Universiteit Utrecht

Department of Mathematics

Master Thesis

Formalisation of Cryptographic Proofs in Agda

Author:
Anton Golov

Supervisors:
Dr. Jaap van Oosten
Dr. Wouter Swierstra
Victor Cacciari Miraldo

December 2017-August 2018

Abstract

The game-based style of proofs [BR06, Sho04] is often used in cryptography to prove properties of cryptographic primitives, such as the security of an encryption scheme. Given the importance of cryptography in the modern world, there is considerable value in being able to verify these proofs automatically. In this thesis, we develop a system for expressing proofs of this form in the Agda programming language.

Contents

Foreword

The goal of this research project was initially to develop a system for cryptographic proofs in the Agda programming language. During the process, it became clear that the construction of the system as a whole would not be feasible, and the project thus became a number of experiments in Agda that were each intended to investigate a particular feature of the design space.

The purpose of this thesis is to write up the results of these experiments and show how they can be brought together. The code is available on GitHub[1], and the text will contain references to the files where relevant.[2]

Since formalisation in Agda is the point of the research, I assume that the reader is familiar with the Agda programming language. There are several good introductions online, for example by Ulf Norell [Nor09]. For later chapters, a passing familiarity with category theory is also beneficial.

I would like to thank dr. Wouter Swierstra for agreeing to be my supervisor (despite my thesis being in maths), and, together with Victor Cacciari Miraldo, for their time and advice throughout the project. I would also like to thank dr. Jaap van Oosten, my tutor and second supervisor, for allowing me to do this project (despite my master's being in maths), and for his guidance throughout the years of my master's degree.

Anton Golov
27 July, 2018

[1] `https://github.com/jesyspa/master-thesis`

[2] All references are to Agda files in the above repository, relative to the `src` directory. For example `Probability/Class` refers to the file `https://github.com/jesyspa/master-thesis/tree/master/src/Probability/Class.agda`.

Chapter 1

Introduction

Cryptography plays an essential role in the modern world: we trust that cryptographic primitives will prevent unauthorised access to our data, securing our online activity, banking information, and whatever else we wish to keep private. As such, it is important to be able to verify that such primitives provide the guarantees they promise.

These guarantees are typically phrased as statements that no program can distinguish between two given possibilities. The standard example, which we will return to often, is that a good encryption scheme should not allow an attacker to tell what message had been encrypted, even if the set of possible messages is very small.

Following Bellare and Rogaway [BR06], we will frame questions of this form as games between a challenger and an adversary. The challenger represents the system we want to prove secure, while the adversary represents an attacker. The challenger poses a challenge to the adversary, and the adversary wins if it can correctly answer the challenge. If we can show that no adversary can reliably win the game, we conclude that our system is secure. On the other hand, we can prove a system to be vulnerable by exhibiting an adversary that has a winning strategy.

To apply this to the aforementioned example, let us specify it in a more formal manner. Let Alice be the challenger and Eve be the adversary. The protocol they follow to play the game is as follows: Eve gives Alice two messages, m_1 and m_2. Alice generates an encryption key and uses it to encrypt one of the messages, chosen at random. Alice gives Eve the resulting ciphertext and poses the challenge: did she encrypt m_1 or m_2? Eve wins if her answer is correct. This game is known as IND-EAV, indistinguishability in the presence of an eavesdropper.

Eve can definitely win half of her games, just by choosing an answer at random. How much better Eve can do is called her *advantage*. In order to show that an encryption scheme is secure, we must show that any adversary's advantage is close to zero. In order to show that a scheme is not secure, we must show that there exists some adversary that has high advantage.

When we want to put an upper bound on the advantage, we could analyse the game and attempt to derive this bound directly. However, it is often simpler to modify the game slightly and show that this modification does not change the advantage considerably. We say that two games between which the difference in advantage is at most ϵ are ϵ-indistinguishable. By constructing a sequence of ϵ-indistinguishable games, we can relate

our initial (complicated) game to a much simpler one, where computing the advantage of the adversary is trivial.

1.1 Games as Programs

We can regard a game as a sequence of actions performed by the players. Players may perform computations, generate random bits, and make use of memory. As such, a description of a game or player can be seen as an imperative program in a non-determinsitic, stateful programming language, the instructions of which correspond to the actions that players may take.

Representing imperative programs in a functional language is a well-studied problem [Wad95], and can be solved using a monad that has operations corresponding to the imperative instructions. We will show how this monad can be constructed explicitly in chapter 2. For now, we will assume that there is a monad CryptoExpr ST that supports the following operations, where ST is the type of the state that the players have access to:[1]

```
uniform   : (n : ℕ) → CryptoExpr ST (BitVec n)
coin      :           CryptoExpr ST Bool
set-state : ST      → CryptoExpr ST ⊤
get-state :           CryptoExpr ST ST
```

A term of type CryptoExpr ST A represents a computation that can generate random bits and store and retrieve values of type ST, and that has a result of type A. We include both uniform and coin for the sake of convenience, although one could be defined in terms of the other.

We will now use this monad to formally specify an encryption scheme, as well as a game between a challenger and an adversary that expresses a security property of this scheme. We use the same example as above, indistinguishability in the presence of an eavesdropper.

Let us begin by assuming that we have some type K for our keys, PT for our plaintext messages, and CT for our ciphertext messages. To define an encryption scheme, we must give the algorithm for generating a new key and for encrypting a message with a given key. We can express this in Agda as a record.[2]

```
record EncScheme : Set₁ where
  field
    keygen  : ∀{st} → CryptoExpr st K
    encrypt : ∀{st} → K → PT → CryptoExpr st CT
```

We express the fact that the encryption scheme should be stateless by requiring that it work for *any* state type. This allows us to let the adversary choose the state type, as we will see shortly.

[1] c.f. `Syntactic/CryptoExpr`. Note that our implementation uses techniques discussed in chapter 5 for this definition. For a more direct implementation, but without support for state, c.f. `Crypto/Syntax`.

