# A Platform for Secure Distributed Programming

**Eine Plattform für die sichere Programmierung verteilter Systeme**

Master-Thesis von Markus Hauck

Oktober 2015

TECHNISCHE UNIVERSITÄT DARMSTADT

Informatik
Software Engineering

A Platform for Secure Distributed Programming
Eine Plattform für die sichere Programmierung verteilter Systeme

Vorgelegte Master-Thesis von Markus Hauck

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. Guido Salvaneschi

Tag der Einreichung:

# Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 30. September 2015

_____

(Markus Hauck)

# Abstract

Cloud computing provides anywhere, anytime availablility of data and processing power. The opportunity to move computations to the cloud, however, forces developers to face a number of security challenges. This is especially the case for sensitive data i.e., data that even the party running the computation should not be allowed to see. Secure function evaluation, an active field in cryptography, offers a viable solution, but current implementations struggle to raise the level of abstraction from the logic circuit level or force developers to adopt external DSLs that hardly integrate with existing programming languages. Another potential solution is offered by another field in cryptography, using homomorphic encryption schemes to perform operations on encrypted data. While fully homomorphic encryption, which supports all operations without decryption, is prohibitively expensive, implementations based on partial homomorphic encryption schemes have proved to scale up to practical performance in recent implementations. Approaches using partial homomorphic encryption, however, commonly adopt special-purpose languages, e.g., SQL in CryptDB, or external DSLs to describe computations. In summary, as of now, supporting a general purpose language – enabling the reuse of existing libraries, fostering modularity and composition – as well as using a mix of plain and encrypted data is an open research problem.

To fill this gap, we present an embedded DSL in Scala, allowing developers to write computations over encrypted data without requiring sophisticated cryptographic knowledge and without inhibiting the usual abstraction mechanisms. We separate the interface for writing programs in our DSL into an applicative and a monadic part, such that the two can be combined, but the applicative part can be statically analyzed and executed in parallel. Programs can be executed both locally for testing purposes and in the cloud running on untrusted machines. We evaluate our DSL using different case studies, demonstrating the benefits of using an embedded DSL in combination with a general purpose language. Case studies include writing programs in our DSL that run implicitly parallel, integration with the CEP engine Esper to perform queries over encrypted events and the use of the reactive programming framework RxScala, to write a reactive graphical user interface that is based on encrypted data. The results demonstrate the practicability of our approach.

# Zusammenfassung

Cloud Computing bietet immer und überall Verfügbarkeit von Daten und Rechenkapazität. Die Möglichkeit Berechnungen in die Cloud zu verlagern konfrontiert Entwickler mit einer Vielzahl an Herausforderungen bezüglich der Sicherheit. Insbesondere gilt dies für private Daten, die nicht mal der Cloud Computing Provider sehen darf. Sichere Funktionsevaluierung, ein aktuelles Forschungsthema aus der Kryptographie, stellt eine mögliche Lösung dar, doch momentane Implementierungen kämpfen mit einem sehr niedrigen Abstrationsniveau auf Schaltungsebene, oder zwingen Entwickler externe domänenspezifische Sprachen zu verwenden, die kaum mit bestehenden Programmiersprachen integrierbar sind. Ein alternativer vielversprechender Ansatz ist die Verwendung von homomorphen Verschlüsselungsschemata. Während vollständig homomorphe Verschlüsselung momentan noch zu rechenintensiv ist, haben Ansätze basierend auf teilweise homomorpher Verschlüsselung bewiesen, dass sie in der Praxis zu praktikabler Leistung fähig sind. Momentan existierende Ansätze basierend auf homomorpher Verschlüsselung sind allerdings auf spezielle Sprachen spezialisiert, die nicht universell einsetzbar sind, wie zum Beispiel SQL, oder verwenden externen DSLs um die Berechnung zu beschreiben. Zusammenfassend: Sowohl die Verwendung einer universell einsetzbaren Sprache, welche es erlaubt bestehende Bibliotheken zu verwenden und sowohl Modularität als auch Komposition ermöglicht, als auch die Möglichkeit eine Mischung aus unverschlüsselten und verschlüsselten Daten zu verwenden, ist eine offenes Forschungsproblem.

Um diese Lücke zu schließen, präsentieren wir eine eingebettete domänenspezifische Sprache in Scala, welche es Entwicklern ermöglicht Berechnungen auf verschlüsselten Daten zu beschreiben, ohne tieferes Wissen in der Kryptographie zu besitzen. Unsere domänenspezifische Sprache unterscheidet dabei zwischen Teilen, welche auf Monaden beruhen und Teilen, die auf Applikativen Funktoren beruhen, so dass beide kombiniert werden können. Teile, die auf Applikativen Funktoren beruhen haben gegenüber den monadischen Teilen den Vorteil, dass sie statisch analysierbar sind und implizit parallel ausgeführt werden können. Programme, welche in unserer domänenspezifischen Sprache verfasst sind, können sowohl lokal, etwa zu Testzwecken, als auch über das Netzwerk auf entfernten, potenziell unsicheren, Maschinen ausgeführt werden. Wir evaluieren unsere Implementierung anhand von verschiedenen Fallstudien, welche die Vorteile einer eingebetteten domänenspezifischen Sprache in Kombination mit einer universell einsetzbaren Programmiersprache aufzeigen. In den Fallstudien demonstrieren wir das Verfassen von Programmen in unserer domänenspezifischen Sprache und deren implizit parallele Ausführung, verwenden Esper, eine Engine für die Verarbeitung komplexer Ereignisse, um Ereignisse mit verschlüsselten Attributen zu verarbeiten und nutzen RxScala, eine Bibliothek für die Reaktive Programmierung um eine grafische Oberfläche zu verfassen, welche auf verschlüsselten Daten beruht. Die Resultate belegen die Anwendbarkeit unserer Implementierung in der Praxis.

# Contents

# List of Figures

# List of Listings

# 1 Introduction

With the increasing availability and performance of cloud computing together with the ever growing volume of data, it becomes attractive to move expensive computations to a remote server, given that providing the required hardware locally as several drawbacks: the necessary hardware has to be bought, installed and requires continuous maintenance. The upfront costs require that the expected system load and the required performance is known in advance and that upper bounds are known. In scenarios where the demand is not known up front at all or if it can vary drastically over time, keeping the hardware available at all times becomes too much of a burden. Cloud computing promises to solve these issues, offering on-demand provisioning of resources, the possibility of seemingly unlimited scalability and other desirable properties, e.g., fault tolerance, minimal maintenance, cost reduction and others. The program representing the computation and the required data is then uploaded into the cloud and executed there, where resources can be allocated as required for the execution of the computation. While this is a viable approach for non-sensitive and public data, it is not for scenarios where the data required by the computation is meant to be kept private i.e., neither the cloud computing provider, nor potential adversaries should have access to the content. We therefore need a way to perform computations securely on private data.

#### Secure computation.

The first naive attempt to avoid, that a third-party gains access to the sensitive data, is to encrypt it before uploading it into the cloud. The problem is, that the program running on the remote hardware is not allowed to decrypt the data during the execution, otherwise the plain data may leak to potential adversaries, something we wanted to avoid primarily. What is needed in addition to the encryption of sensitive data, is a way to perform our computation over the *encrypted* data, then we can encrypt private data and upload it into the cloud, execute our program and no sensitive data will leak.

Currently the state of the art in cryptography offers two different approaches to solve this problem: secure function evaluation using garbled circuits [Sny14] and homomorphic encryption [Gen09].

Over the last few years, garbled circuits have gained a lot of interest and frequent optimization techniques promise increasingly good performance, however many of the current frameworks lower the abstraction to the Boolean circuit level or require the use of external DSLs. This has the undesirable consequence that programmers are not able to reuse existing technologies and libraries, commonly found in general purpose programming languages and mostly absent for highly specialized external DSLs. The only approach that tries to allow the use of a general purpose language is presented in [HFKV12], but it requires the use of a modified compiler and restricts the language constructs that can be used for writing programs.

The second possible solution to the problem of performing computation over encrypted data is to use homomorphic encryption schemes [Gen09]. The idea is to exploit homomorphic operations to perform computations on cipher texts. The schemes differ in the type of operations they support. Fully homomorphic encryption schemes (FHE) support at least multiplication and addition, from which all other operations can be derived. While in theory fully homomorphic encryption schemes seem to be the ultimate solution to the problem of performing computation over encrypted data, in practice the performance has not (yet) reached practical performance levels [GHS12]. Partially homomorphic encryption (PHE) schemes are less powerful, in most cases supporting only one homomorphic operation on the cipher text. On the other hand, there are many schemes that are partially homomorphic with respect to an operation, or can easily be modified for the purpose, e.g., [Pai99] (addition), ElGamal (multiplication), OPE (ordering), AES (equality). PHE schemes, in contrast to FHE, have been used in practice and proved to offer practical performance [PRZB11, JSSSE14, TLMM13].

In summary, both approaches – garbled circuits and homomorphic encryption schemes – require a dedicated language. The former requires extensive knowledge of circuit theory and how to define them for writing programs which achieve acceptable performance, the latter needs the use of an external DSL, thereby neglecting existing libraries or frameworks and instead requiring developers to reinvent the wheel.

#### Contribution.

In this work, we present an embedded DSL to write programs that work over encrypted data in Scala. The DSL is based on a new combination of a well known approach using free monads [SA07, Swi08] and the more recent research on free applicative functors [CK14]. While intuitively applicative functors are less expressive than monads, the latter suffer from the lack of static analyzability and do not allow implicit parallelism during the traversal of the structure. The resulting DSL allows program authors to write programs in either style and combine them at the end, without losing the benefits of using the free applicative functors interface.

Behind the scenes, we use a set of partial homomorphic encryption schemes, where each supports only a limited set of operations on the cipher text. In order to be able to write real programs, we convert between schemes in a manner that is transparent for program authors. We use the same cryptographic schemes as in [JSSSE14] and also make us of an external trusted service to perform conversions between encryption schemes during program execution.

Programs are evaluated using an interpreter. By writing different versions interpreters, programs can be executed differently, for example locally for testing purposes, over the network on remote hardware in a distributed fashion, or with an optional variety of optimization like, implicit parallelism as well as transformations using information from static analysis.

Specifically, this thesis makes the following contributions:

- We develop an embedded DSL for computations over encrypted data in Scala.
- We contrast a version of the DSL using free monads with another version using free applicative functors.
- We show that the version using free applicative functors allows for implicit parallelism during interpretation and features static analyzability at the cost of expressivity.
- We combine the two version of the DSL into one to get the best of both approaches.

The rest of this work is structured as follows. Chapter 2 provides technical background concerning the Scalaz library, the Kind Projector plugin for type lambda syntax and a short introduction to the use of free monads for writing DSLs. In chapter 3 we slowly build up the components of our final DSL. We start with a simple DSL based on free monads, then contrast it with a version using free applicative functors. At the end, we combine both versions to combine the individual advantages. Chapter 4 compares the developed interpretation styles available for our DSL, demonstrates how to write a word count program and presents different case studies, integrating with other libraries like Esper and RxScala. Benchmarks show a significant overhead, but also that real world usage is still practical. Related work is presented in chapter 5, chapter 6 concludes.

# 2 Background

This chapter provides an overview of the technical background, essential for the rest of the thesis. We introduce the Scalaz library, the Kind Projector compiler plugin for the Scala compiler and demonstrate how to write a simple DSL using free monads in Scala with the functionality provided by Scalaz.

## 2.1 Scalaz

Scalaz[1] is a library for Scala that provides a lot of the functional programming techniques most commonly found in Haskell. For example, we will make heavy use of the type class infrastructure of Scalaz. In general, the hierarchy of type classes found in Scalaz is very similar to the one found in Haskell, but differing in some aspects, e.g., it is more fine-grained: the `Applicative` class only adds the `point` function and is based on the `Apply` class providing the equivalent of Haskell's `(<*>)` which in turn is based on the `Functor` class. The rough idea on how to implement type classes in Scala using implicit arguments is presented in [OMO10].

As an example, listing 2.1 shows a simplified version of the `Functor` type class in both Haskell and Scala. An example `Functor` instances for a simple `Box` data type is shown in listing 2.2. How to write a function that requires the argument to be an instance of the `Functor` type class is shown in listing 2.3. The usage of type classes in Scala essentially boils down to requiring an additional implicit argument, representing the dictionary which is hidden in Haskell. We can then use the dictionary from the implicit argument which provides the implementation of the functions in the type class to perform the desired computation, e.g., call the `map` function from the `Functor` type class. Using the dictionary explicitly leads to a very functional style, but we can make it more object-oriented, by using implicits to extend already defined components [Oa04, sec. 9]. By providing a little bit of extra machinery, shown in listing 2.4, we can call `Box("1e3").map(_.toDouble)` and the implicit class will automatically wrap it with `FunctorOps`, calling the `map` function with the correct arguments. What we derived by hand in the previous listings is provided per default in Scalaz, together with predefined instances for most of Scala's standard library data types like `Option`, `List` etc.

In addition to the type class infrastructure, Scalaz also defines immutable data structures, that are compatible with the available type classes. As an example, Scalaz defines an alternative `collection.immutable.Map` that requires an `Order` instance, while the standard Scala version does not. This fact can lead to subtle errors, e.g., with the standard version you will not get a compiler error, because there are no types that cannot be inserted into a map. When using Scalaz's version however, types without an `Order` instance will be rejected by the compiler if the developer tries to insert values of this type into a Scalaz `Map`.

```scala
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}
```

**(a)** Scala

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

**(b)** Haskell

**Listing 2.1:** The "Functor" type class.

## 2.2 Kind Projector

A well-known pattern to simulate type lambdas in Scala uses a combination of anonymous classes, type members and type projection. Unfortunately, the pattern suffers from syntactic noise, a simple type level version of the identity function is written as:

```scala
({type Lambda[A] = A})#Lambda
```

Kind Projector[2] is a compiler plugin for Scala enabling a much more convenient syntax for type lambdas. As an example, listing 2.5 uses a type lambda to define the `identity` instance of `Pointed`. This is even more convenient than in Haskell,

[1] https://github.com/scalaz/scalaz

[2] https://github.com/non/kind-projector

```scala
case class Box[+A](get: A)
object Box {
  implicit val boxFunctor: Functor[Box] =
    new Functor[Box] {
      def map[A,B](fa: Box[A])(f: A => B) =
        Box(f(fa.get))
    }
}
```

**(a)** Scala

```haskell
data Box a = Box a deriving (Show,Read)
instance Functor Box where
  fmap f (Box x) = Box (f x)
```

**(b)** Haskell

**Listing 2.2:** Defining a functor instance for "Box".

```scala
def useFunctor[F[_]](input: F[String])(
  implicit F: Functor[F]) =
    F.map(input)(_.toDouble)

println(useFunctor(Box("1e3")))
// => Box(1000.0)
```

**(a)** Scala

```haskell
useFunctor :: Functor f => f String -> f Double
useFunctor = fmap read

main = print $ useFunctor (Box "1e3")
-- => Box 1000.0
```

**(b)** Haskell

**Listing 2.3:** Using the functor instance for "Box".

because the latter does not have type lambdas. The trick in Haskell is to use data types with type parameters, simulating the lambda, at the cost of some syntactic overhead due to wrapping/unwrapping. A simple use case where we need type lambdas is when defining instances for type classes that require a higher kinded type like `Functor`. If we want to make Scala's `Either` an instance of it, we face the problem that `Either` takes *two* type arguments not one. In Haskell we can make use of the fact that the type arguments are curried to partially apply `Either`:

```haskell
instance Functor (Either a) where ...
```

But in Scala this is not possible and without kind projector we have to either create an one-off type alias or use the inconvenient and verbose type lambda syntax shown in listing 2.6. With kind projector we can use function syntax for type level lambdas or the even more convenient shorthand "?", which behaves much like the underscore in Scala to define anonymous functions, but at the type level. The two available alternatives are shown in listing 2.7.

