd, Celt t q, ...)
=> v -> rnd (Cyc t m q)
errorRounded :: (ToInteger z, ...) => v -> Cyc t m z
errorCoset :: (ToInteger z, ...) => v -> Cyc t m z

-- for extension rings
embed :: (m `Divides` m', Celt t r) => Cyc t m r -> Cyc t m' r
twace :: (m `Divides` m', Celt t r) => Cyc t m' r -> Cyc t m r
coeffsCyc :: (m `Divides` m', Celt t r) => Basis -> Cyc t m' r -> [Cyc t m r]
powBasis :: (m `Divides` m', Celt t r) => Tagged m [Cyc t m' r]
crtSet :: (m `Divides` m', Celt t r, ...) => Tagged m [Cyc t m' r]

Figure 1: Representative functions for the Cyc data type. (The Celt t r constraint is a synonym for a collection of constraints that include Tensor t, along with various constraints on r.)

errorCoset samples an error term from a (discretized) tweaked Gaussian of parameter \( p \cdot r \) over a given coset of \( R_p = R/pR = \mathbb{Z}_p[z_m] \). This operation is often used in encryption schemes when encrypting a desired message from the plaintext space \( R_p \).

Finally, the following functions involve Cyc data types for two indices \( m,m' \); recall that this means we can view the \( m \)th cyclotomic ring as a subring of the \( m' \)th one. Notice that in the type signatures, the divisibility constraint is expressed as \( m \ `\text{Divides}` m' \), and recall from Section 2.6 that this constraint is statically checked by the compiler and carries no runtime overhead.

eMBED, TWACE are respectively the embedding and “tweaked trace” functions between the \( m \)th and \( m' \)th cyclotomic rings.

coeffsCyc expresses an element of the \( m' \)th cyclotomic ring with respect to the relative powerful or decoding basis \( \overline{p}_{m',m} \) and \( \overline{d}_{m',m} \), respectively, as a list of coefficients from the \( m \)th cyclotomic.

powBasis is the relative powerful basis \( \overline{p}_{m',m} \) of the \( m' \)th cyclotomic over the \( m \)th one. Note that the Tagged \( m \) type annotation is needed to specify which subring the basis is relative to.

\[\text{The extra factor of } p \text{ in the Gaussian parameter reflects the connection between coset sampling as used in cryptosystems, and the underlying Ring-LWE error distribution actually used in their security proofs. Therefore, the input } v \text{ has a consistent meaning across all the error-sampling functions.}\]

\[\text{We also could have defined decBasis, but it is slightly more complicated to implement, and we have not needed it in any of our applications.}\]
**crtSet** is the relative CRT set \( \mathcal{E}_{m',m} \) of the \( m' \)th cyclotomic ring over the \( m \)th one, modulo a prime power. (See Section 13.4 for its formal definition and a novel algorithm for computing it.) We have elided some constraints which say that the base type \( r \) must represent \( \mathbb{Z}_{p^e} \) for a prime \( p \).

We emphasize that both **powBasis** and **crtSet** are values (of type **Tagged** \( m \) \([\text{Cyc} \ t \ m' \ r]\)), not functions. Due to Haskell’s laziness, only those values that are actually used in a computation are ever explicitly computed; moreover, the compiler usually ensures that they are computed only once each and then memoized.

In addition to the above, we also could have included functions that apply automorphisms of cyclotomic rings, which would be straightforward to implement in our framework. We leave this for future work, merely because we have not yet needed automorphisms in any of our applications.

### 3.3 Unsafe Interface: UCyc

The **Cyc** data type described in the previous subsection is merely a **newtype** wrapper around another data type **UCyc**, which has a wider but “unsafe” interface. By this we mean that the **UCyc** interface exposes limited control over the underlying representation of ring elements, along with functions whose behavior depends on the current representation. Moreover, for some combinations of operations and representations the behavior is not well defined, so improper usage of **UCyc** can result in a runtime error. Therefore, client code must use unsafe functions correctly, by ensuring that their **UCyc** inputs are in appropriate representations. (**Cyc** is itself a client that does just that, which is why it is safe and recommended over **UCyc**.)

**Instances.** As might be expected, **UCyc** \( t \ m \ r \) is an instance of all the same classes **Eq**, **Additive**, **Ring**, **Reduce**, **Gadget**, **Decompose**, etc. as **Cyc** \( t \ m \ r \) is. (Indeed, **Cyc**’s instances are merely costless wrappers around **UCyc**’s.) The implementations of these classes are described in Sections 3.4.2 and 3.4.3 below. In addition, the partially applied type **UCyc** \( t \ m \) is an instance of the generic “container” classes **Functor**, **Applicative**, **Foldable**, and **Traversable**. This allows us to easily and generically “promote” operations from the base type \( r \) to **UCyc** \( t \ m \), as described in Section 3.4.3 below.

```haskell
forceBasis :: (Fact m, CElt t r) => Maybe Basis -> UCyc t m r -> UCyc t m r
fmapC :: (Fact m, CElt t a, CElt t b) => (a -> b) -> UCyc t m a -> UCyc t m b
```

**Figure 2:** Additional functions for the **UCyc** data type.

**Functions.** The **UCyc** type admits safe functions that have the exact same names and descriptions as those for the **Cyc** type (see Figure[1]). In addition, we have the following unsafe functions (see Figure[2]) for their type signatures:

**forceBasis** returns an equivalent ring element that is internally represented in a specified \( r \)-basis, as determined by the first argument: **Just Pow** for the powerful basis, **Just Dec** for the decoding basis, and **Nothing** for an arbitrary \( r \)-basis of the implementation’s choice. This function has no externally visible effect on the results of arithmetic operations like \((=)\), \((+)\), and \((\times)\), but, like **adviseCRT**, it may affect runtimes by altering the number of basis conversions required by a computation.

More importantly, **forceBasis** does affect the behavior of **UCyc**’s instances of the “container” classes **Functor**, **Applicative**, etc. This is because these instances operate on the vector of \( r \)-coefficients in the current representation. See Section 3.4.3 for further details.
fmapc is an analogue of the fmap function from the Functor class, but restricted to base types that satisfy the CElt constraint (whereas fmap must be defined for arbitrary base types). This restriction allows for more efficient implementations.

3.4 U Cyc Implementation

Here we summarize our implementation of U Cyc. In short, it is a relatively thin wrapper around an instance of the Tensor class. (Recall that a Tensor encapsulates a coefficient vector and all the relevant linear transforms that we may need to perform on it; see Appendix D for full details.) U Cyc mainly manages the choice of internal representation for ring elements, implicitly performing the appropriate conversions (via Tensor methods) to support efficient ring operations. This design has the advantage of decoupling the higher-level management from the computation-intensive work, allowing for multiple implementations of the latter without affecting higher-level code.

3.4.1 Representations

U Cyc t m r can represent an element of the mth cyclotomic ring over base ring r in several ways:

- as a tensor of r-coefficients with respect to either the powerful or decoding basis;
- in one of two possible Chinese Remainder Theorem (CRT) representations, the choice of which depends on the properties of r as described in the next paragraph; or
- when applicable, directly as a scalar from the base ring r, or as an element of the kth cyclotomic (sub)ring for some k|m.

The last of these representations provides a particularly useful optimization, because applications often need to treat scalars and subring elements as residing in a larger ring, yet U Cyc exploits knowledge of their “true” domains to operate more efficiently.

The two possible CRT representations are as follows: if there is a CRT basis of index m over r itself, then U Cyc uses it, employing the Chinese remainder transform to convert between the powerful and CRT bases. Otherwise, U Cyc embeds r into a ring r’ that is guaranteed to have a CRT basis of any index, and stores a tensor of r’-coefficients with respect to this basis. Often, r’ is (some representation of) the complex numbers C, but the choice of r’ is defined by r itself, and U Cyc is entirely agnostic to it. For example, the associated embedding ring of a product ring (r1, r2) is (r1’, r2’), where r1’ is the embedding ring of r1.

3.4.2 Arithmetic Operations

U Cyc implements operations like (==), (+), and (*) in the following way: it converts the inputs to a “compatible” representation in the most efficient way possible, then computes the output in this representation. A few representative rules for how this is accomplished include:

- For two scalars from the base ring r, the result is computed and stored as a scalar.
- Arguments from (possibly different) subrings, of indices k1, k2|m, are embedded into the compositum of the two subrings—i.e., the cyclotomic of index k = lcm(k1, k2), which divides m—and the result is computed there and stored as a subring element.
- For (+), the arguments are converted to a common representation and added entry-wise.
For \( (*) \), if one of the arguments is a scalar from the base ring \( r \), it is simply multiplied by the coefficients of the other argument (in any \( r \)-basis). Otherwise, the two arguments are converted to the same CRT representation and multiplied entry-wise.

\textbf{UCyc} implements the embed and twace operations in the following way: embed from index \( m \) to \( m' \) is "lazy," merely storing its argument as a subring element and returning instantly. For twace from index \( m' \) to \( m \), UCyc typically just computes the result in the current representation by invoking the appropriate linear transformation (from Tensor). However, it is optimized for subring elements: for an element in the \( k' \)-th cyclotomic, UCyc applies twace from index \( k' \) to index \( k = \gcd(m, k') \), where the result is guaranteed to reside, and stores the result as a subring element.

### 3.4.3 Promoting from Base Ring to Cyclotomics

Many operations on cyclotomic rings are defined as entry-wise operations on the ring element’s coefficient vector, with respect to either a particular basis or an arbitrary one. For example, reducing from \( R \) to \( R_q \) is equivalent to reducing the coefficients from \( \mathbb{Z} \) to \( \mathbb{Z}_q \) in any basis, while “decoding” \( R_q \) to \( R \) (used in decryption) means lifting the \( \mathbb{Z}_q \)-coefficients to their smallest representatives in \( \mathbb{Z} \), using the decoding basis. To implement these and other functions for UCyc, we use a very generic and modular mechanism for “promoting” operations on the base ring to corresponding operations on cyclotomic rings. Specifically, we define \textbf{UCyc t m} to be an instance of Haskell’s standard "container" classes \textbf{Functor, Applicative, Foldable, and Traversable}.

To illustrate this approach, consider the Functor class, which introduces the method

\[
\text{fmap :: Functor } f \Rightarrow (a -> b) -> f a -> f b.
\]

The \textbf{Functor} instance for \textbf{UCyc t m} defines \text{fmap} \( g \ c \) to apply \( g \) entry-wise to \( c \)'s vector of coefficients in its \textit{current} representation, namely, the powerful, decoding, or CRT basis (if it exists); other representations yield a runtime error. (Recall from Section 2.3 that we can convert to a particular representation using \text{forceBasis}.) We can therefore easily implement the above-described reduce and decode operations by promoting the methods of our classes from Section 2. An instance \textbf{Reduce} \( z \ zq \) is promoted to an instance \textbf{Reduce \ (UCyc t m z)} \ (\textbf{UCyc t m zq}), and an instance \textbf{Lift} \( zq \ z \) is promoted to the decoding operation, as follows:

\[
\text{reduce} = \text{fmap reduce} \ . \ \text{forceBasis} \ \text{Nothing}
\]

\[
\text{decode} :: (\text{Lift} \ b \ a, ...) \Rightarrow \text{UCyc t m b} \rightarrow \text{UCyc t m a}
\]
\[
\text{decode} = \text{fmap lift} \ . \ \text{forceBasis} \ (\text{Just Dec})
\]

As a richer example, consider gadgets and decomposition (Section 2.5) for a cyclotomic ring \( R_q \) over base ring \( \mathbb{Z}_q \). For any gadget over \( \mathbb{Z}_q \), we get an identical gadget over \( R_q \) simply by embedding the scalar \( \mathbb{Z}_q \)-entries into \( R_q \). This lets us promote an instance of \textbf{Gadget} for \( zq \) to an instance for \textbf{UCyc t m zq} as follows:

\[
\text{gadget} = \text{fmap} \ (\text{fmap scalarCyc}) \ \text{gadget}
\]

(The double use of \text{fmap} is because there are two \textbf{Functor} layers around the \( zq \)-entries of the underlying \textbf{gadget} :: \textbf{Tagged} \ \text{gad} \ [\text{zq}]: the list \([\text{zq}]\), and the \textbf{Tagged} \ \text{gad context}.)

Decomposing an \( R_q \)-element into a short vector over \( R \) works coefficient-wise in the power basis. That is, we decompose each \( \mathbb{Z}_q \)-coefficient into a short vector over \( \mathbb{Z} \), then collect the corresponding entries of
these vectors to yield a vector of short $R$-elements. To implement this strategy, one might try to promote the function (here with slightly simplified signature) $\text{decompose} :: \text{Decompose } zq \ z \Rightarrow zq \to [z]$ to $\text{UCyc } t \ m \ zq$, as we did with reduce and lift above. However, this does not work: $\text{fmap} \ \text{decompose}$ has type $c \ m \ zq \to c \ m \ [z]$, whereas we need output type $[c \ m \ z]$. The solution is to use the $\text{Traversable}$ class, which introduces the method

$$\text{traverse} :: (\text{Traversable } v, \text{Applicative } f) \Rightarrow (a \to f \ b) \to v \ a \to f (v \ b)$$

In our setting, $v$ stands for $\text{UCyc } t \ m$, and $f$ stands for the list type $[]$, which is indeed an instance of $\text{Applicative}$. We therefore easily promote an instance of $\text{Decompose}$ from $zq$ to $\text{UCyc } t \ m \ zq$ via:

$$\text{decompose} = \text{traverse} \ \text{decompose} \ . \ \text{force} \ \text{basis} \ (\text{Just Pow})$$

We promote many other operations on base rings just as easily, including the error-correction operation $\text{correct}$, the rescaling function $\text{rescale}$ (from $\mathbb{Z}_q$ to $\mathbb{Z}_{q'}$), discretization of $\mathbb{Q}$ to $\mathbb{Z}$ or to a desired coset of $\mathbb{Z}_p$, and many more.

**Rescaling.** To rescale a cyclotomic ring $R_q$ to $R_{q'}$, we typically need to apply the integer rescaling operation $[\_]: \mathbb{Z}_q \to \mathbb{Z}_{q'}$ (represented by the function $\text{rescale} :: \text{Rescale } a \ b \Rightarrow a \to b$; see Section 2.3) coordinate-wise in either the powerful or decoding basis, for geometrical reasons. However, rescaling cyclotomics is special, because there are at least two distinct algorithms, depending on the representation of $\mathbb{Z}_q$ and $\mathbb{Z}_{q'}$. First, there is the generic algorithm, which simply converts the input to the desired basis and then rescales coefficient-wise. Second, there is a more efficient, specialized algorithm due to [GHST12] for rescaling a product ring $R_q = R_{q_1} \times R_{q_2}$ to $R_{q_1}$. When rescaling an input in the CRT basis to an output in the CRT basis, this algorithm requires only about half as many CRT transformations over individual moduli.

In more detail, the specialized rescaling algorithm is analogous to the one for product rings $\mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2}$ described at the end of Section 2.3. Specifically, to rescale $a = (a_1, a_2) \in R_{q_1} \times R_{q_2}$ to $R_{q_1}$, we lift $a_2 \in R_{q_2}$ to a relatively short representative $\tilde{a}_2 \in R$ using the powerful or decoding basis; this implicitly involves an inverse-CRT over $R_{q_2}$. We then output $q_2^{-1} \cdot (a_1 - \tilde{a}_2) \in R_{q_1}$; this implicitly involves a CRT over $R_{q_1}$ on $(\tilde{a}_2 \mod q_1 R)$. In total, we perform only one (inverse-)CRT transformation for each $R_{q_1}$ component, whereas the generic algorithm involves a transform in both directions for $R_{q_1}$. Because $R_{q_1}$ is itself often the product of many sub-components, its CRT transforms are the bottleneck, and so the specialized algorithm is nearly twice as fast as the generic one.

To capture the polymorphism represented by above algorithms, we define a class called $\text{RescaleCyc}$, which introduces the method $\text{rescaleCyc}$. We give instances of $\text{RescaleCyc}$ for both the generic and specialized algorithms, and the compiler automatically chooses the appropriate one based on the concrete types representing the moduli.

