Alchemy: A Language and Compiler for Homomorphic Encryption Made easY

Eric Crockett
University of Michigan and
Georgia Institute of Technology

Chris Peikert
University of Michigan

Chad Sharp
University of Michigan

ABSTRACT

Fully Homomorphic Encryption (FHE) is a cryptographic "holy grail" that allows a worker to perform arbitrary computations on client-encrypted data, without learning anything about the data itself. Since the first plausible construction in 2009, a variety of FHE implementations have been given and used for particular applications of interest. Unfortunately, using FHE is currently very complicated, and a great deal of expertise is required to properly implement nontrivial homomorphic computations.

This work introduces Alchemy, a modular and extensible system that simplifies and accelerates the use of FHE. Alchemy compiles "in-the-clear" computations on plaintexts, written in a modular domain-specific language (DSL), into corresponding homomorphic computations on ciphertexts—with no special knowledge of FHE required of the programmer. The compiler automatically chooses (most of the) parameters by statically inferring ciphertext noise rates, generates keys and "key-switching hints," schedules appropriate ciphertext "maintenance" operations, and more. In addition, its components can be combined modularly to provide other useful functionality, such logging the empirical noise rates of ciphertexts throughout a computation, without requiring any changes to the original DSL code.

As a testbed application, we demonstrate fast homomorphic evaluation of a pseudorandom function (PRF) based on Ring-LWR, whose entire implementation is only a few dozen lines of simple DSL code. For a single (non-batched) evaluation, our unoptimized implementation takes only about 10 seconds on a commodity PC, which is more than an order of magnitude faster than state-of-the-art homomorphic evaluations of other PRFs, including some specifically designed for amenability to homomorphic evaluation.

CCS CONCEPTS

• **Security and privacy** → Public key encryption; • **Software and its engineering** → Software design techniques;

KEYWORDS

fully homomorphic encryption; domain-specific languages; compilers

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243828

ACM Reference Format:
Eric Crockett, Chris Peikert, and Chad Sharp. 2018. Alchemy: A Language and Compiler for, Homomorphic Encryption Made easY. In *CCS '18: 2018 ACM SIGSAC Conference on Computer & Communications Security Oct. 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243828

1 INTRODUCTION

Fully Homomorphic Encryption (FHE) is a powerful cryptographic concept that allows a worker to perform arbitrary computations on client-encrypted data, without learning anything about the data itself. Although first envisioned 40 years ago [49] as a cryptographic "holy grail," no plausible candidate FHE scheme was known until Gentry's seminal work in 2009 [26, 27]. Motivated by FHE's potential to enable new privacy-sensitive applications and enhance existing ones, a flurry of research activity has led to FHE schemes with better efficiency, stronger security assurances, and specialized features. (See [3, 4, 11–15, 18–20, 30–32, 51, 52] for a sampling.) In addition, there are a variety of real-world FHE implementations which have been used for particular applications of interest (see, e.g., [17, 22, 25, 29, 33, 38, 46]).

Unfortunately, using current FHE implementations for non-trivial homomorphic computations is quite complicated: First, one must express the desired "in the clear" computation (on plaintexts) in terms of the FHE scheme's "instruction set," i.e., the basic homomorphic operations it natively supports. This is non-trivial because the operations (which can vary based on the scheme) are typically algebraic ones like addition and multiplication on finite fields, and sometimes other functions like permutations on fixed-sized arrays of field elements. Thus, one needs to "arithmetize" the desired computation in terms of these operations, as efficiently as possible for the instruction set at hand. (Moreover, the instruction set can sometimes depend on the choice of plaintext and ciphertext rings, which can also affect the third step below.)

Second, FHE ciphertexts accumulate "errors" or "noise" under homomorphic operations, and too much noise causes the result to decrypt incorrectly—so proper noise management is essential. In addition, the ciphertext "degree" increases under certain operations, but can be brought back down using other means. So along with homomorphic operations that perform meaningful work on the plaintext, one must also carefully schedule appropriate "maintenance" operations, such as *linearization* and other forms of *key switching*, and *rescaling* (also known as *modulus switching*) to keep the ciphertext noise and size under control.

Third, one must choose appropriate ciphertext parameters for the desired level of security, i.e., dimensions and moduli that are compatible with the noise rates at the various stages of the computation (and also consistent with any restrictions inherited from the

first step). Importantly, the choice of parameters itself affects the noise growth incurred by the homomorphic operations, so one may need several cycles of trial and error until the parameters stabilize.

Lastly, one also needs to generate all the needed keys and auxiliary key-switching "hints" that are needed for the maintenance operations, and to encrypt the input plaintexts under the appropriate keys.

In summary, the above process requires a great deal of expertise in both the theory of FHE and the quirks of its particular implementation, in addition to a lot of manual programming and trial-and-error. Perhaps for this reason, most applications of FHE to date have been ad-hoc, one-off implementations, with complex code that is hard to debug and which obscures the nature of the underlying computation.

1.1 Contributions

This work introduces Alchemy, a system that greatly simplifies and accelerates the implementation of homomorphic computations.[1] In short, Alchemy *automatically* and *safely* transforms "in-the-clear" computations on plaintexts into corresponding homomorphic computations on ciphertexts. Crucially, this requires no detailed knowledge of particular FHE schemes on the part of the programmer. One simply writes (and runs, and debugs) a program that describes a desired plaintext computation, and then obtains a matching homomorphic computation with very little additional effort.

At a more technical level, Alchemy consists of two main pieces: (1) *domain-specific languages* (DSLs) for expressing plaintext and ciphertext computations, including both "native" operations and a library of higher-level functions built out of them; and (2) a *compiler* that transforms plaintext programs into corresponding homomorphic ones. The compiler automatically handles the cumbersome and delicate tasks of choosing (most of the) parameters, generating keys and hints, scheduling appropriate "maintenance" operations to control the ciphertext size and noise, encrypting inputs and decrypting outputs under appropriate keys, etc. In addition, it uses a strict type system to *statically*—i.e., at compile time—track the ciphertext noise to choose appropriate moduli, and to check that other parameters satisfy arithmetic conditions required for correct computation. Finally, the various pieces of the compiler can be composed modularly to provide additional useful functionality.

In summary, Alchemy lets programmers write clear and concise code describing what they really care about—the plaintext computation—and produces a corresponding homomorphic computation without requiring expertise in the intricacies of FHE. In Section 1.2 we describe the approach in more detail, and in Sections 1.3 and 5 we give some simple and then more advanced examples to demonstrate Alchemy's convenience and flexibility.

Application: homomorphic PRF evaluation. As a full-scale testbed application, in Section 5 we demonstrate fast homomorphic evaluation of a candidate pseudorandom function (PRF) based on the Learning With Rounding over Rings (Ring-LWR) problem [6]. In the non-batched setting, our (still unoptimized) implementation is *more than one order of magnitude faster* than prior homomorphic evaluations of other PRFs. In addition, the programmer-written implementation of the "plaintext" PRF computation is only a few dozen lines of very simple and transparent code; Alchemy handles everything else.

It has long been understood that homomorphic evaluation of symmetric-key primitives like PRFs is a very useful tool in the theory and practice of FHE (see, e.g., [26, 28, 33]). For example, it allows a client to encrypt its data using *symmetric-key* encryption—which is much faster and more compact than FHE encryption—while still allowing a worker to compute on the data homomorphically. The worker first homomorphically evaluates the symmetric decryption circuit on an FHE encryption of the user's secret key, resulting in an FHE encryption of the user's data. From there the worker can homomorphically evaluate the actual function of interest.

Prior work has homomorphically evaluated PRFs like the AES block cipher [18, 24, 33, 45] and the "LowMC" cipher [1], which was specially designed to have low multiplicative depth for amenability to FHE and multi-party computation (MPC). On standard laptop-class hardware, highly optimized state-of-the-art implementations require a *few minutes* (or much more) for a single AES or LowMC evaluation. They can also use SIMD techniques to compute scores of evaluations at once, resulting in *amortized* rates of a few seconds per block, or even sub-second rates for LowMC amortized over hundreds of blocks. However, the latency remains in the several minutes, which makes its unsuitable for many scenarios.

By constrast, our homomorphic evaluation of a Ring-LWR PRF takes only about *twelve seconds* for a single evaluation on commodity hardware.[2] This is primarily due to its use of rich natively supported FHE operations like "ring switching" [22, 30] and SIMD operations on "slots" [51], its small number of such operations, and its low multiplicative depth (over a large ring). This good "algebraic fit" of Ring-LWR for homomorphic evaluation was previously noted in [3, 5], and our work confirms it in practice.

1.2 Overview of Alchemy

As already mentioned, Alchemy is a collection of *domain-specific languages* (DSLs) for expressing plaintext and ciphertext computations, and *interpreters* that act on programs written in these languages. Here "interpreters" is broadly defined, and encompasses evaluators, optimizers, and, most significantly, compilers that transform programs from one DSL to another.

Alchemy is highly modular and extensible. Each DSL is made up of small, easy-to-define components corresponding to particular operations or language features, which can be combined arbitrarily. Interpreters can support any subset of the language components, and are easy to extend to new ones. In addition, Alchemy's interpreters are easy to compose with each other to perform a variety of different tasks. For example, starting from a single plaintext program one can: evaluate it "in the clear," compile it to a corresponding homomorphic computation, print a representation of both programs, encrypt a plaintext and perform the homomorphic computation on it, and track the ciphertext noise throughout.

Another key property of Alchemy is *static safety*: any well-formed program, and only well-formed programs, should be accepted by an interpreter, and the possibility of runtime errors should

[1] Alchemy is publicly available under a free and open-source license at https://github.com/cpeikert/ALCHEMY.

[2] There are limited opportunities for even faster amortized rates via batch evaluation; we leave these for future work.

be minimized or even eliminated. For these purposes, the Alchemy DSLs are *functional* (pure) and *statically typed*, in a rich type system that supports strong *type inference*.

- Purity means that a function always yields the same output when given the same inputs (no side effects or global variables), which is a good match for the arithmetic functions and circuits that are common targets for homomorphic computation.
- Static typing means that every expression has a type that is known at *compile time*, and only well-formed expressions typecheck. This allows many common programming errors—both in DSL code, and in Alchemy's own interpreters—to be caught early on. The type system is very rich, allowing many safety properties to be encoded into types and automatically verified by the type checker.
- Type inference ensures that (almost) all types in DSL expressions can be automatically determined by the type system, and need not be explicitly specified by the programmer. This makes code more concise, and easier to understand and check for correctness.

All the above-described properties are obtained by defining the DSLs and interpreters in the host language Haskell, from which Alchemy directly inherits its basic syntax, rich data types, and safety features—with no special implementation effort or extra complexity. As its underlying FHE implementation, Alchemy uses a BGV-style [12] cryptosystem as refined and implemented in $\Lambda\circ\lambda$ [22], a recent Haskell framework for FHE and lattice-based cryptography more generally. However, we emphasize that Alchemy compilers can easily target other FHE schemes and implementations, not just those based on $\Lambda\circ\lambda$ and Haskell.

Languages. Domain-specific languages (DSLs) have long been recognized as powerful tools for working in particular problem domains; e.g., LaTeX is a (Turing-complete) DSL for typesetting documents, and the MATLAB language is targeted toward numerical computing and linear algebra.

Alchemy's first main ingredient is a collection of modular and extensible DSLs for expressing both "in the clear" computations on plaintexts, and homomorphic computations on ciphertexts. Following the powerful "typed tagless final" approach to embedded language design [41], each DSL is the union of several independent and composable *language components*. Alchemy defines language components for the following DSL features:

- data types for plaintexts rings and FHE ciphertexts, and simple data structures like tuples and lists;
- basic arithmetic operations like addition and multiplication, along with more advanced ones like arbitrary linear functions between plaintext rings;
- ciphertext operations as supported by the underlying FHE implementation;
- programmer-defined functions, including higher-order functions (i.e., those that operate on other functions);
- and even specific forms of side effects, via monads.

It is easy to introduce additional data types and language features as needed, simply by defining more language components.

Both the plaintext and ciphertext DSLs include the generic language components for data structures, arithmetic operations, and functions. In addition, each one includes the components that relate specifically to plaintext or ciphertext operations. Because the plaintext DSL involves relatively simple data types and operations, it is easy for the programmer to hand-write code to express a desired computation. By contrast, proper use of the ciphertext DSL is significantly more complicated—e.g., ciphertext types involve many more parameters, and FHE operations must be appropriately scheduled—so it is not intended for human use (though nothing prevents this). Instead, it is the target language for Alchemy's plaintext-to-ciphertext compiler. As we will see, having a dedicated ciphertext DSL allows for homomorphic computations to be acted upon in various useful ways beyond just executing them, e.g., tracking noise growth or optimizing away redundant operations.