## 2.3 Writing DSLs using Free Monads

This section serves as a simple example on how to write embedded DSLs using a functor in combination with the free monad. The purpose is to demonstrate the technique later used in chapter 3 in a simpler form.

Every functor `f` gives rise to a monad, the so called free monad on `f`. The naive definition of a free monad together with the instance for the `Monad` type class is shown in listing 2.8. This implementation however suffers from a severe performance problem. For more details of the specifics we refer interested readers to [Voi08] and/or [vdPK14], in the following we will ignore this fact and note that we use the implementation provided by Scalaz, which is already optimized for Scala and does not suffer from the performance problems of the naive definition in listing 2.8. As our example DSL, we will in the following use the `Teletype` example from [SA07] written in Haskell, but instead of building the monad structure right in (the `Return` case), we use Scalaz's free monad implementation, called `Free`.

Listing 2.9 defines a trait `Teletype` together with two case classes that extend it, `GetChar` and `PutChar`. We provide the functor instance in the companion object for `Teletype` and also add two smart constructors, one for each case class, to hide the required lifting into `Free`. With this code in place we can write programs using the teletype DSL, for example a simple `echo` program or programs to print strings, `putStr` and `putStrLn` shown in listing 2.10.

Programs like `echo` represent a `Teletype` program as a value, waiting to be executed. The next step is to write an interpreter. A simple interpreter is shown in listing 2.11, which uses the standard `System.in.read` and `print` methods from the Scala standard library to interpret `GetChar` and `PutChar`, respectively. It is important to note that we use, from the Haskell perspective, "unsafe" IO functions i.e., they are not tracked by the type system. While Scala features a very

```scala
implicit class FunctorOps[F[_]:Functor,A](val self: F[A]) {
  def map[B](f: A => B): F[B] =
    implicitly[Functor[F]].map(self)(f)
}

println(Box("1e3").map(_.toDouble))
// => Box(1000.0)
```

**Listing 2.4:** More convenient syntax for "Functor".

```scala
trait Pointed[F[_]] {
  def point[A](x: A): F[A]
}

val option = new Pointed[Option] {
  def point[A](x: A) = Option(x)
}

val identity = new Pointed[λ[α=>α]] { // type lambda
  def point[A](x: A) = x
}

println(identity.point(5)) // => 5
println(option.point(5))   // => Some(5)
```

**Listing 2.5:** Using type lambda syntax from kind projector.

expressive type system, it does not enforce purity by default in the same way e.g., Haskell does, though Scalaz provides wrappers to manually declare e.g., `IO` actions.

```scala
def eitherFunctor1[C] = {
  // type alias
  type EitherC[X] = Either[C,X]
  new Functor[EitherC] {
    def map[A,B](fa: Either[C,A])(f: A => B) =
      ???
  }
}
```

**(a)** One-off type alias

```scala
def eitherFunctor2[C] = {
  // no type alias but verbose syntax
  new Functor[({type λ[α] = Either[C,α]})#λ] {
    def map[A,B](fa: Either[C,A])(f: A => B) =
      ???
  }
}
```

**(b)** Verbose type lambda

**Listing 2.6:** Functor instance for "Either".

```scala
def eitherFunctor3[C] = {
  // explicit function syntax
  new Functor[λ[α=>Either[C,α]]] {
    def map[A,B](fa: Either[C,A])(f: A => B) =
      ???
  }
}
```

**(a)** Anonymous function syntax

```scala
def eitherFunctor4[C] = {
  // anonymous lambda via '?' shorthand
  new Functor[Either[C,?]] {
    def map[A,B](fa: Either[C,A])(f: A => B) =
      ???
  }
}
```

**(b)** Shorthand with '?'

**Listing 2.7:** Functor instance for "Either" using kind projector.

```haskell
data Free f a = Pure a | Free (f (Free f a))

instance Functor f => Monad (Free f) where
  return = Pure
  Pure a >>= f = f a
  Free m >>= f = Free (fmap (>>= f) m)
```

**Listing 2.8:** Naive definition of free monads in Haskell.

```scala
sealed trait TeletypeF[+A]
case class GetChar[+A](k: Char => A) extends TeletypeF[A]
case class PutChar[+A](char: Char, k: A) extends TeletypeF[A]

type Teletype[A] = Free[TeletypeF,A]

// Functor instance for TeletypeF
object TeletypeF {
  implicit val functor: Functor[TeletypeF] = new Functor[TeletypeF] {
    def map[A,B](fa: TeletypeF[A])(f: A => B): TeletypeF[B] = fa match {
      case GetChar(k) => GetChar(f compose k)
      case PutChar(char,k) => PutChar(char,f(k))
    }
  }
}

def getChar: Teletype[Char] = Free.liftF(GetChar(identity))
def putChar(char: Char): Teletype[Unit] = Free.liftF(PutChar(char,()))
```

**Listing 2.9:** The "Teletype" functor.

```scala
def echo: Teletype[Unit] = for {
  c <- getChar
  _ <- putChar(c)
  r <- echo
} yield r

def putStr(s: String): Teletype[Unit] = s.toList.traverse_(putChar(_))
def putStrLn(s: String): Teletype[Unit] = putStr(s) >> putChar('\n')
```

**Listing 2.10:** Writing programs with Teletype.

```scala
def runTeletype[A](tt: Teletype[A]): A = tt.resume match {
  case -\/(GetChar(k)) => runTeletype(k(System.in.read.toChar))
  case -\/(PutChar(char,k)) =>
    print(char)
    runTeletype(k)
  case \/-(x) => x
}
```

**Listing 2.11:** One possible way wo interpret “Teletype” programs.

# 3 Design and Implementation

In this chapter we describe the implementation of our embedded DSL for writing programs working with encrypted data. After a short overview of our overall architecture, we introduce the encryption schemes. The next section starts with a simple implementation of our DSL using free monads. Afterwards we implement the same simplified DSL using free applicative functors and contrast it with the previous version. Different interpretation styles are presented, as well as how to extend the DSL with additional operations, even if no encryption scheme supports them. We close the chapter with a discussion about the separation between the monadic and applicative DSL, as well as an example of a typical program execution on a remote machine.

## 3.1 Overview

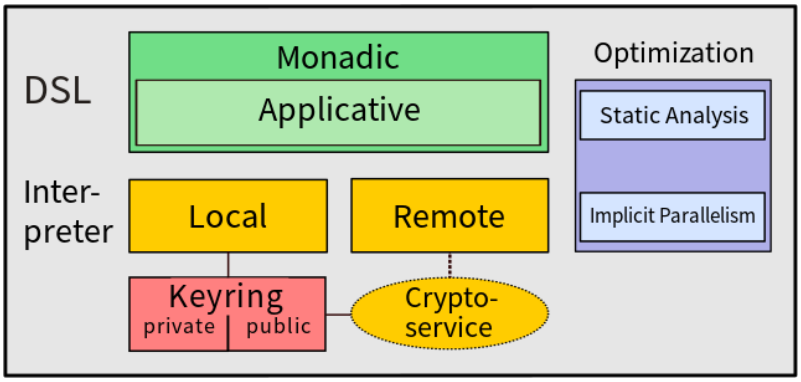



**Figure 3.1.:** Overview of the System.

Figure 3.1 displays an overview of our design. The core of our implementation is the DSL, which allows developers to write programs working with encrypted data. Programs written in the DSL are executed using interpreters, either on the local machine with direct access to the encryption keys or in the cloud, communicating with a trusted cryptographic service handling conversion requests. By using the applicative interface of the DSL, we can statically analyze and transform programs, as well as exploit implicit parallelism during interpretation.

## 3.2 Encryption Schemes

The encryption schemes providing homomorphic operations used in our implementation are essentially the same as in [PRZB11] and [JSSSE14], a summary is shown in table 3.1.

| Scheme | Operation | Types | Asymmetric |
|---|---|---|---|
| Paillier | Addition | Integers | yes |
| ElGamal | Multiplication | Integers | yes |
| AES | Equality | Integers, Strings | no |
| OPE | Ordering | Integers, Strings | no |

**Table 3.1.:** Overview of the encryption schemes used in the implementation.

In the DSL, we have two different types for integers and strings respectively. For each of them we distinguish the concrete scheme type. A slightly simplified representation of encrypted numbers is shown in listing 3.1.

`EncInt` represents an encrypted integer. The four case classes each represent an integer encrypted with a specific encryption scheme. The knowledge of the concrete encryption scheme also allows the use of the respective homomorphic operation, e.g., addition for `PaillierEnc`, given that both operands are of the `PaillierEnc` type. With this knowledge we can write a simple function that tries to add two encrypted integers without knowing the specific scheme by using pattern matching, shown in listing 3.2.

```scala
sealed trait EncInt
case class PaillierEnc(val underlying: BigInt) extends EncInt
case class ElGamalEnc(val ca: BigInt, val cb: BigInt) extends EncInt
case class AesEnc(val underlying: Array[Byte]) extends EncInt
case class OpeEnc(val underlying: BigInt) extends EncInt
```

**Listing 3.1:** Representing encrypted integers.

```scala
def addition(lhs: EncInt, rhs: EncInt): Option[EncInt] = (lhs,rhs) match {
  case (PaillierEnc(l),PaillierEnc(r)) => Some(PaillierEnc(???))
  case (ElGamalEnc(l1,l2),ElGamalEnc(r1,r2)) => None
  case (AesEnc(l),AesEnc(r)) => None
  case (OpeEnc(l),OpeEnc(r)) => None
  case _ => None
}
```

**Listing 3.2:** Addition of encrypted integers.

The first `case` is easy, we just replace the "???" with the operation that uses the homomorphic property of the Paillier encryption scheme. More problematic are the cases where one or more operands are not encrypted under the correct encryption scheme required for the operation. Then we need to convert the operands into the proper scheme, in our case for `addition`, both operands have to be encrypted under the Paillier scheme.

```scala
def ensurePaillier(i: EncInt): PaillierEnc = ???
```

**Listing 3.3:** Forcing a specific underlying encryption scheme.

In theory, we could use a function as shown in listing 3.3 and simply call it for both operands in `add`. The function `ensurePaillier` is then responsible for converting from the `EncInt` type, which admits integers encrypted under an arbitrary scheme, to the `PaillierEnc` type, if necessary. The problem with this approach is that functions like `ensurePaillier` also need access to the public and *private* encryption key, because it has to perform one decryption and one encryption operation in order to re-encrypt the value under the desired scheme. In our setting from chapter 1 however, this might expose our private data, because the untrusted party providing the hardware to run our computation might suddenly be able to observe plain data during the computation. What we actually want, is a way to write programs without worrying about the fact that we might require access to encryption/decryption keys, what specific scheme we need for the computation at hand and how the conversion is handled, such that our data is secure. With this background, we take a step back and first think about a possible signature for an operation that adds two encrypted integers without assuming a specific encryption scheme.

Listing 3.4 shows three different signatures for a function that adds two encrypted integers. If we want to let the computation fail, because we cannot convert between encryption schemes at all, we want the signature from line 1, which has the return type `Option[EncInt]` to model failure, on the other hand if we want to talk to a remote service we might want a signature that looks like the one shown in line 2, modeling the network communication and returning a `Future[EncInt]`. In another case we might want to just test the computation locally where we have all required keys and can encrypt and decrypt at will, then the signature should look as shown in line 3, simply returning a value of `EncInt`.

Using Scala's support for higher-kinded types, we can abstract over all of those possible types by adding a higher kinded type parameter. Additionally this is a typical problem where we want to model effects explicitly in our types, a problem that is optimal suited for the use of monads.

## 3.3 The Monadic DSL

In this section, we implement a simplified version of our DSL using free monads and design interpreters that execute our programs. As an introduction to writing DSLs using free monads with Scalaz, we refer to section 2.3. This section builds

```scala
def addOption(lhs: EncInt, rhs: EncInt): Option[EncInt] = ??? // Failure
def addFuture(lhs: EncInt, rhs: EncInt): Future[EncInt] = ??? // Async
def addPure(lhs: EncInt, rhs: EncInt): EncInt = ???           // Pure
```

**Listing 3.4:** Possible signatures for addition.

up the monadic DSL for our cryptographic framework, which enables developers to write secure computations. We start with an implementation using free monads.

### 3.3.1 Free Monads

Analogous to the `Teletype` DSL from section 2.3, we start with a base functor, which defines the possible operations inside of our DSL, re-using the definition of `EncInt` from section 3.2. We define a new type `CryptoF` and add cases for every operation we want to support i.e., addition (`Plus`), multiplication (`Mult`), equality (`Equals`) and comparisons (`Compare`), as shown in listing 3.5.

```scala
sealed trait CryptoF[+K]
case class Plus[K](lhs: EncInt, rhs: EncInt, k: PaillierEnc => K) extends CryptoF[K]
case class Mult[K](lhs: EncInt, rhs: EncInt, k: ElGamalEnc => K) extends CryptoF[K]
case class Equals[K](lhs: EncInt, rhs: EncInt, k: Boolean => K) extends CryptoF[K]
case class Compare[K](lhs: EncInt, rhs: EncInt, k: Ordering => K) extends CryptoF[K]
```

**Listing 3.5:** A simplified version of the "CryptoF" functor.

```scala
case class Plus[K](lhs:EncInt, rhs:EncInt, k:PaillierEnc => K) extends CryptoF[K]
def add(lhs:EncInt, rhs:EncInt): Crypto[EncInt] = ???
```

**Listing 3.6:** The "Plus" case.

Instead of discussing each case in full detail, in the following we limit ourselves to the `Plus` case and the corresponding smart constructor, the other cases are analogous and therefore left out at this point. Listing 3.6 shows only the `Plus` case and the corresponding smart constructor `add`. The constructor takes two operands, the left- and right-hand operands and an additional argument, a function from `PaillierEnc` to `K`. This last argument is the rest of the program i.e., a continuation which will *receive* an argument of type `PaillierEnc`, representing the result of the interpretation of the `Plus` operation given both operands. Instead of `PaillierEnc` we could have returned `EncInt`, but the additional refinement turns out to be useful when writing programs later. As a note aside, in the `Compare` case the `Ordering` is *not* the type class from `scala.math.Ordering`, it is a simple algebraic data type (or at least the Scala equivalent way to encode them) from Scalaz with three cases: `LT`,`EQ`,`GT`. We also have to write an instance for the `Functor` type class for `CryptoF`, which we omit because it is trivial to derive and mechanical to implement. The different classes representing operations are not exposed directly to the DSL users, instead we only expose the smart constructors as listed in listing 3.7 i.e., one smart constructor for each possible case of `CryptoF`.