## 4 Homomorphic Encryption in $\Lambda \circ \lambda$

In this section we describe a full-featured fully homomorphic encryption implementation in $\Lambda \circ \lambda$, using the interfaces described in the previous sections. At the mathematical level, the system closely follows the Ring-LWE cryptosystem and homomorphic operations developed over a long series of works [LPRT10].

---

11While this is true, the instance of $\text{Applicative}$ for $[]$ actually models *nondeterminism*, not the entry-wise operations we need. Fortunately, there is a standard $\text{newtype}$ wrapper around $[]$, called $\text{Ziplist}$, that instantiates $\text{Applicative}$ in exactly the way we need. So our actual promotion of $\text{decompose}$ converts (for free) between $[]$ and $\text{Ziplist}$ at appropriate points.
In addition, we include some important generalizations and new operations, such as “ring-tunneling,” that have not yet appeared in the literature. Along with the mathematical description of each main component, we present the corresponding Haskell code, showing how the two forms match very closely.

4.1 Keys, Plaintexts, and Ciphertexts

The cryptosystem is parameterized by two cyclotomic rings: $R = \mathcal{O}_m$ and $R' = \mathcal{O}_{m'}$ where $m|m'$, making $R$ a subring of $R'$. The spaces of keys, plaintexts, and ciphertexts are derived from these rings as follows:

- A secret key is an element $s \in R'$. Some operations require $s$ to be “small;” more precisely, we need $s \cdot g_{m'}$ to have small entries in the canonical embedding of $R'$ (see Invariant 4.1 below). Recall that this is indeed the case for theory-recommended Ring-LWE error distributions over $R'$.

- The plaintext ring is $R_p = R/pR$, where $p$ is a (typically small) positive integer, e.g., $p = 2$. For technical reasons, $p$ must be coprime with every odd prime dividing $m'$. A plaintext is simply an element $\mu \in R_p$.

- The ciphertext ring is $R'_{q} = R'/qR'$ for some integer modulus $q \geq p$ that is coprime with $p$. A ciphertext is essentially just a polynomial $c(S) \in R'_{q}[S]$, i.e., one with coefficients from $R'_{q}$ in an indeterminant $S$, which represents the (unknown) secret key. We often identify $c(S)$ with its vector of coefficients $(c_0, c_1, \ldots, c_d) \in (R'_{q})^{d+1}$, where $d$ is the degree of $c(S)$.

In addition, a ciphertext carries a nonnegative integer $k \geq 0$ and a factor $l \in \mathbb{Z}_p$ as auxiliary information. These values are affected by certain operations on ciphertexts, as described below.

Data types. Following the above definitions, our data types for plaintexts, keys, and ciphertexts as follows. The plaintext type $\text{pt}$ is merely a synonym for its argument type $\text{rp}$ representing the plaintext ring $R_p$.

The data type $\text{SK}$ representing secret keys is defined as follows:

\[
\text{data SK} \ r' \ where \ \text{SK} \ :: \ \text{ToRational} \ \nu = \nu \rightarrow r' \rightarrow \text{SK} \ r'
\]

Notice that a value of type $\text{SK} \ r'$ consists of an element from the secret key ring $R'$, and in addition it carries a rational value (of “hidden” type $\nu$) representing the (squared) parameter $\nu = r^2$ of the (tweaked) Gaussian distribution from which the key was sampled. Binding the parameter to the secret key in this way allows us to automatically generate ciphertexts and other key-dependent information using consistent error distributions, thereby relieving client code of the responsibility for managing error parameters across multiple functions.

The data type $\text{CT}$ representing ciphertexts is defined as follows:

\[
\text{data Encoding} \ = \ \text{MSD} \mid \ \text{LSD} \\
\text{data CT} \ m \ zp \ r'q \ = \ \text{CT Encoding} \ \text{Int} \ \text{zp} \ (\text{Polynomial} \ r'q)
\]

The $\text{CT}$ type is parameterized by three arguments: a cyclotomic index $m$ and a $\mathbb{Z}_p$-representation $zp$ defining the plaintext ring $R_p$, and a representation $r'q$ of the ciphertext ring $R'_{q}$. A $\text{CT}$ value has four components: a flag indicating the “encoding” of the ciphertext (MSD or LSD; see below); the auxiliary integer $k$ and factor $l \in \mathbb{Z}_p$ (as mentioned above); and a polynomial $c(S)$. 

21
Decryption relations and error invariant. A ciphertext \( c(S) \) (with auxiliary values \( k \in \mathbb{Z}, l \in \mathbb{Z}_p \)) encrypting a plaintext \( \mu \in \mathbb{R}_p \) under secret key \( s \in \mathbb{R}' \) satisfies the relation

\[
c(s) = c_0 + c_1 s + \cdots + c_d s^d = e \pmod{qR'}\tag{4.1}
\]

for some sufficiently “small” error term \( e \in R' \) such that

\[
e = l^{-1} \cdot g_m^k \cdot \mu \pmod{pR'}\tag{4.2}
\]

More precisely, by “small” we mean that decoding \( c(s) \in \mathbb{R}' \) to \( \mathbb{R}' \) (i.e., lifting using the decoding basis) should yield \( e \) itself. In particular, this holds if all the coefficients of \( e \in R' \) in the decoding basis have magnitudes smaller than \( q/2 \). To control these coefficients as tightly as possible, all our operations maintain the following informal invariant. This invariant is satisfied by “fresh” error terms drawn from tweaked Gaussians over \( \mathbb{R}' \) (see Section 3.1), and is a sufficient condition for obtaining sharp bounds on the decoding-basis coefficients, as shown in [LPR13, Section 6]:

**Invariant 4.1.** For an error term \( e \in \mathbb{R}' \), every complex coordinate of the canonical embedding \( \sigma(e \cdot g_m^k) \in \mathbb{C}^n \) is nearly independent (up to the conjugate pairs), and bounded by a distribution with “light” (e.g., subexponential) tails.

A ciphertext satisfying Equations (4.1) and (4.2) is said to be in “least significant digit” (LSD) form, because the message \( \mu \) is encoded in the mod-\( p \) value of the error term. An alternative form, which is more convenient for certain homomorphic operations, is the “most significant digit” (MSD) form. Here the relation is

\[
c(s) \approx \frac{2}{p} \cdot (l^{-1} \cdot g_m^k \cdot \mu) \pmod{qR'}\tag{4.3}
\]

where the approximation hides a small fractional error term (in \( \frac{1}{p} R' \)) that satisfies Invariant 4.1. Notice that the message is represented as a multiple of \( \frac{2}{p} \) modulo \( q \), hence the name “MSD.” One can losslessly transform between LSD and MSD forms in linear time, just by multiplying by appropriate \( \mathbb{Z}_q \)-elements (see [Ap13, Appendix A]). Each such transformation implicitly multiplies the plaintext by some fixed element of \( \mathbb{Z}_p \), which is why our ciphertexts carry auxiliary factors \( l \in \mathbb{Z}_p \) that must be accounted for upon decryption.

### 4.2 Encryption and Decryption

To encrypt a message \( \mu \in \mathbb{R}_p \) under a key \( s \in \mathbb{R}' \), one does the following:

1. sample an error term \( e \in \mu + pR' \) (from a distribution that should be a \( p \) factor wider than that of the secret key);
2. sample a uniformly random \( c_1 \leftarrow R'_q \);
3. output the LSD-form ciphertext \( c(S) = (e - c_1 \cdot s) + c_1 \cdot S \in \mathbb{R}'_q[S] \), with \( k = 0, l = 1 \in \mathbb{Z}_p \).

(Observe that \( c(s) = e \pmod{qR'} \), as desired.)

This translates directly into just a few lines of Haskell code, which is monadic due to its use of randomness:

```haskell
encrypt :: (m `Divides` m', MonadRandom rnd, ...) => SK (Cyc t m' z) -> PT (Cyc t m zp) -> rnd (CT m zp (Cyc t m' zq))
encrypt (SK v s) mu = do
  e <- errorCoset v (embed mu) -- error from mu + pR'
c1 <- getRandom -- uniform from R'/qR'
return $ CT LSD zero one $ fromCoeffs [reduce e - c1 * reduce s, c1]
```
To decrypt an LSD-form ciphertext $c(S) \in R_q'[S]$ under secret key $s \in R_q'$, we first evaluate $c(s) \in R_q'$ and then lift the result to $R_q'$ (using the decoding basis) to recover the error term $e$, as follows:

$$\text{errorTerm :: (Lift zq z, m `Divides` m', ...)}$$

$$\Rightarrow SK (\text{Cyc t m' z}) \rightarrow CT m zp (\text{Cyc t m' zq}) \rightarrow \text{Cyc t m' z}$$

$$\text{errorTerm (SK - s) (CT LSD _ _ c)} = \text{liftCyc Dec (evaluate c (reduce s))}$$

Following Equation (4.2), we then compute $l \cdot g_{m'}^{k} \cdot e \mod pR_q'$. This yields the embedding of the message $\mu$ into $R_q'$, so we finally take the twace to recover $\mu \in R_q$ itself:

$$\text{decrypt :: (Lift zq z, Reduce z zp, ...)}$$

$$\Rightarrow SK (\text{Cyc t m' z}) \rightarrow CT m zp (\text{Cyc t m' zq}) \rightarrow PT (\text{Cyc t m zp})$$

$$\text{decrypt sk ct@ (CT LSD k 1 _)} =$$

$$\text{let e = reduce (errorTerm sk ct)}$$

$$\text{in (scalarCyc l) * twace (iterate divG e !! k)}$$

### 4.3 Homomorphic Addition and Multiplication

Homomorphic addition of ciphertexts with the same values of $k$ and $l$ is simple: convert the ciphertexts to the same form (MSD or LSD), then add their polynomials. It is also possible adjust the values of $k, l$ as needed by multiplying the polynomial by an appropriate factor, which only slightly enlarges the error. Accordingly, we define $CT m zp (\text{Cyc t m' zq})$ to be an instance of Additive, for appropriate argument types.

Now consider homomorphic multiplication: suppose ciphertexts $c_1(S), c_2(S)$ encrypt messages $\mu_1, \mu_2$ in LSD form, with auxiliary values $k_1, l_1$ and $k_2, l_2$ respectively. Observe that

$$c_1(s) \cdot c_2(s) \cdot g_{m'} = e_1 \cdot e_2 \cdot g_{m'} \pmod{qR_q'}$$

$$e_1 \cdot e_2 \cdot g_{m'} = (l_1 l_2)^{-1} \cdot g_{m'}^{k_1+k_2+1} \cdot (\mu_1 \mu_2) \pmod{pR_q'},$$

and the error term $e = e_1 \cdot e_2 \cdot g_{m'}$ satisfies Invariant[4.1] because $e_1, e_2$ do (recall that multiplication in the canonical embedding is coordinate-wise). Therefore, the LSD-form ciphertext

$$c(S) := c_1(S) \cdot c_2(S) \cdot g_{m'} \in R_q'[S]$$

encrypts $\mu_1 \mu_2 \in R_p$ with auxiliary values $k = k_1 + k_2 + 1$ and $l = l_1 l_2 \in \mathbb{Z}_p$. Notice that the degree of the output polynomial is the sum of the degrees of the input polynomials.

More generally, it turns out that we only need one of $c_1(S), c_2(S)$ to be in LSD form; the product $c(S)$ then has the same form as the other ciphertext[12] All this translates immediately to an instance of Ring for $CT m zp (\text{Cyc t m' zq})$, with the interesting case of multiplication having the one-line implementation

$$\text{(CT LSD k1 l1 c1) * (CT d2 k2 l2 c2) =}$$

$$\text{CT d2 (k1+k2+1) (l1*l2) (mulG <$> c1 * c2)}$$

(The other cases just swap the arguments or convert one ciphertext to LSD form, thus reducing to the case above.)

12 If both ciphertexts are in MSD form, then it is possible to use the “scale free” homomorphic multiplication method of [Bra12], but we have not implemented it because it appears to be significantly less efficient than just converting one ciphertext to LSD form.
4.4 Modulus Switching

Switching the ciphertext modulus is a form of rescaling typically used for decreasing the modulus, which commensurately reduces the absolute magnitude of the error in a ciphertext—though the error rate relative to the modulus stays essentially the same. Because homomorphic multiplication implicitly multiplies the error terms, keeping their absolute magnitudes small can yield major benefits in controlling the error growth. Modulus switching is also sometimes useful to temporarily increase the modulus, as explained in the next subsection.

Modulus switching is easiest to describe and implement for ciphertexts in MSD form (Equation (4.3)) that have degree at most one. Suppose we have a ciphertext $c(S) = c_0 + c_1 S$ under secret key $s \in R'$, where

$$c_0 + c_1 s = d \approx \frac{q}{p} \cdot \gamma \pmod{qR'}$$

for $\gamma = l^{-1} \cdot g_{m'}^t \cdot \mu \in R_p$. Switching to a modulus $q'$ is just a suitable rescaling of each $c_i \in R'_{q'}$ to some $c_i' \in R'_{q'}$ such that $c_i' \approx (q'/q) \cdot c_i$; note that the right-hand sides here are fractional, so they need to be discretized using an appropriate basis (see the next paragraph). Observe that

$$c_0' + c_1's \approx \frac{q'}{q}(c_0 + c_1 s) = \frac{q'}{q} \cdot d \approx \frac{q'}{p} \cdot \gamma \pmod{q'R'},$$

so the message is unchanged but the absolute error is essentially scaled by a $q'/q$ factor.

Note that the first approximation above hides the extra discretization error $e_0 + e_1 s$ where $e_i = c_i' - \frac{q}{q'} c_i$, so the main question is what bases of $R'$ to use for the discretization, to best maintain Invariant 4.1. We want both $e_0$ and $e_1 s$ to satisfy the invariant, which means we want the entries of $\sigma(e_0 \cdot g_{m'})$ and $\sigma(e_1 s \cdot g_{m'}) = \sigma(e_1) \circ \sigma(s \cdot g_{m'})$ to be essentially independent and as small as possible; because $s \in R'$ itself satisfies the invariant (i.e., the entries of $\sigma(s \cdot g_{m'})$ are small), we want the entries of $\sigma(e_1)$ to be as small as possible. It turns out that these goals are best achieved by rescaling $c_0$ using the decoding basis $\tilde{z}$, and $c_1$ using the powerful basis $\tilde{p}$. This is because $g_{m'} \cdot \tilde{d}$ and $\tilde{p}$ respectively have nearly optimal spectral norms over all bases of $g_{m'} R'$ and $R'$, as shown in [LPR13].

Our Haskell implementation is therefore simply

```haskell
rescaleLinearCT :: (Rescale zq zq', ...) => CT m zp (Cyc t m' zq) -> CT m zp (Cyc t m' zq')
rescaleLinearCT (CT MSD k 1 (Poly [c0,c1])) =
  let c'0 = rescaleDec c0
      c'1 = rescalePow c1
  in CT MSD k 1 $ Poly [c'0, c'1]
```

4.5 Key Switching and Linearization

Recall that homomorphic multiplication causes the degree of the ciphertext polynomial to increase. Key switching is a technique for reducing the degree, typically back to linear. More generally, key switching is a mechanism for proxy re-encryption: given two secret keys $s_{\text{in}}$ and $s_{\text{out}}$ (which may or may not be different), one can construct a “hint” that lets an untrusted party convert an encryption under $s_{\text{in}}$ to one under $s_{\text{out}}$, while preserving the secrecy of the message and the keys.

Key switching uses a gadget $g \in (R'_{q'})^{\ell}$ and associated decomposition function $g^{-1} : R'_{q'} \rightarrow (R'')^{\ell}$ (both typically promoted from $\mathbb{Z}_q$; see Sections 2.5 and 3.4.3). Recall that $g^{-1}(c)$ outputs a short vector over $R'$ such that $\tilde{g}^t \cdot g^{-1}(c) = c \pmod{qR'}$. 