[2] c.f. `Crypto/Schemes`

The adversary is given the chance to act twice during the game, first to generate two plaintext messages, and then to guess which message has been encrypted. We again represent this as a record, parametrised by the type of state ST that the adversary uses.[3]

```
record Adversary (ST : Set) : Set where
  field
    A₁ : CryptoExpr ST (PT × PT)
    A₂ : CT → CryptoExpr ST Bool
```

It may seem strange that the adversary is not given access the plaintext messages it generated earlier when it is asked to decide which was encrypted. This is because the adversary can use get-state and set-state to store these messages if it needs to. We could have made this flow of data explicit, but since we are modelling an imperative program that can have internal state, this approach feels more natural.

Now we can introduce the game itself. As before, we let the adversary pick two messages, generate a key, encrypt one of the messages based on a coin flip, and then let the adversary guess which one it was. Altogether, this is a probabilistic computation that returns true iff the adversary wins.[4]

```
IND-EAV : EncScheme → Adversary ST → CryptoExpr ST Bool
IND-EAV enc adv = do
  m₁ , m₂ ← A₁ adv
  k ← keygen enc
  b ← coin
  let pt = if b then m₁ else m₂
  ct ← encrypt enc k pt
  b′ ← A₂ adv ct
  return $ b == b′
```

If we now fix an encryption enc and take an arbitrary adversary adv, we can reason about the probability that evaluating IND-EAV enc adv returns true. If we can bound this probability by $\frac{1}{2}$, then we can conclude that the encryption scheme enc is secure against an eavesdropper attack. On the other hand, if we can find an adversary that wins with high probability, we can conclude that the scheme is vulnerable against this attack.

1.2 Example: One-Time Pad (IND-EAV)

Let us see how we can reason about a game like the one demonstrated in the previous section. For this example, we will use the One-Time Pad encryption scheme, which works by XORing the message with a pre-determined key of the same length. Formally, this can be described as follows. Fix an $n : \mathbb{N}$. To generate the key, we take an n-bit vector uniformly at random. To encrypt some message m of length n with a key k, take the bitwise XOR of m and n. In Agda, this can be expressed as follows.[5]

[3] c.f. `Crypto/EAV`.
[4] c.f. `Crypto/EAV` again.
[5] c.f. `Crypto/OTP`.

```
OTP : EncScheme
keygen OTP = uniform n
encrypt OTP key msg = return $ key ⊗ msg
```

We can now rewrite this game to show that no matter the adversary chosen, it is indistinguishable from a coin flip. Let us start by writing out the game, filling in the definition of the encryption scheme:

```
IND-EAV-OTP₁ : Adversary ST → CryptoExpr ST Bool
IND-EAV-OTP₁ adv = do
  m₁ , m₂ ← A₁ adv
  k ← uniform n
  b ← coin
  let pt = if b then m₁ else m₂
  ct ← return $ k ⊗ pt
  b′ ← A₂ adv ct
  return $ b == b′
```

First of all, we note that k and b are independent random variables and may thus be reversed. By the monad laws, a bind followed by a return can be written as an fmap. This gives us the following code:

```
IND-EAV-OTP₂ adv = do
  m₁ , m₂ ← A₁ adv
  b ← coin
  let pt = if b then m₁ else m₂
  ct ← fmap (λ k → k ⊗ pt) (uniform n)
  b′ ← A₂ adv ct
  return $ b == b′
```

We can show that $\lambda\ k \rightarrow k \otimes m$ is a bijection for any m : BitVec n, and since applying a bijection to a uniform distribution produces another uniform distribution, we may omit the fmap, giving us the following game:

```
IND-EAV-OTP₃ adv = do
  m₁ , m₂ ← A₁ adv
  b ← coin
  ct ← uniform n
  b′ ← A₂ adv ct
  return $ b == b′
```

We now see that b and b' are independent random variables and can reorder these, as well. We can also once more rewrite a bind followed by a return as an application of fmap:

```
IND-EAV-OTP₄ adv = do
  m₁ , m₂ ← A₁ adv
  ct ← uniform n
  b′ ← A₂ adv ct
  fmap (λ b → b == b′) coin
```

Finally, we can show that $\lambda\ b\ \rightarrow\ b == b'$ to be a bijection as well, giving us the last game in the sequence:

```
IND-EAV-OTP₅ adv = do
  m₁ , m₂ ← A₁ adv
  ct ← uniform n
  b′ ← A₂ adv ct
  coin
```

Since the outcome of IND-EAV-OTP$_5$ is independent of adv, it is indistinguishable from coin. Since the games are indistinguishable, the probability of drawing true from IND-EAV-OTP$_1$ is the same as from IND-EAV-OTP$_5$. Since the advantage of any adversary against IND-EAV-OTP$_5$ is 0, it follows that it has advantage 0 against IND-EAV-OTP$_1$ as well, and thus we have shown that OTP is secure against an eavesdropper attack.[6]

1.3 Oracles

The above lets us reason about adversaries expressed in terms of the basic language of non-determinsitic stateful computation. This is useful by itself, but by restricting adversaries to this language, we are only considering the weakest kind of adversary possible. If we want to strengthen our results, we need to develop a way of giving adversaries access to computations they cannot perform themselves.

To give an example, a straightforward strengthening of the IND-EAV game is to give the adversary access to the encryption function used by the challenger. Since this computation depends on the key[7], this cannot be expressed directly as a computation performed by the adversary. Instead, we must give the adversary black-box access to the encryption function.

A function provided to the adversary in this opaque way is called an *oracle*. Oracles have all the power that the challenger and adversary have: they may generate random bitstrings and have access to mutable state. However, the other players cannot inspect the code or state of the oracle. This lets us precisely control the power of the adversary by adjusting the information provided by the oracle. Given the importance of this, a flexible and easy-to-use system for oracles has been a central focus of this work.

For the moment, we will assume that there are two operations provided by the oracle: a way to initialise the oracle state with some value of type OracleState, and a way to query the oracle function using an OracleArg argument to get an OracleResult response. The types in question will depend on the game being played. We can represent this in code by assumpting that CryptoExpr supports an additional two operations:[8]

```
init-oracle : OracleState → CryptoExpr ST ⊤
call-oracle : OracleArg   → CryptoExpr ST OracleResult
```

In chapter 5 we will show how this approach can be generalised to allow oracles with different signatures in a straightforward manner, which is a noteworthy feature of our system.