Alchemy also provides a variety of useful higher-level functions and combinators that are written in the DSLs. These include "arithmetized" versions of functions that are not natively supported by FHE schemes, but can be expressed relatively efficiently in terms of native operations. A particularly important example for our purposes is the "rounding function" from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_2$, where $\mathbb{Z}_q$ denotes the ring of integers modulo q. This function is central to efficient "boostrapping" algorithms for FHE and the related Learning With Rounding problem [6], and has an efficient arithmetization [3, 31]. See Section 5 for further details.

Interpreters and compilers. The other main ingredient of Alchemy is its collection of composable *interpreters* for programs written in its DSLs. In keeping with Alchemy's modular structure, each interpreter separately defines how it implements each relevant language component. In particular, some of the interpreters are actually *compilers* that translate programs from various DSL components to others. Example interpreters provided in Alchemy include:

- an evaluator, which simply interprets the plaintext or ciphertext DSL operations using the corresponding Haskell and $\Lambda\circ\lambda$ operations;
- various utility interpreters that, e.g., print DSL programs, or compute useful metrics like program size, multiplicative depth, etc.;
- a diagnostic compiler that modifies any ciphertext-DSL program to also log the noise rate of every ciphertext it produces;
- most significantly, a compiler that transforms any program written in the plaintext DSL to a corresponding homomorphic computation in the ciphertext DSL.

The plaintext-to-ciphertext compiler is the most substantial and nontrivial of the interpreters, and is one of this work's main contributions. This compiler *automatically* performs several important tasks that in all other FHE systems must handled manually by the programmer. In particular, it:

- generates all necessary *secret keys* and *auxiliary "hints"* for ciphertext operations like key-switching and ring-switching;
- properly schedules all necessary ciphertext maintenance operations like key-switching (e.g., for "linearization" after homomorphic multiplication) and modulus-switching (for noise management);

- *statically* infers, using compile-time type arithmetic, the approximate noise rate of each ciphertext to within a small factor, and selects an appropriate ciphertext modulus from a given pool (and if any inferred rate is too small relative to the available moduli, outputs an informative type error);
- generates encrypted inputs for the resulting homomorphic computation, with appropriate noise rates to ensure correct decryption of the ultimate encrypted output.

1.3 Example Usage

Here we give a few concrete examples of programs in the Alchemy DSLs, and the various ways they can be interpreted and compiled. These examples illustrate the ease of use and flexibility of the approach. We start with the following simple DSL expression:

```
ex1 = lam2 $ \x y -> (var x +: var y) *: var y
```

As expected, `ex1` represents a function of two inputs `x` and `y`, which adds them using the DSL operator `+:`, then multiplies the result by `y` using the DSL operator `*:`. The Haskell typechecker automatically infers the full type of `ex1`, which is:

```
ex1 :: (Lambda_ expr, Mul_ expr a, Add_ expr (PreMul expr a))
    => expr e (PreMul expr a -> PreMul expr a -> a)
```

This type carries a great deal of important information. Let us unpack its various components:

- First, the type is *polymorphic* in the *type variables* `expr`, `e`, and `a`. These type variables can represent arbitrary Haskell types…
- …subject to the *constraints* (`Lambda_ expr, ...`), which say that `expr` must be able to interpret the `Lambda_`, `Mul_`, and `Add_` language components. More specifically, `Lambda_` says that `expr` supports programmer-defined functions (here via `lam2`), `Mul_` says that `expr` can handle multiplication of two values of type `PreMul expr a` to produce one of type `a`, (via `*:`), and `Add_` says that `expr` can handle addition of values of the former type (via `+:`). (The purpose of `PreMul` is explained below, where we describe the plaintext-to-ciphertext compiler.)
- Finally, the type `expr e (PreMul expr a -> ... -> a)` says that `ex1` represents a *DSL function* that takes two input values of type `PreMul expr a` and outputs a value of type `a`. The type argument `e` represents the expression's *environment*, which must list the types of any *unbound* variables used in "open" code. Here both `x` and `y` are bound by the enclosing `lam2`, so there are no unbound variables—the code is "closed"—and hence `e` is completely unconstrained.

1.3.1 One Program, Many Interpretations. Because `ex1` is polymorphic in `expr`, having written it once we can interpret it in several different ways by specializing `expr` to various concrete interpreter types. One simple interpreter is the "not-so-pretty" printer `P`, which trivially implements all the requisite language components. Its public interface

```
print :: P () a -> String
```

converts any closed `P`-expression to a string representing the computation in a "desugared" form. Calling `print ex1` implicitly specializes `ex1`'s interpreter type variable `expr` to `P` and its environment type variable `e` to `()`, resulting in the following:[3]

```
print ex1
-- "(\v0 -> (\v1 -> ((mul ((add v0) v1)) v1)))"
```

Another very simple interpreter is the evaluator `E`, which just interprets each of Alchemy's DSL components using corresponding Haskell (or $\Lambda\circ\lambda$) operations. Its public interface

```
eval :: E () a -> a
```

converts any closed *DSL expression* of arbitrary type `a` to a *Haskell value* of type `a`, as follows:

```
eval ex1
-- (Ring a) => a -> a -> a
eval ex1 7 11
-- 198
```

Because `eval` implicitly specializes `ex1`'s type variable `expr` to `E`, which defines `PreMul E a = a`, the call to `eval ex1` produces a polymorphic Haskell function of type `a -> a -> a`, for an arbitrary `Ring` type `a`. The `Ring` constraint comes from the fact that `E` uses the operators `+` and `*` (introduced by `Ring`) to interpret the DSL operators `+:` and `*:` (introduced by `Add_` and `Mul_`, respectively). The call to `eval ex1 7 11` actually evaluates the Haskell function, yielding $(7 + 11) \cdot 11 = 198$.

We stress that `eval ex1 :: (Ring a) => a -> a -> a` is *polymorphic* in `a`, so it can be applied to elements of any plaintext ring, or even to ciphertexts from $\Lambda\circ\lambda$'s FHE scheme (both of which instantiate `Ring`). However, in the latter case `ex1` still lacks the extra ciphertext "maintenace" operations, like relinearization and modulus-switching, that are needed in typical homomorphic computations. For these we use Alchemy's plaintext-to-ciphertext compiler, described in Section 1.3.3 below.

1.3.2 Ring Switching. Here we exhibit another small program that illustrates another important language component, for "switching" from one cyclotomic ring to another. Ring-switching in homomorphic encryption was developed and refined in a series of works [3, 12, 22, 30], which showed its utility for tasks like "bootstrapping" and efficiently computing a wide class of linear functions.

```
ex2 = linearCyc_ (decToCRT @F28 @F182) .:
      linearCyc_ (decToCRT @F8  @F28)
```

Here `decToCRT @F8 @F28` is a Haskell expression representing a certain linear function from the 8th to the 28th cyclotomic ring, and similarly for `decToCRT @F28 @F182`. (The specific linear functions do not matter here, and could be arbitrary.) The `.:` operator denotes composition of DSL functions. Naturally, the Haskell type checker enforces that the output type of the "inner" function must equal the input type of the "outer" function.

The Haskell compiler automatically infers the following type for `ex2` (we have suppressed some type arguments and constraints for better readability):

```
ex2 :: (LinearCyc_ expr cyc, ...) =>
       expr e (cyc F8 zp -> cyc F182 zp)
```

[3] Notice the automatically indexed variables `v0`, `v1`, and prefix-form functions `mul`, `add` in place of the infix operators `*:`, `+:`.

This type says that `ex2` is a (closed) DSL function that may be interpreted by any interpreter `expr` that can handle the **`LinearCyc_`** language component (which introduces `linearCyc_`). Essentially, the DSL function maps from `cyc F8 zp`, which should represent the 8th cyclotomic ring modulo some integer p, to `cyc F182 zp`, which should represent the 182nd cyclotomic ring modulo p. The type `cyc` is specified in the **`LinearCyc_`** `expr cyc` constraint, and could be, e.g., the concrete type **`Cyc`** from $\Lambda \circ \lambda$, which implements cyclotomic rings.

As with the previous example, we can print `ex2` and evaluate it "in the clear." More interesting is to *homomorphically* evaluate it on FHE ciphertexts, using a special form of key-switching as shown in [22, 30]. For this we use Alchemy's homomorphic compiler, described next.

1.3.3 Compiling to the Ciphertext DSL. We now show how the above example DSL programs, which should now be thought of as functions on plaintexts, can be compiled into programs that operate on FHE ciphertexts to homomorphically evaluate the original programs on their underlying plaintexts. Like the printer and evaluator, the compiler is a data type **`PT2CT`** that can interpret all the appropriate DSL components (**`Add_`**, **`Mul_`**, **`LinearCyc_`**, etc.). Because FHE involves additional parameters like ciphertext rings, ciphertext moduli, and a choice of key-switching "gadget," and because **`PT2CT`** statically tracks the error rate of ciphertexts, we must specify a few additional types. For example, we can define:

```
type Zq1 = Zq $(mkModulus 34594561)
type Zq2 = Zq $(mkModulus 35642881)
...
type CTModuli  = [ Zq1, Zq2, ... ]
type CTRingMap = [ (F8,F512), ... ]
type Gad       = TrivGad

-- specialize examples to cyclotomics with
-- desired ciphertext error rates
ex1' = ex1 :: _ => expr e (_ -> _ -> PNoiseCyc 0 F8 _)
ex2' = ex2 :: _ => expr e (_      -> PNoiseCyc 0 _  _)
```

The type **`Zq1`** represents $\mathbb{Z}_{q_1}$, the ring of integers modulo $q_1 = 34594561$, and similarly for **`Zq2`** etc. (The macro `mkModulus` defines a *type* representing its argument, which is also augmented with the number of "units of noise" the modulus can hold; see below for more details.) The type **`CTRingMap`** specifies that when the plaintext ring is the 8th cyclotomic, the ciphertext ring should be the 512th cyclotomic, etc. (Ciphertext rings must be taken large enough, relative to the moduli, to achieve a desired level of security; there are automated tools to aid their selection.) Finally, **`TrivGad`** indicates a simple kind of gadget that emphasizes speed and compactness.

The definitions of `ex1'`, `ex2'` specialize the output types of `ex1`, `ex2` to a particular cyclotomic data type **`PNoiseCyc`** that specifies a desired noise rate for the corresponding ciphertext; the `0` arguments say that the output ciphertexts should be decryptable but need not support any further homomorphic operations. (The underscores indicate types that will be inferred by the Haskell type checker.) This specialization is needed so that the Alchemy compiler can convert the plaintext types to corresponding ciphertext types, and statically (back)track error rates.

Having defined the needed types, we can compile our plaintext-DSL expressions to corresponding ciphertext-DSL expressions using the public interface `pt2ct` (whose type signature is given in Section 4.1); the resulting ciphertext-DSL program can then be handled by any suitable interpreter. One subtlety is that because the compiler automatically generates all the requisite *random* keys and key-switch hints, it is necessarily *monadic*, i.e., non-pure. We therefore use Haskell's "do notation" to invoke `pt2ct` in an appropriate monadic context. For example, we can compile `ex1'` (which, to recall, represents `(x + y) * y`) and print the resulting "sugar-free" ciphertext-DSL expression as follows:

```
do ct1 <- pt2ct @CTRingMap @CTModuli @Gad ex1'
   return $ print ct1

-- "(\v0 -> (\v1 ->
--    (((\v2 -> (\v3 ->
--      ((\v4 -> (modSwitch
--        ((\v4 -> (keySwitchQuad <HINT> (modSwitch v4))) v4)))
--       ((mul v2) v3))))
--      ((add v0) v1)) v1)))"
```

Despite the abundant variables and parentheses, the structure can be teased out. First, because this is a program in the ciphertext DSL, we should think of all the variables as representing FHE ciphertexts. The expression is a function of two inputs, denoted `v0` and `v1`. In the "inner" layer, the variable `v2` is bound to `((add v0) v1)`, and `v3` is bound to `v1`.[4] These two ciphertexts are multiplied, which results in a "quadratic" ciphertext, to which `v4` is bound. This is modulus-switched to match the key-switching hint, then key-switched to a "linear" ciphertext by `keySwitchQuad`, then finally switched back to an appropriate modulus for its inherent noise rate. We again stress that the creation of the key-switch hints, and the selection of appropriate moduli for hints and ciphertexts, (which are not displayed in the printer output), is handled automatically.