24
The core operations. Let \( s_{in}, s_{out} \in R' \) denote some arbitrary secret values. A key-switching hint for \( s_{in} \) under \( s_{out} \) is a matrix \( H \in (R'_q)^{2\times \ell} \), where each column can be seen as a linear polynomial over \( R'_q \), such that

\[
(1, s_{out}) \cdot H \approx s_{in} \cdot \overline{g}^t \pmod{qR'}.
\]  

(4.4)

Such an \( H \) is constructed simply by letting the columns be Ring-LWE samples with secret \( s_{out} \), and adding \( s_{in} \cdot \overline{g}^t \) to the top row. In essence, such an \( H \) is pseudorandom by the Ring-LWE assumption, and hence hides the secrets.

The core key-switching step takes a hint \( H \) and some \( c \in R'_q \), and simply outputs \( c' = H \cdot g^{-1}(c) \in (R'_q)^2 \), which can be viewed as a linear polynomial \( c'(S) \). Notice that

\[
c'(s_{out}) = (1, s_{out}) \cdot c' = ((1, s_{out}) \cdot H) \cdot g^{-1}(c) \approx s_{in} \cdot \overline{g}^t \cdot g^{-1}(c) = s_{in} \cdot c \pmod{qR'},
\]

(4.5)

where the approximation holds because \( g^{-1}(c) \) is short. More precisely, because the error terms in Equation (4.4) satisfy Invariant 4.1, we want all the elements of the decomposition \( g^{-1}(c) \) to have small entries in the canonical embedding, so it is best to decompose relative to the powerful basis.

Switching ciphertexts. The above tools can be used to switch MSD-form ciphertexts of degree up to \( d \) under \( s_{in} \) as follows: first publish a hint \( H_i \) for each power \( s_{in}^i \), \( i = 1, \ldots, d \), all under the same \( s_{out} \). Then to switch a ciphertext \( c(S) \):

- For each \( i = 1, \ldots, d \), apply the core step to coefficient \( c_i \in R'_q \) using the corresponding hint \( H_i \), to get a linear polynomial \( c'_i = H_i \cdot g^{-1}(c_i) \). Also let \( c'_0 = c_0 \).
- Sum the \( c'_i \) to get a linear polynomial \( c'(S) \), which is the output.

Then \( c'(s_{out}) \approx c(s_{in}) \pmod{qR'} \) by Equation (4.5) above, so the two ciphertexts encrypt the same message.

Notice that the error rate in \( c'(S) \) is essentially the sum of two separate quantities: the error rate in the original \( c(S) \), and the error rate in \( H \) times a factor corresponding to the norm of the output of \( g^{-1} \). We typically set the latter error rate to be much smaller than the former, so that key-switching incurs essentially no error growth. This can be done by constructing \( H \) over a modulus \( q' \gg q \), and scaling up \( c(S) \) to this modulus before decomposing.

Haskell functions. Our implementation includes a variety of key-switching functions, whose types all roughly follow this general form:

\[
\text{keySwitchFoo} :: \text{(MonadRandom} \text{ rnd,} \ldots \text{)} \Rightarrow \text{SK} \text{ r'} \rightarrow \text{SK} \text{ r'} \\
\rightarrow \text{Tagged} \text{ (gad, zq')} \text{ (rnd (CT} \text{ m} \text{ zp} \text{ r'} \text{ q}) \rightarrow \text{CT} \text{ m} \text{ zp} \text{ r'} \text{q})}
\]

Unpacking this, the inputs are the two secret keys \( s_{out}, s_{in} \in R' \), and the output is essentially a re-encryption function that maps one ciphertext to another. The extra Tagged (gad, zq’) context indicates what gadget and modulus are used to construct the hint, while the rnd wrapper indicates that randomness is used in constructing (but not applying) the function; this is because constructing the hint requires randomness.

Outputting a re-encryption function—rather than just a hint itself, which would need to be fed into a separate function that actually does the switching—has advantages in terms of simplicity and safety. First, it reflects the abstract re-encryption functionality provided by key switching. Second, we implement a variety of key-switching functions that each operate slightly differently, and may even involve different types of hints (e.g., see the next subsection). With our approach, the hint is abstracted away entirely, and each style
of key-switching can be implemented by a single client-visible function, instead of requiring two separate functions and a specialized data type.

A prototypical implementation of a key-switching function is as follows, where `ksHint` and `switch` are simple auxiliary functions that perform the core operations described above:

```haskell
-- switch a linear ciphertext from one key to another
keySwitchLinear sout sin = tag $ do -- rnd monad
  hint :: Tagged gad [Polynomial (Cyc t m’ zq’)]] <- ksHint sout sin
  return $ \ (CT MSD k l (Poly [c0,c1])) ->
    CT MSD k l $ Poly [c0] + switch hint c1
```

4.6 Ring Tunneling

The term “ring switching” encompasses a collection of techniques, introduced in [BGV12, GHPS12, AP13], that allow one to change the ciphertext ring for various purposes. These techniques can also induce a corresponding change in the plaintext ring, at the same time applying a desired linear function to the underlying plaintext.

In this subsection we describe a novel method of ring switching, which we call ring tunneling, that is more efficient than the functionally equivalent method of [AP13], which for comparison we call ring hopping. The difference between the two methods is that hopping goes “up and then down” through the compositum of the source and target rings, while tunneling goes “down and then up” through their intersection (the largest common subring). Essentially, tunneling is more efficient because it uses an intermediate ring that is smaller than, instead of larger than, the source or target ring. In addition, we show how the linear function that is homomorphically applied to the plaintext can be integrated into the key-switching hint, thus combining two separate steps into a simpler and more efficient operation overall. We provide a simple implementation of ring tunneling in \(\Lambda \circ \lambda\), which to our knowledge is the first realization of ring-switching of any kind.

Linear functions. We will need some basic theory of linear functions on rings. Let \(E\) be a common subring of some rings \(R, S\). A function \(L : R \rightarrow S\) is \(E\)-linear if for all \(r, r' \in R\) and \(e \in E\),

\[
L(r + r') = L(r) + L(r') \quad \text{and} \quad L(e \cdot r) = e \cdot L(r).
\]

From this it follows that for any \(E\)-basis \(\{b\}_j\) of \(R\), an \(E\)-linear function \(L\) is uniquely determined by its values \(y_j = L(b_j) \in S\). Specifically, if \(r = b \cdot \bar{e} \in R\) for some \(\bar{e}\) over \(E\), then \(L(r) = L(\bar{b}) \cdot \bar{e} = \bar{y} \cdot \bar{e}\).

Accordingly, we introduce a useful abstract data type to represent linear functions on cyclotomic rings:

```haskell
newtype Linear t z e r s = D [Cyc t s z]
```

The parameters \(t, z\) respectively represent the underlying Tensor representation and base type, while the parameters \(e, r, s\) represent the indices of the cyclotomic rings \(E, R, S\). For example, \(\text{Cyc } t s z\) represents the ring \(S\). An \(E\)-linear function \(L\) is internally represented by its list \(\bar{y} = L(\bar{d}_{r,e})\) of values on the relative decoding basis \(\bar{d}_{r,e}\) of \(R/E\), hence the constructor named \(D\). (We could also represent linear functions via the relative powerful basis, but so far we have not needed to do so.) Using our interface for cyclotomic rings (Section 3), evaluating a linear function is straightforward:

```haskell
evallin :: (e `Divides` r, e `Divides` s, ...)
  => Linear t z e r s -> Cyc t r z -> Cyc t s z
evallin (D ys) r = dotprod ys (fmap embed (coeffsCyc Dec r :: [Cyc t e z]))
```

26
Extending linear functions. Now let $E', R', S'$ respectively be cyclotomic extension rings of $E, R, S$ satisfying certain conditions described below. As part of ring switching we will need to extend an $E$-linear function $L: R \to S$ to an $E'$-linear function $L': R' \to S'$ that agrees with $L$ on $R$, i.e., $L'(r) = L(r)$ for every $r \in R$. The following lemma gives a sufficient condition for when and how this is possible. (It is a restatement of Lemma E.1 whose proof appears in Appendix E).

Lemma 4.2. Let $e, r, s, e', r', s'$ respectively be the indices of cyclotomic rings $E, R, S, E', R', S'$, and suppose $e = \gcd(r, e'), r' = \lcm(r, e'),$ and $\lcm(s, e')|s'$. Then:

1. The relative decoding bases $\overline{a}_{r, e}$ of $R/E$ and $\overline{a}_{r', e'}$ of $R'/E'$ are identical.
2. For any $E$-linear function $L: R \to S$, the function $L': R' \to S'$ defined by $L'(\overline{d}_{r', e'}) = L(\overline{a}_{r, e})$ is $E'$-linear and agrees with $L$ on $R$.

The above lemma leads to the following very simple Haskell function to extend a linear function; notice that the constraints use the type-level arithmetic described in Section 2.6 to enforce the hypotheses of Lemma 4.2.

```haskell
extendLin :: (e ~ FGCD r e', r' ~ FLCM r e', (FLCM s e') `Divides` s') => Linear t z e r s → Linear t z e' r' s'
extendLin (Dec ys) = Dec (fmap embed ys)
```

Ring tunneling as key switching. Abstractly, ring tunneling is an operation that homomorphically evaluates a desired $E_p$-linear function $L_p: R_p \to S_p$ on a plaintext, by converting its ciphertext over $R'_q$ to one over $S'_q$. Operationally, it can be described simply as an enhanced form of key switching.

Ring tunneling involves two phases: a preprocessing phase where we use the desired linear function $L_p$ and the secret keys to produce appropriate hints, and an online phase where we apply the tunneling operation to a given ciphertext using the hint. The preprocessing phase is as follows:

1. Extend $L_p$ to an $E'_p$-linear function $L'_p: R'_p \to S'_p$ that agrees with $L_p$ on $R_p$, as described above.
2. Lift $L'_p$ to a “small” $E'$-linear function $L: R' \to S'$ that induces $L'_p$. Specifically, define $L'$ by $L'(\overline{d}_{e', e'}) = \overline{y}$, where $\overline{y}$ (over $S'$) is obtained by lifting $\overline{y}_p = L'_p(\overline{d}_{e', e'})$ using the powerful basis.

The above lifting procedure is justified by the following considerations. We want $L'$ to map ciphertext errors in $R'$ to errors in $S'$, maintaining Invariant 4.1 in the respective rings. In the relative decoding basis $\overline{d}_{e', e'}$, ciphertext error $e = \overline{d}_{e', e'} \cdot \overline{e} \in R'$ has $E'$-coefficients $\overline{e}$ that satisfy the invariant for $E'$, and hence for $S'$ as well. Because we want

$$L'(e) = L'(\overline{d}_{e', e'} \cdot \overline{e}) = \overline{y} \cdot \overline{e} \in S'$$

to satisfy the invariant for $S'$, it is therefore best to lift $\overline{y}_p$ from $S'_p$ to $S'$ using the powerful basis, for the same reasons that apply to modulus switching when rescaling the $c_1$ component of a ciphertext (Section 4.4). \[13\]

\[13\] The very observant reader may notice that because $L'_p(\overline{d}_{e', e'}) = L_p(\overline{d}_{e, e})$ is over $S_p$, the order in which we extend and lift does not matter.
3. **Prepare** an appropriate key-switching hint using keys \( s_{\text{in}} \in R' \) and \( s_{\text{out}} \in S' \). Let \( \tilde{b} \) be an arbitrary \( E' \)-basis of \( R' \) (which we also use in the online phase below). Using a gadget vector \( \tilde{g} \) over \( S_{q'} \), generate key-switching hints \( H_j \) for the components of \( L' \cdot (s_{\text{in}} \cdot \tilde{b}) \), such that

\[
(1, s_{\text{out}}) \cdot H_j \approx L' (s_{\text{in}} \cdot b_j) \cdot \tilde{g} \quad (\text{mod } qS').
\]  

(As usual, the approximation hides appropriate Ring-LWE errors that satisfy Invariant 4.1.) Recall that we can interpret the columns of \( H_j \) as linear polynomials.

The online phase proceeds as follows. As input we are given an MSD-form, linear ciphertext \( c(S) = c_0 + c_1 S \) (over \( R'_q \)) with associated integer \( k = 0 \) and arbitrary \( l \in \mathbb{Z}_p \), encrypting a message \( \mu \in R_p \) under secret key \( s_{\text{in}} \).

1. Express \( c_1 \) uniquely as \( c_1 = \tilde{b} \cdot \bar{c} \) for some \( \bar{c} \) over \( E'_q \) (where \( \tilde{b} \) is the same \( E' \)-basis of \( R' \) used in Step 3 above).

2. Compute \( L'(c_0) \in S'_{q'} \), apply the core key-switching operation to each \( e_j \) with hint \( H_j \), and sum the results. Formally, output a ciphertext having \( k = 0 \), the same \( l \in \mathbb{Z}_p \) as the input, and the linear polynomial

\[
c'(S) = L'(c_0) + \sum_j H_j \cdot g^{-1}(e_j) \quad (\text{mod } qS').
\]  

(4.7)

For correctness, notice that we have

\[
c_0 + s_{\text{in}} \cdot c_1 \approx \frac{q}{p} \cdot l^{-1} \cdot \mu \quad (\text{mod } qR')
\]

\[
\implies L'(c_0 + s_{\text{in}} \cdot c_1) \approx \frac{q}{p} \cdot l^{-1} \cdot L(\mu) \quad (\text{mod } qS'),
\]  

(4.8)

where the error in the second approximation is \( L' \) applied to the error in the first approximation, and therefore satisfies Invariant 4.1 by design of \( L' \). Then we have

\[
c'(s_{\text{out}}) \approx L'(c_0) + \sum_j L'(s_{\text{in}} \cdot b_j) \cdot \tilde{g} \cdot g^{-1}(e_j) \quad \text{(Equations (4.7), (4.6))}
\]

\[
= L'(c_0 + s_{\text{in}} \cdot \tilde{b} \cdot \bar{c})
\]

\[
= L'(c_0 + s_{\text{in}} \cdot c_1)
\]

\[
\approx \frac{q}{p} \cdot l^{-1} \cdot L(\mu) \quad (\text{mod } qS')
\]  

(E'-linearity of \( L' \))

(definition of \( \bar{c} \))

(4.8)

as desired, where the error in the first approximation comes from the hints \( H_j \).

**Comparison to ring hopping.** We now describe the efficiency advantages of ring tunneling versus ring hopping. We analyze the most natural setting where both the input and output ciphertexts are in CRT representation; in particular, this allows the process to be iterated as in [AP13].

Both ring tunneling and ring hopping convert a ciphertext over \( R'_q \) to one over \( S'_{q'} \), either via the greatest common subring \( E'_q \) (in tunneling) or the compositum \( T'_{q'} \) (in hopping). In both cases, the vast majority of the work happens during key-switching, where we compute one or more values \( H \cdot g^{-1}(c) \) for some hint \( H \) and ring element \( c \) (which may be over different rings). This proceeds in two main steps:

1. We convert \( c \) from CRT to powerful-basis representation for \( g^{-1} \)-decomposition, and then convert each entry of \( g^{-1}(c) \) to CRT representation. Each such conversion takes \( \Theta(n \log n) = \tilde{\Theta}(n) \) time in the dimension \( n \) of the ring that \( c \) resides in.
2. We multiply each column of \( H \) by the appropriate entry of \( g^{-1}(c) \), and sum. Because both terms are in CRT representation, this takes linear \( \Theta(n) \) time in the dimension \( n \) of the ring that \( H \) is over.

The total number of components of \( g^{-1}(c) \) is the same in both tunneling and hopping, so we do not consider it further in this comparison.

In ring tunneling, we switch \( \dim(R' / E') \) elements \( e_j \in E'_q \) (see Equation (4.7)) using the same number of hints over \( S'_q \). Thus the total cost is

\[
\dim(R' / E') \cdot (\tilde{\Theta}(\dim(S'_q)) + \Theta(\dim(R'))) = \tilde{\Theta}(\dim(R')) + \Theta(\dim(T')).
\]

By contrast, in ring hopping we first embed the ciphertext into the compositum \( T'_q \) and key-switch there. Because the compositum has dimension \( \dim(T') = \dim(R' / E') \cdot \dim(S') \), the total cost is

\[
\tilde{\Theta}(\dim(T'_q)) + \Theta(\dim(T')).
\]

The second (linear) terms of the above expressions, corresponding to Step 2, are essentially identical. For the first (superlinear) terms, we see that Step 1 for tunneling is at least a \( \dim(T' / R') = \dim(S' / E') \) factor faster than for hopping. In typical instantiations, this factor is a small prime between, say, 3 and 11, so the savings can be quite significant in practice.