[6] c.f. `Crypto/OTP`.

[7] Which the adversary should *not* have access to!

[8] c.f. `Syntactic/OracleExpr` and `Syntactic/OracleEval`.

We can now express a game that expresses a stronger security condition than IND-EAV. In IND-EAV, we assumed that the adversary could choose two messages for the challenger to encrypt, but could not perform the encryption. If the adversary also has the power to encrypt messages of its choice, the game is known as indistinguishability under a chosen plaintext attack, abbreviated IND-CPA. The name comes from the fact that the adversary is allowed to choose one or more plaintext messages to be encrypted by the oracle. Apart from the fact that the challenger has to initialise the oracle and the adversary may query it, the game is identical.

Let us now look at the code. Since the oracle must have the key to encrypt messages, OracleState $=$ K. The query takes a plaintext and yields a ciphertext, so OracleArg $=$ PT and OracleResult $=$ CT. The code is as follows:

```
IND-CPA : EncScheme → Adversary ST
          → CryptoExpr ST Bool
IND-CPA enc adv = do
  m₁ , m₂ ← A₁ adv
  k ← keygen enc
  init-oracle k
  b ← coin
  let pt = if b then m₁ else m₂
  ct ← encrypt enc k pt
  b′ ← A₂ adv ct
  return $ b == b′
```

However, we are not yet done. Apart from specifying the interaction between the challenger and the adversary, we must also specify the behaviour of the oracle. Just like the challenger and adversary, the oracle may store some state, in this case the key. In this case, the oracle definition is straightforward: initialisation stores the encryption key, and a query takes a plaintext message and returns it encrypted with the stored key. This can be expressed in code as follows:

```
init-oracle-impl : K → CryptoExpr K ⊤
init-oracle-impl = set-state

call-oracle-impl : PT → CryptoExpr K CT
call-oracle-impl pt = do
  k ← get-state
  encrypt enc k pt
```

We have now specified the IND-CPA game, just as we had specified IND-EAV earlier, and can reason about it in the same ways, by fixing an encryption scheme and considering an arbitrary adversary. Our goal is again to either upper bound the probability of any adversary winning, or show that an adversary exists that wins the game with certainty.

If an adversary wins the IND-EAV game against some encryption scheme *enc*, the same adversary can win the IND-CPA game against *enc* by ignoring the oracle. Conversely, any game that is secure against IND-CPA is also secure against IND-EAV. We can thus regard IND-CPA as a stronger claim about an encryption scheme, and we will see that it is strictly stronger by showing that the One-Time Pad scheme is not secure against it.

Before we go on, let us note that we have not specified how the implementation of an oracle can be combined with the implementation of the game, since one has state type ST

and the other has state type K. The full details will be worked out in section 2.2; for now, it suffices to remark that we can store both at once in a state of type $ST \times K$. We will use get-adv-state and get-oracle-state instead of get-state (and analogously for set-state) when we want to disambiguate which component we are referring to.

1.4 Example: One-Time Pad (IND-CPA)

Let us now show that the One-Time Pad encryption scheme presented above is not secure with respect to the IND-CPA game by constructing an adversary that wins the game with probability 1. As before, we fix an $n : \mathbb{N}$ and set $K = CT = PT =$ BitVec n. We assume $n > 0$, since otherwise there exists only one plaintext message.

The encryption scheme can be broken using the fact that encrypt OTP is deterministic. Since the adversary has access to the encryption function, it can use the oracle to find the ciphertext that corresponds to each message. As long as the two chosen messages m_1 and m_2 are distinct, the challenger chose to encrypt m_1 iff the given ciphertext is equal to the ciphertext of m_1.

In code, we need to choose two specific messages. Let all-zero n and all-one n be the n-bit vectors that consist entirely of zeroes and ones respectively. They are distinct, since $n > 0$. To decide which message the challenger chose to encrypt, we query the oracle to encrypt all-zero n, and respond with true iff this is equal to the given ciphertext.

This can be expressed in Agda as follows. Note that since we have chosen our messages a priori, we do not need to store any state, and can choose $ST = \top$.

```
IND-CPA-OTP-ADV : Adversary ⊤
A₁ IND-CPA-OTP-ADV = return (all-zero n , all-one n)
A₂ IND-CPA-OTP-ADV ct = do
  r ← call-oracle (all-zero n)
  return $ ct == r
```

Let us write out IND-CPA with OTP, IND-CPA-OTP-ADV, and the definition of the oracle filled in. As we mentioned, the combination of the adversary and oracle state results in the state type being $\top \times K$.

It may seem strange to go to such lengths to define the oracle separately, only to immediately inline it when we begin with the proof. However, recall that the purpose of the separation was to prevent the adversary from accessing the oracle state. Since we have chosen an adversary that does not does this, the separation does not play any further role in this example.

The resulting code is as follows:

```
IND-CPA-OTP₁ : CryptoExpr (⊤ × K) Bool
IND-CPA-OTP₁ = do
  m₁ , m₂ ← return $ all-zero n , all-one n
  k ← uniform n
  set-oracle-state k
  b ← coin
  let pt = if b then m₁ else m₂
  ct ← return $ k ⊗ pt
```

```
  k' ← get-oracle-state
  b' ← return $ ct == k' ⊗ (all-zero n)
  return $ b == b'
```

We use set-oracle-state instead of set-state here to disambiguate whose state we are talking about. Since k is the last value stored in the oracle state, we know $k = k'$. We can also inline the definitions of m_1 and m_2. This gives us the following game, indistinguishable from the previous:

```
IND-CPA-OTP₂ = do
  k ← uniform n
  b ← coin
  let pt = if b then m₁ else m₂
  ct ← return $ k ⊗ pt
  b' ← return $ ct == k ⊗ (all-zero n)
  return $ b == b'
```

By the monad laws, we can rewrite this game as follows, translating return statements into let bindings and inlining the definition of ct:

```
IND-CPA-OTP₃ = do
  k ← uniform n
  b ← coin
  let pt = if b then all-zero n else all-one n
      b' = k ⊗ pt == k ⊗ (all-zero n)
  return $ b == b'
```

We know that $k \otimes \cdot$ is an injective function, so $k \otimes v == k \otimes w$ holds iff $v == w$ holds. We can thus simplify the above to the following:

```
IND-CPA-OTP₄ = do
  k ← uniform n
  b ← coin
  let pt = if b then all-zero n else all-one n
      b' = pt == all-zero n
  return $ b == b'
```

If b is true, then we compare all-zero n to all-zero n and get true, so the expression as a whole is true. On the other hand, if b is false, we compare all-one n to all-zero n and get false, so the expression as a whole is true as well. It follows that this game always yields true, and so this adversary is guaranteed to win it.