Based on `CryptoF`, we define a convenient type alias for the free monad arising from it:

```scala
type Crypto[A] = Free[CryptoF,A]
```

### 3.3.2 Writing Programs

With our minimal DSL from the previous section, we can write first small programs, for now we ignore how to create encrypted integers like `one`, this will be discussed later. The program in listing 3.8 defines a function `plus1` that increments

```
def add(lhs: EncInt, rhs: EncInt): Crypto[EncInt] = Free.liftF(Plus(lhs,rhs,identity))
def multiply(lhs: EncInt, rhs: EncInt): Crypto[EncInt] = Free.liftF(Mult(lhs,rhs,identity))
def equal(lhs: EncInt, rhs: EncInt): Crypto[Boolean] = Free.liftF(Equals(lhs,rhs,identity))
def compare(lhs: EncInt, rhs: EncInt): Crypto[Ordering] = Free.liftF(Compare(lhs,rhs,identity))
```

**Listing 3.7:** Smart constructors perform the lifting.

```
lazy val one: EncInt = ???
def plus1(in: EncInt): Crypto[EncInt] = add(in,one)
```

**Listing 3.8:** A simple program.

the given encrypted integer. The result type is `Crypto[EncInt]`, which means that given an `EncInt`, running the program will result in a value of type `EncInt`. For the reason that we use a free monad, we can also make use of monadic combinators, as well as Scala's for-comprehensions, providing syntactic sugar. Listing 3.9 shows a larger example, making use of three different DSL primitives to calculate a result which involves addition, multiplication and equality. The result is a program that will yield a value of type `Boolean` after interpretation.

```
lazy val two: EncInt = ???
lazy val fortyTwo: EncInt = ???

// (in + 1) * 2 == 42
def isAnswer(in: EncInt): Crypto[Boolean] = for {
  x1 <- plus1(in)
  x2 <- multiply(x1,two)
  x3 <- equal(x2,fortyTwo)
} yield x3
```

**Listing 3.9:** Using Scala's for-comprehensions.

### 3.3.3 The KeyRing

Before we look at how to execute our programs, we need to first explain the interface to our encryption scheme public and private keys.

In general we have one *secret* key for each **symmetric** encryption scheme and a public/private key pair for each *asymmetric* encryption scheme. As listed in table 3.1 we have two schemes that are asymmetric and two schemes that are symmetric. Whether a scheme is asymmetric or symmetric is an important point. If it is asymmetric, that means we can encrypt plain data on the untrusted machine using the public key without revealing any private information, if it is symmetric on the other hand, we can under no circumstance distribute our secret key and therefore we cannot perform encryption operations on the untrusted machine. If we cannot encrypt plain values under a specific scheme, this means we have to provide encrypted versions for all required values (constants like e.g., 0,1,etc.) in advance.

The `KeyRing` class encapsulates encryption keys for each scheme, essentially it consists of two components, `PubKey`s for each asymmetric encryption scheme, as well as `PrivKey`s for every scheme. The default key ring can be obtained calling `create` on the `KeyRing` companion object. By default this uses 1024b for Paillier and ElGamal, 256b for AES and 128b for OPE. If we extend our DSL with a new operation to encrypt values as shown in listing 3.10, we are able to write a version of the `isAnswer` function that does not depend on encrypted constants to be available in advance and does instead encrypt the desired constants by itself, prior to the computation, as shown in listing 3.11.

### 3.3.4 Local Program Execution

Looking at the result types of the programs from the previous sections, we see that it is always of type `Crypto[A]`, for some type `A`, where `A` is the type we would normally expect, e.g., `Int`, `Double` etc. This is because the example programs

```scala
sealed trait Scheme
case object Paillier extends Scheme
case object ElGamal extends Scheme

case class Encrypt[K](s: Scheme, plain: Int, k: EncInt => K) extends CryptoF[K]

def encrypt(s: Scheme)(plain: Int): Crypto[EncInt] = Free.liftF(Encrypt(s,plain,identity))
```

**Listing 3.10:** Extending the DSL with encryption.

```scala
def isAnswer_(in: EncInt): Crypto[Boolean] = for {
  one <- encrypt(Paillier)(1)
  two <- encrypt(ElGamal)(2)
  fortyTwo <- encrypt(Paillier)(42)
  x1 <- add(in,one)
  x2 <- multiply(x1,two)
  x3 <- equal(x2,fortyTwo)
} yield x3
```

**Listing 3.11:** An improved version of "isAnswer".

are only a *description* of a program i.e., it is either a computation in suspension represented by our functor or it ends with a concrete result, representing the final value of the computation due to the definition of a free monad.

To actually run our program descriptions, we have to write an interpreter for `Crypto` programs, in essence the scheme is the same as in section 2.3 to run `Teletype` programs. We start with an interpretation that has access to *both private and public* keys. The following sections will expand on this by using an external cryptographic service to model execution on remote hardware without requiring access to the private keys. Listing 3.12 shows an interpreter for the `Plus` and `Encrypt` case of `CryptoF`, the other cases are left out but straightforward, following the same principles and differing only in the concrete encryption scheme required for the operation at hand. We assume two helper functions:

- `convertToPaillier`, which converts between `EncInt` and `PaillierEnc` essentially by performing a decryption followed by an encryption operation
- `asymEncrypt`, which is an internal function to encrypt under the desired encryption scheme. `asymEncrypt` only needs access to *public keys*, because it supports only the two asymmetric schemes which allow encryption via the respective public key.

With these two helper functions in place, we can interpret the `Encrypt` case of our DSL simply by calling `asymEncrypt` and passing the result into the continuation `k` of the current suspension. Interpreting `Plus` is straightforward if both operators are already encrypted under the Paillier encryption scheme: we can simply use the "+" defined on `PaillierEnc`, which exploits the homomorphic property. If at least one of the operators is not encrypted under the correct encryption scheme, we have to call the `convertToPaillier` helper function to perform the conversion, which requires access to the private keys, after that we can again exploit the homomorphic property.

Because of the assumption that access to both public and private keys is given, this interpretation style is not suitable for execution in the cloud, where private keys will not be available. Nevertheless, the interpreter is still useful, because we might want to write tests for our programs to assure that they work correctly and those tests are obviously not meant to be run on the remote hardware in the cloud, instead we want to run it locally on a trusted hardware. For this use case, the above interpreter fits perfectly, because access to private keys is not a critical issue.

### 3.3.5 Remote Program Execution

In section 3.3.4 we demonstrate how to interpret programs locally. As stated earlier this might be convenient to test programs before running them in the cloud, but it does not solve our original problem of secure computation over encrypted data on untrusted remote machines. In this section we will develop another style of interpretation, which will not require access to the private keys and is therefore suitable to be run by an untrustworthy third party. Because all of

```scala
def convertToPaillier(keyRing: KeyRing)(enc: EncInt): PaillierEnc = ???
def asymEncrypt(scheme: Scheme, keyRing: PubKeys)(plain: BigInt): EncInt = ???

def interpret[A](keyRing: KeyRing)(program: Crypto[A]): A = program.resume match {
  case \/-(x) => x
  case -\/(Encrypt(s,plain,k)) => interpret(keyRing)(k(asymEncrypt(s,keyRing.pub)(plain)))
  case -\/(Plus(l,r,k)) => (l,r) match {
    case (lhs@PaillierEnc(_),rhs@PaillierEnc(_)) => interpret(keyRing)(k(lhs + rhs))
    case _ =>
      val lhs = convertToPaillier(keyRing)(l)
      val rhs = convertToPaillier(keyRing)(r)
      interpret(keyRing)(k(lhs + rhs))
  }
  case _ => sys.error("Unhandled case")
}
```

**Listing 3.12:** Interpreting encryption and addition.

the semantics of our DSL programs are defined by the interpreter, we can add a new interpretation without changing anything inside the DSL functor `CryptoF`.

```scala
def remote[A](pub: PubKeys, srv: CryptoService)(program: Crypto[A])(
  implicit ec: ExecutionContext): Future[A] = program.resume match {

  case \/-(x) => Future.successful(x)
  case -\/(Encrypt(s,plain,k)) => remote(pub,srv)(k(asymEncrypt(s,pub)(plain)))
  case -\/(Plus(l,r,k)) => for {
    lhs <- srv.toPaillier(l)
    rhs <- srv.toPaillier(r)
    res <- remote(pub,srv)(k(lhs + rhs))
  } yield res
  case _ => sys.error("Unhandled case.")
}
```

**Listing 3.13:** A remote interpreter.

The first thing to notice in the implementation of a remote interpreter in listing 3.13, is that `remote` will return a `Future[A]` for a program with result `A`. This is because we internally use `Future`s to model network communication to the external `CryptoService`. The interpretation of `Encrypt` is analogous to the local interpretation, because as said above we can encrypt under an asymmetric encryption schemes using only the public keys, which are also available on the untrusted third party's machine. The `Plus` case is more interesting this time: without access to the private keys, we are not able to simply re-encrypt values under the necessary scheme. Instead we delegate the conversion to the additional argument of `remote`, `srv` with type `CryptoService`. The `CryptoService` handles the network communication with a trusted external party, sending requests for the re-encryption of values under a desired encryption scheme. Ignoring the `CryptoService` for a moment, the interpretation of the `Plus` case in the remote interpreter looks quite similar to the one defined previously for local execution. The difference is that here we use Scala's for-comprehension, because every call of `toPaillier` results in a `Future[PaillierEnc]`. Apart from this difference we again call a function to make sure that the operands have the correct encryption scheme and then pass the result of the addition into the continuation and continue with the interpretation.

The `CryptoService`'s task is to act as a trusted access point for the program running on the untrusted party's hardware in order to perform conversion between encryption schemes. Therefore it has to provide an interface as shown in listing 3.14, defining methods to convert from `EncInt` to each of the concrete cases with known encryption scheme e.g., `PaillierEnc` for Paillier and `ElGamalEnc` for ElGamal. If the interpreter requests a conversion between two encryption schemes, an concrete implementation of `CryptoService` has to know how to talk to the trusted server that is able to

```scala
trait CryptoService {
  def toPaillier(enc: EncInt): Future[PaillierEnc] = ???
  def toElGamal(enc: EncInt): Future[ElGamalEnc] = ???
  // ...
}
```

**Listing 3.14:** The CryptoService.

re-encrypt values. An example implementation could do this in an RPC style, though this is not fixed and e.g., a REST API would be possible as well.

### 3.3.6 Using Monadic Combinators

We already used Scala's for-comprehension to get syntactic sugar in listing 3.13. For-comprehensions in essence translate to underlying function calls to `flatMap` (`(>>=)`), `map` (`fmap`), `filter` etc. However, there are even more functions we can use in our DSL simply because of the free monad structure. In Haskell and via Scalaz also in Scala, there is another type class called **`Foldable`**, which is described rather short in Haskell's **`Data.Foldable`** module as: "Data structures that can be folded." We can use the fact that Scala's **`List`** type is an instance of **`Foldable`** and therefore use *monadic* folds provided by the **`Foldable`** type class to sum a list of encrypted numbers, shown in listing 3.15.

```scala
def sumList(zero: EncInt)(xs: List[EncInt]): Crypto[EncInt] =
  xs.foldLeftM(zero)(add(_,_))
```

**Listing 3.15:** The "sum" program using a monadic fold.

In this example we decided to pass in an encrypted version of `0`, although `sumList` could request an encrypted version by itself as shown in a previous example. Under real conditions, however it is obviously better to request encrypted version of constants like `0`,`1`, which are used repeatedly all over the computation, only once and subsequently pass it to the functions needing them. Back to the concrete example, we claim that any programmer familiar with monads can write programs like this, without having to worry about the required conversions between the underlying encryption schemes. Other functions we can use for writing programs in our DSL without doing any work include: `findM` to find elements based on a monadic predicate, `filterM` the monadic version of the well-known `filter` function, `takeWhileM`, `partitionM` etc. all predefined in Scalaz.

```scala
def randomEnc: EncInt = ???
lazy val result = sumList(zero)(List.fill(100)(randomEnc))
```

**Listing 3.16:** Example program using the monadic DSL.

In summary, up until this point we defined a small embedded DSL in Scala for writing programs working with encrypted data. Due to the free monad structure, we are able to use monadic combinators like monadic folds to get a lot of predefined functionality essentially for free and we can interpret the resulting program in different ways: locally for testing or on remote hardware in the cloud. While it seems like we only benefit from the free monad structure arising from our **`CryptoF`** functor, there is a major catch. When writing a program as shown in listing 3.16, the *whole* execution of the program is done *sequentially* i.e., look at the head of the list, if the underlying scheme is not **`Paillier`** convert it, add it to the accumulator of the fold (`zero` initially) and repeat for the tail of the list. While this does give the right answer, it exhibits horrible performance, especially when we take into account that this problem of summing up a list of numbers is, in fact, embarrassingly parallel. The issue though is not within the interpretation function, instead it is an inherent trait of the monadic approach: using the monadic bind operator we can depend on previous monadic values, but this is inherently *sequential* and there is nothing we can do about this fact by changing the interpreter, we have to change the DSL. On top of the forced sequential execution, another disadvantage of the free monad DSL design is that we cannot inspect **`Crypto`** programs *without* executing them.

To see where this limitation comes from, we try to statically analyze the program from listing 3.15. The `result` value from listing 3.16 has the type `Crypto[EncInt]`, which is a type alias for `Free[CryptoF,EncInt]`. The `Free` type has two alternatives, it can either be a `Return`, returning a value of type `EncInt`, or it can be a `Suspend`, which represents a suspended computation of `Free[CryptoF,EncInt]` wrapped inside a functor. For the `sumList` program, we know that the outermost layer of our program will be a `Suspend`, because `Return` only occurs at the end of the program. More specifically we also know that the suspension functor, which is of type `CryptoF`, is the `Plus` case (listing 3.5). The `Plus` case class has three members, the two operands for the operation and as a third argument the continuation which receives the result of the operation. Inspecting the outermost layer of the program is no problem, but after that we cannot proceed further, without passing the result to the continuation of `Plus`. Therefore we have no choice, but to perform the actual computation, in this case the addition of the two operands, before we can proceed to the next instruction of our program. We can try to pass stub values into the continuation e.g., by always using the first operand and not perform any addition, but this is not a viable approach, because the remaining program might choose alternative execution paths depending on the value of the addition and therefore we would execute different programs depending on the fact, whether we perform the addition correctly, or supply a stub value, rendering the results of our inspection useless.

## 3.4 The Applicative DSL

This section implements a different version of our DSL. While in section 3.3 we use free monads, now we switch to free applicative functors. This gives us the ability to perform static analysis and execute programs implicitly parallel.

### 3.4.1 From Free Monads to Free Applicative Functors

In the last section we discovered the major limitation of our approach using free monads i.e., the inability to exploit parallelism and the inability to statically analyze `Crypto` programs without executing them.

Given that the limitation stems from the fact that we use a free monad, it becomes clear that this is the part we have to replace. Luckily the `Applicative` type class is exactly what we want. In essence, applicative functors make it easy to lift *pure* functions such that they accept *effectful* arguments. In contrast to monads however, this does not allow dependencies on previous effectful computations, but we gain the flexibility to evaluate those effects in parallel if we want. Therefore we have to restructure our DSL and use free applicative functors instead of free monads. In [CK14], Capriotti and Kaposi define a free applicative functor and demonstrate their usage to write DSLs that can be analyzed statically. By virtue of using Scala, we can use the version defined in Scalaz, called `FreeAp`. We can completely reuse our previously defined functor `CryptoF`. Instead of providing smart constructors that lift into the *free monad* arising from `CryptoF`, we now change those to lift into the *free applicative functor* instead, an example of the necessary change is shown in listing 3.17, i.e., replacing `Free` with `FreeAp` and `Free.liftF` with `FreeAp.lift`.

```
1  type Crypto[K] = Free[CryptoF,K]
2  def add(lhs: EncInt, rhs: EncInt):
3      Crypto[EncInt] =
4        Free.liftF(Plus(lhs,rhs,identity))
```

**(a)** free monad

```
1  type Crypto[K] = FreeAp[CryptoF,K]
2  def add(lhs: EncInt, rhs: EncInt):
3      Crypto[EncInt] =
4        FreeAp.lift(Plus(lhs,rhs,identity))
```

**(b)** free applicative functor

**Listing 3.17:** Lifting for free monads and free applicative functors.

We do this for each smart constructor, and change the `Crypto` type alias such that it uses the free applicative `FreeAp` instead of the free monad `Free`. The old `Crypto` type alias is renamed to `CryptoM` and will be adapted slightly in a later section. With those changes we are almost done changing the structure of our DSL, though we still have to adapt our interpreters.