As another example, we can compile the expression `ex2'` (which represents two successive ring-switches) and print the result:

```
do ct2 <- pt2ct @CTRingMap @CTModuli @Gad ex2'
   return $ print ct2

-- "(\v0 ->
--    ((\v0 -> (modSwitch (tunnel <HINT> (modSwitch v0))))
--     ((\v0 -> (modSwitch (tunnel <HINT> (modSwitch v0))))
--      v0)))"
```

The program takes a ciphertext denoted `v0` as input, switches it to the modulus of the "tunneling hint" that encodes the desired linear function, then switches rings by "tunneling" with the hint, then switches back to an appropriate ciphertext modulus. The same cycle is repeated for the next tunneling step. (Note that some of the modulus-switches may turn out to be null operations, depending on the noise rates.)

1.3.4 Evaluating and Logging. While it is nice to be able to see a representation of ciphertext-DSL programs, we are more interested in *evaluating* them to perform a homomorphic computation on

[4] Note that the expression is highly amenable to "inlining" using β-reduction; while our compiler does not currently perform such optimizations at the DSL level (though that could be done by an additional interpreter), the Haskell compiler itself would likely do so. In any case, the performance cost of not inlining is negligible when compared with homomorphic operations.

ciphertexts. Fortunately, this is extremely simple: just replace `print` with `eval` in the above code! This specializes the (polymorphic) interpreter of the output ciphertext-DSL program to the evaluator `E` rather than the printer `P`.

In addition, for diagnostic purposes we may wish to measure the "error rates" of the ciphertexts as homomorphic evaluation proceeds. (E.g., during development this was very helpful for designing our static noise tracker to closely approximate the true noise growth, and for finding subtle implementation bugs.) Such logging is very easy using the `ERW` interpreter, which transforms any ciphertext-DSL program to an equivalent one that additionally logs the error rates of all generated ciphertexts. The transformed program can then be evaluated (or printed, or sized, etc.) as usual. One important subtlety is that *the transformed program itself*—rather than just the process of generating it—is *monadic*, because it uses the side effect of writing DSL values to a log. Therefore, our ciphertext DSL needs to support monadic computation, which it does modularly via the `Monad_` and related language components. Continuing our previous example:

```
do logct2  <- writeErrorRates ct2
   inputCT <- encrypt inputPT
   (result, log) = runWriter $ eval logct2 >>= ($ inputCT)
   return log

-- "Error rates:
-- ("modSwitch_Q539360641*Q537264001",6.8495e-7),
-- ("tunnel_Q539360641*Q537264001",3.3651e-6),
-- ("modSwitch_Q537264001",7.3408e-6),
-- ("modSwitch_Q537264001",7.3408e-6),
-- ("tunnel_Q537264001",1.8010e-4),
-- ("modSwitch_Q537264001",1.8010e-4)"
```

The log shows the error rates of the ciphertexts produced by each ciphertext-DSL operation (conveniently augmented by its modulus). We can see that the first `tunnel` operation increases the error rate by roughly 5x, the switch to the smaller modulus increases the rate by roughly 2x, etc. (The other `modSwitch` operations do not actually change the modulus, and are therefore null.)

1.4 Limitations and Future Work

As explained above, Alchemy represents significant progress toward making FHE usable for non-experts. Here we describe some of its present limitations and directions for potential improvement.

Fully automating FHE parameters. Although the `PT2CT` compiler automatically chooses parameters for each ciphertext and hint in the computation, the programmer still must provide a collection of sufficiently large (and arithmetically valid) cyclotomic indices, and a pool of sufficiently many moduli to support all the statically estimated error rates. Ideally, Alchemy would choose all of these parameters automatically, targeting a desired security level without unnecessarily sacrificing efficiency. Because in our examples we generated these parameters semi-automatically using scripts, this ultimate goal may not be too far out of reach. One potential route is to use Template Haskell to programmatically generate types meeting the required constraints at compile time. (Alchemy already uses Template Haskell in some basic ways, e.g., to compute the "noise capacity" of the provided moduli.)

Scalability. We have successfully demonstrated Alchemy's utility on moderate-sized computations, like homomorphic evaluation of a Ring-LWR-based PRF. However, Alchemy is not yet suitable for much larger or more complex functions, due to the long compilation times under the ghc Haskell compiler (which unfortunately have become worse in recent versions). The main bottleneck is Alchemy's extensive use of type-level arithmetic for static error-rate tracking and encoding arithmetic constraints, which severely stresses the Haskell compiler. (This is a known performance issue that has been under active consideration for some time.) Relaxing or simplifying the error-rate arithmetic might help significantly.

Alternative backends. While Alchemy currently targets the FHE implementation of $\Lambda\circ\lambda$, nothing in its design requires this. Because the plaintext and ciphertext DSLs are polymorphic, one could potentially write interpreters that output, e.g., valid C++ code targeting HElib [36]. Such a staged-compilation approach could combine the strong static safety properties of Alchemy's DSLs and convenience of its compilers with the high performance of optimized lower-level libraries.

Formal proofs. Alchemy's DSL interpreters have very concise implementations (of at most a few lines of code per method) that are easy to audit by inspection, but are not formally verified. Therefore, there is the possibility that Alchemy will transform some program incorrectly, perhaps for some unusual choices of parameters or unlikely choices of randomness. Given the complexity of FHE, and especially of error accumulation, at this stage it is far from clear what a meaningful formal proof of correctness would entail.

Automating and optimizing arithmetization. Lastly, Alchemy and all other existing FHE implementations still require the programmer to arithmetize the desired plaintext computation into relatively low-level operations (notwithstanding recent progress like [21, 37]). At present this process is very ad-hoc and manually driven: the "native instruction sets" of FHEs consist of a motley assembly of low- and medium-level operations with varying cost metrics (including "bootstrapping"), and it is not at all clear how best to arithmetize even some basic computations of interest. An ambitious goal would be to devise compilers that convert high-level DSL code into particular FHE instruction sets, and optimize their performance according to various objectives.

1.5 Related Work

Fully homomorphic encryption. As far as we know, there are no prior domain-specific languages or compilers for FHE; all implementations require the programmer to "arithmetize" the desired function by hand and then write code in a general-purpose language, manually scheduling appropriate homomorphic and ciphertext-maintenance operations, generating keys and hints, etc.

Probably the most well-known and mature FHE implementation is HElib [36], an "assembly language" for fully homomorphic encryption, which is implemented in C++ on top of NTL [50]. HElib has been used for many homomorphic computations of interest [21, 33, 37, 38], but it requires quite a lot of expertise in FHE and the library itself to use, because computations must be written directly in the "assembly language."

Alchemy is built on top of the $\Lambda \circ \lambda$ library for lattice-based cryptography [22] and its FHE implementation. To our knowledge, this is the only implementation that supports ring-switching, which we use for homomorphic PRF evaluation. However, up until now those who wished to use $\Lambda \circ \lambda$ for FHE still had to write code directly to its interface, which is roughly at the same level of abstraction as HElib's.

FHEW [25] is a refinement and fast implementation of an efficient bootstrapping algorithm [4] for "third-generation" FHE schemes [34]. However, it is not yet appropriate for general-purpose homomorphic computations, because it encrypts only one bit (or just a few bits) per ciphertext, and supports only basic logic gates.

The SEAL library [17] provides heuristic parameter selection, an important part of practically usable FHE. However, users must still manually arithmetize and implement their computations, and generate keys and hints. SEAL is also limited to power-of-two cyclotomic rings, which do not support SIMD "slots" (for characteristic-two plaintext rings) or the most useful forms of ring-switching.

Secure computation. Secure two- and multi-party computation (2PC and MPC) is similar to FHE, in that it allows mutually distrustful *interacting* parties to compute a function on their private inputs while revealing nothing more than the function output (and what is implied by it). The history of 2PC/MPC stretches back to the 1980s [8, 35, 53], and its implementations are more mature. Like FHE, secure computation also requires "in-the-clear" functions to be converted to (arithmetic or boolean) circuits and compiled into "encrypted" versions, e.g., garbled circuits. Tools for these tasks have evolved over many years (e.g., [7, 9, 23, 39, 43, 44, 47, 48]), and the approach of using specification languages and compilers has proven to be very powerful.

As a few examples, Fairplay [7, 44], TASTY [39], ShareMonad [42], and Wysteria [48] all provide high-level domain-specific languages for expressing computations, and compilers for transforming these into executable protocols that satisfy the desired security properties. In particular, TASTY compiles to two-party protocols that use a mix of garbled circuits and (semi-)homomorphic encryption, but not fully homomorphic encryption. Wysteria goes further to support "mixed-mode" programs, and has a strong type system and a proof of type soundness, which implies formal secrecy guarantees for the parties' private data. The use of strong static type systems to provide higher levels of safety and trustworthiness is also a theme in Alchemy, though our work does not (yet) provide the kinds of formal guarantees that Wysteria has.

The rest of the paper is organized as follows.

Section 2 gives the relevant background on the *(typed) tagless final* approach of DSL design and implementation, which Alchemy is based on.

Section 3 describes the various components of Alchemy's plaintext and ciphertext DSLs.

Section 4 describes the flagship Alchemy compiler, the plaintext-to-ciphertext compiler.

Section 5 describes a full-scale application in Alchemy, namely (homomorphic) "ring rounding," and gives an evaluation.

The appendices contain additional background and technical material.

2 TAGLESS-FINAL BACKGROUND

Here we give the necessary background on the elegant and powerful "(typed) tagless-final" approach [16, 41] to the design of domain-specific languages (DSLs), also called *object languages*, in a *host language*. The well-known "initial" approach to DSLs represents object-language expressions as *values* of a corresponding data type/structure in the host language, e.g., an abstract syntax tree. By contrast, the tagless, or "final," approach represents object-language expressions as ordinary combinations of *polymorphic terms* in the host language. The polymorphism allows a DSL expression to be written once and interpreted in many different ways.

The tagless-final approach makes language design and interpretation highly modular, extensible, and safe. Object-language features can be defined independently and combined together arbitrarily. Interpreters can be defined to handle any subset of the available language components, and extended to support new ones without changing existing code. If an interpreter does not implement all the language components used in an expression, type checking fails at compile time with an informative error. And the full strength of the host language's type system, including type inference, is directly inherited by the object language with no special effort.

Below we give an introduction to the approach by providing a running example of several general-purpose language components and interpreters from [16, 41], which are also included in Alchemy. In Section 3 we describe more specialized language components for the plaintext and ciphertext languages.

2.1 Language Components and Interpreters

Language components. In the tagless-final approach for the host language Haskell, an object-language component is defined by a *type class*, or *class* for short. A class defines an abstract interface that introduces one or more polymorphic *methods*, which may be functions or just values. Concrete data types can then *instantiate* the class, by implementing its methods in an appropriate way. For example, `Int` and `Bool` both instantiate the `Additive` class representing additive groups, defining its addition operator `+` as ordinary integer addition and exclusive-or, respectively.

In the tagless-final context, a language component is defined by a class. For example, operations related to pairs in the object language are defined by[5]

```
class Pair_ expr where
  pair_ :: expr e (a -> b -> (a,b))
  fst_  :: expr e ((a,b) -> a)
  snd_  :: expr e ((a,b) -> b)
```

Here `pair_` is a polymorphic host-language term representing an *object-language function*, which maps (object-language) values of arbitrary types `a` and `b` to an (object-language) value of pair type `(a,b)`. (All Haskell types are automatically inherited by the object language.) Naturally, `fst_` and `snd_` are similar.

Notice the common form `expr e t` of the method types. The type `expr` is an *instance* of the `Pair_` class, and serves as the *interpreter* of `pair_`, `fst_`, and `snd_`. The type `t` represents the type of the object-language term, and the type `e` represents its *environment* (discussed below in Section 2.2).

[5] By convention, names of object-language components and terms always end in an underscore, to distinguish them from host-language names.