1.5 Weaker Notions of Security

Having introduced oracles to allow for the strengthening of our security conditions, let us consider the problem of expressing weaker notions of security. So far, we have looked at concrete, information-theoretic security: the error term ϵ gave an explicit bound on the advantage of the adversary, and our results followed from information theory. In this section, we will look at asymptotic and computational security.

A strong motivation to consider these topics arises in public key cryptography. Instead of generating a single key that is used for both encryption and decryption, a public key encryption scheme generates a public key that can be used for encrypting messages and a private key that is needed to decrypt them. As the name suggests, the public key is available to both the challenger and adversary, while only the challenger has the private key.

In this setting, the adversary can guess a private key and check whether it corresponds to the challenger's public key. This improves the advantage of the adversary somewhat, since the private key makes it possible to decrypt the message. If the adversary is allowed to attempt this repeatedly, then the advantage will compound. Neverthelesss, public key cryptography is effective in practice: this is because the probability of guessing the private correctly is so small, the adversary would have to make a very large number of attempts, making the attack impractical.

It is thus desirable to be able to express bounds on the resources an adversary may use. We will see a simple form of this in section 2.3, where we will show how we can bound the number of oracle queries an adversary makes.

In practice, we generally cannot say that an attacker will be able to perform at most some number of queries. Instead, some encryption schemes allow the user to choose the key length they want, letting them increase security as necessary. In this setting, we want to show that the advantage of the adversary can be made arbitrarily small as the key length goes to infinity. This also suggests a new class of constraints on the adversary: we can require that the adversary only uses an amount of resources polynomial in the key length.

Finally, arguments in cryptography often rely on conjectures such as $P \neq NP$. In order to be able to formalise these arguments we need to be able to represent these conjectures, and it turns out that this question is highly relevant to the question of asymptotic bounds.

Ensuring that the above notions can be represented within our system was a key design constraint for us. We will discuss how we achieve them in detail in section 3.5, but they are worth keeping in mind throughout the development.

1.6 Generalised Games

In the introduction, we described games as being a formalisation of the notion that no adversary can distinguish between two situations. We then introduced the challenger, who would decide which of the situations occured. However, we could have also posed the question more directly: is there any adversary that can reliably give different results for the two situations?

The benefit of this formulation is that a claim phrased this way is much easier to reuse in a portion of another game. If we know that no adversary can distinguish between games X and Y, then we can replace X by Y in the context of another game and know that this is a sound rewriting step. This also relates to the question of how we can express security assumptions that we discussed in the previous section.

The games we have seen so far can all be expressed in this manner. In the case of IND-EAV and IND-CPA, instead of asking whether the adversary can tell which message was encrypted, we ask whether the adversary can tell whether they are in a game where

the encrypted message is always the first or always the second. These two games that correspond to IND-EAV are as follows:

```
IND-EAV₁ : EncScheme → Adversary ST → CryptoExpr ST Bool
IND-EAV₁ enc adv = do
  m , _ ← A₁ adv
  k ← keygen enc
  ct ← encrypt adv k m
  A₂ adv ct
IND-EAV₂ : EncScheme → Adversary ST → CryptoExpr ST Bool
IND-EAV₂ enc adv = do
  _ , m ← A₁ adv
  k ← keygen enc
  ct ← encrypt adv k m
  A₂ adv ct
```

If an adversary *adv* can distinguish between IND-EAV$_1$ *enc* and IND-EAV$_2$ *enc* for some encryption scheme *enc*, then we can construct an adversary that reliably wins IND-EAV *enc* by noting that IND-EAV can be rewritten as the following game:

```
IND-EAV : EncScheme → Adversary ST → CryptoExpr ST Bool
IND-EAV enc adv = do
  b ← coin
  b′ ← if b then IND-EAV₁ enc adv else IND-EAV₂ enc adv
  return $ b ≡ b′
```

As such, this really is a generalisation of the notion of games we had considered up to this point.

1.7 Summary

This concludes our exploration of the problem space. We have seen that to express game-based proofs we need to be able to represent games, and need some set of rules specifying when two games are ϵ-indistinguishable. If we wish to express stronger security guarantees then our system must support the use of oracles, while expressing weaker guarantees requires a way of imposing constraints on the class of adversaries. In the next two chapters we will lay out the groundwork of our system, in chapter 4 we will discuss how we can show that our system is correct, and finally in chapters 5 and 6 we will show some techniques that can aid in the implementation of the system.

We are not the first to approach this problem. Several systems already exist for the formalisation of cryptographical proofs, EasyCrypt[9] and FCF [PM14] being two notable examples. In this work, we have focused in particular on how we can leverage the power of dependent types to simplify the problem. The primary difficulty of building such a proof system is the verbosity of the resulting proofs:[10] steps that were trivial on paper may nevertheless require extensive proof in a formal setting. We will discuss these aspects of our approach, and what can be done to improve on this point, in chapter 7.

[9] `http://www.easycrypt.info`

[10] c.f. `Crypto/OTP`.

Chapter 2

Representing Games

In chapter 1, we modelled games as imperative programs, represented by terms in a suitable monad. We had assumed the existence of a monad that supported the operations we require. In this chapter, we will see how the free monad construction [Swi08, McB15] can be used to define this monad explicitly. A monad constructed this way supports all the required operations, but treats them syntactically, without giving them any further interpretation.