References


A Haskell Background

In this section we give a brief primer on the basic syntax, concepts, and features of Haskell needed to understand the material in the rest of the paper. For further details, see the excellent tutorial [Lip11].

A.1 Types

Every well-formed Haskell expression has a particular type, which is known statically (i.e., at compile time). An expression’s type can be explicitly specified by a type signature using the :: symbol, e.g., \texttt{3 :: Integer} or \texttt{True :: Bool}. However, such low-level type annotations are usually not necessary, because Haskell has very powerful type inference, which can automatically determine the types of arbitrarily complex expressions (or declare that they are ill-typed).

Every \texttt{function}, being a legal expression, has a type, which is written by separating the types of the input(s) and the output with the \texttt{->} symbol, e.g., \texttt{xor :: Bool -> Bool -> Bool}. Functions can be either fully or only partially applied to arguments having the appropriate types, e.g., we have the expressions \texttt{xor False False :: Bool} and \texttt{xor True :: Bool -> Bool}, but not the ill-typed \texttt{xor 3}. Partial application works because \texttt{->} is right-associative, so the “true” type of \texttt{xor} is \texttt{Bool -> (Bool -> Bool)}, i.e., it takes a boolean as input and outputs a \texttt{function} that itself maps a boolean to a boolean. Functions can also take functions as inputs, e.g.,

\[
\texttt{selfCompose :: (Integer -> Integer) -> (Integer -> Integer)}
\]

takes any \texttt{f :: Integer -> Integer} as input and outputs another function (presumably representing \texttt{f \circ f}).

The names of \texttt{concrete} types, such as \texttt{Integer} or \texttt{Bool}, are always capitalized. This is in contrast with lower-case \texttt{type variables}, which can stand for any type (possibly subject to some constraints; see the next subsection). For example, the function \texttt{alwaysTrue :: a -> Bool} takes a value of any type, and outputs a boolean value (presumably \texttt{True}). More interestingly, \texttt{cons :: a -> [a] -> [a]} takes a value of any type, and a \texttt{list} of values all having that \texttt{same} type, and outputs a \texttt{list} of values of that type.

Types can be parameterized by other types. For example:

- The type \texttt{[]} seen just above is the generic “(ordered) list” type, whose single argument is the type of the listed values, e.g., \texttt{[Bool]} is the “list of booleans” type. (Note that \texttt{[a]} is just syntactic sugar for \texttt{[a] a}.)

- The type \texttt{Maybe} represents “either a value (of a particular type), or nothing at all;” the latter is typically used to signify an exception. Its single argument is the underlying type, e.g., \texttt{Maybe Integer}.


• The generic “pair” type (,) takes two arguments that specify the types being paired together, e.g., (Integer,Bool).

Only fully applied types can admit values, e.g., there are no values of type [], Maybe, or (Integer,).

A.2 Type Classes

Type classes, or just classes, define abstract interfaces that types can implement, and are therefore a primary mechanism for obtaining polymorphism. For example, the Additive class (from the numeric prelude [1]) represents types that form abelian additive groups. As such, it introduces the terms

\[
\begin{align*}
\text{zero} & \quad :: \text{Additive} \quad a \Rightarrow a \\
\text{negate} & \quad :: \text{Additive} \quad a \Rightarrow a \rightarrow a \\
(+) & \quad , \quad (-) \quad :: \text{Additive} \quad a \Rightarrow a \rightarrow a \rightarrow a
\end{align*}
\]

In type signatures like the ones above, the text preceding the \(\Rightarrow\) symbol specifies the class constraint(s) on the type variable(s). The constraints Additive a seen above simply mean that the type represented by a must be an instance of the Additive class. A type is made an instance of a class via an instance declaration, which simply defines the actual behavior of the class’s terms for that particular type. For example, Integer and Double are instances of Additive. While Bool is not, it could be made one via the instance declaration

\[
\text{instance Additive Bool where}
\begin{align*}
\text{zero} & \quad = \text{False} \\
\text{negate} & \quad = \text{id} \\
(+) & \quad , \quad (-) \quad = \text{xor} \quad -- \text{same for ()}
\end{align*}
\]

Using class constraints, one can write polymorphic expressions using the terms associated with the corresponding classes. For example, we can define double :: Additive a => a -> a as double x = x + x. The use of (+) here is legal because the input x has type a, which is constrained to be an instance of Additive by the type of double. As a slightly richer example, we can define

\[
\begin{align*}
\text{isZero} & \quad :: (\text{Eq} \ a, \ \text{Additive} \ a) \Rightarrow a \rightarrow \text{Bool} \\
\text{isZero} \ x & \quad = x \Rightarrow \text{zero}
\end{align*}
\]

where the class Eq introduces the function (==) :: Eq a => a -> a -> Bool to represent types whose values can be tested for equality.

The definition of a class C can declare other classes as superclasses, which means that any type that is an instance of C must also be an instance of each superclass. For example, the class Ring from numeric prelude, which represents types that form rings with identity, has Additive as a superclass; this is done by writing class Additive r => Ring r in the class definition. One advantage of superclasses is that they help reduce the complexity of class constraints. For example, we can define f :: Ring r => r -> r as f x = one + double x, where the term one :: Ring r => r is introduced by Ring, and double is as defined above. The use of (+) and double is legal here, because f’s input x has type r, which (by the class constraint on f) is an instance of Ring and hence also of Additive.

---

14Operators like +, -, *, /, and == are merely functions introduced by various type classes. Function names consisting solely of special characters can be used in infix form in the expected way, but in all other contexts they must be surrounded by parentheses.

15Notice the type inference here: the use of (==) means that x and zero must have the same type a (which is an instance of Additive), so there is no ambiguity about which implementation of zero to use.

16It is generally agreed that the arrow points in the wrong direction, but for historical reasons we are stuck with this syntax.
So far, the discussion has been limited to *single-parameter* classes: a type either is, or is not, an instance of the class. In other words, such a class can be seen as merely the set of its instance types. More generally, *multi-parameter* classes express *relations* among types. For example, the two-argument class definition

```haskell
class (Ring r, Additive a) => Module r a where
  (*>) :: Module r a => r -> a -> a.
```

## B  More on Type-Level Cyclotomic Indices

Picking up from Section 2.6, in Section B.1 we give more details on how cyclotomic indices are represented and operated upon at the type level. Then in Section B.2 we describe how all this is used to generically derive algorithms for arbitrary cyclotomics.

### B.1  Promoting Factored Naturals

Operations in a cyclotomic ring are largely governed by the prime-power factorization of its index. Therefore, we define the data types *PrimePower* and *Factored* to represent factored positive integers (note that type *Nat* is a standard Peano encoding of the nonnegative integers, though any other representation would work just as well):

```haskell
newtype PrimePower = PP (Nat, Nat)
newtype Factored = F [PrimePower]
```

To enforce the invariants, we hide the `PP` and `F` constructors from clients, and instead export only legal values and operations that maintain the invariants. For example, we have the following values and functions, whose implementations are straightforward:

```haskell
f1, f2, f3, f4, ... :: Factored -- naturals in factored form
fDivides :: Factored -> Factored -> Bool
fMul, fGCD, fLCM :: Factored -> Factored -> Factored
```

We use data kinds and singletons to mechanically promote all these terms to the type level. Concretely, the above values `f1`, `f2`, `f3`, etc. yield the types `F1`, `F2`, `F3`, etc., whose inhabiting values are just the singletons `sf1 :: Sing m`, `sf2 :: Sing m`, etc. Note that we also can obtain the singleton value of *any* promoted type in a uniform manner via the term `sing`; e.g., `sing :: Sing m` yields the singleton value of promoted type `m`. We can also go in the reverse direction using the “magic” `withSingI` function, which lets us use a singleton *value* to set a corresponding *type variable* in an expression, e.g., `withSingI sf5 (one :: Cyc RT m Int)`. Finally, we can `reflect` any singleton value back to the original value that defined the singleton’s type, via the function `fromSing`; e.g., `fromSing (sf2 :: F2)` yields `f2`.

Analogously, promoting the above functions yields the type families `FDivides`, `FMul`, `FGCD`, and `FLCM`, which we can apply to the promoted types. For example, `FMul F2 F2` yields the type `F4`, as does `FGCD F12 F8`. Similarly, `FDivides F5 F30` yields the type `True`. (Nearly all values from Haskell’s standard types, like `Bool` in this case, are themselves automatically promoted to types.)
B.2 Applying the Promotions

Here we summarize how we use the promoted types and singletons to generically derive algorithms for working in arbitrary cyclotomics. We also use the "sparse decomposition" framework described in Appendix C below; for our purposes here, we only need that the type Trans r represents linear transforms over base ring r via their sparse decompositions.

A detailed example will illustrate our approach: consider the polymorphic function

\[ \text{crt} :: \text{(Fact } m, \text{ CRTrans } r, \ldots) \Rightarrow \text{Tagged } m \text{ (Trans } r) \]

which represents the index-m Chinese Remainder Transform (CRT) over a base ring r (e.g., \( \mathbb{Z}_q \) or \( \mathbb{C} \)). Equation (D.7) gives a sparse decomposition of CRT in terms of prime-power indices, and Equations (D.8) and (D.9) give sparse decompositions for the prime-power case in terms of CRT and DFT for prime indices, and "twiddle" transforms for prime-power indices.

Following these decompositions, our implementation of \( \text{crt} \) works as follows:

1. It first reflects the Factored value represented by type \( m \), using fromSing (sing :: Sing m), and extracts the list of PrimePower factors. For each of these, it tensors the appropriate specializations of the prime-power CRT function

\[ \text{crtPP} :: \text{(PPow } pp, \text{ CRTrans } r, \ldots) \Rightarrow \text{Tagged } pp \text{ (Trans } r) \]

The correct specializations are obtained by "elevating" the PrimePower values to the pp type variable using withSingI, as described above.

In fact, this reduction from Factored to PrimePower types applies equally well to all our transforms of interest. Therefore, we implement a completely generic combinator that builds a transform indexed by arbitrary (factored) \( m \) from one indexed by prime powers.

2. Similarly, \( \text{crtPP} \) reflects the PrimePower value represented by type \( pp \), extracts the Nat values of its prime and exponent, and composes the appropriate specializations of the prime-index CRT and DFT functions

\[ \text{crtP}, \, \text{dftP} :: \text{(NatC } p, \text{ CRTrans } r, \ldots) \Rightarrow \text{Tagged } p \text{ (Trans } r) \]

along with transforms that apply the appropriate "twiddle" factors.

3. Finally, \( \text{crtP} \) and \( \text{dftP} \) reflect the prime Nat value represented by type \( p \), and actually apply the CRT/DFT transformations indexed by this value (using the naive algorithms). This requires the \( p \)th roots of unity in \( r \), which are obtained via the CRTrans interface.

C Sparse Decompositions and Haskell Framework

As shown in Appendix D, the structure of the powerful, decoding, and CRT bases yield sparse decompositions, and thereby efficient algorithms, for cryptographically important linear transforms relating to these bases. Here we explain the principles of sparse decompositions, and summarize our Haskell framework for expressing and evaluating them.
C.1 Sparse Decompositions

A sparse decomposition of a matrix (or the linear transform it represents) is a factorization into sparser or more “structured” matrices, such as diagonal matrices or Kronecker products. Recall that the Kronecker (or tensor) product $A \otimes B$ of two matrices or vectors $A \in \mathbb{R}^{m_1 \times n_1}, B \in \mathbb{R}^{m_2 \times n_2}$ over a ring $\mathbb{R}$ is a matrix in $\mathbb{R}^{m_1 m_2 \times n_1 n_2}$. Specifically, it is the $m_1$-by-$n_1$ block matrix (or vector) made up of $m_2$-by-$n_2$ blocks, whose $(i, j)$th block is $a_{i,j} \cdot B \in \mathbb{R}^{m_2 \times n_2}$, where $A = (a_{i,j})$. The Kronecker product satisfies the properties

$$(A \otimes B)^t = (A^t \otimes B^t)$$
$$(A \otimes B)^{-1} = (A^{-1} \otimes B^{-1})$$

and the mixed-product property

$$(A \otimes B) \cdot (C \otimes D) = (AC) \otimes (BD),$$

which we use extensively in what follows.

A sparse decomposition of a matrix $A$ naturally yields an algorithm for multiplication by $A$, which can be much more efficient and parallel than the naïve algorithm. For example, multiplication by $I_n \otimes A$ can be done using $n$ parallel multiplications by $A$ on appropriate chunks of the input, and similarly for $A \otimes I_n$ and $I_l \otimes A \otimes I_r$. More generally, the Kronecker product of any two matrices can be expressed in terms of the previous cases, as follows:

$$A \otimes B = (A \otimes I_{\text{height}(B)}) \cdot (I_{\text{width}(A)} \otimes B) = (I_{\text{height}(A)} \otimes B) \cdot (A \otimes I_{\text{width}(B)}).$$

If the matrices $A, B$ themselves have sparse decompositions, then these rules can be applied further to yield a “fully expanded” decomposition. All the decompositions we consider in this work can be fully expanded as products of terms of the form $I_l \otimes A \otimes I_r$, where multiplication by $A$ is relatively fast, e.g., because $A$ is diagonal or has small dimensions.

C.2 Haskell Framework

We now describe a simple, deeply embedded domain-specific language for expressing and evaluating sparse decompositions in Haskell. It allows the programmer to write such factorizations recursively in natural mathematical notation, and it automatically yields fast evaluation algorithms corresponding to fully expanded decompositions. For simplicity, our implementation is restricted to square matrices (which suffices for our purposes), but it could easily be generalized to rectangular ones.

As a usage example, to express the decompositions

$$A = B \otimes C$$
$$B = (I_n \otimes D) \cdot E$$

where $C, D, E$ are “atomic,” one simply writes

```haskell
transA = transB @* transC -- A ⊗ B
transB = (Id n @* transD) .* transE -- (I_n ⊗ D) · E
transC = trans functionC -- similarly for transD, transE
```
where `functionC` is (essentially) an ordinary Haskell function that left-multiplies its input vector by \( C \). The above code causes `transA` to be internally represented as the fully expanded decomposition

\[
A = (I_n \otimes D \otimes I_{\dim(C)}) \cdot (E \otimes I_{\dim(C)}) \cdot (I_{\dim(E)} \otimes C).
\]

Finally, one simply writes `eval transA` to get an ordinary Haskell function that left-multiplies by \( A \) according to the above decomposition.

**Data types.** We first define the data types that represent transforms and their decompositions (here `Array r` stands for some arbitrary array type that holds elements of type `r`)

```haskell
-- (dim(f), f) such that (f l r) applies \( I_l \otimes f \otimes I_r \)
type Tensorable r = (Int, Int -> Int -> Array r -> Array r)

type TransC r = (Tensorable r, Int, Int)

data Trans r = Id Int   -- identity sentinel
| TSnoc (Trans r) (TransC r)
```

- The client-visible type alias `Tensorable r` represents an “atomic” transform (over the base type `r`) that can be augmented (tensored) on the left and right by identity transforms of any dimension. It has two components: the dimension \( d \) of the atomic transform \( f \) itself, and a function that, given any dimensions \( l, r \), applies the \( ldr \)-dimensional transform \( I_l \otimes f \otimes I_r \) to an array of \( r \)-elements. (Such a function could use parallelism internally, as already described.)

- The type alias `TransC r` represents a *transform component*, namely, a `Tensorable r` with particular values for \( l, r \). `TransC` is only used internally; it is not visible to external clients.

- The client-visible type `Trans r` represents a full transform, as a sequence of zero or more components terminated by a sentinel representing the identity transform. For such a sequence to be well-formed, all the components (including the sentinel) must have the same dimension. Therefore, we export the `Id` constructor, but not `TSnoc`, so the only way for a client to construct a nontrivial `Trans r` is to use the functions described below (which maintain the appropriate invariant).