Interpreters. An interpreter of a language component is just a data type that instantiates the class defining the component. As running examples, we describe two simple interpreters. The *evaluator* `E` is defined as

```
newtype E e a = E (e -> a)
```

which says that a value of type `E e a` is just a function mapping `e`-values to `a`-values. For example, when `e` is the null type `()` (indicating a "closed" expression with no free variables), the function will map its (null) input to the `a`-value represented by the object-language expression. The (not-so-pretty) *printer* `P` is defined as

```
newtype P e a = P String
```

which says that a value of type `P e a` is just a `String`, i.e., the printed representation of the object-language expression.

We make `E` and `P` interpreters of the pair-related DSL operations by defining them as instances of the `Pair_` class. Observe that when `expr` is specialized to `E`, the type of `pair_` is equivalent to `e -> a -> b -> (a,b)`. This leads to the following (partial) `Pair_` instance (the definitions of `fst_` and `snd_` are also trivial):[6]

```
instance Pair_ E where
  pair_ = E $ \e -> \a -> \b -> (a,b)
```

When `expr` is specialized to `P`, the type of `pair_` is equivalent to just `String`, which leads to the easy (partial) instance definition

```
instance Pair_ P where
  pair_ = P $ "pair"
```

Extending the language and interpreters. We can introduce more DSL operations simply by defining more classes, e.g., for addition and multiplication:

```
class Add_ expr a where
  add_ :: expr e (a -> a -> a)
  neg_ :: expr e (a -> a)

class Mul_ expr a where
  type PreMul_ expr a
  mul_ :: expr e (PreMul_ expr a -> PreMul_ expr a -> a)
```

Notice that here the type `a` is *specific to the instance*, not arbitrary: it is an argument to the `Add_` class. This means that an interpreter may support `add_` and `neg_` for certain types `a`, but not others.

The type of `mul_` is similar to that of `add_`, except that the two (object-language) inputs have type `PreMul_ expr a` instead of just `a`. This allows the interpreter to define the input type as a function of the output type. We take advantage of this in plaintext-to-ciphertext compilation, where the types carry static information about ciphertext error rates (see Section 4).

The instances of `Add_` for `E` and `P` are again trivial. We show `E`'s to highlight one small subtlety:

```
instance Additive a => Add_ E a where
  add_ = E $ \e -> \x -> \y -> x+y -- or: E $ pure (+)
  neg_ = E $ \e -> \x -> negate x
```

Here `E` is an instance of `Add_` only for types `a` that are instances of the class `Additive`. This is necessary because `E` interprets `add_` using `Additive`'s addition operator `+`. By contrast, `P` can interpret `add_` for *any* type `a`, because it just produces a string.

The instances of `Mul_` are almost identical, except that we use the constraint `Ring a` and its multiplication operator `*`. This also means we must define `PreMul_ E a = a`. By contrast, the `P` interpreter represents any object-language term as a `String`, so we are free to define `PreMul_ P a` arbitrarily.

[6] A shorter definition is `pair_ = E $ pure (,)`.

2.2 Functions and HOAS

Suppose we want to write an object-language function `double_` that adds an input value to itself. We may at first be tempted to write this as a host-language function on object-language values:

```
double_ :: Add_ expr a => expr e a -> expr e a
double_ x = x +: x
```

Unfortunately, there is a subtle problem: when we apply `double_`, the object-language argument is "inlined" into the resulting object-language expression (i.e., call-by-name evaluation), so the argument is re-evaluated every time it appears in the body of the function:

```
print $ double_ (3 *: 4)
-- add (mul 3 4) (mul 3 4)
```

Instead of multiplying 3 by 4 and passing the result to `double_` as we would want, the object-language expression `3 *: 4` is passed to `double_` unevaluated, which ultimately results in two multiplications instead of one. (Examples involving an exponential blowup in size are easy to construct.) While this does not change the expression's value, it may dramatically harm its computational efficiency.

To resolve this problem we need the *object language* to support programmer-defined functions—i.e., function abstraction—and function application. Ideally, writing and applying functions in the object language would be as natural as for the host language, via a direct translation from the latter to the former. Essentially, we seek to use *higher-order abstract syntax* (HOAS). In Appendix B, building heavily on ideas from [40] we present a solution that provides nearly this degree of ease of use. Importantly, it even works for *effectful* (monadic) host-language functions, like our plaintext-to-ciphertext compiler (which creates random keys and hints during its code transformations), via simpler interfaces and types than in [40]:

```
lam    :: Lambda_ expr
       => (forall x . expr (e,x) a -> expr (e,x) b)
       -> expr e (a -> b)

lamM   :: (Lambda_ expr, Functor m)
       => (forall x . expr (e,x) a -> m (expr (e,x) b))
       -> m (expr e (a -> b))

double_ = lam $ \x -> var x +: var x
print $ double_ $: (3 *: 4)
-- (\v0 -> add v0 v0) (mul 3 4)
```

2.3 Generic Language Components

In the full version we give other general-purpose language components that embed basic data types like lists and strings into the object language. We also define language components for *category-theoretic* abstractions like *(applicative) functors* and *monads*, which can model fine-grained (side) effects in the object language. For example, we use these to modularly add logging of ciphertext error rates during homomorphic computation (see Appendix C).

3 ALCHEMY DOMAIN-SPECIFIC LANGUAGES

In this section we describe Alchemy's specialized DSL language components for expressing computations on FHE plaintexts and ciphertexts. Recall that in tagless-final style, language components can be combined arbitrarily. Loosely speaking, "plaintext DSL" (respectively, "ciphertext DSL") refers to the union of the generic language components supporting arithmetic, basic data structures, etc., and the language components for specialized plaintext operations described in Section 3.1 (respectively, homomorphic operations on ciphertexts described in Section 3.2). Naturally, not every program written in these DSLs will use every component.

3.1 Plaintext DSL

BGV-style FHE systems [12, 22] natively support homomorphic evaluation of several operations. We define language components that model the induced functions on the *plaintexts*, and instantiate them for the appropriate Alchemy interpreters; in most cases this is completely straightforward. More interestingly, the plaintext-to-ciphertext compiler instantiates plaintext-DSL components by translating their operations to the ciphertext DSL (see Section 4).

Arithmetic with public values. In FHE one can homomorphically add or multiply a *public* value from the plaintext ring with an encrypted plaintext, yielding an encrypted result. We model this by treating the public value as residing in the *host language*, while the encrypted plaintext resides in the *object language*. When transforming from the plaintext DSL to the ciphertext DSL, the public (host-language) value remains "in the clear," whereas the object-language plaintext type is transformed to a ciphertext type.

```
class AddLit_ expr a where
  addLit_ :: a -> expr e (a -> a)

class MulLit_ expr a where
  mulLit_ :: a -> expr e (a -> a)
```

Division by two. When the plaintext modulus p is even and the plaintext itself is known to also be even, it is possible to homomorphically divide both the modulus and plaintext by two. This is a useful operation in the context of bootstrapping and the homomorphic evaluation of PRFs (see Section 5).

The following `Div2_` language component introduces an object-language function that models the divide-by-two operation on plaintexts. Like the `Mul_` class, `Div2_` has an associated type family `PreDiv2_` that allows the interpreter to define the input plaintext type as a function of the output type.

```
class Div2_ expr a where
  type PreDiv2_ expr a
  div2_ :: expr e (PreDiv2_ expr a -> a)
```

Linear functions. In BGV-style FHE over cyclotomic rings, one can homomorphically apply (to an encrypted plaintext) any function from the rth cyclotomic to the sth cyclotomic that is linear over a common subring. The following (somewhat simplified) `LinearCyc_` language component introduces the `linearCyc_` function, which models this operation. Notice that `linearCyc_` takes a *host-language* value representing the desired linear function (via the $\Lambda\circ\lambda$ data type `Linear`), and produces an *object-language* function on plaintexts. Analogously to `addLit_` and `mulLit_`, this reflects the fact that the choice of linear function always remains "in the clear," whereas the object-language plaintext type can be transformed, e.g., to a ciphertext type. Notice also that, like `PreMul_` and `PreDiv2_`, the `PreLinearCyc_` type family gives the interpreter some control over the type of the input as a function of the output type. The plaintext-to-ciphertext compiler uses this to statically track error rates; see Section 4.2.2.

```
class LinearCyc_ expr cyc where
  type PreLinearCyc expr cyc

  linearCyc_ :: ... => Linear cyc e r s zp ->
    expr env ((PreLinearCyc expr cyc) r zp -> cyc s zp)
```

Higher-level operations. There are a variety of other useful higher-level operations on plaintexts that, while not natively supported by BGV-style FHE, have reasonably efficient "arithmetizations" in terms of native operations. Alchemy includes a few such operations, which are implemented entirely using standard combinators on plaintext-DSL terms, and can be used just as easily as the native operations.

One important example is the mod-p rounding function $\lfloor\cdot\rceil_2 : \mathbb{Z}_p \to \mathbb{Z}_2$, which is defined as $\lfloor x\rceil_2 = \lfloor \frac{2}{p} \cdot x \rceil = \lfloor \frac{2}{p} \cdot x + \frac{1}{2} \rfloor$. This function plays an important role in bootstrapping for FHE [3, 31], as well as in the Learning With Rounding problem [6]. While rounding is trivial to implement in the clear, it is not natively supported by BGV-stlye FHE. However, when $p = 2^k$ is a power of two, there are known arithmetizations as low-depth circuits with native FHE operations as the gates. One was given in [31], and was subsequently improved in [3]; in the full version of this work we provide one that is even more efficient for $p \leq 32$.

The Alchemy implementation of the rounding tree (slightly simplified for readability) is given by:

```
rescaleTree_ :: (Lambda_ expr, Div2_ expr r2, ...) =>
  expr e (PreRescaleTree_ expr k r2 -> r2)
rescaleTree_ = ...

type family PreRescaleTree_ expr k r2 where ...
```

The type `k` is a positive natural number representing the power of two associated with the input modulus $p = 2^k$. The term `rescaleTree_` represents a function from `(PreRescaleTree expr k r2)` to `r2`, which should represent a ring modulo $p = 2^k$ and two, respectively.[7] Observe that the `PreRescaleTree_` type family is recursively defined in terms of `PreMul_` and `PreDiv2_`.

The following small example shows the type and printed representation in terms of native operations, for $p = 4$. Note that the entire type of `round4` is inferred by the Haskell compiler.

```
round4 = rescaleTree_ @2  -- choose k = 2 for p = 2^2 = 4
-- (Lambda_ expr, AddLit_ expr (PreMul_ expr (PreDiv2_ expr r2)),
--  Mul_ expr (PreDiv2_ expr r2), Div2_ expr r2, ...)
-- => expr e (PreMul_ expr (PreDiv2_ expr r2) -> r2)
print round4
-- "(\v0 -> (div2 ((mul v0) (addLit (Scalar ZqB 1) v0))))"
```

[7] Note that in our application in Section 5, the ring is not just the integers $\mathbb{Z}$, but an appropriate cyclotomic ring that has many mod-2 "slots." The very same `rescaleTree_` function operates in parallel over the slots, without modification.

3.2 Ciphertext DSL

For the ciphertext DSL, we define language components that model the operations that can be performed on BGV-style FHE ciphertexts. These include arithmetic operations from the generic **Add_** and **Mul_** classes, and ciphertext "maintenance" operations like key-switching and linearization. The latter are defined in a language component called **SHE_** (for "somewhat homomorphic encryption," another name for encryption that supports a bounded amount of homomorphic computation). Because Alchemy currently targets the FHE implementation from $\Lambda\circ\lambda$ [22], the ciphertext DSL operations use its types.

FHE types in $\Lambda\circ\lambda$. A plaintext is an element of the mth *cyclotomic ring* modulo an integer p, denoted $R_p = \mathbb{Z}_p[X]/(\Phi_m(X))$, where $\Phi_m(X)$ is the mth cyclotomic polynomial. In $\Lambda\circ\lambda$, this ring is represented with the data type `Cyc m zp`, where `m` is a type representing the *cyclotomic index* m, and `zp` is a type representing $\mathbb{Z}_p$, the ring of integers modulo p.

A ciphertext is a (usually linear) polynomial over R'_q, the m'th cyclotomic ring modulo some $q \gg p$, where the index m' must be divisible by the plaintext index m. Ciphertexts are represented by the type `CT m zp (Cyc m' zq)`. A secret key for a ciphertext of this type has type `SK (Cyc m' z)`, where `z` represents the ring of integers $\mathbb{Z}$ (not modulo anything).