Given that there exist monads both for stateful and probabilistic [EK06] computations, a natural question is why we do not define our games in terms of those. This would be possible, but makes the subsequent development considerably harder. The syntactic approach we take allows us to inspect our games with greater detail. For example, there is no way to check whether a term in the state monad makes use of the state, while being able to do so is convenient for our purposes. Furthermore, as the term 'free monad' suggests, we can map our games into any monad that supports the required operations, and so we lose nothing by delaying this interpretation until it is unavoidable.

We will start by constructing a monad for games that do not make use of an oracle. In this case, the only operations we require are uniform, get-state and set-state from chapter 1. We will then show how the same techniques can be used to extend this language to support games that do make use of an oracle. Finally, we will show how the syntactic nature of this representation can be used to impose constraints on an adversary.

2.1 Free Monads

From a syntactic point of view, a game with result type A can do one of two things: immediately yield a value of type A, or execute some command and then map the response to another game with result type A. Treating this as an inductive definition is the key insight of the free monad construction [McB15]. For our three commands, the free monad can be defined as follows:[1]

```
data CryptoExpr (ST : Set) : Set → Set where
  Return    : A                                  → CryptoExpr ST A
```

[1] c.f. `Syntactic/CryptoExpr`; again, note that in the code, these are defined using techniques from chapter 5.

```
Uniform  : (n : ℕ) → (BitVec n → CryptoExpr ST A) → CryptoExpr ST A
GetState :           (ST       → CryptoExpr ST A) → CryptoExpr ST A
SetState : ST      → (⊤        → CryptoExpr ST A) → CryptoExpr ST A
```

When talking about games, we will refer to the constructor and first argument as the *command*, and the second argument as the *response handler* or *continuation*. Note that since $\top \to X$ is isomorphic to X for every type X, we could have used $ST \to$ CryptoExpr $ST\ A \to$ CryptoExpr $ST\ A$ as the type of SetState. However, we use the more verbose form for the sake of consistency with the other constructors.

Although this definition is entirely syntactic, there is an intended semantic meaning we keep in mind: Uniform represents the generation of a uniformly random bit vector, GetState represents a read from the state and SetState represents a write to the state. We will only define this interpretation in chapter 4, but provides a useful intuition for the constructions we do in this chapter and the next.

We can define the monadic actions uniform, set-state and get-state as terms in the CryptoExpr ST monad by passing Return as the response handler:[2]

```
uniform : (n : ℕ) → CryptoExpr ST (BitVec n)
uniform n = Uniform n Return

get-state : CryptoExpr ST ST
get-state = GetState Return

set-state : ST → CryptoExpr ST ⊤
set-state st = SetState st Return
```

In order to show that CryptoExpr ST is indeed a monad, we take return $=$ Return and define fmap and $\gg\!=$ as follows:[3]

```
fmap : (A → B) → CryptoExpr ST A → CryptoExpr ST B
fmap f (Return a)        = Return (f a)
fmap f (Uniform n cont)  = Uniform n  λ v → fmap f (cont v)
fmap f (GetState cont)   = GetState   λ st → fmap f (cont st)
fmap f (SetState st cont) = SetState st λ t → fmap f (cont t)

_>>=_ : CryptoExpr ST A → (A → CryptoExpr ST B) → CryptoExpr ST B
Return a        >>= f = f a
Uniform n cont  >>= f = Uniform n  λ v →  cont v >>= f
GetState cont   >>= f = GetState   λ st → cont st >>= f
SetState st cont >>= f = SetState st λ t →  cont t >>= f
```

We will see how we can avoid the repetitiveness of these definitions in chapter 5. For convenience, we also define some derived operations:

```
coin : CryptoExpr ST Bool
coin = fmap head (uniform 1)

modify : (ST → ST) → CryptoExpr ST ST
modify f = do
```

[2] c.f. `Syntactic/CryptoExprHelpers`.

[3] Since we use the techniques outlined in chapter 5, these functions are generated for us; c.f. `Syntactic/CommandStructure`.

```
st ← get-state
let st′ = f st
set-state st′
return st′
```

Since proving the equality between two games involves proving that their response handlers are equal, we need functional extensionality to prove that CryptoExpr ST satisfies the monad laws. Except for this, the proofs are straightforward.

This concludes the definition of CryptoExpr ST, bringing us again to where we were at the beginning of section 1.1, but this time with no assumptions. The motivated reader may wish to go back and check that the IND-EAV game presented there can be expressed in this system.

2.2 Representing Oracles

We will now define another monad for games, which will allow us to express games where the adversary has access to an oracle. As in section 1.3, we will assume that oracles support two operations: init-oracle to initialise the oracle state and call-oracle to perform a query to the oracle. We will consider how this set of operations can be extended in chapter 5.

Recall that init-oracle takes an OracleState value to initialise the oracle with and gives no response, while call-oracle takes an OracleArg and responds with an OracleResult. We define OracleExpr ST, the type of games that use an oracle, as follows:[4]

```
data OracleExpr (ST : Set) : Set → Set where
  Return     : A                                                    → OracleExpr ST A
  Uniform    : (n : ℕ)      → (BitVec n      → OracleExpr ST A) → OracleExpr ST A
  GetState   :                  (ST           → OracleExpr ST A) → OracleExpr ST A
  SetState   : ST           → (⊤            → OracleExpr ST A) → OracleExpr ST A
  InitOracle : OracleState  → (⊤            → OracleExpr ST A) → OracleExpr ST A
  CallOracle : OracleArg    → (OracleResult → OracleExpr ST A) → OracleExpr ST A
```

We could have used different names for the constructors that were already used in CryptoExpr, but since the behaviour in each case is practically identical, we expect the ambiguity to not cause any issues.

As in the case of CryptoExpr, we can define uniform, get-state, and set-state, and we can now also define init-oracle and call-oracle in the same way. Similarly, the definitions of fmap and $>>=$ are straightforward extensions of those for CryptoExpr.

We can now specify games such as IND-CPA from section 1.3, but we must also be able to define the behaviour of the oracles themselves. We do this much the same way we specify adversaries, by defining a record that has interpretations for the operations. Note that we define oracles using the CryptoExpr monad, not OracleExpr: the latter would allow an oracle to call itself, potentially leading to a non-terminating game. The definition of an oracle implementation is as follows:[5]

[4] c.f. `Syntactic/OracleExpr`.