### 3.4.2 Interpreting Free Applicative Functors

We start with an overview of the changes that occur by switching from free monads to free applicative functors. First of all, we can no longer use for-comprehensions to write programs, because we cannot define a monadic bind (`flatMap`), this limitation will be addressed in later sections. Secondly, we have to change our interpretation functions. The required adaptions to go from a local interpretation style to a remote are the same, therefore we only show an example for local interpretation limited to the `Encrypt` and `Plus` cases in listing 3.18.

```scala
def interpret[A](keyRing: KeyRing)(program: Crypto[A]): A = {
  program.foldMap(new (CryptoF ~> Id) {
    def apply[B](fa: CryptoF[B]): B = fa match {
      case Encrypt(s,plain,k) => k(asymEncrypt(s,keyRing.pub)(plain)).point(Id.id)
      case Plus(l,r,k) => (l,r) match {
        case (lhs@PaillierEnc(_),rhs@PaillierEnc(_)) => k(lhs + rhs)
        case _ =>
          val lhs = convertToPaillier(keyRing)(l)
          val rhs = convertToPaillier(keyRing)(r)
          k(lhs + rhs).point(Id.id)
      }
      case _ => sys.error("Unhandled case.")
    }
  })
}
```

**Listing 3.18:** Interpretation with free applicative functors.

Scalaz's `FreeAp` does not expose the concrete constructors and instead provides a function `foldMap` that encapsulates the recursion scheme for us. We pass as argument to `foldMap` an anonymous instance for the `~>[CryptoF,Id]` class, which can be written infix as `CryptoF ~> Id`. In essence, it is an encoding of a polymorphic function value, originally developed by Miles Sabin in shapeless[1] and here used to represent a natural transformation between `CryptoF` and our target semantics. We override the `apply` method and match on the concrete cases of our by know well-known `CryptoF` type. If we compare this version to the one presented in section 3.3.4, we can see that both versions are almost identical, modulo the fact that `foldMap` handles the recursion for us and the explicit usage of `Id`'s `Applicative` instance. Despite the marginal change on the surface, we can now execute programs implicitly parallel, given that the `Applicative` instance is defined to do it. This is already the case for `Future`, which means that e.g., for the `Plus` case we get implicit parallelism for free, converting both operands in parallel.

### 3.4.3 Using the Applicative DSL

On account of the fact that we are using the free *applicative* functor arising from `CryptoF`, we can no longer use Scala's for-comprehensions, which require (in addition to others) the `flatMap`, the *monadic* bind, to be defined. Here it becomes obvious what it means that we are no longer able to depend on effectful previous values: programs like `isAnswer` from section section 3.3.2 can no longer be expressed using our DSL. We will remedy this limitation later, but first let's have a look at programs that we can express. In both Haskell and, by using Scalaz in Scala, there is a type class called `Traversable` (Haskell) / `Traverse` (Scala). In Scala it is not to be mistaken for the `scala.collection.Traversable` traits which clashes with the Haskell name. In the following we will use `Traversable` and `Traverse` interchangeably, ignoring the trait from `scala.collection` and assuming no name conflict. Where `Foldable` is for data structures that can only be folded, `Traversable` represents data structures that can be traversed from left to right while performing an action on each element.

Before we turn to an alternative version of the previously defined `sumList` function, we expand our DSL with explicit conversion commands, one for each encryption scheme as in listing 3.20. Using `traverse` from the `Traversable` type class, we can write `sumList` as shown in listing 3.19. Instead of using a monadic fold, we traverse the list with an effectful operation, a conversion to the `PaillierEnc`, such that subsequently we can make use of the fact that `Crypto` is also a functor. This allows us to call `map` and sum up the list of numbers, now with type `List[PaillierEnc]`, simply by exploiting the homomorphic property of the Paillier encryption scheme. The resulting program to sum a list of encrypted numbers is semantically equivalent to the previous definition using a monadic fold, but it now admits implicit parallelism during interpretation, as well as static analysis. In fact, our DSL based on free applicative functors *guarantees* that there are no computations depending on previous effectful values. The advantage is that all programs using the DSL based on free applicative functors can be statically analyzed.

[1] https://github.com/milessabin/shapeless/

```scala
def sumList(zero: PaillierEnc)(xs: List[EncInt]): Crypto[PaillierEnc] =
  xs.traverse(toPaillier).map(_.foldLeft(zero)(_+_))
```

**Listing 3.19:** Sum a list using traverse.

```scala
case class ToPaillier[K](v: EncInt, k: PaillierEnc => K) extends CryptoF[K]
def toPaillier(v: EncInt): Crypto[PaillierEnc] = FreeAp.lift(ToPaillier(v,identity))

case class ToGamal[K](v: EncInt, k: ElGamalEnc => K) extends CryptoF[K]
def toGamal(v: EncInt): Crypto[ElGamalEnc] = FreeAp.lift(ToGamal(v,identity))

case class ToAes[K](v: EncInt, k: AesEnc => K) extends CryptoF[K]
def toAes(v: EncInt): Crypto[AesEnc] = FreeAp.lift(ToAes(v,identity))

case class ToOpe[K](v: EncInt, k: OpeEnc => K) extends CryptoF[K]
def toOpe(v: EncInt): Crypto[OpeEnc] = FreeAp.lift(ToOpe(v,identity))
```

**Listing 3.20:** Extending the DSL with explicit conversions.

### 3.4.4 Static Analysis

As claimed earlier, by using free applicative functors, we gain the ability to perform static analysis. This is possible, because an applicative program can always be modelled as a *pure* function that receives a number of arguments, represented by *independent* effectful computations, thereby allowing us to look at each effectful argument without running the program. `FreeAp` provides two functions that make it easy to express the desired analysis, the first is the already mentioned `foldMap`, which gives the program the semantics of a provided applicative functor and `analyze`, which "performs a monoidal analysis". As an example, we can count the required number of conversions of a program in advance *without interpreting* it, shown in listing 3.21 by using the monoid instance that maps the binary operation to addition and the identity element to `0`.

```scala
def requiredConversions[A](p: Crypto[A]): Int = {
  p.analyze(new (CryptoF ~> λ[α => Int]) {
    def apply[B](fa: CryptoF[B]): Int = fa match {
      case ToPaillier(PaillierEnc(_),_) => 0
      case ToPaillier(_,_) => 1
      // for each encryption scheme ...
      case Plus(PaillierEnc(_),PaillierEnc(_),_) => 0
      case Plus(_,PaillierEnc(_),_) => 1
      case Plus(PaillierEnc(_),_,_) => 1
      case Plus(_,_,_) => 2
      // for every operation in the DSL ...
      case _ => sys.error("Unhandled case.")
    }
  })
}
```

**Listing 3.21:** Determine the number of required conversions using static analysis.

This analysis can be used to estimate the number of required conversions in advance, which indicates how "costly" in terms of network overhead the interpretation will be when executed in the cloud. A sample usage of `requiredConversions` is shown in listing 3.22. We construct an program in our DSL using the applicative builder syntax from Scalaz, which converts the three encrypted integers to the Paillier scheme and then exploits the homomorphic property to add up the

three operands. While seemingly not extremely useful at first glance, `requiredConversions` is an important building block for other static analysis functions.

```
lazy val (x,y,z): (EncInt,EncInt,EncInt) = ??? // three encrypted numbers

val conversions: Int = requiredConversions {
  (toPaillier(x) |@| toPaillier(y) |@| toPaillier(z)){_+_+_}
}
```

**Listing 3.22:** Using information from static analysis.

```
  def convert(keyRing: KeyRing)(s: EncScheme, v: EncInt): EncInt = ???
  def preconvert[A](keyRing: KeyRing): Crypto[A] => Crypto[A] =
    _.foldMap(new (CryptoF ~> Crypto) {
      def apply[B](fa: CryptoF[B]): Crypto[B] = {
        fa match {
          case ToPaillier(v,k) =>
            val r@PaillierEnc(_) = convert(keyRing)(Additive, v)
            FreeAp.point(k(r))
          case ToGamal(v,k) =>
            val r@ElGamalEnc(_,_) = convert(keyRing)(Multiplicative, v)
            FreeAp.point(k(r))
          case ToAes(v,k) =>
            val r@AesEnc(_) = convert(keyRing)(Equality, v)
            FreeAp.point(k(r))
          case ToOpe(v,k) =>
            val r@OpeEnc(_) = convert(keyRing)(Comparable, v)
            FreeAp.point(k(r))
          case x => FreeAp.lift(x)
        }
      }
    })
```

**Listing 3.23:** Using static analysis to perform encryption scheme conversion before execution.

The previous example does only perform static analysis on the program, but in fact we can also modify it, for example we can perform all required conversions before running the program. A function to do this is shown in listing 3.23. The `preconvert` function takes as input the key ring and results in another function that transforms programs of our DSL. When passed a program as input, the result is a new program which, when run, does not need to convert numbers between encryption schemes. Static analysis is especially helpful for programs running on remote hardware and therefore communicating with an encryption service. If we return to our example of `sumList`, recall that in the monadic DSL, we are limited to *one* conversion at a time, because we are not able to look at the next "instruction" without having the result of the current one. With the applicative DSL however, we can do better, e.g., by first looking at the total number of required conversions via `requiredConversions` and then doing the conversion of each individual number *in parallel*, or we can group conversion requests into fixed size packets in order to better utilize network bandwidth when the number of conversions is extremely high.

### 3.4.5 Recovering Monadic Expressivity

As stated earlier, switching from free monads to free applicative functors enables implicit parallelism and static analysis, but this comes at the cost of the ability to depend on previous effectful values. A DSL that is based on an applicative functor interface seems to be too restrictive to write larger programs, where it is rather common that we depend on effectful results from previous computations. On the other side, we stated earlier that a DSL based on free monads can definitely not be analyzed statically and by deviating from laws, in order to make use of implicit parallelism, we open

ourselves up to subtle issues, of which some are discussed in section 3.7. It turns out that we can combine our two DSLs, therefore allowing programs to depend on previous effects, while still being able to to perform static analysis for the parts that only use the applicative interface. The rest of this section outlines the necessary changes to our DSL.

We introduce a new functor **Embed**, which will embed programs written using the free applicative functor into a new DSL based on free monads. We use the technique from [Swi08] to then combine our previous **CryptoF** and the new **Embed** functor into another by using Scalaz's **Coproduct**. We create a new type alias for convenience, **CryptoM**, which is the free monad over our functor coproduct:

```
type Crypto[A] = FreeAp[CryptoF, A]
type CryptoM[A] = Free[Coproduct[CryptoF,Embed,?], A]
```

Listing 3.24 demonstrates the use of the smart constructor for **Embed**. We use the `sumList` program from section 3.4.3 (listing 3.19) which uses the applicative style DSL. When introducing this version earlier, we remarked that the only difference to the monadic version in section 3.3.6 (listing 3.15) is, that the first argument has to be of type **PaillierEnc** i.e., the encryption scheme is fixed, because we cannot perform the conversion as well as the sum without requiring the power of monads to depend on the conversion result. By using the embedding, we can remedy this flaw and write yet another version of `sumList`, shown in listing 3.24. The first argument to `sumList2` is now of type **EncInt**, which means we do not care about the concrete encryption scheme of the integer, allowing any **EncInt** to be passed in as an argument. The first step in the program then is to make sure that it is encrypted under the Paillier encryption scheme. Here we have to use the monadic interface, because we depend on the effectful result of `toPaillierM`. The second step is to use the result and pass it to the embed *applicative* version of `sumList`. Any argument to `embed` has to be written using the applicative interface, enforced by Scala's type system. Taken together, the interpretation of the `sumList2` program will proceed roughly as follows:

1. convert the first argument to the Paillier encryption scheme
2. run the embedded applicative program
   a) convert all numbers, *in parallel*
   b) sum the list of numbers

While the concrete behaviour ultimately depends on the concrete choice of the interpreter, we can still exploit the advantages of the applicative style using the embedding inside monadic programs. The result type of `sumList2` is **CryptoM**[**PaillierEnc**], this means that we have a program using the monadic interface, which can no longer be embedded using **Embed** and can also not be statically analyzed. We therefore have a strict separation of the monadic and applicative DSL with the ability to use an applicative program inside a monad program, but not vice versa. This guarantees that programs of type **Crypto**[**A**] for some **A** can make use of implicit parallelism during interpretation and can be statically analyzed. Calls to `embed` cannot be left out, if we forget it, we will get a type error. For more convenience, we can make the `embed` call implicit, such that in the previous example developers could leave it out completely. In practice however we advocate the explicit use of `embed` because it makes it more obvious which parts of the program use the monadic, respectively the applicative interface.

```
1 def sumList2(zero: EncInt)(xs: List[EncInt]): CryptoM[EncInt] = for {
2   z <- toPaillierM(zero)
3   result <- embed(sumList(z)(xs))
4 } yield result
```

**Listing 3.24:** Embedding applicative programs.

## 3.5 The Complete DSL

In previous sections, we use a simplified version of our DSL, limiting the number of operations available for developers. After combining the monadic and applicative DSL, in this section we remedy this fact and introduce additional operations available in the full implementation. As the available operations of the DSL are defined by the base functor, we start wit the full version of **CryptoF** as shown in listing 3.25.