**Evaluation.** Evaluating a transform is straightforward. Simply evaluate each component in sequence:

```haskell
evalC :: TransC r -> Array r -> Array r
evalC ((\_, f), l, r) = f l r

eval :: Trans r -> Array r -> Array r
eval (Id \_) = id   -- identity function
neval (TSnoc rest f) = eval rest . evalC f
```

**Constructing transforms.** We now explain how transforms of type `Trans r` are constructed. The function `trans` wraps a `Tensorable` as a full-fledged transform:
More interesting are the functions for composing and tensoring transforms, respectively denoted by the operators (\(\cdot\)), (\(\ast\)) :: \textbf{Trans} \(r\) \(\rightarrow\) \textbf{Trans} \(r\) \(\rightarrow\) \textbf{Trans} \(r\). Composition just appends the two sequences of components, after checking that their dimensions match; we omit its straightforward implementation. The Kronecker-product operator (\(\ast\)) simply applies the appropriate rules to get a fully expanded decomposition:

\[ I_m \otimes I_n = I_{mn} \]
\[ (\text{Id} \ m) \ast (\text{Id} \ n) = \text{Id} \ (m \ast n) \]

\[ (A \otimes B) = (I_n \otimes A) \cdot (I_n \otimes B), \text{ and similarly} \]
\[ i@\text{Id} \ n \ast (\text{TSnoc} \ a \ (b, 1, r)) = \text{TSnoc} \ (i \ast a) \ (b, (n \ast 1), r) \]
\[ (\text{TSnoc} \ a \ (b, 1, r)) \ast i@\text{Id} \ n = \text{TSnoc} \ (a \ast i) \ (b, 1, (r \ast n)) \]

\[ (A \otimes B) = (A \otimes I) \cdot (I \otimes B) \]
\[ a \ast b = (a \ast \text{Id} \ (\dim b)) \ast (\text{Id} \ (\dim a) \ast b) \]

(The \textbf{dim} function simply returns the dimension of a transform, via the expected implementation.)

\section{Tensor Interface and Implementation}

In this section we detail the “backend” representations and algorithms for computing in cyclotomic rings. We implement these algorithms using the sparse decomposition framework outlined in Appendix C.

An element of the \(m\)th cyclotomic ring over a base ring \(r\) (e.g., \(\mathbb{Q}\), \(\mathbb{Z}\), or \(\mathbb{Z}_q\)) can be represented as a vector of \(n = \varphi(m)\) coefficients from \(r\), with respect to a particular \(r\)-basis of the cyclotomic ring. We call such a vector a \textbf{(coefficient) tensor} to emphasize its implicit multidimensional nature, which arises from the tensor-product structure of the bases we use.

The class \textbf{Tensor} (see Figure 3) represents the cryptographically relevant operations on coefficient tensors with respect to the powerful, decoding, and CRT bases. An instance of \textbf{Tensor} is a data type \(t\) that itself takes two type parameters: an \(m\) representing the cyclotomic index, and an \(r\) representing the base ring. So the fully applied type \(t \ m \ r\) represents an index-\(m\) cyclotomic tensor over \(r\).

The \textbf{Tensor} class introduces a variety of methods representing linear transformations that either convert between two particular bases (e.g., \texttt{linv}, \texttt{crt}), or perform operations with respect to certain bases (e.g., \texttt{mulGpow}, \texttt{embedDec}). It also exposes some important fixed values related to cyclotomic ring extensions (e.g., \texttt{powBasisPow}, \texttt{crtSetDec}). An instance \(t\) of \textbf{Tensor} must implement all these methods and values for arbitrary (legal) cyclotomic indices.

\subsection{Mathematical Background}

Here we recall the relevant mathematical background on cyclotomic rings, largely following [LPR13, AP13] (with some slight modifications for convenience of implementation).

\subsubsection{Cyclotomic Rings and Powerful Bases}

\textbf{Prime cyclotomics.} The first cyclotomic ring is \(O_1 = \mathbb{Z}\). For a prime \(p\), the \(p\)th cyclotomic ring is \(O_p = \mathbb{Z}[\zeta_p]\), where \(\zeta_p\) denotes a primitive \(p\)th root of unity, i.e., \(\zeta_p\) has multiplicative order \(p\). The minimal
class Tensor t where
  -- single-index transforms
  scalarPow :: (Ring r, Fact m) => r -> t m r
  scalarCRT :: (CRTrans r, Fact m, ...) => Maybe (r -> t m r)
  l, lInv :: (Ring r, Fact m) => t m r -> t m r
  mulGPow, mulGDec :: (Ring r, Fact m) => t m r -> t m r
  divGPow, divGDec :: (IDZT r, Fact m) => t m r -> Maybe (t m r)
  crt, crtInv, mulGCRT, divGCRT :: (CRTrans r, IDZT r, Fact m)
    => Maybe (t m r -> t m r)
  tGaussianDec :: (OrdFloat q, Fact m, MonadRandom rnd, ...)
    => v -> rnd (t m q)
  -- two-index transforms and values
  embedPow, embedDec :: (Ring r, m `Divides` m') => t m r -> t m' r
  twacePowDec :: (Ring r, m `Divides` m') => t m' r -> t m r
  embedCRT :: (CRTrans r, IDZT r, m `Divides` m') => Maybe (t m r -> t m' r)
  twaceCRT :: (CRTrans r, IDZT r, m `Divides` m') => Maybe (t m' r -> t m r)
  coeffs :: (Ring r, m `Divides` m') => t m' r -> [t m r]
  powBasisPow :: (Ring r, m `Divides` m') => Tagged m [t m' r]
  crtSetDec :: (PrimeField fp, m `Divides` m', ...) => Tagged m [t m' fp]

Figure 3: Representative methods from the Tensor class. For the sake of concision, the constraint TElt t r is omitted from every method. The constraint IDZT r is a synonym for IntegralDomain r, ZeroTestable r.
polynomial over \(\mathbb{Z}\) of \(\zeta_p\) is \(\Phi_p(X) = 1 + X + X^2 + \cdots + X^{p-1}\), so \(\mathcal{O}_p\) has degree \(\varphi(p) = p - 1\) over \(\mathbb{Z}\), and we have the ring isomorphism \(\mathcal{O}_p \simeq \mathbb{Z}[X]/\Phi_p(X)\) by identifying \(\zeta_p\) with \(X\). The power basis \(\bar{p}_p\) of \(\mathcal{O}_p\) is the \(\mathbb{Z}\)-basis consisting of the first \(p - 1\) powers of \(\zeta_p\), i.e.,
\[
\bar{p}_p := (1, \zeta_p, \zeta_p^2, \ldots, \zeta_p^{p-2}).
\]

**Prime-power cyclotomics.** Now let \(m = p^e\) for \(e \geq 2\) be a power of a prime \(p\). Then we can inductively define \(\mathcal{O}_m = \mathcal{O}_{m/p}[\zeta_m]\), where \(\zeta_m\) denotes a primitive \(p\)th root of \(\zeta_{m/p}\). Its minimal polynomial over \(\mathcal{O}_{m/p}\) is \(X^p - \zeta_{m/p}\), so \(\mathcal{O}_m\) has degree \(p\) over \(\mathcal{O}_{m/p}\), and hence has degree \(\varphi(m) = (p - 1)p^{e-1}\) over \(\mathbb{Z}\).

The above naturally yields the relative power basis of the extension \(\mathcal{O}_m/\mathcal{O}_{m/p}\), which is the \(\mathcal{O}_{m/p}\)-basis
\[
\bar{p}_{m,m/p} := (1, \zeta_m, \ldots, \zeta_m^{p-1}).
\]

More generally, for any powers \(m, m'\) of \(p\) where \(m|m'\), we define the relative power basis \(\bar{p}_{m',m}\) of \(\mathcal{O}_{m'}/\mathcal{O}_m\) to be the \(\mathcal{O}_m\)-basis obtained as the Kronecker product of the relative power bases for each level of the tower:
\[
\bar{p}_{m',m} := \bar{p}_{m',m/p} \otimes \bar{p}_{m/p,m'/p^2} \otimes \cdots \otimes \bar{p}_{m/p,m}. \tag{D.1}
\]

Notice that because \(\zeta_{p^i} = \zeta_{m'}^{m'/p^i}\) for \(p^i \leq m'\), the relative power basis \(\bar{p}_{m',m}\) consists of all the powers \(0, \ldots, \varphi(m')/\varphi(m) - 1\) of \(\zeta_{m'}\), but in “base-\(p\) digit-reversed” order (which turns out to be more convenient for implementation). Finally, we also define \(\bar{p}_m := \bar{p}_{m,1}\) and simply call it the powerful basis of \(\mathcal{O}_m\).

**Arbitrary cyclotomics.** Now let \(m\) be any positive integer, and let \(m = \prod_{\ell=1}^t m_\ell\) be its factorization into maximal prime-power divisors \(m_\ell\) (in some canonical order). Then we can define
\[
\mathcal{O}_m := \mathbb{Z}[\zeta_{m_1}, \zeta_{m_2}, \ldots, \zeta_{m_t}] \tag{17}
\]

It is known that the rings \(\mathbb{Z}[\zeta_\ell]\) are linearly disjoint over \(\mathbb{Z}\), i.e., for any \(\mathbb{Z}\)-bases of the individual rings, their Kronecker product is a \(\mathbb{Z}\)-basis of \(\mathcal{O}_m\). In particular, the powerful basis of \(\mathcal{O}_m\) is defined as the Kronecker product of the component powerful bases:
\[
\bar{p}_m := \bigotimes_{\ell} \bar{p}_{m_\ell}. \tag{D.2}
\]

Similarly, for \(m|m'|\) having factorizations \(m = \prod_{\ell} m_\ell, m' = \prod_{\ell} m'_\ell\), where each \(m_\ell, m'_\ell\) is a power of a distinct prime \(p_\ell\) (so some \(m_\ell\) may be 1), the relative powerful basis of \(\mathcal{O}_{m'}/\mathcal{O}_m\) is
\[
\bar{p}_{m',m} := \bigotimes_{\ell} \bar{p}_{m'_\ell,m_\ell}. \tag{D.3}
\]

Notice that for \(m|m'|m''\), we have that \(\bar{p}_{m'',m}\) and \(\bar{p}_{m'',m'} \otimes \bar{p}_{m',m}\) are equivalent up to order, because they are tensor products of the same components, but possibly in different orders.

---

17Equivalently, \(\mathcal{O}_m = \bigotimes_\ell \mathcal{O}_{m_\ell}\) is the ring tensor product over \(\mathbb{Z}\) of all the \(m_\ell\)th cyclotomic rings; see Appendix E.
Canonical embedding. The $m$th cyclotomic ring $R$ has $n = \varphi(m)$ distinct ring embeddings (i.e., injective ring homomorphisms) into the complex numbers $\mathbb{C}$. Concretely, if $m$ has prime-power factorization $m = \prod \ell^e m_\ell$, then these embeddings are defined by mapping each $\zeta_m$ to each of the primitive $m_\ell$th roots of unity in $\mathbb{C}$, in all combinations. The canonical embedding $\sigma : R \to \mathbb{C}^n$ is defined as the concatenation of all these embeddings, in some standard order. (Notice that the embeddings come in conjugate pairs, so $\sigma$ actually maps into an $n$-dimensional real subspace $H \subseteq \mathbb{C}^n$.) The canonical embedding endows the ring (and its ambient number field) with a canonical geometry, i.e., all geometric quantities on $R$ are defined in terms of the canonical embedding. E.g., we have the Euclidean norm $\|x\| := \|\sigma(x)\|_2$. A key property is that both addition and multiplication in the ring are coordinate-wise in the canonical embedding:

\[
\sigma(a + b) = \sigma(a) + \sigma(b) \\
\sigma(a \cdot b) = \sigma(a) \circ \sigma(b).
\]

This property aids analysis and allows for sharp bounds on the growth of errors in cryptographic applications.

D.1.2 (Tweaked) Trace, Dual Ideal, and Decoding Bases

In what follows let $R = \mathcal{O}_m$, $R' = \mathcal{O}_{m'}$ for $m | m'$, so we have the ring extension $R' / R$. The trace function $\text{Tr}_{R'/R} : R' \to R$ is the $R$-linear function defined as follows: fixing any $R$-basis of $R'$, multiplication by an $x \in R'$ can be represented as a matrix $M_x$ over $R$ with respect to the basis, which acts on the multiplicand’s vector of $R$-coefficients. Then $\text{Tr}_{R'/R}(x)$ is simply the trace of $M_x$, i.e., the sum of its diagonal entries. (This is invariant under the choice of basis.) Because $R'/R$ is Galois, the trace can also be defined as the sum of the automorphisms of $R'$ that fix $R$ pointwise. All of this extends to the field of fractions of $R'$ (i.e., its ambient number field) in the same way.

Notice that the trace does not fix $R$ (except when $R' = R$), but rather $\text{Tr}_{R'/R}(x) = \deg(R'/R) \cdot x$ for all $x \in R$. For a tower $R^m / R'/R$ of ring extensions, the trace satisfies the composition property

\[
\text{Tr}_{R''/R'} = \text{Tr}_{R'/R} \circ \text{Tr}_{R''/R'}.
\]

The dual ideal, and a “tweak.” There is a special fractional ideal $R^\vee$ of $R$, called the codifferent or dual ideal, which is defined as the dual of $R$ under the trace, i.e.,

\[
R^\vee := \{ \text{fractional } a : \text{Tr}_{R/Z}(a \cdot R) \subseteq \mathbb{Z} \}.
\]

By the composition property of the trace, $(R')^\vee$ is the set of all fractional $a$ such that $\text{Tr}_{R'/R}(a \cdot R') \subseteq R^\vee$. In particular, we have $\text{Tr}_{R''/R}(R^\vee') = R^\vee$.

Concretely, the dual ideal is the principal fractional ideal $R^\vee = (g_m / \hat{m}) R$, where $\hat{m} = m / 2$ if $m$ is even and $\hat{m} = m$ otherwise, and the special element $g_m \in R$ is as follows:

- for $m = p^e$ for prime $p$ and $e \geq 1$, we have $g_m = g_p := 1 - \zeta_p$ if $p$ is odd, and $g_m = g_p := 1$ if $p = 2$; 
- for $m = \prod \ell m_\ell$ where the $m_\ell$ are powers of distinct primes, we have $g_m = \prod \ell g_{m_\ell}$.

The dual ideal $R^\vee$ plays a very important role in the definition, hardness proofs, and cryptographic applications of Ring-LWE (see [LPR10, LPR13] for details). However, for implementations it seems preferable to work entirely in $R$, so that we do not have to contend with fractional values or the dual ideal explicitly. Following [API13] and the discussion in Section 3.1 we achieve this by multiplying all
values related to $R^\vee$ by the “tweak” factor $t_m = \hat{m}/g_m \in R$; recall that $t_m R^\vee = R$. To compensate for this implicit tweak factor, we replace the trace by what we call the twace (for “tweaked trace”) function $T_{w_{m'},m} = Tw_{R'/R} : R' \to R$, defined as

$$
Tw_{R'/R}(x) := t_m \cdot Tr_{R'/R}(x/t_{m'}) = (\hat{m}/\hat{m}') \cdot Tr_{R'/R}(x \cdot g_{m'}/g_m).
$$

(D.4)

A nice feature of the twace is that it fixes the base ring pointwise, i.e., $Tw_{R'/R}(x) = x$ for every $x \in R$. It is also easy to verify that it satisfies the same composition property that the trace does.

We stress that this “tweaked” perspective is mathematically and computationally equivalent to using $R^\vee$, and all the results from [LPR10, LPR13] can translate to this setting without any loss.