Arithmetic operations. The ciphertext data type **CT** (with appropriate arguments) is an instance of Haskell's **Additive** and **Ring** classes, so we can use the `+` and `*` operators on ciphertexts to perform homomorphic addition and multiplication, respectively. Therefore, the **Add_** and **Mul_** instances for, say, the evaluator **E** and printer **P** described in Section 2 already handle object-language addition `+:` and multiplication `*:` of ciphertexts, with no additional code.

Other homomorphic operations. The remainder of the ciphertext DSL is (almost) entirely represented by the **SHE_** language component, which closely corresponds to the public interface of $\Lambda\circ\lambda$'s implementation. Due to space restrictions, we defer its formal definition and a description of its methods to the full version.

Measuring ciphertext error. Ciphertexts have an implicit error term that grows as homomorphic operations are performed. If this error becomes too large relative to the ciphertext modulus, the ciphertext does not decrypt correctly to the intended plaintext. Therefore, it is important to control the error growth during homomorphic computation. For diagnostic purposes, it can be helpful to just decrypt the ciphertext and observe the empirical error. This operation is captured by the following language component:

```
class ErrorRate_ expr where
  errorRate_ :: (...)
    => SK (Cyc m' z)
    -> expr e (CT m zp (Cyc m' zq) -> Double)
```

Because extracting the error term requires the decrypting the ciphertext, `errorRate_` requires the secret key for the ciphertext. But observe that the secret key properly resides in the *host* language because it is generated prior to the evaluation of the homomorphic computation.

4 PLAINTEXT-TO-CIPHERTEXT COMPILER

In this section we describe the design and implementation of Alchemy's "plaintext-to-ciphertext" compiler **PT2CT**, which, given an "in-the-clear" program in the plaintext DSL, interprets it as a corresponding homomorphic computation in the ciphertext DSL. In keeping with Alchemy's modular design, the resulting program can in turn be handled by any ciphertext-DSL interpreter, such as the evaluator, the printer, or another transformation like an optimizer or the error-rate logger described in Appendix C.

The **PT2CT** compiler automatically performs a number of tasks to reduce the burden on the programmer and the complexity of application code: it generates and manages all necessary keys, key-switching and tunneling hints, and input ciphertexts to the homomorphic computation. And it *statically* (i.e., at compile time) infers quite sharp upper bounds on the error rates of every ciphertext in the homomorphic computation, using these to choose appropriate ciphertext moduli based on their "error capacity." If the programmer has not supplied type-level moduli with enough error capacity for the desired homomorphic computation, **PT2CT** emits an informative compile-time type error.

4.1 Interface and Design

We now provide more detail on **PT2CT**'s public interface and the key considerations affecting its design. Using **PT2CT** is as simple as calling `pt2ct`, with certain types specifying additional FHE parameters, on a plaintext-DSL expression that uses the cyclotomic type **PNoiseCyc** to allow static tracking of ciphertext error rates:

```
-- define a plaintext computation
ex1  = lam2 $ \x y -> (var x +: var y) *: var y
-- specialize ex1's output type for error tracking
ex1' = ex1 :: _ => expr e (_ -> _ -> PNoiseCyc 0 F8 _)

do -- compile to homomorphic computation
   exct1 <- pt2ct @CTRingMap @CTModuli @Gad ex1'
   xct   <- encrypt x           -- encrypt plaintext inputs
   yct   <- encrypt y
   return $ eval exct1 xct yct  -- evaluate homomorphically
```

To generate an encrypted input for the generated homomorphic computation one simply invokes `encrypt` on a plaintext value, and invokes `decrypt` on the ultimate encrypted output to recover the plaintext; no explicit keys or extra parameters are required.

The design of **PT2CT** is guided by a few key technical challenges and our solutions to them, summarized as follows and elaborated upon below. First, the exposed type of the plaintext-DSL program that **PT2CT** interprets needs to be statically transformed to a corresponding type of the output ciphertext-DSL program, which involves additional ciphertext parameters. In particular, plaintext types must be converted to ciphertext types over an appropriate cyclotomic ring, and having large enough moduli for their error rates. Similarly, *function* types on plaintexts must be converted to function types on ciphertexts. We accomplish this by parameterizing **PT2CT** by a corpus of additional ciphertext parameters, and by defining a sophisticated *type family*—i.e., a function on types applied at compile time—that performs the desired plaintext-to-ciphertext type conversions.

Second, we want **PT2CT** to statically track ciphertext error rates and choose corresponding moduli for each ciphertext. To do this,

```
newtype PT2CT
  m'map     -- | type list of (PT index m, CT index m')
  zqs       -- | type list of pairwise coprime ℤ_q components
  gad       -- | gadget type for key-switch hints
  mon       -- | monad for creating keys and hints
  ctex      -- | ciphertext-DSL interpreter
  e         -- | environment
  a         -- | object-language plaintext type
  = PC (mon (ctex (Cyc2CT m'map zqs e) (Cyc2CT m'map zqs a)))

pt2ct :: PT2CT m'map zqs gad ctex mon e a
      -> mon (ctex (Cyc2CT m'map zqs e) (Cyc2CT m'map zqs a))
pt2ct (PC ex) = ex

type family Cyc2CT m'map zqs a = ct where ...

encrypt :: ... => Cyc m zp -> mon (CT m zp (Cyc m' zq))
decrypt :: ... => CT m zp (Cyc m' zq) -> mon (Maybe (Cyc m zp))
```

Figure 1: Definition and public interface of the plaintext-to-ciphertext compiler PT2CT.

PT2CT requires the types in the plaintext-DSL program to be augmented by what are essentially (mild upper bounds on) the error rates of the corresponding ciphertexts. Given a desired error rate for the ultimate output, the Haskell compiler "backtracks" through the computation to compute error-rate bounds for all the preceding ciphertexts. This is done via PT2CT's instances of the type families PreMul_, PreLinearCyc_, etc., which, to recall, are associated with the various plaintext-DSL components. For example, these type families may determine that for mul_ to produce an output with error rate at most 2^{-50}, the inputs should have error rates at most 2^{-68}. (See Section 4.2.2 for details.)

Third, we want PT2CT to generate and have access to the random keys and hints that are used in the homomorphic computation, and for encrypting inputs and decrypting outputs. These values properly reside in the *host* language because they are used only to *construct* the ciphertext-DSL program and its encrypted inputs. Therefore, all this is best modeled by embedding the ciphertext-DSL program in a host-language *"accumulator" monad* that provides append-only generation and reading of keys and hints.

Figure 1 shows the definition and public interface of PT2CT, which is parameterized by several types, whose meanings are given in the comments. Based on these parameters, PT2CT represents a plaintext-DSL program as a ciphertext-DSL program of type Cyc2CT m'map zqs a, interpreted by ctex, and embedded in the host-language monad mon. The function pt2ct just returns this representation.

Cyc2CT is a *type family*—i.e., a function from types to types—that converts an "in-the-clear" type to a corresponding "homomorphic" type. In particular, it converts the error-rate–augmented cyclotomic ring type PNoiseCyc p m zp to the ciphertext type over the cyclotomic ring of index m' = Lookup m m'map, with a large enough ciphertext modulus as determined by the error rate corresponding to p (see Section 4.2.2 for details). Similarly, it converts the *function* type a -> b by recursing on both arguments a, b, so that functions on plaintexts (even higher-order functions) map to functions on ciphertexts of corresponding types.

4.2 Implementation

4.2.1 Instantiations of Language Components. We now show how PT2CT interprets some instructive plaintext-DSL components. Some plaintext operations, like addition, translate directly to ciphertext-DSL addition of ciphertexts. This leads to a trivial Add_ instance:

```
instance (Add_ ctex (Cyc2CT m'map zqs a), Applicative mon)
  => Add_ (PT2CT m'map zqs gad ctex mon) a where
  add_ = PC $ pure add_
  neg_ = PC $ pure neg_
```

Here add_ just embeds the ciphertext-DSL function add_ into the host-language (applicative) monad mon, and similarly for neg_.

Multiplication. By contrast, translating plaintext multiplication to a full homomorphic ciphertext multiplication is much more involved, but still has relatively concise code for the amount of work it does:

```
instance (Lambda_ ctex, SHE_ ctex, Mul_ ctex (PreMul_ ...),
          MonadAccumulator Hints mon,
          MonadAccumulator Keys  mon,
          MonadRandom            mon, ...)
  => Mul_ (PT2CT m'map zqs gad ctex mon) (PNoiseCyc p m zp) where

  mul_ = PC $
    lamM $ \x -> lamM $ \y -> do
      hint <- getQuadCircHint   -- lookup/gen key-switch hint
      return $ modSwitch_ .:    -- switch to output modulus
        keySwitchQuad_ hint .:  -- switch quad ctext to linear
          modSwitch_ $:         -- switch to hint modulus
            var x *: var y      -- multiply input ciphertexts
```

The body of mul_ first multiplies (using *:) the ctex terms bound to the input variables x and y; this requires Mul_ ctex (PreMul_ ...). The result is a *quadratic* ciphertext, which is switched back to a linear ciphertext using keySwitchQuad_; hence the SHE_ ctex constraint. Key-switching requires an appropriate "hint," which is looked up using getQuadCircHint, a monadic function that looks up (or generates and stores) an appropriate hint by looking up (or generating and storing) the appropriate secret keys. (All this requires the various Monad... mon constraints.)

Linear functions. Another plaintext-DSL operation with a nontrivial homomorphic implementation is linearCyc_ f, which, to recall from Section 3.1, represents applying a desired linear function f from one (plaintext) cyclotomic ring to another:

```
instance (Lambda_ ctex, SHE_ ctex,
          MonadAccumulator Keys mon,
          MonadRandom           mon, ...)
  => LinearCyc_ (PT2CT m'map zqs gad ctex mon) (PNoiseCyc p) where
  linearCyc_ f = PC $
    lamM $ \x -> do
      hint <- getTunnelHint f   -- generate a hint for tunneling
      return $ modSwitch_ .:    -- switch to the output modulus
        tunnel_ hint .:         -- tunnel with the hint
          modSwitch_ $: var x   -- switch to the hint modulus
```

Here linearCyc_ applies a special type of ring-switching [30] called *ring tunneling* [22]. This is implemented as a special form of key switching, which requires an appropriate hint that is generated using the monadic function getTunnelHint.

4.2.2 Statically Tracking Error Rates. Most homomorphic operations induce error growth in the resulting ciphertexts. The precise amount of error growth depend in a rather complex way on the operation, the error in the input ciphertext(s), the choice of gadget, the ciphertext ring and moduli, and more. When composing homomorphic operations, we need the error in the ultimate output ciphertext to be small enough (relative to the ciphertext modulus) that it will decrypt correctly.

To ensure correct decryption for a given homomorphic computation, **PT2CT** uses Haskell's type system to *statically* compute error-rate bounds, working backwards from the output to its inputs. Given a DSL expression and a desired output error rate, **PT2CT** inductively does *type-level arithmetic*—via the type families **PreMul_**, **PreLinearCyc_**, etc.—to compute sufficient bounds for all the preceding ciphertexts to yield the desired output rate. It then chooses sufficiently large ciphertext moduli to support these error rates, by combining component moduli from its given corpus. If there are insufficiently many (or insufficiently large) moduli in the corpus, compilation fails and an informative type error is raised.

To facilitate these type computations, **PT2CT** requires the use of a special data type **PNoiseCyc** `p m zp` for plaintext cyclotomic rings. This type is parameterized by a type `p` representing a natural number p, called the "pNoise bound" of the corresponding ciphertext. It is essentially a bound on the *negative logarithm* of the error rate α of the ciphertext, i.e., $\alpha \lesssim 2^{-Cp}$, where C is a constant representing one multiplicative "unit" of error rate. (The name "pNoise" reflects this logarithmic relationship.) For pNoise p, the corresponding ciphertext modulus would need to be $\gtrsim Cp$ bits long. We typically set the ultimate output pNoise $p = 0$ to indicate that the ciphertext need not be used for any further homomorphic computation; intermediate ciphertexts generally have increasing pNoise values from the output back to the inputs.

5 HOMOMORPHIC PRF EVALUATION

Having surveyed the different components of Alchemy on toy programs, we now demonstrate its use with a full-size example. Specifically, we demonstrate fast homomorphic evaluation of a candidate pseudorandom function (PRF) based on the Learning With Rounding (LWR) lattice problem [6]. The heart of this PRF is a "ring-rounding" function, which independently rounds each $\mathbb{Z}_p$-coefficient (relative to a certain basis) of a cyclotomic ring element modulo some $p \gg 2$ to $\mathbb{Z}_2$, essentially keeping only the most significant bit. This ring-rounding functionality is also the heart of the asymptotically efficient "bootstrapping" method for BGV-style FHE systems developed in [3, 31], so our approach here applies equally well to bootstrapping, though significantly larger parameters would be needed.