The **CryptoF** functor can be roughly divided into three parts:

```scala
sealed trait CryptoF[+K]

sealed trait CryptoRatio[+K] extends CryptoF[K]
case class CeilRatio[K](r: EncRatio, k: EncInt => K) extends CryptoRatio[K]
case class FloorRatio[K](r: EncRatio, k: EncInt => K) extends CryptoRatio[K]

sealed trait CryptoString[+K] extends CryptoF[K]
case class CompareStr[K](lhs: EncString, rhs: EncString,k: Ordering => K) extends CryptoString[K]
case class EqualsStr[K](lhs: EncString, rhs: EncString, k: Boolean => K) extends CryptoString[K]
case class ToAesStr[K](v: EncString, k: AesString => K) extends CryptoString[K]
case class ToOpeStr[K](v: EncString, k: OpeString => K) extends CryptoString[K]
case class ConcatStr[K](s1: EncString, s2: EncString, k: EncString => K) extends CryptoString[K]
case class SplitStr[K](s: EncString, regex: String, k: List[EncString] => K) extends CryptoString[K]

sealed trait CryptoNumber[+K] extends CryptoF[K]
case class Mult[K](lhs: EncInt, rhs: EncInt, k: ElGamalEnc => K) extends CryptoNumber[K]
case class Plus[K](lhs: EncInt, rhs: EncInt, k: PaillierEnc => K) extends CryptoNumber[K]
case class Equals[K](lhs: EncInt, rhs: EncInt, k: Boolean => K) extends CryptoNumber[K]
case class Compare[K](lhs: EncInt, rhs: EncInt, k: Ordering => K) extends CryptoNumber[K]

case class Encrypt[K](s:Scheme, v: Int, k: EncInt => K) extends CryptoNumber[K]
case class ToPaillier[K](v: EncInt, k: PaillierEnc => K) extends CryptoNumber[K]
case class ToGamal[K](v: EncInt, k: ElGamalEnc => K) extends CryptoNumber[K]
case class ToAes[K](v: EncInt, k: AesEnc => K) extends CryptoNumber[K]
case class ToOpe[K](v: EncInt, k: OpeEnc => K) extends CryptoNumber[K]

case class Sub[K](lhs: EncInt, rhs: EncInt, k: EncInt => K) extends CryptoNumber[K]
case class Div[K](lhs: EncInt, rhs: EncInt, k: EncRatio => K) extends CryptoNumber[K]
case class IsEven[K](v: EncInt, k: Boolean => K) extends CryptoNumber[K]
case class IsOdd[K](v: EncInt, k: Boolean => K) extends CryptoNumber[K]

abstract case class Embed[K]() {
  type I
  val v: Crypto[I]
  val k: CryptoM[I] => CryptoM[K]
}
```

**Listing 3.25**: The complete DSL.

- the `CryptoRatio` trait, defining rounding operations on ratios of encrypted integers
- `CryptoString`, defining operations for Strings and `CryptoNumber` supporting the already introduced operations on encrypted integers as well as additional operations not used before like subtraction, division, etc
- the `Embed` class to embed applicative programs into the monadic DSL

Together these parts define all available operations or our DSL. As before, we do not expose the constructors directly, instead we define corresponding smart constructors for both the monadic and applicative DSL, as well as infix operators where applicable. As an example, for the `Mult` class, we define the following:

```scala
def multiply(lhs: EncInt, rhs: EncInt): Crypto[EncInt]
def multiplyM(lhs: EncInt, rhs: EncInt): CryptoM[EncInt]
implicit class InfixOps(self: EncInt) { def *(that: EncInt) = multiply(self,that) }
```

Some operations, like `isEven` are not actually supported by any encryption scheme, we will discuss their implementation in section 3.6.

```scala
trait CryptoInterpreter[F[_]] {
  def interpret[A](p: CryptoM[A]): F[A]
  def interpretA[A](p: Crypto[A]): F[A]
}
```

**Listing 3.26:** The interface for interpreters.

Having defined our basic operations of the DSL, we still have to look at the interpretation. In general, interpreters extend the basic `CryptoInterpreter` trait as shown in listing 3.26. The trait defines two different interpretation functions `interpret` and `interpretA` which are responsible for the interpretation of monadic and applicative programs respectively. In theory, we can use the implementation of `interpret` just as well for `interpretA`, but the distinction allows to exploit the fact that we can interpret applicative programs in a more flexible way, as shown in previous sections. `CryptoInterpreter` is parameterized over a higher kinded type constructor `F`, which appears in the result type of the interpretation functions. This higher kinded type parameter is useful, because it allows different result types for different interpretation styles, e.g., local and remote, as shown in listing 3.27.

```scala
case class LocalInterpreter(
  keyRing: KeyRing)
    extends CryptoInterpreter[λ[α=>α]] {
  def interpret[A](
    p: CryptoM[A]): A = ???
  def interpretA[A](
    p: Crypto[A]): A = ???
}
```

**(a)** Local

```scala
case class RemoteInterpreter(
  service: CryptoService)(
  implicit ec: ExecutionContext)
    extends CryptoInterpreter[Future] {
  def interpret[A](
    p: CryptoM[A]): Future[A] = ???
  def interpretA[A](
    p: Crypto[A]): Future[A] = ???
}
```

**(b)** Remote

**Listing 3.27:** "CryptoInterpreter" instances for local and remote execution.

In this work, we only presented local and remote versions of interpretation for DSL programs. Those two examples should not be considered the only possible ways though, in fact we could allow operations for only a subset of the encryption schemes used in our implementation and fail at runtime if we cannot perform the operation, or we could mix local and remote interpretation, for example if we trust the third party with the access to at least a part of the private encryption keys. Other options include the decision what operations are to be performed in parallel, what aspects of static analysis are used and also whether we use only a central CryptoService for all operations, or if we have different services depending on the scheme or the operation. With these examples we hope to demonstrate the flexibility we gain with our approach.

After the interpreters are defined, we next look at the `CryptoService` type, which is responsible for handling conversion requests and might also be asked to perform computations that are not supported by any scheme. In listing 3.28 we show the interface that provides the necessary functions to convert between encryption schemes. As noted earlier, in some cases we may be faced with a huge number of required conversions when interpreting a program. Simply flooding the network with potentially hundreds of conversion requests at the same time is undesirable, therefore we provide the functions `batchConvert` and `batchEncrypt` in addition to their variants handling only one input, `convert` and `encrypt`. The last two functions `decryptAndPrint` and `println` can be used to communicate between the currently being interpreted program and the hardware running the `CryptoService`.

## 3.6 Additional Operations

If we take a closer look at the operations of our DSL in listing 3.25, we can see operations like `SplitStr` which are clearly not supported by any of our encryption schemes. The trick to make it possible to use those operations is that we support and expose them to developers inside the DSL, but this does not change the fact that at the end we have to perform the operation over *unencrypted* data. The problem is therefore simplified to an implementation, that allows the operation to be performed over unencrypted data without leaking it to adversaries. In the local interpretation style we do not have network communication and we are running on a local trusted machine, therefore the implementation is easy as:

```scala
trait CryptoService {
  def publicKeys: Future[PubKeys]

  def toPaillier(in: EncInt): Future[PaillierEnc]
  def toElGamal(in: EncInt): Future[ElGamalEnc]
  def toAes(in: EncInt): Future[AesEnc]
  def toOpe(in: EncInt): Future[OpeEnc]
  def toAesStr(in: EncString): Future[AesString]
  def toOpeStr(in: EncString): Future[OpeString]

  def convert(s: Scheme)(in: EncInt): Future[EncInt]
  def batchConvert(xs: List[(Scheme,EncInt)]): Future[List[EncInt]]

  def encrypt(s: Scheme)(in: Int): Future[EncInt]
  def batchEncrypt(xs: List[(Scheme,Int)]): Future[List[EncInt]]

  def decryptAndPrint(v: EncInt): Unit
  def println[A](a: A): Unit
}
```

**Listing 3.28:** The full "CryptoService" interface.

```scala
case SplitStr(string,regex,k) =>
  val plain: String = decryptStr(keyRing)(string)
  val split: Array[EncString] = plain.split(regex).map(encryptStrOpe(keyRing))
  interpret(k(split.toList))
```

For remote interpretation the case is different, due to the network communication and the fact that the hardware running the program is considered not trustworthy, we have to fall back on respective methods from the `CryptoService`, providing an equivalent function that will perform the operation by communicating with a trusted party. The operation will still be performed on unencrypted data, but it is done on the machine running the `CryptoService`, which has to be trustworthy in any case, given that it has access to private encryption keys. While this is certainly better than not allowing these operations at all, it has the drawback that instead of using the cloud hardware, we now rely on the machine running the `CryptoService` to perform possibly quite expensive operations, which may quickly lead to a bottleneck. By allowing operations on the `CryptoService` machine though, we gain a lot of functionality, because operations like e.g., division can be performed in the course of a DSL program, while other systems [PRZB11, JSSSE14] don't allow this type of operations at all.

## 3.7 The Role of Separation

In our system, we have a forced distinction between programs written using the monadic DSL and programs written using the applicative DSL. In theory we could combine the two and provide only the monadic interface, because every monad is also an applicative functor. As an example, Haxl's `Fetch` datatype [MBCP14] is an instance of `Monad` and therefore also has to be an instance of `Applicative`. In the `Applicative` instance, in order to make use of the implicit parallelism, the authors deviate from one of the rules from Haskell's `Control.Applicative`, which says:

> *If `f` is also a Monad, it should satisfy*
>
> - *`pure = return`*
> - *`(<*>) = ap`*

This however rules out the use of concurrency in the `Applicative` instance, because `ap` is defined as shown in listing 3.29 and the use of `(>>=)` (which is used after desugaring of the do-notation) is inherently sequential, as discussed before.

One may argue, that it is only a rule and rules can be broken. While this is certainly possible, it leads to unexpected problems later. As an example, in Haxl[MBCP14], there are subtle issues developers have to take into account when

```haskell
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf mx = do
  f <- mf
  x <- mx
  return (f x)
```

**Listing 3.29:** Definition of "ap".

writing programs using the `Fetch` type. To see what those issues are, consider the following two functions that exist in the Haskell standard library:

```haskell
traverse :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f (t b)
mapM     :: (Traversable t, Monad m)       => (a -> m b) -> t a -> m (t b)
```

The type signatures are almost identical, differing only in the class context. The `mapM` function requires a monad instance, while `traverse` only requires applicative. Both functions furthermore serve the same purpose: to loop over a structure while applying an effect to each element. Historically however, the `Monad` type class and therefore `mapM` predates the `Applicative` type class used by `traverse`. Due to this, the `mapM` function was written first, with the `traverse` function being introduced later. Up to and including version 7.8 of GHC, the two implementations differ, such that a program written in such a way that `mapM` is used, the program would run sequential. Only if the developer replaces `mapM` with `traverse`, the program would exhibit the expected parallelism during runtime [MBCP14]. To avoid subtleties like this and in order to be more explicit about what kind of behaviour one can expect from a program written in the DSL, we separate the two interfaces. The benefit is that a program of type `Crypto[A]` is *guaranteed* to be statically analyzable and will exhibit concurrent behaviour during runtime (of course depending on the interpreter). `CryptoM[A]` instead, does not give those guarantees, but provides more expressive power in the sense that we can depend on previous effectful results. As desirable as this is, sometimes it might be preferable to make this separation less intrusive for developers, therefore our implementation can perform the lifting from `Crypto[A]` to `CryptoM[A]` implicitly, hiding the necessary wrapping inside the `Embed` data type. In Haskell, the plan is to introduce a new language pragma, `ApplicativeDo` [App], which tries to desugar do-notation such that the result uses applicative combinators where possible.

## 3.8 Execution Model

This section discusses a typical execution model of a program written in our DSL and running on a third party's hardware. The sequence diagram in figure 3.2 shows such a typical execution. The first step is for the client to write the program using our DSL. Then a `CryptoService` has to be started. In theory, the `CryptoService` can simply be run on the Client i.e., a 1:1 correspondence, though other scenarios are also possible like 1:n and n:m. The next step is for the Client to provide the `CryptoService` with the necessary cryptographic schemes. After the `CryptoService` is ready, the Client sends the program to the remote hardware (into the cloud). The Client then uploads the encrypted data required for the program as well. In figure 3.2 this is marked as optional, because we can in theory also execute programs where the *input* is unencrypted but the final result returned from the program is encrypted. The program is then started on the remote hardware and communicates as necessary with the provided `CryptoService` instance for encryption scheme conversion and to perform operations not supported by partial homomorphic encryption. After the program as finished, the result, normally encrypted, is sent back to the Client.

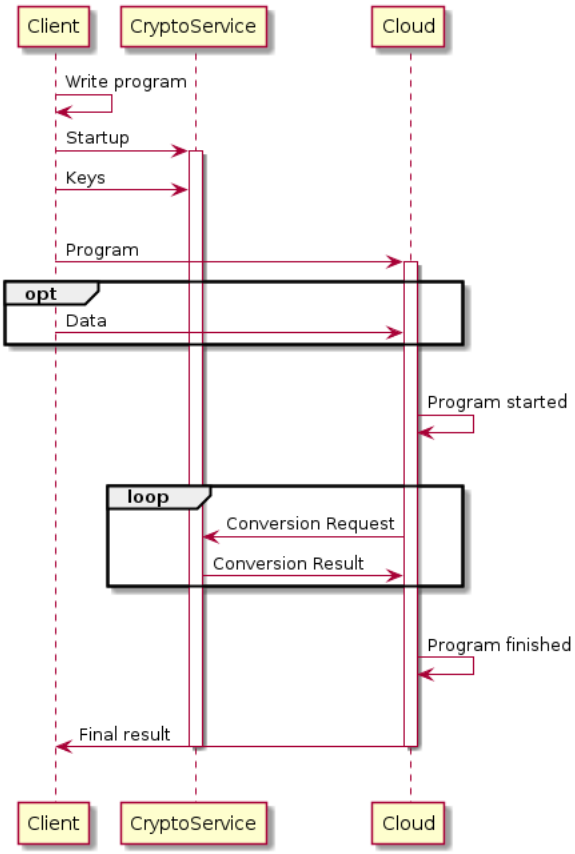



**Figure 3.2.**: A sample execution model for remote execution.

# 4 Evaluation

This chapter evaluates our implementation of the DSL. After showing an example of a program written in the DSL, we compare the performance of different interpretation styles and evaluate several case studies demonstrating the practical use of our DSL.

The benchmarks were run on a Thinkpad X220 with an Intel dual core i7 2620M 2.7 GHz processor and 8GB of RAM. The code was compiled and executed with Scala 2.11.7, the OpenJDK Runtime Environment (IcedTea 2.5.6) and the OpenJDK 64-Bit Server VM (build 24.79-b02). For the benchmark execution, we use the ScalaMeter[1] framework in version 0.6. We use the predefined `PerformanceTest.OfflineReport` test configuration to run each benchmark in a separate JVM.

## 4.1 Writing Programs

We now show a larger example of a program following the approach introduced in chapter 3. As an example, we use a program calculating the n-th Fibonacci number, shown in listing 4.1.

```
1  // Encrypted
2  def fib(n: EncInt): CryptoM[EncInt] = for  {
3    (one,two) <- encrypt(Additive)(1) tuple encrypt(Additive)(2)
4    r <- fibHelper(one,two)(n)
5  } yield r
6
7  def fibHelper(one: EncInt, two: EncInt)(n: EncInt): CryptoM[EncInt] = for  {
8    cmp <- n <= one
9    r <- if (cmp) {
10     one.lifted
11   } else for {
12     (n1,n2) <- (n-one) tuple (n-two)
13     (f1,f2) <- fibHelper(one,two)(n1) tuple fibHelper(one,two)(n2)
14     s <- f1 + f2
15   } yield s
16 } yield r
17
18 // Normal
19 def fibVerbose(n: Int): Int =  {
20   val cmp = n <= 1
21   if (cmp) {
22     1
23   } else {
24     val (n1,n2) = (n-1,n-2)
25     val (f1,f2) = (fibVerbose(n1),fibVerbose(n2))
26     val s = f1 + f2
27     s
28   }
29 }
```

**Listing 4.1:** A program for Fibonacci numbers.

The definition of `fib` in line 2 is a rather simple wrapper, that requests encrypted versions of the literals "1" and "2" which are required for the calculation inside `fibHelper`. At this point it is much better to separate the request of encrypted literals and the recursive calculation, because otherwise the program would have to request them in *each* recursive call.