(Tweaked) decoding basis. The work of [LPR13] defines a certain $\mathbb{Z}$-basis $\tilde{b}_m = (b_j)$ of $R^\vee$, called the decoding basis. It is defined as the dual of the conjugated powerful basis $\tilde{p}_m = (p_j)$ under the trace:

$$
Tr_{R/\mathbb{Z}}(b_j \cdot p_j^{-1}) = \delta_{j,j'}
$$

for all $j, j'$. The key geometric property of the decoding basis is, informally, that the $\mathbb{Z}$-coefficients of any $e \in R^\vee$ with respect to $\tilde{b}_m$ are optimally small in relation to $\sigma(x)$, the canonical embedding of $e$. In other words, short elements like Gaussian errors have small decoding-basis coefficients.

With the above-described “tweak” that replaces $R^\vee$ by $R$, we get the $\mathbb{Z}$-basis

$$
\tilde{d}_m = (d_j) := t_m \cdot \tilde{b}_m,
$$

which we call the (tweaked) decoding basis of $R$. By definition, this basis is dual to the conjugated powerful basis $\tilde{p}_m$ under the twace:

$$
Tw_{R/\mathbb{Z}}(d_j \cdot p_j^{-1}) = \delta_{j,j'}.
$$

Because $g_m \cdot t_m = \hat{m}$, it follows that the coefficients of any $e \in R$ with respect to $\tilde{d}_m$ are identical to those of $g_m \cdot e \in g_m R = \hat{m} R^\vee$ with respect to the $\mathbb{Z}$-basis $g_m \cdot \tilde{d}_m = \hat{m} \cdot \tilde{b}_m$ of $g_m R$. Hence, they are optimally small in relation to $\sigma(g_m \cdot e)$\footnote{This is why Invariant 4.1 of our fully homomorphic encryption scheme (Section 4) requires $\sigma(e \cdot g_m)$ to be short, where $e$ is the error in the ciphertext.}.

Relative decoding basis. Generalizing the above, the relative decoding basis $\tilde{d}_{m',m}$ of $R'/R$ is dual to the (conjugated) relative powerful basis $\tilde{p}_{m',m}$ under $T_{w_{R'/R}}$. As such, $\tilde{d}_{m',m}$ (and in particular, $\tilde{d}_{m'}$ itself) has a Kronecker-product structure mirroring that of $\tilde{p}_{m',m}$ from Equations (D.1) and (D.3). Furthermore, by the results of [LPR13, Section 6], for a positive power $m$ of a prime $p$ we have

$$
\tilde{d}^t_{m',m/p} = \begin{cases} 
\tilde{p}^t_{m,m/p} \cdot L_p & \text{if } m = p \\
\tilde{p}^t_{m,m/p} & \text{otherwise},
\end{cases}
$$

(D.5)

where $L_p$ is the lower-triangular matrix with 1s throughout its lower triangle.
D.1.3 Chinese Remainder Bases

Let $m$ be the index of cyclotomic ring $R = \mathcal{O}_m$, let $q = 1 \pmod{m}$ be a prime integer, and consider the quotient ring $R_q = R/qR$, i.e., the $n$th cyclotomic over base ring $\mathbb{Z}_q$. This ring has a Chinese remainder (or CRT) $\mathbb{Z}_q$-basis $\bar{c} = \bar{c}_m \in R^\varphi(m)$, whose entries are indexed by $\mathbb{Z}_m^*$. The key property satisfied by this basis is

$$c_i \cdot c_i' = \delta_{i,i'} \cdot c_i \quad (D.6)$$

for all $i, i' \in \mathbb{Z}_m^*$. Therefore, multiplication of ring elements represented in the CRT basis is coefficient-wise (and hence linear time): for any coefficient vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^\varphi(m)$, we have

$$(\bar{c}^t \cdot \mathbf{a}) \cdot (\bar{c}^t \cdot \mathbf{b}) = \bar{c}^t \cdot (\mathbf{a} \odot \mathbf{b}).$$

Also by Equation (D.6), the matrix corresponding to multiplication by $c_i$ (with respect to the CRT basis) has one in the $i$th diagonal entry and zeros everywhere else, so the trace of every CRT basis element is unity: $\text{Tr}_{\mathbb{R}/\mathbb{Z}}(\bar{c}) = 1 \pmod{q}$. For completeness, in what follows we describe the explicit construction of the CRT basis.

**Arbitrary cyclotomies.** For an arbitrary index $m$, the CRT basis is defined in terms of the prime-power factorization $m = \prod_{\ell} m_\ell$. Recall that $R_q = \mathbb{Z}_q[\zeta_{m_1}, \ldots, \zeta_{m_\ell}]$, and that the natural homomorphism $\phi: \mathbb{Z}_m^* \to \prod_{\ell} \mathbb{Z}_{m_\ell}^*$ is a group isomorphism. Using this, we can equivalently index the CRT basis by $\prod_{\ell} \mathbb{Z}_{m_\ell}^*$. With this indexing, the CRT basis $\bar{c}_m$ of $R_q$ is the Kronecker product of the CRT bases $\bar{c}_{m_\ell}$ of $\mathbb{Z}_q[\zeta_{m_\ell}]$:

$$\bar{c}_m = \bigotimes_\ell \bar{c}_{m_\ell},$$

i.e., the $\phi(i)$th entry of $\bar{c}_m$ is the product of the $\phi(i)$th entry of $\bar{c}_{m_\ell}$, taken over all $\ell$. It is easy to verify that Equation (D.6) holds for $\bar{c}_m$, because it does for all the $\bar{c}_{m_\ell}$.

**Prime-power cyclotomies.** Now let $m$ be a positive power of a prime $p$, and let $\omega_m \in \mathbb{Z}_q^*$ be an element of order $m$ (i.e., a primitive $m$th root of unity), which exists because $\mathbb{Z}_q^*$ is a cyclic group of order $q - 1$, which is divisible by $m$. We rely on two standard facts:

1. the Kummer-Dedekind Theorem, which implies that the ideal $qR = \prod_{\ell \mid q} q_\ell$ factors into the product of $\varphi(m)$ distinct prime ideals $q_\ell = (\zeta_m - \omega_m^{q_\ell})R + qR \subset R$; and

2. the Chinese Remainder Theorem (CRT), which implies that the natural homomorphism from $R_q$ to the product ring $\prod_{\ell} \mathbb{Z}_{m_\ell}^* \otimes q_\ell$ is a ring isomorphism.

Using this isomorphism, the basis $\bar{c}_m$ is defined so that its $i$th entry $c_i \in R_q$ satisfies $c_i = \delta_{i,i'} \pmod{q_\ell}$ for all $i, i' \in \mathbb{Z}_m^*$. Observe that this definition clearly satisfies Equation (D.6).

Like the powerful and decoding bases, for any extension $R'/R_q$ where $R' = \mathcal{O}_{m'}$, $R = \mathcal{O}_m$ for powers $m|m'$ of $p$, there is a relative CRT $R_q$-basis $\bar{c}_{m',m}$ of $R'_q$, which has a Kronecker-product factorization mirroring the one in Equation (D.1). The elements of this $R_q$-basis satisfy Equation (D.6), and hence their traces into $R_q$ are all unity. We defer a full treatment to Section D.4, where we consider a more general setting (where possibly $q \not\equiv 1 \pmod{m}$) and define and compute relative CRT sets.
D.2 Single-Index Transforms

In this and the next subsection we describe sparse decompositions for all the Tensor operations. We start here with the dimension-preserving transforms involving a single index \( m \), i.e., they take an index-\( m \) tensor as input and produce one as output.

D.2.1 Prime-Power Factorization

For an arbitrary index \( m \), every transform of interest factors into the tensor product of the corresponding transforms for prime-power indices. More specifically, let \( T_m \) denote the matrix for any of the linear transforms on index-\( m \) tensors that we consider below. Then letting \( m = \prod \ell \ m_\ell \) be the factorization of \( m \) into its maximal prime-power divisors \( m_\ell \) (in some canonical order), we have the factorization

\[
T_m = \bigotimes \ell T_{m_\ell} \quad .
\] (D.7)

This follows directly from the Kronecker-product factorizations of the powerful, decoding, and CRT bases (e.g., Equation (D.2)), and the mixed-product property. Therefore, for the remainder of this subsection we only deal with prime-power indices \( m = p^e \) for a prime \( p \) and positive integer \( e \).

D.2.2 Embedding Scalars

Consider a scalar element \( a \) from the base ring, represented relative to the powerful basis \( \bar{p}_m \). Because the first element of \( \bar{p}_m \) is unity, we have

\[
a = \bar{p}_m^t \cdot (a \cdot e_1),
\]

where \( e_1 = (1, 0, \ldots, 0) \). Similarly, in the CRT basis \( \bar{c}_m \) (when it exists), unity has the all-ones coefficient vector \( 1 \). Therefore,

\[
a = \bar{c}_m^t \cdot (a \cdot 1).
\]

The Tensor methods scalarPow and scalarCRT use the above equations to represent a scalar from the base ring as a coefficient vector relative to the powerful and CRT bases, respectively. Note that scalarCRT itself is wrapped by Maybe, so that it can be defined as Nothing if there is no CRT basis over the base ring.

D.2.3 Converting Between Powerful and Decoding Bases

Let \( L_m \) denote the matrix of the linear transform that converts from the decoding basis to the powerful basis:

\[
\bar{d}_m^t = \bar{p}_m^t \cdot L_m
\]

i.e., a ring element with coefficient vector \( \bar{v} \) in the decoding basis has coefficient vector \( L_m \cdot \bar{v} \) in the powerful basis. Because \( \bar{d}_m = \bar{p}_{m,p} \otimes \bar{d}_{p,1} \) and \( \bar{d}_{p,1} = \bar{p}_{p,1}^t \cdot L_p \) (both by Equation (D.5)), we have

\[
\bar{d}_m^t = (\bar{p}_{m,p}^t \cdot I_{m/p}) \otimes (\bar{p}_{p,1}^t \cdot L_p)
= \bar{p}_m^t \cdot (I_{m/p} \otimes L_p)
\]

L_{m/P}
Recall that \( L_p \) is the square \( \varphi(p) \)-dimensional lower-triangular matrix with 1s throughout its lower-left triangle, and \( L_p^{-1} \) is the lower-triangular matrix with 1s on the diagonal, \(-1\)s on the subdiagonal, and 0s elsewhere. We can apply both \( L_p \) and \( L_p^{-1} \) using just \( p - 1 \) additions, by taking partial sums and successive differences, respectively.

The Tensor methods 1 and 1Inv represent multiplication by \( L_m \) and \( L_m^{-1} \), respectively.

### D.2.4 Multiplication by \( g_m \)

Let \( G_p^\text{pow} \) denote the matrix of the linear transform representing multiplication by \( g_m \) in the powerful basis, i.e.,

\[
g_m \cdot \overrightarrow{p}_m = \overrightarrow{p}_m \cdot G_m^\text{pow}.
\]

Because \( g_m = g_p \in \mathcal{O}_p \) and \( \overrightarrow{p}_m = \overrightarrow{p}_{m,p} \otimes \overrightarrow{p}_p \), we have

\[
g_m \cdot \overrightarrow{p}_m = (\overrightarrow{p}_{m,p} \cdot I_{m/p}) \otimes (\overrightarrow{p}_p \cdot G_p^\text{pow}) = \overrightarrow{p}_m \cdot (I_{m/p} \otimes G_p^\text{pow}) = \overrightarrow{p}_m \cdot (G_p^\text{pow}^{-1}) = \overrightarrow{p}_m \cdot (G_p^\text{pow})^{-1},
\]

where \( G_p^\text{pow} \) and its inverse (which represents division by \( g_p \) in the powerful basis) are the square \( (p - 1) \)-dimensional matrices

\[
G_p^\text{pow} = \begin{pmatrix}
1 & 1 \\
-1 & \ddots & 1 \\
& \ddots & \ddots & \ddots \\
& & -1 & 1 & 1 \\
& & & -1 & 2
\end{pmatrix}, \quad (G_p^\text{pow})^{-1} = \begin{pmatrix}
p - 1 & \cdots & -1 & -1 & -1 \\
\vdots & \ddots & \ddots & \ddots & \ddots \\
3 & \cdots & 3 - p & 3 - p \\
2 & \cdots & 2 & 2 & 2 - p \\
1 & \cdots & 1 & 1 & 1
\end{pmatrix}.
\]

Identical decompositions hold for \( G_p^\text{dec} \) and \( G_p^\text{crt} \) (which represent multiplication by \( g_m \) in the decoding and CRT bases, respectively), where

\[
G_p^\text{dec} = \begin{pmatrix}
2 & 1 & \cdots & 1 \\
-1 & 1 & \ddots & \ddots \\
& \ddots & \ddots & \ddots \\
& & -1 & 1 & 1 \\
& & & -1 & 1
\end{pmatrix}, \quad (G_p^\text{dec})^{-1} = \begin{pmatrix}
1 & 2 - p & 3 - p & \cdots & -1 \\
1 & 2 & 3 - p & \cdots & -1 \\
1 & 2 & 3 & \cdots & -1 \\
\vdots & \vdots & \vdots & \ddots & \ddots \\
1 & 2 & 3 & \cdots & p - 1
\end{pmatrix},
\]

and \( G_p^\text{crt} \) is the diagonal matrix with \( 1 - \omega_p^i \) in the \( i \)th diagonal entry (indexed from 1 to \( p - 1 \)), where \( \omega_p \) is the same primitive \( p \)th root of unity in the base ring used to define the CRT basis.

The linear transforms represented by the above matrices can be applied in time linear in the dimension. For \( G_p^\text{pow} \), \( G_p^\text{dec} \), and \( G_p^\text{crt} \) and its inverse this is obvious, due to their sparsity. For \((G_p^\text{dec})^{-1}\), this follows from the fact that every row (apart from the top one) differs from the preceding one by a single entry. For \((G_p^\text{pow})^{-1}\), we can compute the entries of the output vector from the bottom up, by computing the sum of all the input entries and their partial sums from the bottom up.

The Tensor methods mulGPow and mulGDec represent multiplication by \( G_m^\text{pow} \) and \( G_m^\text{dec} \), respectively. Similarly, the methods divGPow and divGDec represent division by these matrices; note that their outputs are
wrapped by Maybe, so that the output can be Nothing when division fails. Finally, mulGCRT and divGCRT represent multiplication and division by \(G_{m}^{\text{crt}}\); note that these methods themselves are wrapped by Maybe, because \(G_{m}^{\text{crt}}\) and its inverse are well-defined over the base ring exactly when a CRT basis exists. (In this case, division always succeeds, hence no Maybe is needed for the output of divGCRT.)

### D.2.5 Chinese Remainder and Discrete Fourier Transforms

Consider a base ring, like \(\mathbb{Z}_q\) or \(\mathbb{C}\), that admits an invertible index-\(m\) Chinese Remainder Transform \(\text{CRT}_m\), defined by a principal \(m\)th root of unity \(\omega_m\). Then as shown in [LPR13 Section 3], this transform converts from the powerful basis to the CRT basis (defined by the same \(\omega_m\)), i.e.,

\[
p^i_m = \hat{c}^t_m \cdot \text{CRT}_m .
\]

Also as shown in [LPR13 Section 3], \(\text{CRT}_m\) admits the following sparse decompositions for \(m > p\),

\[
\text{CRT}_m = (\text{DFT}_{m/p} \otimes I_{p-1}) \cdot \hat{T}_m \cdot (I_{m/p} \otimes \text{CRT}_p) \quad (D.8)
\]

\[
\text{DFT}_m = (\text{DFT}_{m/p} \otimes I_p) \cdot T_m \cdot (I_{m/p} \otimes \text{DFT}_p) . \quad (D.9)
\]

(These decompositions can be applied recursively until all the CRT and DFT terms have subscript \(p\).) Here \(\text{DFT}_p\) is a square \(p\)-dimensional matrix with rows and columns indexed from zero, and \(\text{CRT}_p\) is its lower-left \((p-1)\)-dimensional square submatrix, with rows indexed from one and columns indexed from zero. The \((i,j)\)th entry of each matrix is \(\omega_p^{ij}\), where \(\omega_p = \omega_m^{m/p}\). Finally, \(\hat{T}_m, T_m\) are diagonal “twiddle” matrices whose diagonal entries are certain powers of \(\omega_m\).