5.1 Ring-Rounding In the Clear

We first need to arithmetize the ring-rounding function in terms of plaintext operations that the target FHE scheme natively supports, then program it in the plaintext DSL. For a modulus $p = 2^k$ (e.g., $p = 32$ or $p = 64$), the integer rounding function $\lfloor \cdot \rceil_p : \mathbb{Z}_p \to \mathbb{Z}_2$ has an efficient arithmetization that is implemented as the plaintext-DSL function `rescaleTree_`; see Section 3.1. However, this function is defined for *integers* modulo p; when the same arithmetic operations are applied to a cyclotomic ring element in R_p, the result is nonsense because the $\mathbb{Z}_p$-coefficients "mix" together.

To map the $\mathbb{Z}_p$-rounding function over the coefficients of a ring element, we first apply a linear function that moves the coefficients into the "$\mathbb{Z}_p$-slots" of another carefully-chosen ring. Concretely, we use the efficient method described in [3] and refined in [22], which switches through a sequence of "hybrid" cyclotomic rings that convert the source ring to the target ring in small steps. Once the coefficients are in slots, applying the arithmetized $\mathbb{Z}_p$-rounding function to the entire ring element causes each slot to be rounded independently, due to the product-ring structure of the slots.[8]

The precise sequence of hybrid rings that move the coefficients into slots depends on the particular choice of source ring: in particular, the target ring needs to have at least as many $\mathbb{Z}_p$-slots as the degree of the source ring. Therefore, the programmer must supply a suitable sequence of hybrid rings. Fortunately, these are not hard to find with a little trial and error using the principles laid out in [3], and several example sequences are available in the literature.

Plaintext-DSL implementation. Here we give what is essentially our entire plaintext-DSL implementation of the (arithmetized) ring-rounding function. The clarity and concision of the code makes its intent apparent, and demonstrates the ease of expressing nontrivial computations.

First we define (type-level) indices for an appropriate sequence of hybrid cyclotomic rings (see Figure 2):

```
type H0 =  F128
type H1 =  ...
```

Next, we define a plaintext-DSL function `coeffsToSlots` as a composition of ring-switches through the hybrid rings, which maps the 64 coefficients of the **H0**th ring to the slots of the **H5**th ring:

```
coeffsToSlots =
  linearCyc_ (decToCRT @H4) .: linearCyc_ (decToCRT @H3) .:
  linearCyc_ (decToCRT @H2) .: linearCyc_ (decToCRT @H1) .:
  linearCyc_ (decToCRT @H0)
```

Recall from Section 3.1 that `linearCyc_ f` yields a plaintext-DSL function for a given linear function `f` from one cyclotomic ring to another. Here we take each `f` to be a specialization of the polymorphic function `decToCRT`, which essentially maps a portion of one hybrid-ring coefficients to a portion of the slots of the next hybrid ring. The type applications **@H0**, **@H1** etc. are necessary to specify the concrete choices of rings for the polymorphic `decToCRT`, but all other types and constraints are inferred by the Haskell compiler.

Finally, we define the full ring-rounding function as a composition of the above linear function and the arithmetized rescaling tree for our choice of modulus p:

```
type K    = 5  -- modulus p = 2^5 = 32
ringRound :: _ => expr env (_ -> PNoiseCyc 0 H5 (Zq 2))
ringRound = (rescaleTree_ @K) .: coeffsToSlots
```

Note that we have monomorphized the output type as required for plaintext-to-ciphertext compilation, setting the pNoise parameter to zero to indicate that no further homomorphic operations are needed on the output.

[8] Optionally, we could switch back to the original cyclotomic ring to move the rounded slot entries back to coefficients, but this is not needed for our application so we omit it. Switching back *would* be required for bootstrapping, but fortunately, switching is a small fraction of the overall running time in the bootstrapping scenario.

5.2 Rounding Homomorphically

Next, we want to use **PT2CT** to compile the plaintext-DSL function ringRound to a ciphertext-DSL function that homomorphically rounds the coefficients of an encrypted input. Recall that to use its interface pt2ct, we need to specify types for the cyclotomic indices of ciphertext rings and available ciphertext moduli (see Figure 2):

```
type H0'   = H0 * 7 * 13
type H1'   = ...
type M'Map = [(H0,H0'), (H1,H1'), ...]
-- corpus of ciphertext moduli
type Zqs = [ Zq $(mkModulus 1543651201), ... ]
```

The type **M'Map** associates each plaintext ring index with its ciphertext ring index. The type **Zqs** is a collection of $\mathbb{Z}_q$-types that **PT2CT** combines to assign large enough moduli to each ciphertext.

Having defined the necessary types, it is now trivial to compile the plaintext computation to its homomorphic counterpart:

```
homomRingRound = pt2ct @M'Map @Zqs @TrivGad ringRound
```

Recall that pt2ct yields a monadic value, where the monad needs to support generation and accumulation of keys and hints, so we need to use homomRingRound in an appropriate monadic context.

5.3 Homomorphic PRF Evaluation

Now that we have homomRingRound, we can easily implement homomorphic evaluation of RLWR-based PRFs. The simplest such PRF is defined as $F_s(x) = \lfloor H(x) \cdot s \rceil_2$, where $H\colon \mathcal{X} \to R_p$ is a hash function modeled as a random oracle mapping the PRF input space to R_p, and $\lfloor \cdot \rceil_2$ denotes the ring-rounding function. (There are also variants where H is replaced with a publicly evaluable function [5, 10], which has no effect on homomorphic evaluation.)

To homomorphically compute an FHE encryption of $F_s(x)$ given an FHE encryption $\widetilde{s}$ of the secret key $s \in R_p$ and an input x, we first compute the hash value $a = H(x) \in R_p$, then we use the FHE's multiply-by-a-public-value operation to get an encryption $\widetilde{a \cdot s}$. Finally, we apply homomorphic ring rounding to get an encryption of the PRF output: $\widetilde{\lfloor a \cdot s \rceil_2} = \widetilde{F_s(x)}$.

The following code chooses a uniformly random PRF key $s \in R_p$ and returns s, a host-language function f that maps any $a \in R_p$ to $\widetilde{\lfloor a \cdot s \rceil_2}$, and the corpus of generated FHE keys that allows for decrypting the result:

```
homomRLWR = do
  s <- getRandom
  (f, keys, _) <- runKeysHints $
    liftM2 (.) (eval <$> homomRingRound) $
               flip mulPublic <$> encrypt s
  return (s, f, keys)
```

Here homomRLWR is a monadic value, where the monad just needs to provide a source of randomness. The call to runKeysHints sets up the additional "accumulator monad" context required by homomRingRound, and outputs the desired host-language function f along with all the generated FHE keys and hints (the latter of which we ignore as unneeded).

5.4 Parameters, Security and Performance

The concrete cyclotomic ring indices and ring dimensions we use are given in Figure 2. Our PRF uses a modulus of $p = 2^5 = 32$. Our entire corpus of ciphertext moduli is altogether less than 2^{180}. According to the "core-SVP" methodology [2] for estimating the security of LWE/LWR parameters, our FHE and PRF parameters have at least 100 bits of security (and this is likely a significant underestimate).

On an iMac (Retina 5k) late-2015 model with 4 GHz Core i7 and 16 GB RAM, homomorphic evaluation of the PRF itself (after generating all keys and hints) takes only *10–11 seconds* for each of several runs. Generating the keys and hints takes about 150 seconds, due primarily to a very naive implementation of finite-field arithmetic in $\Lambda \circ \lambda$, which is used for computing decToCRT. These performance figures were achieved with no compiler optimization flags turned on (ghc-8.0.2), so even better performance may be possible; however, certain flags cause compilation to take a great deal of time and memory.

PT ring index m	CT ring index m'	dimension $\varphi(m')$
$H_0 = 2^7$	$H'_0 = H_0 \cdot 7 \cdot 13$	4,608
$H_1 = 2^6 \cdot 7$	$H'_1 = H_1 \cdot 5 \cdot 13$	9,216
$H_2 = 2^5 \cdot 7 \cdot 13$	$H'_2 = H_2 \cdot 3 \cdot 5$	9,216
$H_3 = 2^3 \cdot 5 \cdot 7 \cdot 13$	$H'_3 = H_3 \cdot 3 \cdot 5$	11,520
$H_4 = 2^2 \cdot 3 \cdot 5 \cdot 7 \cdot 13$	$H'_4 = H_4 \cdot 5$	5,760
$H_5 = 2^0 \cdot 3^2 \cdot 5 \cdot 7 \cdot 13$	$H'_5 = H_5 \cdot 5$	8,640

Figure 2: Cyclotomic ring indices used for the homomorphic evaluation of the Ring-LWR PRF.

REFERENCES

[1] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2015. Ciphers for MPC and FHE. In *EUROCRYPT*. 430–454.

[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. 2016. Post-quantum Key Exchange - A New Hope. In *USENIX Security Symposium*.

[3] Jacob Alperin-Sheriff and Chris Peikert. 2013. Practical Bootstrapping in Quasi-linear Time. In *CRYPTO*. 1–20.

[4] Jacob Alperin-Sheriff and Chris Peikert. 2014. Faster Bootstrapping with Polynomial Error. In *CRYPTO*. 297–314.

[5] Abhishek Banerjee and Chris Peikert. 2014. New and Improved Key-Homomorphic Pseudorandom Functions. In *CRYPTO*. 353–370.

[6] Abhishek Banerjee, Chris Peikert, and Alon Rosen. 2012. Pseudorandom Functions and Lattices. In *EUROCRYPT*. 719–737.

[7] Assaf Ben-David, Noam Nisan, and Benny Pinkas. 2008. FairplayMP: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security*. 257–266.

[8] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *STOC*. 1–10.

[9] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS*. 192–206.

[10] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. 2013. Key Homomorphic PRFs and Their Applications. In *CRYPTO*. 410–428.

[11] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *CRYPTO*. 868–886.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *TOCT* 6, 3 (2014), 13. Preliminary version in ITCS 2012.

[13] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In *CRYPTO*. 505–524.

[14] Zvika Brakerski and Vinod Vaikuntanathan. 2014. Efficient Fully Homomorphic Encryption from (Standard) LWE. *SIAM J. Comput.* 43, 2 (2014), 831–871. Preliminary version in FOCS 2011.

[15] Zvika Brakerski and Vinod Vaikuntanathan. 2014. Lattice-Based FHE as Secure as PKE. In *ITCS*. 1–12.

[16] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543.