[1] https://scalameter.github.io/

The main calculation starts at line 7 in `fibHelper`. In line 19 we include the definition of an unencrypted version doing the same calculation as a reference. An important detail in `fibHelper` is the use of `(n-one) tuple (n-two)` in line 12. In Haskell, an equivalent version, given an infix operator `(!-)` for subtraction of encrypted integers and the `embed` function, would look like this:

```
1 embed :: Crypto a -> CryptoM a
2 (!-) :: EncInt -> EncInt -> Crypto EncInt
3 (n1,n2) <- embed ((,) `fmap` (n !- one) <*> (n !- two) :: Crypto (EncInt,EncInt))
```

**Listing 4.2:** "tuple" in Haskell using Applicative.

The necessary call to `embed` is explicit in the Haskell version, while in Scala, it is added automatically by an implicit. The fact that we are using the applicative interface and that the result of the subtraction is a program of type `Crypto[EncInt]` leads to a program structure that can perform the two operations, "`n-one`" and "`n-two`" in parallel. In contrast, the use of `tuple` in line 13 will *not* allow parallel execution, because `fibHelper`'s result type is `CryptoM`, the monadic DSL, and therefore we use the `Applicative` instance that is compatible with the `Monad` instance, forbidding implicit parallelism. This example demonstrates that developers writing the DSL have to take care about minimizing the parts using the monadic interface in order to get better performance. On the upside, this means that it is easier to reason about the expected runtime behaviour, because every time we have a computation of type `Crypto[A]` we can be sure that it can be executed implicitly parallel and analyzed statically.

## 4.2 Styles of Interpretation

| style | description |
|---|---|
| local | Local interpretation |
| remote | Use `CryptoService`, otherwise like *local* |
| remoteOpt | Use `CryptoService` but makes use of parallelism when possible |
| remoteOptAnalysis | Like *remoteOpt*, but also starts to batch requests into chunks of up to 15, if the total number is higher than a certain threshold, in this case the threshold was also set to 15. |

**Table 4.1.:** Comparison of interpretation styles.

Figure 4.1 compares the performance of interpreting a simple program that sums up an increasing number of encrypted integers. The program is evaluated in a version using the monadic style from listing 3.15 as well as the applicative version from listing 3.19. The encrypted numbers are encrypted under a random encryption scheme, where each scheme is equally probable. For both the applicative and monadic program, we compare different interpretation styles, an overview with the differences is shown in table 4.1. The results show that for the first two interpretation styles – "local" and "remote" – there is almost no performance difference between the applicative and the monadic version of the program. For the "remoteOpt" case however, there suddenly is a difference of about factor two between the two programs. The reason is that in the applicative version of `sum` we can use implicit parallelism to perform the encryption scheme conversions for all numbers in parallel, leading to a speedup of two, limited by the dual core architecture of the test hardware. The "remoteOptAnalysis" style exhibits the same behaviour as the previous style, although it performs static analysis in addition to exploiting implicit parallelism as explained in table 4.1. In this case however, there was no network delay for encryption scheme conversion and therefore the interpreter does not gain any benefits from grouping conversion requests when compared to the previous style. Therefore, figure 4.2 shows the results of another benchmark, where we introduce an artificial conversion request delay of *50ms*, which simulates network latency when performing conversion requests over the network i.e., talking to a remote `CryptoService`. Looking at the result, we can see that the two optimizing interpreters "remoteOpt" and "remoteOptAnalysis" both perform significantly better than the naive "remote" interpreter performing no optimization. With the delay for conversion requests, we can see that the interpreter using static analysis starts to outperform the interpreter exploiting only implicit parallelism starting from an input size that is larger than ten. The specific point at which the static analysis interpreter improves is explained by the fact that we start grouping conversion requests at a specific size, in this case only if the number is greater than fifteen. With the growing input size, static analysis increases the performance gains, because the "remoteOpt" interpreter will simply perform *all* conversion requests in parallel and therefore it has to pay the network delay for each request, while the "remoteOptAnalysis" groups the requests

into groups of up to fifteen and therefore has to pay the cost of the network delay not as many times as when compared to the “remoteOpt” style of interpretation.

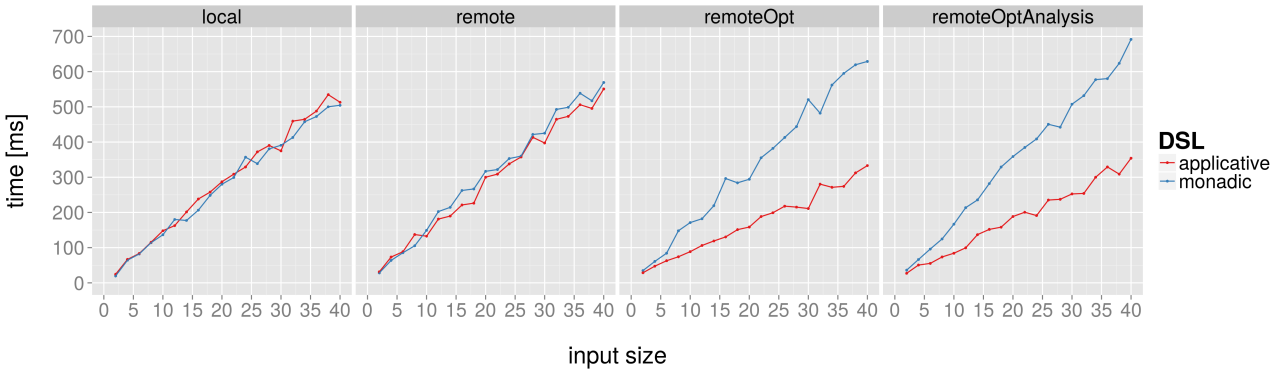



**Figure 4.1.:** Benchmark showing different interpretation styles.

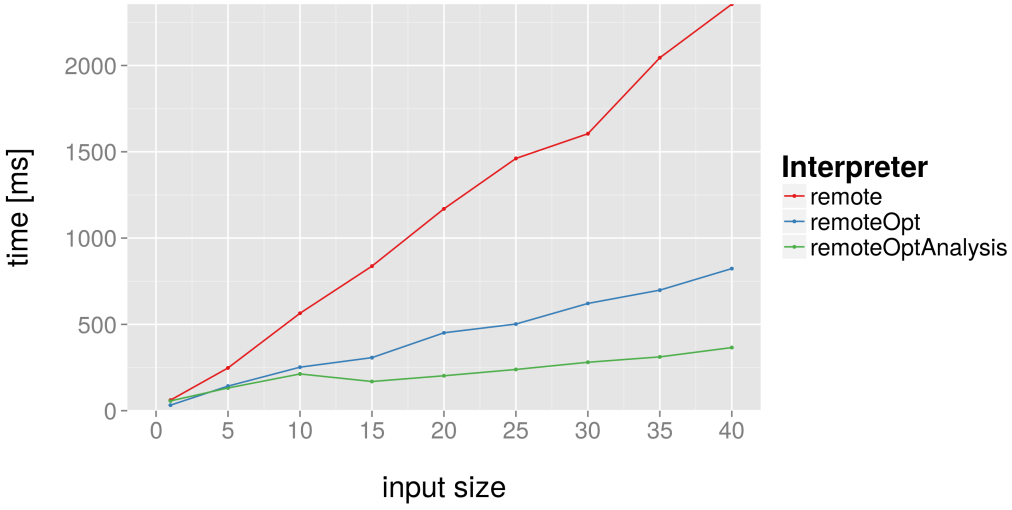



**Figure 4.2.:** Interpretation with 50ms simulated conversion delay.

## 4.3 Case Studies

This section evaluates three case studies for writing larger programs with our DSL. The first example is a program to count the words in a file that is encrypted under the OPE encryption scheme. The result is a grouping of encrypted words to their unencrypted count. The second case study uses the complex event processing engine Esper[2] to process a series of events with encrypted attributes. The final case study uses RxScala to write a GUI showing an ongoing series of encrypted transactions between bank accounts using a reactive style.

### 4.3.1 Counting the Words in a Text - Word Count

The input is an encrypted text from file, the output is a file with the encrypted word and associated word count as a number in plaintext.

The core word count function can be written in the DSL as shown in listing 4.3. The structure of the whole program is described as follows:

1. read the file with encrypted text

[2] http://www.espertech.com/esper/

2. split the text into words
3. group by word
4. associate each word with the number of occurrences
5. write the result into a file or print them

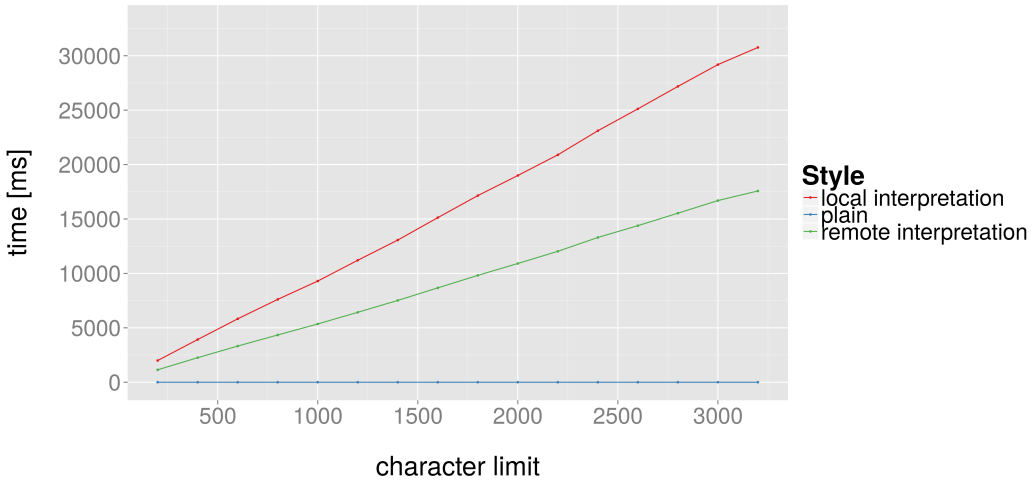



**Figure 4.3.:** Benchmark results for Word Count.

The complete text used in this case study is included in section A.1. Figure 4.3 shows the result of running the `wordCountText` program on an increasing portion of the sample text. We included measurements for two interpretation styles, local interpretation without any optimization,remote interpretation using implicit parallelism as well as an implementation in plain Scala for reference. From the results, we can see that the remote interpretation style applying implicit parallelism performs about twice as fast as the local interpreter, which is very close to the optimum on a dual core processor. Of course the plain Scala version without any kind of encryption, shown as *plain* is much faster, we include it here to show the impact of the encryption and the DSL on performance.

```
1 def wordCountText(input: EncString): CryptoM[List[(OpeString,Int)]] =
2   input.split("""\s+""") >>= wordCount_
3
4 def wordCount_(input: IList[EncString]): Crypto[List[(OpeString,Int)]] =
5   input.traverse(toOpeStr).map(_.groupBy(x => x).map(_.length).toList)
```

**Listing 4.3:** Counting words.

### 4.3.2 Complex Event Processing - License Plates

The License Plates case study combines the complex event processing engine Esper[3] with our DSL to perform queries over events containing encrypted fields. Esper allows the usage of user defined functions inside EPL queries and therefore we can also call – with some restrictions – Scala methods from within queries, thereby using our DSL programs to work with encrypted data.

Figure 4.4 visualizes the situation. We have a track with a start and a goal with three checkpoints at which speed cameras measure the speed of passing cars. If the speed is higher than a threshold, the camera will trigger and capture an image. Events consist of three attributes, an encrypted string for the license plate, an encrypted integer as the speed of the car and a plain integer as a timestamp for simulation purposes. More concretely an event is represented by the types in listing 4.4.

[3] http://www.espertech.com/esper/

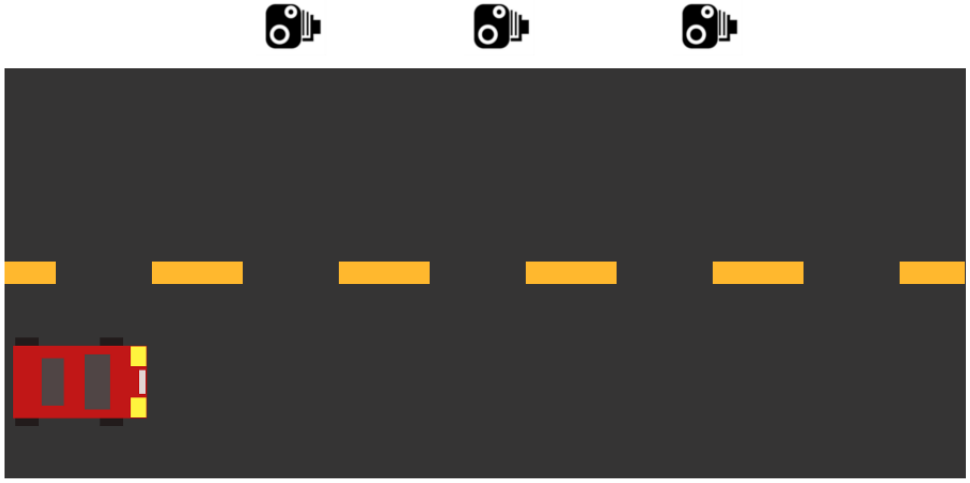


**Figure 4.4.:** Scenario for the License Plates case study.

```
sealed trait LicensePlateEventEnc {
  @BeanProperty def car: EncString
  @BeanProperty def time: Long
  @BeanProperty def speed: EncInt
}

case class CarStartEventEnc(...) extends LicensePlateEventEnc
case class CheckPointEventEnc(...) extends LicensePlateEventEnc
case class CarGoalEventEnc(...) extends LicensePlateEventEnc
```

**Listing 4.4:** Structure of events in the License Plate case study.

The three different case classes represent the start event, the goal event, triggered once the car is at the end of the track and a checkpoint event. Both the start and goal event occur only once per car, while the checkpoint event occurs once for every checkpoint on the track, i.e., three times per execution in our case.

A query that selects all checkpoint events with a speed above the threshold in Esper is shown in figure 4.5. The important part is inside the `WHERE Interp.isTooFast(speed)`. Figure 4.6 shows the relevant Scala code, the `Interp` object defines the `isTooFast` method which runs the interpreter to check the result of the DSL program, indicating if the speed is higher than the allowed `speedLimit`.

```
SELECT car AS license, number, speed
FROM CheckPointEventEnc
WHERE Interp.isTooFast(speed)
```

**Figure 4.5.:** Example query for cars going faster than a threshold.

In figure 4.7, the results for the license plates benchmark are shown. We compare the performance of two semantically equivalent versions of the same program. The program's task is it to monitor car events. For each unique license plate, it tracks the start time, the three checkpoint times and the end time. At each checkpoint, a query will check the speed of the car with the associated threshold, on violation of the speed limit, it will trigger an action with the license plate of the offending car. A version that does not use encrypted events is compared against a version that makes use of our DSL as described to check the *encrypted* integer with the speed of the car event with the threshold and groups the stream on the *encrypted* license plates. The encrypted version was run with a *local* interpreter i.e., without implicit parallelism and without talking to a remote service. However choosing an interpreter with implicit parallelism does not give us a significant performance boost, because all of the uses of the DSL involve simple checks like "greater than" or "equality".

```
object Interp {
  val keyRing: KeyRing = ???
  val interpret: LocalInterpreter = ???
  val speedLimit: EncInt = ???

  def isTooFast(s: EncInt): Boolean = interpret(s > speedLimit)
}
```

**Figure 4.6.:** Interpreter object for Esper.

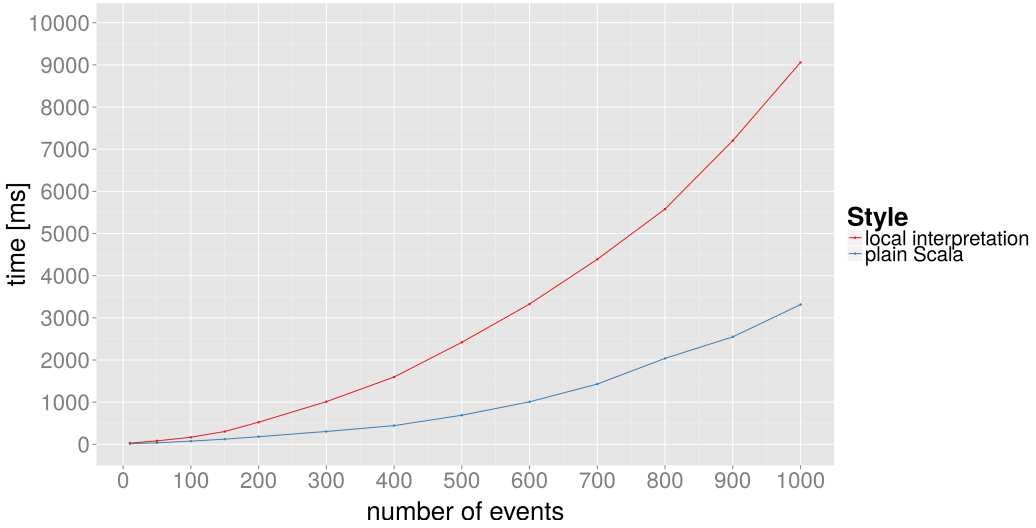



**Figure 4.7.:** Benchmark of the License Plates case study.

These small programs have almost no implicit parallelism, so the opportunities for improving the performance are equally low.