For the inverses \(\text{CRT}_m^{-1}\) and \(\text{DFT}_m^{-1}\), by standard properties of matrix and Kronecker products, we have sparse decompositions mirroring those in Equations (D.8) and (D.9). Note that \(\text{DFT}_p\) is invertible if and only if \(p\) is invertible in the base ring, and the same goes for \(\text{CRT}_p\), except that \(\text{CRT}_2\) (which is just unity) is always invertible. More specifically, \(\text{DFT}_p^{-1} = p^{-1} \cdot \text{DFT}_p^*\), the (scaled) conjugate transpose of \(\text{DFT}_p\), whose \((i,j)\)th entry is \(\omega_p^{-ij}\). For \(\text{CRT}_p^{-1}\), it can be verified that for \(p > 2\),

\[
\text{CRT}_p^{-1} = p^{-1} \cdot (X-1 \cdot (\omega_p^1, \omega_p^2, \ldots, \omega_p^{p-1}))^{-1},
\]

where \(X\) is the upper-right \((p-1)\)-dimensional square submatrix of \(\text{DFT}_p^*\). Finally, note that in the sparse decomposition for \(\text{CRT}_m^{-1}\) (for arbitrary \(m\)), we can collect all the individual \(p^{-1}\) factors from the \(\text{CRT}_p^{-1}\) and \(\text{DFT}_p^{-1}\) terms into a single \(\hat{m}^{-1}\) factor. (This factor is exposed by the \text{CRTrans} interface; see Section 2.4)

The Tensor methods \text{crt} and \text{crtInv} respectively represent multiplication by \(\text{CRT}_m\) and its inverse. These methods themselves are wrapped by Maybe, so that they can be Nothing when there is no CRT basis over the base ring.

### D.2.6 Generating (Tweaked) Gaussians in the Decoding Basis

Cryptographic applications often need to sample secret error terms from a prescribed distribution. For RingLWE, error distributions \(D_r\) that correspond to (continuous) spherical Gaussians in the canonical embedding...
are particularly useful, and for sufficiently large \( r \) are supported by worst-case hardness proofs \([LPR10]\).
(The error can then be discretized in a variety of ways, with no loss in hardness.) Note, however, that all this is for the original definition of Ring-LWE involving the dual ideal \( R' \) (see Sections D.1.2 and 3.1).

With the “tweaked” perspective that replaces \( R' \) by \( R \) via the tweak factor \( t_m \in R \), we are interested in sampling from tweaked distributions \( t_m \cdot D_r \). More precisely, we want a randomized algorithm that samples a coefficient vector over \( R \), with respect to one of the standard bases of \( R \), of a random element that is distributed as \( t_m \cdot D_r \). This is not entirely trivial, because except in the power-of-two case, \( R \) does not have an orthogonal basis, and so the output coefficients will not be independent.

The material in \([LPR13, \text{Section 6.3}]\) yields a specialized, fast algorithm for sampling from \( D_r \) with output represented in the decoding basis \( b_m \) of \( R \). Equivalently, the very same algorithm samples from the tweaked Gaussian \( t_m \cdot D_r \) relative to the decoding basis \( d_m = t_m \cdot b_m \) of \( R \). The algorithm is faster (often much moreso) than the naïve one that applies a full \( \text{CRT}^* \) (over \( \mathbb{C} \)) to a Gaussian in the canonical embedding. The efficiency comes from skipping several layers of orthogonal transforms (namely, scaled DFTs and twiddle matrices), which is possible due to the rotation-invariance of spherical Gaussians. The algorithm also avoids complex numbers entirely, instead using only reals.

The algorithm. The sampling algorithm simply applies a certain linear transform over \( \mathbb{R} \), whose matrix \( E_m \) has a sparse decomposition as described below, to a vector of i.i.d. real Gaussian samples with parameter \( r \), and outputs the resulting vector. The Tensor method \( \text{tGaussianDec} \) implements the algorithm, given \( v = r^2 \).
(\text{Note that its output type \( \text{rnd Ht m qI} \) for \( \text{MonadRandom} \) \( \text{rnd} \) is necessarily monadic, because the algorithm is randomized.})

As with all the transforms considered above, we describe the sparse decomposition of \( E_m \) where \( m \) is a power of a prime \( p \), which then generalizes to arbitrary \( m \) as described in Section D.2.1 For \( m > p \), we have

\[
E_m = \sqrt{m/p} \cdot (I_{m/p} \otimes E_p),
\]

where \( E_2 \) is unity and \( E_p \) for \( p > 2 \) is

\[
E_p = \frac{1}{\sqrt{2}} \cdot \text{CRT}_p \cdot \begin{pmatrix} I & \sqrt{-1} J \\ J & -\sqrt{-1} I \end{pmatrix} \in \mathbb{R}^{(p-1) \times (p-1)},
\]

where \( \text{CRT}_p \) is over \( \mathbb{C} \), and \( J \) is the “reversal” matrix obtained by reversing the columns of the identity matrix. Expanding the above product, \( E_p \) has rows indexed from zero and columns indexed from one, and its \((i,j)\)th entry is

\[
\sqrt{2} \cdot \begin{cases} \cos \theta_{i,j} & \text{for } 1 \leq j < p/2 \\
\sin \theta_{i,j} & \text{for } p/2 < j \leq p - 1 \end{cases}, \quad \theta_k = 2\pi k/p.
\]

Finally, note that in the sampling algorithm, when applying \( E_m \) for arbitrary \( m \) with prime-power factorization \( m = \prod_{\ell} m_\ell \), we can apply all the \( \sqrt{m_\ell} \) scaling factors (from the \( E_{m_\ell} \) terms) to the parameter \( r \) of the Gaussian input vector, i.e., use parameter \( r \sqrt{m} / \text{rad}(m) \) instead.

---

20We remark that the signs of the rightmost block of the above matrix (containing \( -\sqrt{-1} J \) and \( \sqrt{-1} I \)) is swapped as compared with what appears in \([LPR13, \text{Section 6.3}]\). The choice of sign is arbitrary, because any orthonormal basis of the subspace spanned by the columns works equally well.
D.3 Two-Index Transforms and Values

We now consider transforms and special values relating the \( m \)th and \( m' \)th cyclotomic rings, for \( m | m' \). These are used for computing the embed and twace functions, the relative powerful basis, and the relative CRT set.

D.3.1 Prime-Power Factorization

As in the Section \[D.2\] every transform of interest for arbitrary \( m | m' \) factors into the tensor product of the corresponding transforms for prime-power indices having the same prime base. More specifically, let \( T_{m,m'} \) denote the matrix of any of the linear transforms we consider below. Suppose we have factorization \( m = \prod \ell m_\ell, \ m' = \prod \ell m'_\ell \) where each \( m_\ell, m'_\ell \) is a power of a distinct prime \( p_\ell \) (so some \( m_\ell \) may be 1). Then we have the factorization

\[
T_{m,m'} = \bigotimes_\ell T_{m_\ell,m'_\ell},
\]

which follows directly from the Kronecker-product factorizations of the powerful and decoding bases, and the mixed-product property. Therefore, from this point onward we deal only with prime-power indices \( m = p^e, \ m' = p^{e'} \) for a prime \( p \) and integers \( e' > e \geq 0 \).

We mention that for the transforms we consider below, the fully expanded matrices \( T_{m,m'} \) have very compact representations and can be applied directly to the input vector, without computing a sequence of intermediate vectors via the sparse decomposition. For efficiency, our implementation does exactly this.

D.3.2 Coefficients in Relative Bases

We start with transforms that let us represent elements with respect to relative bases, i.e., to represent an element of the \( m' \)th cyclotomic as a vector of elements in the \( m \)th cyclotomic, with respect to a relative basis. Due to the Kronecker-product structure of the powerful, decoding, and CRT bases, it turns out that the same transformation works for all of them. The \texttt{coeffs} method of \texttt{Tensor} implements this transformation.

One can verify the identity \((\vec{x} \otimes \vec{y})^t \cdot a = \vec{x}^t \cdot A \cdot \vec{y} \), where \( A \) is the “matricization” of the vector \( a \), whose rows are (the transposes of) the consecutive \( \dim(\vec{y}) \)-dimensional blocks of \( a \). Letting \( \vec{b}_\ell \) denote either the powerful, decoding, or CRT basis in the \( \ell \)th cyclotomic, which has factorization \( \vec{b}_{m'} = \vec{b}_{m',m} \otimes \vec{b}_m \), we have

\[
\vec{b}_{m'}^t \cdot a = \vec{b}_{m',m}^t \cdot (A \cdot \vec{b}_m).
\]

Therefore, \( A \cdot \vec{b}_m \) is the desired vector of \( R \)-coefficients of \( a = \vec{b}_{m'}^t \cdot a \in R' \). In other words, the \( \varphi(m) \)-dimensional blocks of \( a \) are the coefficient vectors (with respect to basis \( \vec{b}_m \)) of the \( R \)-coefficients of \( a \) with respect to the relative basis \( \vec{b}_{m',m} \).

D.3.3 Embed Transforms

We now consider transforms that convert from a basis in the \( m \)th cyclotomic to the same type of basis in the \( m' \)th cyclotomic. That is, for particular bases \( \vec{b}_{m'}, \vec{b}_m \) of the \( m' \)th and \( m \)th cyclotomics (respectively), we write

\[
\vec{b}_{m'}^t = \vec{b}_{m'}^t \cdot T
\]

for some integer matrix \( T \). So embedding a ring element from the \( m \)th to the \( m' \)th cyclotomic (with respect to these bases) corresponds to left-multiplication by \( T \). The \texttt{embedB} methods of \texttt{Tensor}, for \( B \in \{\text{Pow}, \text{Dec}, \text{CRT}\} \), implement these transforms.
We start with the powerful basis. Because \( \bar{p}_{m',m} = \bar{p}_{m',m} \otimes \bar{p}_m \) and the first entry of \( \bar{p}_{m',m} \) is unity,

\[
\bar{p}'_{m,m} = (\bar{p}'_{m',m} \cdot e_1) \otimes (\bar{p}_m \cdot I_{\varphi(m)}) = \bar{p}'_{m'} \cdot (e_1 \otimes I_{\varphi(m)}) ,
\]

where \( e_1 = (1, 0, \ldots, 0) \in \mathbb{Z}^{\varphi(m')/\varphi(m)} \). Note that \( (e_1 \otimes I_{\varphi(m)}) \) is the identity matrix stacked on top of an all-zeros matrix, so left-multiplication by it simply pads the input vector by zeros.

For the decoding bases \( \bar{m}_{m'}^t, \bar{m}_m \), an identical derivation holds when \( m > 1 \), because \( \bar{m}_{m'}^t = \bar{p}_{m',m} \otimes \bar{d}_m \). Otherwise, we have \( \bar{m}_{m'}^t = \bar{p}_{m',p} \otimes \bar{d}_m \) and \( \bar{m}_m = (1) = \bar{d}_p \cdot v \), where \( v = (1, -1, 0, \ldots, 0) \in \mathbb{Z}^{\varphi(p)} \).

Combining these cases, we have

\[
\bar{d}_m^t = \bar{d}_m^t \cdot \begin{cases} e_1 \otimes I_{\varphi(m)} & \text{if } m > 1 \\ e_1 \otimes v & \text{if } m = 1. \end{cases}
\]

For the CRT bases \( \bar{c}_{m'}, \bar{c}_m \), because \( \bar{c}_m = \bar{c}_{m',m} \otimes \bar{c}_m \) and the sum of the elements of any (relative) CRT basis is unity, we have

\[
\bar{c}_m^t = (\bar{c}_{m',m} \cdot 1) \otimes (\bar{c}_m \cdot I_{\varphi(m)}) = \bar{c}_m^t \cdot (1 \otimes I_{\varphi(m)}) .
\]

Notice that \( (1 \otimes I_{\varphi(m)}) \) is just a stack of identity matrices, so left-multiplication by it just stacks up several copies of the input vector.

Finally, we express the relative powerful basis \( \bar{p}_{m',m} \) with respect to the powerful basis \( \bar{p}_{m'} \); this is used in the powBasisPow method of Tensor. We simply have

\[
\bar{p}'_{m',m} = (\bar{p}'_{m',m} \cdot I_{\varphi(m')/\varphi(m)}) \otimes (\bar{p}_m \cdot e_1) = \bar{p}'_{m'} \cdot (I_{\varphi(m')/\varphi(m)} \otimes e_1),
\]

### D.3.4 Twace Transforms

We now consider transforms that represent the twace function from the \( m' \)th to the \( m \)th cyclotomic for the three basis types of interest. That is, for particular bases \( \bar{b}_{m'}, \bar{b}_m \) of the \( m' \)th and \( m \)th cyclotomics (respectively), we write

\[
Tw_{m',m}(\bar{b}_{m'}) = \bar{b}_m \cdot T
\]

for some integer matrix \( T \), which by linearity of twace implies

\[
Tw_{m',m}(\bar{b}_{m'}^t \cdot v) = \bar{b}_m^t \cdot (T \cdot v).
\]

In other words, the twace function (relative to the these bases) corresponds to left-multiplication by \( T \). The twacePowDec and twaceCRT methods of Tensor implement these transforms.

To start, we claim that

\[
Tw_{m',m}(\bar{p}_{m',m}) = Tw_{m',m}(\bar{d}_{m',m}) = e_1 \in \mathbb{Z}^{\varphi(m')/\varphi(m)} .
\] (D.10)

This holds for \( \bar{d}_{m',m} \) because it is dual to (conjugated) \( \bar{p}_{m',m} \) under \( Tw_{m',m} \), and the first entry of \( \bar{p}_{m',m} \) is unity. It holds for \( \bar{p}_{m',m} \) because \( \bar{p}_{m',m} = \bar{d}_{m',m} \) for \( m > 1 \), and for \( m = 1 \) one can verify that

\[
Tw_{m',1}(\bar{p}_{m',1}) = Tw_{p,1}(Tw_{m',p}(\bar{p}_{m',p} \otimes \bar{p}_{p,1})) = (1, 0, \ldots, 0) \otimes Tw_{p,1}(\bar{p}_{p,1}) = e_1 .
\]
Now for the powerful basis, by linearity of twace and Equation (D.10) we have

\[ Tw_{m', m}(p^{t}_{m'}) = Tw_{m', m}(p^{t}_{m', m}) \otimes p^{t}_{m} \]
\[ = (1 \cdot e_{1}^{t}) \otimes (p^{t}_{m} \cdot I_{\varphi(m)}) \]
\[ = p^{t}_{m} \cdot (e_{1}^{t} \otimes I_{\varphi(m)}) . \]

An identical derivation holds for the decoding basis as well. Notice that left-multiplication by the matrix \((e_{1}^{t} \otimes I_{\varphi(m)})\) just returns the first \(\varphi(m')/\varphi(m)\) entries of the input vector.

Finally, we consider the CRT basis. Because \(g_{m'} = g_{p}\) (recall that \(m' \geq p\)), by definition of twace in terms of trace we have

\[ Tw_{m', m}(x) = (\hat{m}/\hat{m}') \cdot g_{m}^{-1} \cdot Tr_{m', m}(g_{p} \cdot x). \]  
\[ \text{(D.11)} \]

Also recall that the traces of all relative CRT set elements are unity: \(Tr_{m', \ell}(\varpi_{m', \ell}) = 1 \cdot \varphi(m')/\varphi(\ell)\) for any \(\ell | m'\).