[5] c.f. `Syntactic/OracleEval`, where this is called OracleImpl.

```
record Oracle (OST : Set) : Set where
  field
    Init : OracleState → CryptoExpr OST ⊤
    Call : OracleArg   → CryptoExpr OST OracleResult
```

We will use OST to refer to the type of the state that the oracle stores, and use AST (instead of ST) for the state type of the adversary.

In order to reason about a game that involves oracles, we would like to merge all the above components into a single game, much the same way we inlined the definitions of the encryption scheme and adversary when reasoning about IND-EAV in section 1.2. However, the process of combining a game expressed as an OracleExpr AST term and an Oracle implementation is somewhat more complicated than the usage of an adversary in a game, since we need to reconcile the types OracleExpr AST and CryptoExpr OST. In section 1.3, we remarked this can be done by using CryptoExpr $(AST \times OST)$ to store both states at once. Let us now formalise this approach.

We start by defining the operations get-oracle-state, set-oracle-state, get-adv-state, and set-adv-state, which we use for operating on the oracle and adversary components of the state respectively. The definitions are as follows:

```
get-oracle-state : CryptoExpr (AST × OST) OST
get-oracle-state = fmap snd get-state

get-adv-state : CryptoExpr (AST × OST) AST
get-adv-state = fmap fst get-state

set-oracle-state : OST → CryptoExpr (AST × OST) ⊤
set-oracle-state ost = do
  ast ← get-adv-state
  set-state $ ast , ost

set-adv-state : AST → CryptoExpr (AST × OST) ⊤
set-adv-state ast = do
  ost ← get-oracle-state
  set-state $ ast , ost
```

One last function we need before we can define the gluing is a way of embedding the terms that implement the oracle, which have type CryptoExpr OST A, into the game as a whole, which has type CryptoExpr $(AST \times OST)$ A. Fortunately, this is straightforward given the functions we defined above, since we can replace all uses of GetState by get-oracle-state and set-state by set-oracle-state:[6]

```
embed : CryptoExpr OST A → CryptoExpr (AST × OST) A
embed (Return a)        = Return a
embed (Uniform n cont)  = Uniform n λ v → embed (cont v)
embed (GetState cont)   = get-oracle-state   >>= λ st → embed (cont st)
embed (SetState st cont) = set-oracle-state st >>= λ t → embed (cont t)
```

Now we have all the tools necessary to define a function that combines a game that uses an oracle and a definition of that oracle into a single game. The idea is much the same

[6] c.f. `Syntactic/OracleEval` again.

as in embed: we replace uses of get-state and set-state by get-adv-state and set-adv-state respectively, and use the embed function to embed the oracle implementation where it is used.

```
eval : Oracle OST → OracleExpr AST A → CryptoExpr (AST × OST) A
eval ocl (Return a) = Return a
eval ocl (Uniform n cont)     = Uniform n λ v → eval ocl (cont v)
eval ocl (GetState cont)      = get-adv-state    >>= λ st → eval ocl (cont st)
eval ocl (SetState st cont)   = set-adv-state st >>= λ t  → eval ocl (cont t)
eval ocl (InitOracle st cont) = embed (Init ocl st)   >>= λ t → eval ocl (cont t)
eval ocl (CallOracle arg cont) = embed (Call ocl arg) >>= λ r → eval ocl (cont r)
```

With this in place, we can represent the examples from section 1.3 within our system. Most importantly, this gives us an automatic way of performing the inlining that we did by hand in section 1.4, allowing us to detect when we have made a mistake.

We would like to highlight the role of the free monad construction in enabling this straightforward approach to oracles. Had we defined CryptoExpr in a less syntactic way, extending it with further operations would be far more difficult. As it is, such an extension can be performed by adding more constructors to OracleExpr and adding corresponding constructors to Oracle. We will see how this system can be improved further in chapter 5, so that the definitions of fmap and >>= do not need to be updated when such changes are made.

2.3 Constraints on Adversaries

We can now express all the games we have outlined in chapter 1. Let us tackle the opposite problem: how can we place a restriction on what a CryptoExpr or OracleExpr term may do, for example to restrict the class of adversaries?

We can start by considering a simple example: suppose that we want some portion of our game to not have access to the state; for example, if we want to express that an implementation of the oracle may not use the state. We could achieve this by removing state from the games entirely, or by setting its type to ⊤, but both of these are big changes that affect the system as a whole. Instead, we can define a Stateless predicate on terms ce : CryptoExpr ST A that holds only if ce does not use the GetState or SetState constructors. We can define this as follows:[7]

```
data Stateless : CryptoExpr ST A → Set where
  ReturnS  : ∀ a → Stateless (Return a)
  UniformS : ∀ n
           → {cont : BitVec n → CryptoExpr ST A}
           → (∀ v → Stateless (cont v))
           → Stateless (Uniform n cont)
```

If a game is expressed exclusively in terms of Return and Uniform, then we can construct a corresponding proof term using ReturnS and UniformS by following the recursive

[7] c.f. `Syntactic/StateBounds`; note that in the code we separate the property of not writing to the state and not reading from the state, but the difference is not significant.

structure. However, if the game uses GetState or SetState, then no proof can exist, since Stateless (GetState *cont*) and Stateless (SetState *st cont*) can be shown to be empty. We can thus use the type Σ (CryptoExpr *ST A*) Stateless to represent, within Agda, the type of games that use no state.