### 4.3.3 Reactive Programming - RxBank

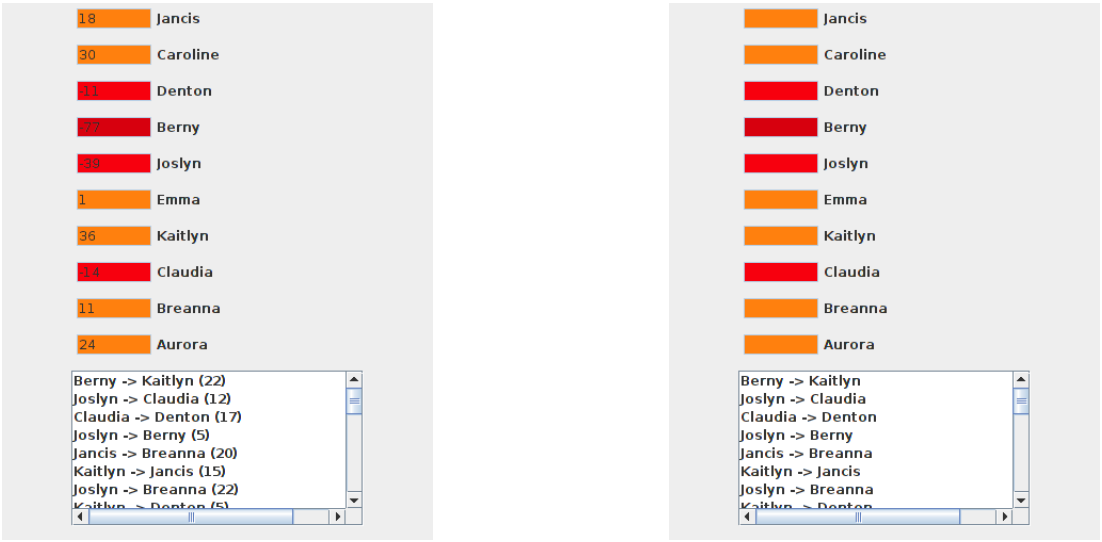
The RxBank case study uses RxScala[4] in combination with our DSL to simulate a number of bank accounts with random transactions between them. The focus is the GUI visualizing those transactions by showing the account balances. Figure 4.8 shows an example of the running GUI. In figure 4.8a we can see both the balance of the accounts and the amount transferred in the message log, while in figure 4.8b we cannot, because the balance and the transactions are encrypted. Bank accounts are highlighted depending on the current balance, from dark red (very negative) over orange (neutral) to light green (very positive).

| transactions | plain | delay | encryped |
|---|---|---|---|
| 10 | 34 | 2508 | 3804 |
| 25 | 48 | 6256 | 9434 |
| 50 | 77 | 12505 | 18427 |
| 75 | 108 | 18761 | 27162 |
| 100 | 133 | 25006 | 36811 |
| 125 | 166 | 31256 | 45430 |
| 150 | 198 | 37504 | 54273 |

**Table 4.2.:** Table of the benchmark results for the RxBank case study.

We evaluate the performance of RxBank by measuring the time required to process a variable number of transactions, which trigger changes in the GUI after each change to the account balances. We measure three different versions as described in :

[4] http://reactivex.io/rxscala/




**(a)** Plain **(b)** Encrypted

**Figure 4.8.:** Screenshot of the RxBank GUIs.

| | |
|---|---|
| **plain** | Plain Scala, process as fast as possible |
| **delay** | Plain Scala, limit transactions to 4 per second |
| **encrypted** | Encrypted version using a local interpreter, process as fast as possible |

The plain version *without any delay* is about 270x as fast as the encrypted version. While this performance difference is huge, it is important to remember that the transactions are processed as fast as possible. If we throttle the rate of transactions to about four per seconds, resulting in 8 updates per second to the GUI, we can see in figure 4.9 that the "delay" version of the plain RxBank is still faster than the encrypted version, but the difference is not as large as previously. Looking at table 4.2, we can see that in fact, the encrypted version is able to process transactions with a rate of approximately 2.7 transactions per second, which is still fast enough to display an animated version of the transactions in the GUI. The significant overhead does render the encrypted version useless for extreme high throughput scenarios, but if the update rate is slower, e.g., eight times per second as above, our implementation allows the practical use of encrypted data and their real time visualization in a graphical interface. Additionally, the RxBank case study shows that programs written in our DSL can easily be integrated with the RxScala library and the Scala wrapper for the Swing library.

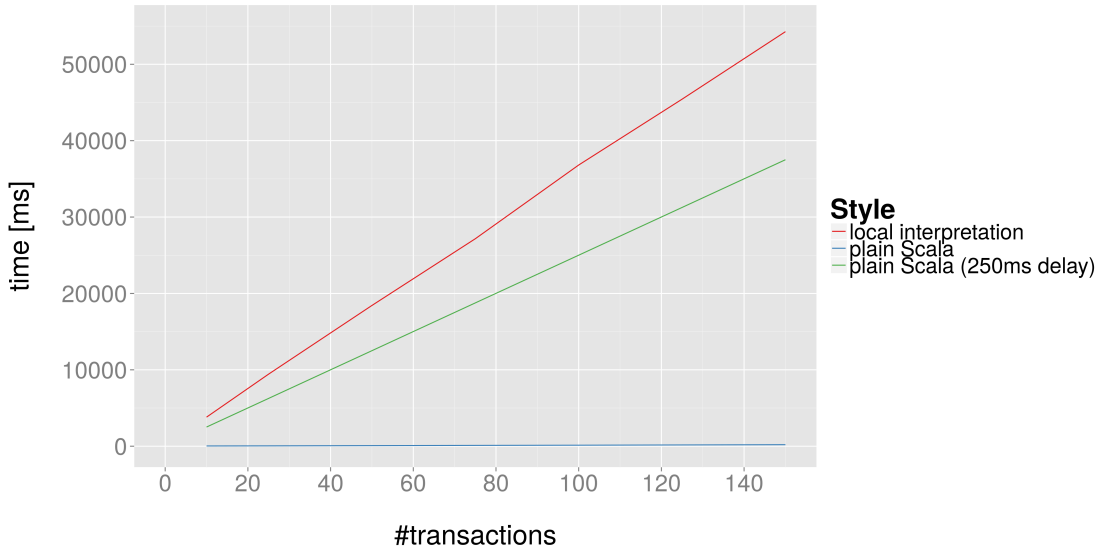



**Figure 4.9.:** Benchmark results for the RxBank case study.

# 5 Related work

This chapter presents work related to the core ideas underlying the components of this work: homomorphic encryption and techniques from functional programming. In addition, we also present related work for garbled circuits.

## 5.1 Homomorphic encryption

This section discusses related work that uses homomorphic encryption to perform computation over encrypted data.

### 5.1.1 CryptDB

CryptDB [PRZB11] is a system that enables computations over encrypted data in SQL backed systems. The domain of the project is focused on converting SQL queries so that they can be used with encrypted database records.

To handle dynamic conversion of the encryption scheme, CryptDB uses *onions*. Values are wrapped in layers of encryption, where the outermost layer is the most secure but provides no operation on the encrypted data and inner layers provide a certain kind of operation like addition, equality, and so on. Based on the received SQL queries CryptoDB's proxy server determines what onion layer is needed to perform the computation. The decryption of onion layers is performed by the DBMS by calling an user defined function and saving the layer of the column. Subsequent operations that make use of the same operations therefore do not require additional conversions, which is important to lower the overhead of computations over encrypted data.

A limitation of this approach is that for example averages cannot be computed in a direct way. The solution in this case is to return the sum and count separately so that the receiving side can perform the final division by itself. Furthermore, any queries that require operations provided by mutually exclusive encryption schemes are not directly supported in CryptDB. A possible way to overcome this limitation is to generate auxiliary columns for intermediate steps and divide the query into parts that require only one encryption scheme.

CryptDB operating in "multi-principal" mode does not support server-side computations because the ciphertexts may be encrypted with different keys. Due to this limitation, CryptDB can run only handle four out of 22 TPC-H [TPC] queries.

### 5.1.2 Monomi

Monomi is "a system for securely executing analytical workloads over sensitive data on an untrusted database server" [TKMZ13].

Monomi is similar in spirit to CryptDB [PRZB11] but introduces "split client/server exectuion" to be able to process queries that cannot be processed in CryptDB. The key idea is to execute as much of the computation as possible over encrypted data and if required send the intermediate result to a trusted client that performs the rest of the computation on the decrypted data. This execution model makes it possible to execute 19 out of the 22 TPC-H [TPC] queries. A number of techniques for performance improvement are introduced in addition to a *designer* to optimize physical data layout and a *planner* to optimize the partition of the query execution between client and server.

In contrast to our approach, Monomi tries to do as much of the computation as possible on the database server and leaves the remaining part of the computation to the trusted client that receives the data and continues after decryption. There is no communication back from the trusted client to the database server, the trusted client has to perform the whole rest of the computation.

### 5.1.3 MrCrypt

MrCrypt [TLMM13] transforms Java programs using static analysis. The target is to infer the set of operations for the individual input columns and finding a suitable homomorphic encryption scheme. After successful selection of a matching encryption scheme, a source-to-source transformation changes the original program to work on encrypted data. The encrypted data and the transformed program are uploaded into the cloud, the final result can be decrypted on the client.

If a column of the input data requires multiple operations that are not supported by a partial homomorphic encryption scheme, MrCrypt will infer fully homomorphic encryption as the required scheme for this column. Given the prohibitive computational costs using fully homomorphic encryption is not yet a practical solution. Therefore, MrCrypt's ability to

transform programs to work on encrypted data is drastically limited to those program that only make use of operations provided by a single partial homomorphic encryption scheme or it will infer fully homomorphic encryption.

In contrast to CryptDB which acts as a proxy together with a database containing encrypted data, the target domain of MrCrypt is cloud computing. The authors evaluate the system for Java programs running on the Hadoop implementation MapReduce [DG10].

### 5.1.4 Program Analysis for Secure Big Data Processing

In [JSSSE14], the authors present a program analysis and transformation scheme for programs as a runtime for Pig Latin [ORS+08] called SPR. In contrast to MrCrypt that does not handle computations requiring multiple encryption schemes for the same data, SPR can perform subcomputations on the client, thereby bypassing the restrictions imposed on programs that can be executed without falling back to fully homomorphic encryption.

The result of the transformation is a program where operations on encrypted data are replaced by calls to user defined functions (UDFs). If during execution a value is encrypted under an encryption scheme that does not allow the operation that is to be performed, SPR sends the encrypted data to an encryption service. The task of the encryption service is to decrypt the received data and re-encrypting it under the scheme required by the pending operation. After the re-encryption has completed, the result is sent back to the cloud and execution can continue.

SPR's applicability is limited to Pig Latin scripts and therefore tightly coupled to the MapReduce framework. While SPR makes use of an encryption service, it does not allow requests for computations over encrypted data that.

### 5.1.5 Information-flow control for programming on encrypted data

In [MSSZ12] the authors present "an expressive core language for secure cloud computing". The language's target is to make it easy for programmers to write secure programs without requiring knowledge of the underlying cryptographic means to ensure it. The language offers protection against control flow observations of the server that is running the program operating on encrypted data. A big advantage of this approach is that the EDSL, implemented in Haskell, is separated from the execution environment. This flexibility is exploited by providing different back ends that can use different cryptographic measures to execute the program. Therefore, it is not limited to homomorphic encryption, the same program can be run using secure multiparty computation techniques as well given a back end that makes us of it.

The EDSL approach is more flexible than our work with regard to the underlying cryptographic system, but at the cost of programmer friendliness. By decoupling the program from its interpretation we can additionally switch between different styles, e.g., local or remote and control the optimization that is performed, e.g., implicit parallelism if there are enough cores and static analysis if the network is a bottleneck. However, our implementation does not provide any protection for control flow, and therefore information may leak to the executing party.

## 5.2 Garbled Circuits

Garbled circuits are a solution to the problem of secure function evaluation. The protocol was developed by Yao, but never published, the first formal definition can be found in [GMW87]. The rough idea is to model the function as a Boolean circuit, garble the truth table as well as the own input and send both to the second party which evaluates the circuit without learning anything about the other party's input. As an overview, [Sny14] is a good introduction.

We continue this section with a brief summary of the protocol and related work that improves on it. To achieve the goal of secure function evaluation with inputs of two parties one party plays the role of the *generating* party, while the other plays the role of the *evaluating* party. The generating party's task is to model the function as a circuit, garble the truth table as well as his private input and send both the garbled circuit and the garbled input to the evaluating party. In turn, the evaluating party has to communicate with the generating party to determine the corresponding garbled value for his input to the circuit, to do this secretly they use a 1-out-of-2 oblivious transfer protocol [Rab05]. Once the evaluating party has both inputs in garbled form, it can start the evaluation of the circuit, resulting in a decrypted output value which is sent to the generating party to complete the protocol.

### 5.2.1 Fairplay-Secure Two-Party Computation System

Fairplay [MNPS04] introduces a "full fledged system that implements generic secure function evaluation (SFE)". Functions are modeled using a separate procedural definition language called SFDL. A compiler then translates this into SHDL, a language to represent Boolean circuits, from which Java objects are generated. The circuit is send to the evaluating party as is i.e., the whole circuit has to be transferred before evaluation can start. In this sense, Fairplay follows YAO's general

protocol as described above. This leads to some problems, e.g., by forcing a separation between generation and evaluation of garbled circuits, Fairplay requires the parties to have enough memory to fit the whole circuit into memory, which is impractical for large problems.