We now need to consider two cases. For \(m > 1\), we have \(g_{m} = g_{p}\), so by Equation (D.11) and linearity of trace,

\[ Tw_{m', m}(\varpi_{m', m}) = (\hat{m}/\hat{m}') \cdot 1 \cdot \varphi(m')/\varphi(m) . \]

For \(m = 1\), we have \(g_{m} = 1\), so by \(\varpi_{m', 1} = \varpi_{m', p} \otimes \varpi_{p, 1}\) and linearity of trace we have

\[ Tw_{m', 1}(\varpi_{m', 1}) = (\hat{m}/\hat{m}') \cdot Tr_{p, 1}(Tr_{m', p}(\varpi_{m', p} \otimes (g_{p} \cdot \varpi_{p, 1})) \]
\[ = (\hat{m}/\hat{m}') \cdot 1 \cdot \varphi(m')/\varphi(\ell) \otimes Tr_{p, 1}(g_{p} \cdot \varpi_{p, 1}) . \]

Applying the two cases, we finally have

\[ Tw_{m', m}(\varpi^{t}_{m'}) = (1 \cdot Tw_{m', m}(\varpi^{t}_{m', m})) \otimes (\varpi^{t}_{m} \cdot I_{\varphi(m)}) \]
\[ = \varpi^{t}_{m} \cdot (\hat{m}/\hat{m}') \cdot \begin{cases} 1 \cdot \varphi(m')/\varphi(m) \otimes I_{\varphi(m)} \text{ if } m > 1 \\ 1 \cdot \varphi(m')/\varphi(\ell) \otimes Tr_{p, 1}(g_{p} \cdot \varpi_{p, 1}) \text{ if } m = 1. \end{cases} \]

Again because \(Tr_{p, 1}(\varpi_{p, 1}) = 1 \cdot \varphi(\ell)\), the entries of \(Tr_{p, 1}(g_{p} \cdot \varpi_{p, 1})\) are merely the CRT coefficients of \(g_{p}\). That is, the \(i\)th entry (indexed from one) is \(1 - \omega_{p}^{m'}/\omega^{m'}_{p}\), where \(\omega_{p} = \omega_{m'}/\omega^{m'}_{p}\) for the value of \(\omega_{m'}\) used to define the CRT set of the \(m'\)th cyclotomic.

**D.4 CRT Sets**

In this final subsection we describe an algorithm for computing a representation of the relative CRT set \(\varpi_{m', m}\) modulo a prime-power integer. CRT sets are a generalization of CRT bases to the case where the prime modulus may not be 1 modulo the cyclotomic index (i.e., it does not split completely), and therefore the cardinality of the set may be less than the dimension of the ring. CRT sets are used for homomorphic SIMD operations [SV11] and in the bootstrapping algorithm of [AP13].

**D.4.1 Mathematical Background**

For a positive integer \(q\) and cyclotomic ring \(R\), let \(qR = \prod_{i} q_{i}^{c_{i}}\) be the factorization of \(qR\) into powers of distinct prime ideals \(q_{i} \subset R\). Recall that the Chinese Remainder Theorem says that the natural homomorphism from \(R_{q} = R/qR\) to the product ring \(\prod_{i}(R/q_{i}^{c_{i}})\) is a ring isomorphism.

**Definition D.1.** The CRT set of \(R_{q}\) is the vector \(\varpi\) over \(R_{q}\) such that \(c_{i} = \delta_{i, i'} (\mod q_{i}^{c_{i}'})\) for all \(i, i'\).
For a prime integer $p$, the prime-ideal factorization of $pR$ is as follows. For the moment assume that $p \nmid m$, and let $d$ be the order of $p$ modulo $m$, i.e., the smallest positive integer such that $p^d = 1 \pmod{m}$. Then $pR$ factors into the product of $\varphi(m)/d$ distinct prime ideals $p_i$, as described below:  

$$pR = \prod_i p_i.$$ 

Observe that the finite field $\mathbb{F}_{p^d}$ has a principal $m$th root of unity $\omega_m$, because $\mathbb{F}_{p^d}^*$ is cyclic and has order $p^d - 1 = 0 \pmod{m}$. Therefore, there are $\varphi(m)$ distinct ring homomorphisms $\rho_i : R \to \mathbb{F}_{p^d}$ indexed by $i \in \mathbb{Z}_m^*$, where $\rho_i$ is defined by mapping $\zeta_m$ to $\omega_m^i$.

The prime ideal divisors of $pR$ are indexed by the quotient group $G = \mathbb{Z}_m^*/\langle p \rangle$, i.e., the multiplicative group of cosets $i \langle p \rangle$ of the subgroup $\langle p \rangle = \{1, p, p^2, \ldots, p^{d-1}\}$ of $\mathbb{Z}_m^*$. For each coset $i = \bar{i} \langle p \rangle \in G$, the ideal $p_i$ is simply the kernel of the ring homomorphism $\rho_i$, for some arbitrary choice of representative $\bar{i} \in i$.

It is easy to verify that this is an ideal, and that it is invariant under the choice of representative, because $\rho_i(r) = \rho_i(r)p$ for any $r \in R$. (This follows from $(a + b)p = ap + bp$ for any $a, b \in \mathbb{F}_{p^d}$.)

Because $p_i$ is the kernel of $\rho_i$, the induced ring homomorphisms $\rho_i : R/p_i \to \mathbb{F}_{p^d}$ are in fact isomorphisms. In combination with the Chinese Remainder Theorem, their concatenation yields a ring isomorphism $\rho : R_p \to (\mathbb{F}_{p^d})^{\varphi(m)/d}$. In particular, for the CRT set $\bar{c}$ of $R_p$, for any $z \in R_p$, we have

$$\text{Tr}_{R_p/\mathbb{Z}_{p^d}}(z \cdot \bar{c}) = \text{Tr}_{\mathbb{F}_{p^d}/\mathbb{F}_p}(\rho(z)). \quad \text{(D.12)}$$

Finally, consider the general case where $p$ may divide $m$. It turns out that this case easily reduces to the one where $p$ does not divide $m$, as follows. Let $m = p^k \cdot \bar{m}$ for $p \nmid \bar{m}$, and let $\bar{R} = \mathcal{O}_{\bar{m}}$ and $p\bar{R} = \prod_i \bar{p}_i$, be the prime-ideal factorization of $p\bar{R}$ as described above. Then the ideals $\bar{p}_i \subset \bar{R}$ are totally ramified in $\bar{R}$, i.e., we have $\bar{p}_i \bar{R} = \bar{p}_i^{\varphi(\bar{m})/\varphi(m)}$ for some distinct prime ideals $\bar{p}_i \subset \bar{R}$. This implies that the CRT set for $\bar{R}_p$ is exactly the CRT set for $\bar{R}_p$, embedded into $\bar{R}_p$. Therefore, in what follows we restrict our attention to the case where $p$ does not divide $m$.

### D.4.2 Computing CRT Sets

We start with an easy calculation that, for a prime integer $p$, “lifts” the mod-$p$ CRT set to the mod-$p^e$ CRT set.

**Lemma D.2.** For $R = \mathcal{O}_m$, a prime integer $p$ where $p \nmid m$, and a positive integer $e$, let $(c_i)_i$ be the CRT set of $R_{p^e}$, and let $\bar{c}_i \in R$ be any representative of $c_i$. Then $(\bar{c}_i^p \mod p^{e+1} R)_i$ is the CRT set of $R_{p^{e+1}}$.

**Corollary D.3.** If $\bar{c}_i \in R$ are representatives for the mod-$p$ CRT set $(c_i)_i$ of $R_p$, then $(\bar{c}_i^{p-1} \mod p^e R)_i$ is the CRT set of $R_{p^e}$.

**Proof of Lemma D.2.** Let $pR = \prod_i p_i$ be the factorization of $pR$ into distinct prime ideals $p_i \subset R$. By hypothesis, we have $\bar{c}_i \in \delta_{i,i'} + p_i^{e'}$ for all $i, i'$. Then

$$\bar{c}_i^p \in \delta_{i,i'} + p \cdot p_i^{e'} + p_i^{e+1} \subseteq \delta_{i,i'} + p_i^{e+1},$$

because $p$ divides the binomial coefficient $\binom{p}{k}$ for $0 < k < p$, because $pR \subseteq p_i$, and because $p_i^{e'} \subseteq p_i^{e+1}$. □
CRT sets modulo a prime. We now describe the mod-$p$ CRT set for a prime integer $p$, and an efficient algorithm for computing representations of its elements. To motivate the approach, notice that the coefficient vector of $x \in R_p$ with respect to some arbitrary $\mathbb{Z}_p$-basis $\vec{b}$ of $R_p$ can be obtained via the twice and the dual $\mathbb{Z}_p$-basis $\vec{b}'$ (under the twice):

$$x = \vec{b}' \cdot \text{Tw}_{R_p/\mathbb{Z}_p}(x \cdot \vec{b}').$$

In what follows we let $\vec{b}$ be the decoding basis, because its dual basis is the conjugated powerful basis, which has a particularly simple form. The following lemma is a direct consequence of Equation (D.12) and the definition of twice (Equation (D.4)).

**Lemma D.4.** For $R = \mathcal{O}_m$ and a prime integer $p \mid m$, let $\vec{c} = (c_i)$ be the CRT set of $R_p$, let $\vec{d} = \vec{d}_m$ denote the decoding $\mathbb{Z}_p$-basis of $R_p$, and let $\tau(\vec{p}) = (p_j^{-1})$ denote its dual, the conjugate powerful basis. Then

$$\vec{c}' = \vec{d}' \cdot \text{Tw}_{R_p/\mathbb{Z}_p}(\tau(\vec{p}) \cdot \vec{c}') = \vec{d}' \cdot \hat{m}^{-1} \cdot \text{Tr}_{\mathbb{F}_{p^d}/\mathbb{F}_p}(C),$$

where $C$ is the matrix over $\mathbb{F}_{p^d}$ whose $(j, \vec{i})$th element is $\rho_i(g_m) \cdot \rho_i(p_j^{-1})$.

Notice that $\rho_i(p_j^{-1})$ is merely the inverse of the $(i, j)$th entry of the matrix $\text{CRT}_m$ over $\mathbb{F}_{p^d}$, which is the Kronecker product of $\text{CRT}_{m_\ell}$ over all maximal prime-power divisors of $m$. In turn, the entries of $\text{CRT}_{m_\ell}$ are all just appropriate powers of $\omega_m \in \mathbb{F}_{p^d}$. Similarly, $\rho_i(g_m)$ is the product of all $\rho_i \mod m_\ell(g_m) = 1 - \omega_m^i$. So we can straightforwardly compute the entries of the matrix $C$ and takes their traces into $\mathbb{F}_p$, yielding the decoding-basis coefficient vectors for the CRT set elements.

Relative CRT sets. We conclude by describing the relative CRT set $\vec{c}_{m', m}$ modulo a prime $p$, where $R = \mathcal{O}_m, R' = \mathcal{O}_{m'}$ for $m | m'$ and $p \mid m'$. The key property of $\vec{c}_{m', m}$ is that the CRT sets $\vec{c}_{m'}$, $\vec{c}_m$ for $R_p, R'_p$ (respectively) satisfy the Kronecker-product factorization

$$\vec{c}_{m'} = \vec{c}_{m', m} \otimes \vec{c}_m. \tag{D.13}$$

The definition of $\vec{c}_{m', m}$ arises from the splitting of the prime ideal divisors $p_\ell$ (of $pR$) in $R'$, as described next.

Recall from above that the prime ideal divisors $p'_\ell \subset R'$ of $pR'$ and the CRT set $\vec{c}_{m'} = (c'_\ell)$ are indexed by $\ell' \in G' = \mathbb{Z}_{m'}^*/(p)$, and similarly for $p_\ell \subset R$ and $\vec{c}_m = (c_\ell)$. For each $i \in G = \mathbb{Z}_m^*/(p)$, the ideal $p_\ell R'$ factors as the product of those $p'_\ell$, such that $i' = i \pmod{m}$, i.e., those $i' \in \phi^{-1}(i)$ where $\phi : G' \to G$ is the natural mod-$m$ homomorphism. Therefore,

$$c_\ell = \sum_{i' \in \phi^{-1}(i)} c'_{i'} \tag{D.14}.$$

To define $\vec{c}_{m', m}$, we partition $G'$ into a collection $\mathcal{I}'$ of $|G'|/|G|$ equal-sized subsets $I'$, such that $\phi(I') = G$ for every $I' \in \mathcal{I}'$. In other words, $\phi$ is a bijection between each $I'$ and $G$. This induces a bijection $\psi : G' \to \mathcal{I}' \times G$, where the projection of $\psi$ onto its second component is $\phi$. We index the relative CRT set $\vec{c}_{m', m} = (c_{I'})$ by $I' \in \mathcal{I}'$, defining

$$c_{I'} := \sum_{i' \in I'} c'_{i'} \tag{D.14}. $$

By Equation (D.14) and the fact that $(c'_\ell)$ is the CRT set of $R'_p$, it can be verified that $c_{I'} = c_{I'} \cdot c_\ell$ for $\psi(i') = (I', i)$, thus confirming Equation (D.13).
E  Tensor Product of Rings

Here we restate and prove Lemma 4.2 using the concept of a tensor product of rings.

Let $R, S$ be arbitrary rings with common subring $E \subseteq R, S$. The ring tensor product of $R$ and $S$ over $E$, denoted $R \otimes_E S$, is the set of $E$-linear combinations of pure tensors $r \otimes s$ for $r \in R, s \in S$, with ring operations defined by $E$-bilinearity, i.e.,

\[
(r_1 \otimes s) + (r_2 \otimes s) = (r_1 + r_2) \otimes s
\]

\[
(r \otimes s_1) + (r \otimes s_2) = r \otimes (s_1 + s_2)
\]

\[
e(r \otimes s) = (er) \otimes s = r \otimes (es)
\]

for any $e \in E$, and the mixed-product property

\[
(r_1 \otimes s_1) \cdot (r_2 \otimes s_2) = (r_1 r_2) \otimes (s_1 s_2).
\]

We need the following facts about tensor products of cyclotomic rings. Let $R = \mathcal{O}_{m_1}$ and $S = \mathcal{O}_{m_2}$. Their largest common subring and smallest common extension ring (called the compositum) are, respectively,

\[
E = \mathcal{O}_{m_1} \cap \mathcal{O}_{m_2} = \mathcal{O}_{\gcd(m_1, m_2)}
\]

\[
T = \mathcal{O}_{m_1} + \mathcal{O}_{m_2} = \mathcal{O}_{\operatorname{lcm}(m_1, m_2)}.
\]

Moreover, the ring tensor product $R \otimes_E S$ is isomorphic to $T$, via the $E$-linear map defined by sending $r \otimes s$ to $r \cdot s \in T$. In particular, for coprime $m_1, m_2$, we have $\mathcal{O}_{m_1} \otimes \mathcal{O}_{m_2} \cong \mathcal{O}_{m_1 m_2}$.

Now let $E', R', S'$ with $E' \subseteq R' \cap S'$ respectively be cyclotomic extensions of $E, R, S$. As part of ring tunneling we need to extend an $E$-linear function $L: R \rightarrow S$ to an $E'$-linear function $L': R' \rightarrow S'$ that agrees with $L$ on $R$, i.e., $L'(r) = L(r)$ for every $x \in R$. The following lemma gives sufficient conditions for when and how this is possible.

**Lemma E.1.** Adopt the above notation, and suppose $E = R \cap E'$ and $R' = R + E'$ (so that $R' \cong R \otimes_E E'$), and $(S + E') \subseteq S'$. Then:

1. The relative decoding bases of $R/E$ and of $R'/E'$ are identical.
2. For any $E$-linear function $L: R \rightarrow S$, the $E'$-linear function $L': R' \rightarrow S'$ defined by $L'(r \otimes e') := L(r) \cdot e'$ is $E'$-linear and agrees with $L$ on $R$.

**Proof.** First observe that $L'$ is indeed well-defined and is $E$-linear, by definition of the ring operations of $R' \cong R \otimes_E E'$. Now observe that $L'$ is in fact $E'$-linear: any $e' \in E'$ embeds into $R'$ as $1 \otimes e'$, so $E'$-linearity follows directly from the definition of $L'$ and the mixed-product property. Also, any $r \in R$ embeds into $R'$ as $r \otimes 1$, and $L'(r \otimes 1) = L(r) \cdot 1$, so $L'$ agrees with $L$ on $R$.

Finally, observe that because $R' \cong R \otimes_E E'$, the index of $E$ is the gcd of the indices of $R, E'$, and the index of $R'$ is their lcm. Then by the Kronecker-product factorization of decoding bases, the relative decoding bases of $R/E$ and of $R'/E'$ are the Kronecker products of the exact same components, in the same order. (This can be seen by considering each prime divisor of the index of $R'$ in turn.)