We can also use this technique to bound the number of times an adversary queries the oracle, and restrict the adversary from reinitialising the oracle. We can make a predicate BoundedOracleUse *k b* on terms *ce* : OracleExpr *ST A* that expresses that *ce* uses CallOracle at most *k* times, and only uses InitOracle if *b* is true. Just like with Stateless, we represent this in Agda by creating a datatype that mimics the recursive structure of OracleExpr.[8]

```
data BoundedOracleUse : Bool → ℕ → OracleExpr A → Set₁ where
  ReturnBOU     : ∀ a → BoundedOracleUse b k (Return a)
  UniformBOU    : {cont : BitVec n → OracleExpr A}
                → (∀ v → BoundedOracleUse b k (cont v))
                → BoundedOracleUse b k (Uniform n cont)
  GetStateBOU   : {cont : ST → OracleExpr A}
                → (∀ st → BoundedOracleUse b k (cont st))
                → BoundedOracleUse b k (GetState cont)
  SetStateBOU   : {ce : OracleExpr A}
                → BoundedOracleUse b k ce
                → BoundedOracleUse b k (SetState st ce)
  InitOracleBOU : {ce : OracleExpr A}
                → BoundedOracleUse false k ce
                → BoundedOracleUse true k (InitOracle st ce)
  CallOracleBOU : {cont : OracleResult → OracleExpr A}
                → (∀ r → BoundedOracleUse b k (cont r))
                → BoundedOracleUse b (suc k) (CallOracle arg cont)
```

Note that in the InitOracleBOU case we prohibit the continuation from performing any further InitOracleBOU calls, thus forcing initialisation to happen at most once, and in CallOracleBOU, we decrease the number of allowed calls to the oracle by one. It is worth mentioning that this is only a restriction on what the game is *allowed* to do: since the ReturnBOU case does not restrict *b* or *k*, we do not *require* the game to perform any actions. If we wanted to, we could achieve the latter effect by changing the ReturnBOU constructor to have type ∀ *a* → BoundedOracleUse false *0* (Return *a*).

This approach works very well when we want to restrict some property of a *ce* : CryptoExpr *ST A* (or OracleExpr) we receive as an input, since we can add an extra parameter that represents a proof that *ce* satisfies some property. However, a drawback of this approach is that when *we* construct a *ce* : CryptoExpr *ST A*, we must also construct the corresponding proof. This is not difficult, since the proof term is completely determined by *ce* itself, but it the kind of work we would like to automate.[9]

The straightforward way to do this in Agda would be to traverse the CryptoExpr *ST A* structure and check that the conditions we impose are satisfied. In the case of Stateless, this can be done, since BitVec *n* is finite. Given an enumeration all-bitvecs : (*n* : ℕ) →

[8] c.f. `Syntactic/BoundedOracleUse` and `Syntactic/BoundedOracleUseExample`.

[9] A partial implementation of these ideas can be found in `Syntactic/BoundedOracleUseExample`.

List (BitVec n) of all bit vectors of a given length, we can define a runtime predicate isStateless? by recursion:

```
isStateless? : CryptoExpr ST A → Bool
isStateless? (Return a) = true
isStateless? (Uniform n cont)
  = all (map (λ v → isStateless? (cont v)) (all-bitvecs n)
isStateless? (GetState _) = false
isStateless? (SetState _ _) = false
```

We can express the soundness of isStateless? in Agda by defining a function that takes a proof that isStateless? *ce* is true and gives a proof of Stateless *ce*. Unfortunately, even though this allows us to compute these proofs automatically, this approach is not useful in practice: enumerating all bit vectors of length n will take time $\Omega(2^n)$.

Instead, we require some way of generating a proof term based on the syntactic structure of a term. In chapter 6 we will show how we can use indexed monads to incorporate a predicate such as BoundedOracleUse in the definition of OracleExpr. An alternative approach, which we have not had time to explore, is to use reflection to obtain a description of the structure of a term as it is in the program code. This would allow the automatic generation of proof terms using considerably less time.

Despite these shortcomings, the predicates we defined in this section satisfy the requirements we posed in section 1.5, and thus we can now represent all games we had set out to. In the next chapter, we will work out the system for reasoning about these games.

Chapter 3

The Logic of Games

Throughout chapter 1, we argued that rewrite steps were valid because the games they produced did not significantly differ from the games they were performed on. In this chapter, we will define the notion of ϵ-indistinguishable to formalise this concept of 'not significantly different', thereby specifying the logic that can be used to reason about games. Keeping with the style of the previous chapter, our definition of ϵ-indistinguishability will be purely syntactic, with no reference to an interpretation, which we will discuss in chapter 4.

Before we dive into the technical details, let us consider what relation we would like to capture. Two games with result type A being ϵ-indistinguishable means that up to some error term ϵ, the probability of sampling any $a : A$ from one is close to the probability of sampling a from the other. An interesting point in our construction is that we will formalise this notion with no reference to the individual elemenst of A.

We will start this chapter by looking at how we can define ϵ-indistinguishability in classical probability theory, and what results hold there. We will then define the ϵ-indistinguishability relation $\equiv_\epsilon^E$ on terms of type CryptoExpr $ST\ A$ and prove some results about it. In particular, we will show that the full power of monads is not necessary to represent arbitrary games: up to indistinguishability, every game has a fixed structure.

We will proceed to consider two basic variations of the ϵ-indistinguishability relation. First of all, we will note that the relation we defined is too strong: there are games we considered ϵ-indistinguishable in chapter 1, but which are not ϵ-indisindistinguishable in this new system. This is because in our earlier examples, we ignored the final adversary state when we reasoned about the games. To correct this, we will introduce a new relation called result ϵ-indistinguishability that will make our proofs go through again. For the second variation, we will show how ϵ-indistinguishability can be extended to games that make use of an oracle.

Finally, we will consider how this system handles the requirements we posed in section 1.5, such as asymptotic complexity and the usage of security assumptions in proofs. We also discuss a number of problems we have been unable to solve.

Throughout this chapter, we will make use of the type $\mathbb{Q}$ of rational numbers to represent probabilities. Our construction is independent of the implementation of the rationals used, as long as arithmetic and ordering is supported.

3.1 Properties of Distributions

Before we start on a formalisation in Agda, let us recall how probability distributions behave in classical mathematics. All of the material in this section is long-known and completely standard, but we feel that this brief recap will serve as motivation for our definition of $\equiv_\epsilon^E$ In particular, we are interested in exploring how $\equiv_\epsilon^E$ should interact with monadic binding.