[17] Hao Chen, Kim Laine, and Rachel Player. 2017. Simple Encrypted Arithmetic Library - SEAL v2.1. In *Financial Cryptography and Data Security*. 3–18.
[18] Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrède Lepoint, Mehdi Tibouchi, and Aaram Yun. 2013. Batch Fully Homomorphic Encryption over the Integers. In *EUROCRYPT*. 315–335.
[19] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. 2011. Fully Homomorphic Encryption over the Integers with Shorter Public Keys. In *CRYPTO*. 487–504.
[20] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. 2012. Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers. In *EUROCRYPT*. 446–464.
[21] Jack L.H. Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup. 2018. Doing Real Work with FHE: The Case of Logistic Regression. Cryptology ePrint Archive, Report 2018/202. https://eprint.iacr.org/2018/202.
[22] Eric Crockett and Chris Peikert. 2016. $\Lambda \circ \lambda$: Functional Lattice Cryptography. In *ACM CCS*. 993–1005. Full version at http://eprint.iacr.org/2015/1134.
[23] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. 2009. Asynchronous Multiparty Computation: Theory and Implementation. In *PKC*. 160–179.
[24] Yarkin Doroz, Yin Hu, and Berk Sunar. 2014. Homomorphic AES Evaluation using NTRU. Cryptology ePrint Archive, Report 2014/039. https://eprint.iacr.org/2014/039.
[25] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *EUROCRYPT*. 617–640.
[26] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Ph.D. Dissertation. Stanford University. http://crypto.stanford.edu/craig.
[27] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *STOC*. 169–178.
[28] Craig Gentry, Jens Groth, Yuval Ishai, Chris Peikert, Amit Sahai, and Adam D. Smith. 2015. Using Fully Homomorphic Hybrid Encryption to Minimize Non-interative Zero-Knowledge Proofs. *J. Cryptology* 28, 4 (2015), 820–843.
[29] Craig Gentry and Shai Halevi. 2011. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *EUROCRYPT*. 129–148.
[30] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. 2013. Field switching in BGV-style homomorphic encryption. *Journal of Computer Security* 21, 5 (2013), 663–684. Preliminary version in SCN 2012.
[31] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Better Bootstrapping in Fully Homomorphic Encryption. In *Public Key Cryptography*. 1–16.
[32] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Fully Homomorphic Encryption with Polylog Overhead. In *EUROCRYPT*. 465–482.
[33] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. In *CRYPTO*. 850–867.
[34] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *CRYPTO*. 75–92.
[35] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*. 218–229.
[36] Shai Halevi and Victor Shoup. [n. d.]. HElib: an implementation of homomorphic encryption. https://github.com/shaih/HElib, last retrieved August 2016.
[37] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *CRYPTO*. 554–571.
[38] Shai Halevi and Victor Shoup. 2015. Bootstrapping for HElib. In *EUROCRYPT*. 641–670.
[39] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. 2010. TASTY: Tool for Automating Secure Two-partY computations. In *ACM Conference on Computer and Communications Security*. 451–462.
[40] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2015. Combinators for impure yet hygienic code generation. *Sci. Comput. Program.* 112 (2015), 120–144.
[41] Oleg Kiselyov. 2010. Typed Tagless Final Interpreters. In *Generic and Indexed Programming - International Spring School, SSGIP 2010*. 130–174. http://okmij.org/ftp/tagless-final/.
[42] John Launchbury, Dave Archer, Thomas DuBuisson, and Eric Mertens. 2014. Application-Scale Secure Multiparty Computation. In *ESOP*. 8–26.
[43] Philip D. MacKenzie, Alina Oprea, and Michael K. Reiter. 2003. Automatic generation of two-party computations. In *ACM Conference on Computer and Communications Security*. 210–219.
[44] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - Secure Two-Party Computation System. In *USENIX Security Symposium*. 287–302.
[45] Silvia Mella and Ruggero Susella. 2013. On the Homomorphic Computation of Symmetric Cryptographic Primitives. In *IMACC*. 28–44.
[46] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. 2011. Can homomorphic encryption be practical?. In *CCSW*. 113–124.
[47] Janus Dam Nielsen and Michael I. Schwartzbach. 2007. A domain-specific programming language for secure multiparty computation. In *Workshop on Programming Languages and Analysis for Security*. 21–30.
[48] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *IEEE Symposium on Security and Privacy*. 655–670.
[49] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. 1978. On Data Banks and Privacy Homomorphisms. *Foundations of secure computation* 4, 11 (1978), 169–180.
[50] Victor Shoup. 2006. A library for doing number theory. http://www.shoup.net/ntl/, version 9.8.1.
[51] Nigel P. Smart and Frederik Vercauteren. 2014. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography* 71, 1 (2014), 57–81. Preliminary version in ePrint Report 2011/133.
[52] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. 2010. Fully Homomorphic Encryption over the Integers. In *EUROCRYPT*. 24–43.
[53] Andrew Chi-Chih Yao. 1982. Theory and Applications of Trapdoor Functions (Extended Abstract). In *FOCS*. 80–91.

A BASIC INTERPRETERS

In this section we describe the implementation of a few simple object-language interpreters included with Alchemy, which are due to [16, 41]. (We have already seen one interpreter in its entirety: the evaluator `E` described in Section 2.)

A.1 Printer

The printer converts an object-language expression into a (host-language) string representing the expression. This interpreter was presented in a simplified form in Section 2; we describe the actual implementation here. It has the following formal definition and public interface:

```
newtype P e a = P (Int -> String)

print :: P () a -> String
print (P f) = f 0  -- closed code has lambda depth 0
```

This says that an object-language expression is represented by an `Int -> String` function. The `Int` argument indicates how many variables have already been bound "outside" the expression, and is used only by the `Lambda_` instance to unambiguously name new variables:

```
instance Add_ P a where
  add_ = P $ \i -> "add"   -- or: P $ pure "add"

-- instances of Mul_ etc. are similar

instance Lambda_ P where
  lamDB (P f) =
    P $ \i -> "(\\v" ++ show i ++ " -> " ++ f (i+1) ++ ")"
  (P f) $: (P a) =
    P $ \i -> "(" ++ f i ++ " " ++ a i ++ ")"
  v0           = P $ \i -> "v" ++ show (i-1)
  weaken (P v) = P $ \i -> v (i-1)
```

Here `lamDB` creates (a function mapping an `Int` i to) a string representing a function of variable `v`i, where the body of the lambda is in a context where $i + 1$ variables have been bound. Function application (the `$:` operator) produces (a function mapping to) a string that just appends the function string and the argument string; the number of bound variables does not change. For `v0`, because the body of any function appears in a context with at least one bound variable (by definition), we print index $i - 1$ so that variables are zero-indexed. Finally, because `weaken` refers to a variable one step "down" the environment stack, it recursively interprets the subexpression in a context with one fewer bound variable.

A.2 Expression Size

A useful metric for the complexity of an expression is its *size* in term of the number of "primitive" DSL terms used. The interpreter `S` that converts an expression to its size is defined as follows:

```
newtype S e a = S Int

size :: S () a -> Int
size (S i) = i
```

This says that an expression is simply represented by an `Int`, which can be extracted using `size`. With this definition it is trivial to interpret all the language components:

```
instance Add_ S a where
  add_ = S 1

-- instances of Mul_ etc. are similar

instance Lambda_ S where
  lamDB (S i) = S $ i+1
  (S f) $: (S a) = S $ f + a
  v0 = S 1
  weaken (S i) = S i
```

The most interesting instance is `Lambda_`. The size of a function definition (i.e., a call to `lamDB`) is one larger than the size of its body, and the size of a function application (the `$:` operator) is the sum of the sizes of the function and the argument. As for uses of variables, `v0` clearly should have unit size. The same should be true for any other variable regardless of its location in the environment, so `weaken` does not change the size of its subexpression.

Using `S` is easy:

```
ex1 = lam $ \x -> lam $ \y -> var y *: (var y +: var x)
size ex1
-- 7
```

A.3 Duplicator

The above interpreters each use a concrete host-language representation (a `String`, an `Int`) as their interpretations of object-language expressions. Such interpreters are "terminal" in that they produce a result in the host language. However, we also have interpreters (like the error-rate logger `ERW` and plaintext-to-ciphertext compiler `PT2CT`) that *transform* object-language code into other object-language code, i.e., "compilers." Importantly, we want the produced code to itself be interpretable in a variety of ways, i.e., it should be also polymorphic in its interpreter. Unfortunately, this is not currently possible in mainstream Haskell: while we can write polymorphic "*top-level*" DSL code that can be interpreted in multiple ways by monomorphizing the interpreter, any DSL code *output* by an interpreter must be *monomorphic*; therefore, it can be handled by only one interpreter.[9]

Fortunately, [41] provides a simple way to work around this restriction via a special interpreter that *duplicates* any object-language expression into two equivalent expressions. These expressions, while necessarily monomorphic, can use different interpreters. This duplicating interpreter `Dup` has the following definition and public interface:

[9]The functionality we seek is known as *impredicative polymorphism*, which is quite poorly supported in the GHC Haskell compiler.

```
data Dup expr1 expr2 e a = Dup (expr1 e a) (expr2 e a)

dup :: Dup expr1 expr2 e a -> (expr1 e a, expr2 e a)
dup (Dup ex1 ex2) = (ex1, ex2)
```

Notice that the `Dup` interpreter is parameterized by two other interpreters, `expr1` and `expr2`, and represents an object-language expression simply as a pair of such expressions, one for each of these interpreters. Naturally this idea can be applied recursively to interpret an expression in arbitrarily many ways, by letting one or both of `expr1`, `expr2` themselves be a `Dup` (with appropriate arguments).

The instances for `Dup` are all very simple and completely mechanical. For example:

```
instance (Add_ expr1 a, Add_ expr2 a)
  => Add_ (Dup expr1 expr2) a where
  add_ = Dup add_ add_
  neg_ = Dup neg_ neg_
```

The constraints on the instance say that in order for `Dup expr1 expr2` to be able to interpret `Add_` for (object-language) type `a`, the interpreters `expr1`, `expr2` must be able to do the same. The implementation mirrors these constraints: the `add_` term for `Dup expr1 expr2` is simply the pair of `add_` terms for the `expr1` and `expr2` interpreters.

Using `Dup` is very simple:

```
ex = functionThatOutputsSomeDSLExpression someArg

-- the following code infers the *monomorphic* type
-- ex :: Dup P E () Int

(ex1, ex2) = dup ex
print ex1
-- "(\\v0 -> ((add v0) v0))"
eval ex2 3
-- 6
```

B OBJECT-LANGUAGE FUNCTIONS AND HOAS

Recall that we wish to have function abstraction and application in our object languages, using the host language's facilities for the same. There is a standard, very elegant solution:

```
class LambdaPure_ expr where
  lamPure :: (expr a -> expr b) -> expr (a -> b)
  ($:)    :: expr (a -> b) -> expr a -> expr b

double_ :: (Lambda_ expr, Add_ expr a) => expr e (a -> a)
double_ = lamPure $ \x -> x +: x

pprint $ double_ $: (3 *: 4)
-- (\v0 -> add v0 v0) (mul 3 4)

addMulPure = lamPure $ \x -> lamPure $ \y -> (x +: y) *: y
```

Notice that there is no need for an explicit environment argument to `expr`; `LambdaPure_` lets us rely entirely on the host language for creating functions and naming variables, resolving variable references, etc. This approach is known as *higher-order abstract syntax* (HOAS).

Unfortunately, `LambdaPure_` is too weak for our needs. While *pure* host-language functions present no problem, difficulties arise

with *effectful* (monadic) ones that use side-effects to produce object-language code, such as our plaintext-to-ciphertext compiler (which creates random keys and hints during its code transformations). The problem is most easily seen with an example: given some monadic function

```
foo :: Monad m => expr a -> m (expr b)
```

we would like to convert it to an object-language function (in a monadic context) of type `m (expr (a -> b))`. Unfortunately, this is impossible in general: if `expr a` is isomorphic to `a` (as is the case with evaluators), then we are asking to transform a monadic function of type `a -> m b` into a *pure* function in a monadic context, of type `m (a -> b)`. This cannot be done because the former type allows the monadic output to depend upon the input `a`-value, whereas the latter type allows only the *function itself* to depend on the monadic context.

The difficulty of using HOAS with effectful generators has long been recognized in the literature as a thorny problem, and various approaches—none of them completely satisfactory—have been proposed; see [40] for an overview. In what follows we improve upon the solution offered in [40, 41], by allowing interpreters to hide the fact that they are effectful, and significantly simplifying the type signatures and syntax of object-language code.

Functions and environments. As an alternative to **LambdaPure_**, following [16] and [41, Section 3.3] we start from an object-language representation `expr e a`, where the *environment* argument `e` has a nested-pair structure reflecting de Bruijn-indexed variables. More specifically:

```
class Lambda_ expr where
  lamDB  :: expr (e,a) b -> expr e (a -> b)
  ($:)   :: expr e (a -> b) -> expr e a -> expr e b
  v0     :: expr (e,a) a
  weaken :: expr e a -> expr (e,x) a
```

The host-language function `lamDB` creates an object-language function (i.e., lambda abstraction). Notice its use of the environment: it converts any object-language term of type `b`, *in any environment whose "topmost" entry has type* `a`, into an object-language function of type `a -> b`.

Similarly, *in any environment whose topmost entry has type* `a`, the object-language term `v0` represents that topmost entry. Essentially, an environment can be thought of as a stack of values of specific types, and `v0` is the object-language term representing the value at the top. To allow access to values farther down, `weaken` "demotes" any object-language term in environment `e` to an equivalent term in a modified environment with an additional value (of arbitrary type) pushed on top. For example, `v1 = weaken v0` has type `expr ((e,a),x) a` and represents the second value on the stack, `v2 = weaken v1` represents the third, etc. Putting these pieces together, for example, `lamDB v0` has type `expr e (a -> a)` and represents the identity function. Finally, the `$:` operator applies an object-language function of type `a -> b` to an object-language value of type `a` to yield an object-language value of type `b`.