### 5.2.2 Faster Secure Two-Party Computation Using Garbled Circuits

In [HEKM11] Huang et al present a novel approach to make secure computation using garbled circuits more practical. In contrast to systems like Fairplay [MNPS04] the authors do not require the presence of the whole circuit in memory for evaluation. Instead, the process of circuit generation and evaluation is done incrementally, leading to a significant performance improvement. An example is given why the requirement of in-memory evaluation in systems like Fairplay is so problematic. According to the authors, an AES program written in SFDL takes several hours to compile, on a machine with 40 GB of memory. Evaluation shows that the pipeline approach is an order of magnitude more efficient than prior work, problems evaluated include “Hamming distance”, “Levenshtein edit distance”, the “Smith-Waterman algorithm” and “AES”. On the developer side, the framework does require a fairly low level of abstraction. Programmers are essentially writing circuits using Java objects and therefore it requires thinking in terms of circuits and gates rather than higher-level concepts. A disadvantage of the incremental nature is that there is no way to perform optimization that require a global view of the circuit.

### 5.2.3 VMCrypt

VMCrypt [Mal11] is a library that aims to provide an API to perform secure computation using garbled circuits in an efficient and scalable way. Main design goals are that the algorithms used by the library have a very small memory footprint even for large circuits, do not require disk storage and can easily be integrated with existing systems via the API. VMCrypt uses an underlying virtual machine to “load and destruct hardware descriptions dynamically” [Mal11] during the evaluation of the circuit, thereby eliminating the prohibitively high memory requirements for large programs as seen in [MNPS04]. To overcome disk storage requirements on the side of the evaluating party, VMCrypt interleaves oblivious transfer in Yao’s protocol thereby improving performance due to bulk execution and opportunities for multi-threading. The performance evaluation shows that VMCrypt is much more scalable than systems like Fairplay, which quickly has to give up due to memory requirements when the problem size rises.

### 5.2.4 Secure Two-Party Computations in ANSI C

In [HFKV12], Holzer et al use the existing model checker [CKL04] and on top of it develop CBMC-GC, a tool to compile ANSI C programs into garbled circuits. This approach enables programmers to write C programs instead of being forced to write in an even lower level language or having to write gates explicitly for the final circuit representation. Conveniently this approach benefits from CBMC’s techniques to minimize the size of the formulas representing the C program such that the generated circuits are more size efficient, which directly translates into improved performance for the generated Boolean circuit. Finally by optimizing the output even further based on the circuit structure, the authors achieve additional performance gains. The programs that can be translated are subject to some constraints due to the CBMC model checker, for example the program must provably terminate in a finite number of steps. Other limitations are mostly due to CBMC i.e., the public version of the source code does not support floating point arithmetic, so CBMC-GC is also limited in this way, though it is not a limitation of the GC approach. Finally, the framework separates generation and evaluation, such that the evaluation can be performed by a specialized framework for secure two-party computations, while for simplicity in their case STC [HEKM11] is used.

## 5.3 Functional Programming

In this section, we discuss related work for the functional programming aspect of our work, which includes free monads, their usage and optimization as well as free applicative functors.

### 5.3.1 Free Monads

Every functor gives rise to a free monad [Awo10]. Free Monads are very interesting because they provide an easy way to define embedded domain specific language using monadic combinators while separating the program construction from the interpretation. In [SA07] Swierstra et al. use this to great effect. The paper defines three different languages,

modeling teletype IO, mutable state and concurrency. Every language is defined using a suitable base functor, the result is a monadic DSL. The beauty of this approach is that it not only allows the expected interpretation, e.g., one with side effects in the case of teletype IO, but in addition also to keep the interpretation totally pure. This possibility of a pure interpretation is used by Swierstra et al to prove properties about programs and also allows easy testing using frameworks like QuickCheck [CH00].

Furthermore free monads can be composed easily, a very hard problem in general for monads [JD93, KW93, KSS13]. The trick is to use the fact that the functor which gives rise to the free monad in question composes. Therefore the techniques from [Swi08] can be used to achieve modularity and to compose languages defined by free monads over individual functors into a free monad combining the two languages into one.

Unfortunately the naive implementation of free monads (as in listing 2.8) yields very bad performance [Voi08]. There are at least two possible ways to improve on this situation, which will be described in the following sections.

### 5.3.2 Asymptotic Improvement of Computations over Free Monads

Computations involving free monads suffer from a severe performance problem, if the binds are associated in the wrong way. To remedy this, [Voi08] presents a technique to re-associate the monadic binds such that the asymptotics of computations using free monads are improved. The whole program-transformation is packed up behind one function, called `improve` which given a computation that only makes us of the free monad functions defined in a type class will return the equivalent computation, but may improve the performance, in some cases even the asymptotics. The technique essentially makes use of CPS to defer the construction of intermediate structures, which are teared down and rebuilt from scratch between monadic computations if the binds are associated in the wrong order.

### 5.3.3 Reflection without Remorse

In [vdPK14], Ploeg and Kiselyov present another optimization for free monads. Continuation-passing style is a way to improve the performance, as shown in [Voi08], but if one uses monadic reflection however, CPS will no longer improve the efficiency, but rather make the computation perform even worse than without the optimization applied. Monad reflection is supported by some monads to observe the current state of the monadic computation. The paper therefore distinguishes between building and observing. Observing is cheap using the naive definition of free monads, while building is problematic (slow) but can be solved using CPS. On the other hand using CPS solves the building problem but in exchange observing becomes now problematic. Therefore, Ploeg and Kiselyov use type aligned sequences to explicate the previously implicit tree of the computation and allow efficient switching between the building/observing.

### 5.3.4 There is no Fork

Haxl [MBCP14] uses an applicative abstraction to exploit the implicit concurrency resulting from the use of applicative combinators. In essence, Haxl provides an instance of a free monad to define an embedded DSL for data fetching from different sources, making use of concurrency where possible. Possibility arises when the applicative combinators are used, which guarantee that the whole applicative computation does not depend on previous effectful operations. The Haxl implementation however does violate a written rule of the Applicative type class, namely the fact that if `f` is an instance of both `Applicative` and `Monad`, then `(<*>) = ap`. This rule however forbids the exploitation of concurrency in the `Applicative` instance. This is also noted in the paper as a consciously committed sin ([MBCP14, sec. 5.4]). An in depth discussion of this problem can be found in section 3.7.

### 5.3.5 Free Applicative Functors

Capriotti et al present the free applicative functor that rises from a functor in [CK14]. Like free monads, free applicative functors can be used to define embedded DSLs easily and separate program construction from interpretation. While applicative functors are less expressive than monads, this limitation is the added benefit one gains by using free applicative functors instead of free monads. By not being able to depend on previous effectful values, essentially provided by `\((>>=)\)/join` from the Monad class, programs written in a DSL from a free applicative functor can be statically analyzed *without* executing the program.

# 6 Conclusion & Future Work

We define an embedded DSL in Scala, allowing developers to write programs that work on encrypted data without requiring cryptographic knowledge. We first presented a simplified version of the DSL using the well-known free monad scheme and discussed the drawbacks inherent to the underlying monad abstraction i.e., the inability of static analysis and the forced sequential nature of interpretation. Improving on this, we presented another version of the DSL, this time based on free applicative functors, which remedies the drawbacks. Using the DSL based on free applicative functors, we presented an interpreter that allows implicit parallelism and static analysis without execution of the program. Finally, we regain the power of the monadic interface, by combining the two DSL variants, allowing for the embedding of programs written in the free applicative functor style into larger programs using the free monads variant.

The evaluation shows the advantage of implicit parallelism as well as the performance improvements from the possibility to perform static analysis and program transformation of programs written in the applicative DSL. In different case studies, we show that our DSL integrates transparently with other technologies like Esper for complex event processing and RxScala for reactive programming. In another case study we evaluate the performance of a word count program written entirely in the DSL to count the words in a file on disk. Performance results show that while the overhead with encrypted data is significant, the use of it is still practical.

## 6.1 Future Work

For free monads, there are at least two optimization, one analog to the difference list technique for lists, making use of CPS [Voi08]. The problem of the CPS optimization is, that introspection of intermediate values becomes very expensive. Therefore [vdPK14] presents another technique, which exhibits better performance for scenarios where we have a mixture of introspection and program construction. On the other hand, for free applicative functors, there is not yet any work on optimizing the representation and the implementation in Scalaz has the unfortunate quality to overflow the stack rather fast if the size of the free applicative structure is large.

Furthermore, the DSL is currently limited to using encrypted integers and strings. For integers, the main limitation is that common calculations typically involve the use of percentages and floating point numbers. While it is possible to e.g., encode a fractional representation, the result requires a large number of conversions even for simple operations. Naive adoption of the common techniques for floating point representation leads to an extremely high performance loss, because intermediate calculation and normalization steps require the use of different operators. In turn, each of those requires a different partial homomorphic encryption scheme and therefore a large number of conversions have to be performed, which might involve network communication in the distributed scenario. Finding an efficient way to work with floating point numbers in the context of partial homomorphic encryption schemes is therefore still an open research problem.

# Appendices

# A Complete File Listings

## A.1 Word Count

The original text can be found online. The slightly adapted file used in the “Word Count” case study section 4.3.1 is shown in listing A.1 and listing A.2.

```
2.5. The GHCi Debugger

GHCi contains a simple imperative-style debugger in which you can stop
a running computation in order to examine the values of variables. The
debugger is integrated into GHCi, and is turned on by default: no
flags are required to enable the debugging facilities. There is one
major restriction: breakpoints and single-stepping are only available
in interpreted modules; compiled code is invisible to the debugger[5].

The debugger provides the following:

    The ability to set a breakpoint on a function definition or
    expression in the program. When the function is called, or the
    expression evaluated, GHCi suspends execution and returns to the
    prompt, where you can inspect the values of local variables before
    continuing with the execution.

    Execution can be single-stepped: the evaluator will suspend
    execution approximately after every reduction, allowing local
    variables to be inspected. This is equivalent to setting a
    breakpoint at every point in the program.

    Execution can take place in tracing mode, in which the evaluator
    remembers each evaluation step as it happens, but doesn't suspend
    execution until an actual breakpoint is reached. When this
    happens, the history of evaluation steps can be inspected.

    Exceptions (e.g. pattern matching failure and error) can be
    treated as breakpoints, to help locate the source of an exception
    in the program.

There is currently no support for obtaining a "stack trace", but the
tracing and history features provide a useful second-best, which will
often be enough to establish the context of an error. For instance, it
is possible to break automatically when an exception is thrown, even
if it is thrown from within compiled code (see Section 2.5.6,
"Debugging exceptions").

2.5.1. Breakpoints and inspecting variables

Let's use quicksort as a running example. Here's the code:

qsort [] = []
qsort (a:as) = qsort left ++ [a] ++ qsort right
  where (left,right) = (filter (<=a) as, filter (>a) as)
```

**Listing A.1:** Full text from the Word Count Case Study. Part 1/2.

```
46
47  main = print (qsort [8, 4, 0, 3, 1, 23, 11, 18])
48
49  First, load the module into GHCi:
50
51  Prelude> :l qsort.hs
52  [1 of 1] Compiling Main             ( qsort.hs, interpreted )
53  Ok, modules loaded: Main.
54  *Main>
55
56
57  Now, let's set a breakpoint on the right-hand-side of the second
58  equation of qsort:
59
60  *Main> :break 2
61  Breakpoint 0 activated at qsort.hs:2:15-46
62  *Main>
63
64  The command :break 2 sets a breakpoint on line 2 of the most
65  recently-loaded module, in this case qsort.hs. Specifically, it picks
66  the leftmost complete subexpression on that line on which to set the
67  breakpoint, which in this case is the expression (qsort left ++ [a] ++
68  qsort right).
69
70  Now, we run the program:
71
72  *Main> main
73  Stopped at qsort.hs:2:15-46
74  _result :: [a]
75  a :: a
76  left :: [a]
77  right :: [a]
78  [qsort.hs:2:15-46] *Main>
79
80  Execution has stopped at the breakpoint. The prompt has changed to
81  indicate that we are currently stopped at a breakpoint, and the
82  location: [qsort.hs:2:15-46]. To further clarify the location, we can
83  use the :list command:
84
85  j[qsort.hs:2:15-46] *Main> :list
86  1  qsort [] = []
87  2  qsort (a:as) = qsort left ++ [a] ++ qsort right
88  3    where (left,right) = (filter (<=a) as, filter (>a) as)
```

**Listing A.2:** Full text from the Word Count Case Study. Part 2/2.

# Bibliography


[App] The ApplicativeDo language pragma for ghc. https://ghc.haskell.org/trac/ghc/wiki/ApplicativeDo. Accessed: 2015-08-04.

[Awo10] Steve Awodey. *Category theory*. Oxford University Press, 2010.

[CH00] Koen Claessen and John Hughes. QuickCheck. *ACM SIGPLAN Notices*, 35(9):268–279, sep 2000.

[CK14] Paolo Capriotti and Ambrus Kaposi. Free applicative functors. *Electronic Proceedings in Theoretical Computer Science*, 153:2–30, jun 2014.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-c programs. In *Lecture Notes in Computer Science*, pages 168–176. Springer Science + Business Media, 2004.

[DG10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. *Commun. ACM*, 53(1):72–77, January 2010.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC '09*. ACM Press, 2009.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology – CRYPTO 2012*, pages 850–867. Springer Science + Business Media, 2012.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC '87*. ACM Press, 1987.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[HFKV12] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ansi c. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 772–783, New York, NY, USA, 2012. ACM.

[JD93] Mark P. Jones and Luck Duponcheel. Composing Monads. Technical report, 1993.

[JSSSE14] Julian James Stephen, Savvas Savvides, Russell Seidel, and Patrick Eugster. Program analysis for secure big data processing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 277–288, New York, NY, USA, 2014. ACM.

[KSS13] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell - Haskell '13*. Association for Computing Machinery (ACM), 2013.

[KW93] David J. King and Philip Wadler. Combining monads. In *Functional Programming, Glasgow 1992*, pages 134–143. Springer Science + Business Media, 1993.

[Mal11] Lior Malka. Vmcrypt: Modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 715–724, New York, NY, USA, 2011. ACM.

[MBCP14] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. There is no fork. *ACM SIGPLAN Notices*, 49(9):325–337, aug 2014.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[MSSZ12] John C. Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. Information-flow control for programming on encrypted data. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 45–60, Washington, DC, USA, 2012. IEEE Computer Society.

[Oa04] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[OMO10] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *ACM SIGPLAN Notices*, 45(10):341, oct 2010.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*. ACM Press, 2008.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, pages 223–238, Berlin, Heidelberg, 1999. Springer-Verlag.

[PRZB11] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[Rab05] Michael O. Rabin. How To Exchange Secrets with Oblivious Transfer. Cryptology ePrint Archive, Report 2005/187, June 2005.

[SA07] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop - Haskell '07*. Association for Computing Machinery (ACM), 2007.

[Sny14] Peter Snyder. Yao's garbled circuits: Recent directions and implementations. 2014.

[Swi08] Wouter Swierstra. Data types a la carte. *Journal of Functional Programming*, 18(04):423–436, 2008.

[TKMZ13] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. *Proceedings of the VLDB Endowment*, 6(5):289–300, mar 2013.

[TLMM13] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. Mrcrypt: Static analysis for secure cloud computations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 271–286, New York, NY, USA, 2013. ACM.

[TPC] TPC-H benchmark specification. http://www.tpc.org/tpch/. Accessed: 2015-07-04.

[vdPK14] Atze van der Ploeg and Oleg Kiselyov. Reflection without remorse. *ACM SIGPLAN Notices*, 49(12):133–144, sep 2014.

[Voi08] Janis Voigtländer. Asymptotic Improvement of Computations over Free Monads. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction*, volume 5133 of *Lecture Notes in Computer Science*, chapter 20, pages 388–403. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.