The definitions of `lamDB`, `v0`, `$:`, and `weaken` are trivial for the evaluator **E**. They are almost as trivial for the printer, but the **P** type needs to be redefined as a *function* from the "`lamDB` depth" of the term to **String**, so that proper variable indices can be generated. See Appendix A.1 for the actual definition of this interpreter.

Higher-order abstract syntax (HOAS).. Referencing variables by their positions in the environment (i.e., de Bruijn indexing) soon becomes painful, because the same index, e.g., `v0` or `v2`, can represent different object-language values depending on its lexical scope, and tracking down the correct binding location in the code can be quite difficult. We would instead like to use the host language for creating, binding, and referencing arbitrary human-readable variable names (i.e., HOAS), just as we could with `lamPure`, but for effectful generators.

By adapting a key idea from [40], it turns out that we can obtain HOAS as a relatively simple layer around our existing **Lambda_** class. The core idea is this: suppose we have a host-language function `f` of (for the moment underspecified) type `expr ? a -> expr ? b`, which is defined as `f = \x -> ...` We can think of the function body as an object-language expression in the "topmost" *host-language* variable `x`. We can substitute `x` with the "topmost" *object-language* variable `v0` simply by invoking `f v0`. If we can arrange for the result to have type `expr (e,a) b`, then applying `lamDB` to it yields a value of the desired type `expr e (a -> b)`. The type of `v0 :: expr (e,a) a` gives us a clue about what the full type of `f` could be, yielding the following candidate implementation:

```
unsafeLam :: Lambda_ expr
  => (expr (e,a) a -> expr (e,a) b)
  -> expr e (a -> b)
  unsafeLam f = lamDB $ f v0
```

However, as its name suggests and as shown in [40, Section 4.1], `unsafeLam` has a subtle but serious flaw: it allows variable bindings to be "mixed up." For example, consider the function

```
ex f = unsafeLam (\y -> unsafeLam (\x -> f (var x)))
```

Surprisingly, certain function arguments `f` cause `f (var x)` to evaluate to `y`! The problem, essentially, is that `f` can have a type that swaps the topmost two environment variables. This ought not be possible under the expected scoping rules of our DSL.

To rememdy the problem, we use a key idea from [40], which is to make `lam` a *higher-rank* function:

```
lam :: Lambda_ expr
    => (forall x . expr (e,x) a -> expr (e,x) b)
    -> expr e (a -> b)
lam f = lamDB $ f v0
```

The only difference with `unsafeLam` is that the host-language function `f` has a more general type: its input's topmost environment variable may have *arbitrary* type `x`. Essentially, this prevents `f` from "misbehaving" by letting `x` escape its scope or swapping it with entries of the environment, because `x` cannot be unified with them. Invoking `f v0` specializes `x` to `a`, so the same implementation as `unsafeLam` works.

Perhaps surprisingly, the above easily generalizes to our ultimate goal of HOAS for *effectful* generators!

```
lamM :: (Lambda_ expr, Functor m)
=> (forall x . expr (e,x) a -> m (expr (e,x) b))
-> m (expr e (a -> b))
lamM f = lamDB <$> f v0
```

Essentially, the type variable `x` acts as a "hole" allowing us to bring the type `a` inside the monad.

These abstractions do not yield quite the same level of simplicity as `lamPure` did above. In particular, replacing `lamPure` with `lam` in

`addMulPure` fails to typecheck because `x` and `y` necessarily have different environments: `y`'s strictly contains `x`'s. In order to add these two values, we must "weaken" `x` by extending its environment:

```
addMul = lam $ \x -> lam $ \y -> ((weaken x) +: y) *: y
```

This is not much more usable than the original index-based approach, because the required number of `weaken`s depends on the lexical scope. To remedy this, we take one more idea from [40]. We define the `Extends m n` typeclass, which allow us to inductively weaken a term until its environment is compatible with the surrounding context.

```
class Extends m n where
var :: expr m a -> expr n a

instance {-# OVERLAPS #-} Extends m m where
var = id

instance (Extends m n, x ~ (n, e)) => Extends m x where
var = weaken . var
```

We adopt the general rule of always using `var` to automatically extend the environment of any variable (if necessary). The above example then becomes:

```
addMul = lam $ \x -> lam $ \y -> (var x +: var y) *: var y
```

Though this is not quite as simple as `lamPure`, the use of `var` is mechanical and context-independent, and gives us foolproof HOAS.

C LOGGING ERROR RATES

Recall that in FHE, ciphertexts have some internal error that grows under homomorphic operations. If the error grows too large relative to the ciphertext modulus, the ciphertext will not decrypt to the correct plaintext. The exact error growth incurred by a homomorphic computation depends in a rather complex way on the particular sequence of homomorphic operations and the various parameters of the system, so it can be difficult to predict in advance exactly what parameters, especially ciphertext moduli, should be used: moduli that are too small will make it impossible to decrypt the result, while unnecessarily large moduli induce (for security reasons) other large parameters and less efficiency.

To aid a good selection of parameters, Alchemy includes a ciphertext-DSL interpreter that logs the empirical *error rate*—essentially, the ratio of the error to the ciphertext modulus—of every ciphertext created during a homomorphic computation. By observing these error rates for a particular computation of interest, the programmer can easily adjust the parameters upward or downward as needed.

In keeping with Alchemy's modular approach, the error-rate logger is actually a *compiler* that transforms any ciphertext-DSL program into an equivalent one that additionally logs the error rates of any ciphertexts it generates. The output program can in turn be passed on to any suitable interpreter, e.g., the evaluator, the printer, other DSL transformations, etc.

C.1 Usage Example

Figure 3 gives a slightly simplified usage example of the logger's public interface `writeErrorRates`. (See Section 5 for a more involved example.) The code first defines a toy object-language function `ex` that simply adds its argument to itself. It then compiles `ex` into a homomorphic (ciphertext-DSL) computation, and

```
ex = lam $ \x -> var x +: var x

((exct, ctarg), keys, hints) <- runKeysHints $ do
    exct <- pt2ct ex
    ctarg <- encrypt =<< getRandom
    return (exct, ctarg)

exct' = runReader keys $ writeErrorRates exct

(result, errors) = runWriter $ eval exct' >>= ($ ctarg)
print errors
-- [("add_Q268440577",7.301429694065961e-7)]
```

Figure 3: Example usage of the error-rate logger.

encrypts a random value to serve as the input. These computations require keys (and in other cases, key-switching hints) to be generated and stored; `runKeysHints` sets up the monadic context for doing this, and ultimately outputs all the generated keys and hints. Next, the code invokes `writeErrorRates exct` to convert the ciphertext-DSL function into one that also logs the error rate of any ciphertext it produces (which in this case is just the output ciphertext); the call to `runReader keys` sets up the necessary monadic context that gives "reader" access to the secret keys. Lastly, the code evaluates the augmented ciphertext-DSL function on the encrypted input `ctarg`, where `runWriter` sets up the necessary monadic context for logging the error rates. This produces both the (encrypted) result and a log, which consists of a list of pairs describing the DSL operation that produced each ciphertext (helpfully augmented with the value of its modulus), and the ciphertext's error rate.

We point out that the monadic ciphertext-DSL code produced automatically from a plaintext-DSL program by `pt2ct` and `writeErrorRates` could in principle instead be written manually be the programmer. However, due to the lack of do-notation "syntactic sugar" in the object language, this would be quite burdensome on the programmer, and would greatly conceal the meaning and intent of the code.

C.2 Interface and Implementation

The design and implementation of our error-rate logger involves a number of technical challenges:

(1) To measure ciphertext error rates we need access to the corresponding decryption keys, which are properly values in the *host* language because they are generated during the *construction* of the homomorphic computation, not its *evaluation*. Therefore, access to the keys is best modeled by embedding the transformed DSL expression in a host-language *"reader" monad* that provides such access.
(2) Creating a log of error rates is most naturally seen as a *side effect* of the main homomorphic computation, which is best modeled by embedding the computation in a *"writer" monad*. Because the error rates depend on the ciphertexts, which are values in the *object language*, we need the object language itself to support monads. (The DSL components for object-language monads are given in Section 2.3.)

```
newtype ERW
  expr -- | the underlying interpreter
  k    -- | (reader) monad that supplies keys
  w    -- | (writer) monad for logging error rates
  e    -- | environment
  a    -- | object-language type
  = ERW (k (expr (Kleislify w e) (w (Kleislify w a))))

writeErrorRates ::
     ERW expr k w e a
  -> k (expr (Kleislify w e) (w (Kleislify w a)))
writeErrorRates (ERW ex) = ex

type family Kleislify w a = r | r -> a where
  Kleislify w (a -> b) = Kleislify w a -> w (Kleislify w b)
  Kleislify w (a,b)    = (Kleislify w a, Kleislify w b)
  Kleislify w [a]      = [Kleislify w a]
  Kleislify _ a        = a
```

Figure 4: Definition and public interface of the error-rate writer ERW.

(3) The exposed object-language type of the original ciphertext-DSL program must be rewritten as an appropriate object-language type of the transformed expression, incorporating all the monadic embeddings (and similarly for their environments).

The data type ERW (short for "error-rate writer") resolves all of the above issues; see Figure 4 for its formal definition and public interface. The (partially applied) type `ERW expr k w` is an interpreter which, as usual, is additionally parameterized by an environment type `e` and an object-language type `a`. It represents an object-language expression of exposed type `a` by another one of type `w (Kleislify w a)`, as interpreted by `expr`. (This representation is also embedded in the host-language reader monad `k`, modeling the fact that we need access to the decryption keys.) The `Kleislify` type family lifts the type `a -> b` of any *pure* object-language function into the *monadic* type `a -> w b`, recursing into pairs and lists to find all appearances.[10] This models the fact that the transformed function may now use the side effect of writing error rates to a log. For example, supposing that `CT` denotes the type of ciphertexts, the exposed object-language type `CT` is internally represented by the type `w CT`, and `CT -> CT -> CT` by the type `w (CT -> w (CT -> w CT))`.

Interpretation of language components. Naturally, `ERW expr k w` is an instance of all our generic and ciphertext-DSL language components, subject to appropriate constraints on `expr`, `k`, and `w`. Specifically, we need:

- `MonadReader Keys k`, which says that `k` provides access to a corpus of FHE secret keys, and
- `MonadWriter [(String,Double)] w`, which says that `w` supports the side-effect of append-only writes of `(String,Double)` pairs to a log. Here `Double` represents the empirical error rate of an intermediate ciphertext, and the matching `String` is an annotation to identify the step in the larger expression that produced it.

[10] Formally, `Kleislify` recursively lifts into the *Kleisli category* of `w`.

```
instance (Add_ expr CT, ...) => Add_ (ERW expr k w) CT where
  neg_ = ERW $ liftWriteError  "neg_" neg_
  add_ = ERW $ liftWriteError2 "add_" add_

-- | Given an annotation string, and an object-lang function that
-- | outputs a ciphertext, lift it to one that also logs the error
liftWriteError :: (ErrorRate_ expr, ...)
  => String                    -- | annotation
  -> expr e (a -> CT)          -- | the function to lift
  -> k (expr e (w (a -> w CT)))

liftWriteError str f_ = do
  key <- lookupKey
  return $ return_ $:
    case key of
      Just sk -> (after_ $: tellError_ str sk) .: f_
      Nothing -> return_ .: f_  -- | no key, so can't log

-- | Given an annotation string and a secret key, produce an
-- | object-lang function that writes a ciphertext's error rate
tellError_ :: (ErrorRate_ expr, ...)
  => String -> SK -> expr e (CT -> w ())

-- | Apply an action to a value, then return the original value
after_ :: (Monad_ expr w, ...) => expr e ((a -> w b) -> a -> w a)
```

Figure 5: Simplified partial implementation of ERW. (Many constraints are elided for brevity.)

As an illustrative example, Figure 5 shows the instantiation of `Add_` for ciphertexts, and the key supporting functions. The most important of these is `liftWriteError`, a host-language function whose type signature hints at its functionality: it transforms any object-language *pure* function that produces a ciphertext into a *monadic* one that also logs the ciphertext's error, if a suitable decryption key is available. First, `liftWriteError` uses `lookupKey` (associated with the reader monad `k`) to try to obtain a key that can decrypt the ciphertext. (This lookup is based on the ciphertext's type parameters, which we have suppressed here for readability.) If a key is found, `liftWriteError` produces a new object-language function that applies the original function, calls `tellError_` on its result, and returns that result. (The generic `after_` combinator performs this sequencing.) If no key is found, it just returns the original function, but modified so that its output is monadic.