Since all of the distributions we will consider arise from a bounded number of coin flips, we are interested exclusively in discrete probability distributions with finite support. We will model a probability distribution X over a set A as a function $f_X : A \to \mathbb{R}$, where for every $a \in A$, $f_X(a)$ gives the probability of drawing a from X. As usual, these functions satisfy $f_X(a) \in [0, 1]$ and $\sum_{a \in A} f_X(a) = 1$. If there is no risk of confusion, we will write $X(a)$ for $f_X(a)$.

To give a better understanding of this presentation, let us consider two examples that we will need later:

- For any set A and any $a \in A$, the Dirac delta distribution 1_a is the distribution that always gives A, given by $1_a(a) = 1$ and $1_a(x) = 0$ if $x \neq a$.
- Let 2^n be the set of bit vectors of length n. The uniform distribution U_n is defined by $U_n(v) = 2^{-n}$ for every $v \in 2^n$.

Given a distribution X over A and an A-indexed family of distributions Y over B, we define the composite distribution XY over B by

$$XY(b) = \sum_{a \in A} X(a)Y_a(b).$$

It is an easy exercise to show that this this defines a probability distribution, and that if X and each Y_a have finite support, then XY also has finite support.

Given two distributions X and Y over the same set A, we denote the distance between them as $d(X, Y)$ and define it as

$$d(x, y) = \frac{1}{2} \sum_{a \in A} |X(a) - Y(a)| .$$

The reader may notice that this is simply the l^1 or Manhattan norm from linear algebra scaled by $\frac{1}{2}$. The following theorem motivates this scaling:

Theorem 1. *For every two probability distributions X, Y over some set A,*

$$d(X, Y) \leq 1.$$

Proof. Since X and Y are distributions, they take values in $[0, 1]$ and sum to 1. It follows by the triangle inequality that

$$\frac{1}{2} \sum_{a \in A} |X(a) - Y(a)| \leq \frac{1}{2} \left(\sum_{a \in A} |X(a)| + \sum_{a \in A} |Y(a)| \right) \leq 1.$$

□

There is a connection between this notion of composition of probability distributions and the way in which our games form a monad. Namely, let D_A denote the set of probability distributions over a set A. We can regard D_- as a functor on **Set** and define the action on a function $f : A \to B$ by

$$D_f(X)(b) = \sum_{a \in f^{-1}(b)} X(a).$$

More importantly for us, D_- has the structure of a monad, with 1_- being the unit and composition of distributions being the bind!

We encourage the motivated reader to derive the monad multiplication and check the monad laws. Our focus, however, lies on the interaction between the monadic structure and the norm we defined.

Let us say that two probability distributions X and Y are *ϵ-indistinguishable* iff $d(X, Y) \leq \epsilon$. We will now demonstrate a number of properties that show how composition of probability distributions interacts with ϵ-indistinguishibility of distributions. While this does not directly prove anything about games, it suggests what properties we can reasonably expect to hold for them, and can thus guide what assumptions we make.

For the rest of this section, let A and B denote arbitrary sets.

Theorem 2. *Let X, Y be distributions over A and let Z be an A-indexed family of distributions over B. Then*

$$d(XZ, YZ) \leq d(X, Y).$$

Proof. Writing out the definition, for any $b \in B$,

$$|XZ(b) - YZ(b)| = \left| \sum_{a \in A} (X(a) - Y(a)) Z_a(b) \right|.$$

By the triangle inequality,

$$\sum_{b \in B} |XZ(b) - YZ(b)| \leq \sum_{b \in B} \sum_{a \in A} |X(a) - Y(a)| \, |Z_a(b)| \, .$$

Since each Z_a is a probability distribution, $\sum_{b \in B} |Z_a(b)| = 1$, hence

$$\sum_{b \in B} |XZ(b) - YZ(b)| \leq \sum_{a \in A} |X(a) - Y(a)| \, .$$

□

Theorem 3. *Let X be a distribution over A and let Y, Z be A-indexed families of distributions over B. Then*

$$d(XY, XZ) \leq \sum_{a \in A} X(a) d(Y_a, Z_a).$$

Proof. Writing out the definition, for any $b \in B$,

$$|XY(b) - XZ(b)| = \left| \sum_{a \in A} X(a)(Y_a(b) - Z_a(b)) \right|.$$

As above, by the triangle inequality we get

$$\sum_{b\in B} |XY(b) - XZ(b)| \leq \sum_{b\in B}\sum_{a\in A} |X(a)|\,|Y_a(b) - Z_a(b)|$$

and now using the non-negativity of X we get

$$\frac{1}{2}\sum_{b\in B} |XY(b) - XZ(b)| \leq \frac{1}{2}\sum_{a\in A} X(a)\left(\sum_{b\in B} |Y_a(b) - Z_a(b)|\right) = \sum_{a\in A} X(a)d(Y_a, Z_a)$$

□

This has two useful consequences.

Corollary 4. *Let X be a distribution over A and let Y, Z be A-indexed families of distributions over B. If there is an ϵ such that $d(Y_a, Z_a) \leq \epsilon$ for every $a \in A$, then*

$$d(XY, XZ) \leq \epsilon.$$

Corollary 5. *Let $n \in \mathbb{N}$ and let X, Y be 2^n-indexed families of probability distributions over A. Then*

$$d(U_n X, U_n Y) \leq \frac{1}{2^n}\sum_{v\in 2^n} d(X_v, Y_v).$$

We hope that by providing this brief overview of the properties that hold of probability distributions in classical mathematics, we have given the reader an intuition for what can be expected from the Agda formalisation.

3.2 ϵ-Indistinguishability

With this classical intuition in hand, we can now define the relation of ϵ-indistinguishability on games. Just like we defined games to be purely syntactic constructs, we define this relation in a syntactic manner, by specifying an inductive data type that represents the proofs of ϵ-indistinguishability.

The definition, in full detail, is available in the Agda code. However, the formulation given there is too verbose to be insightful. We present the same inductive rules here in a more understandable manner. After the definition, we will show the definition of one such rule in its entirety; this should make clear both the precise meaning of the other rules, and our reluctance to write them out in full.

As a final preparation, let us introduce two abbreviations that will be useful in the (recursive) definition of $\equiv_\epsilon^E$ itself. Let A and B be arbitrary types. Firstly, given two games[1] G and H, we will say "G and H are ϵ-indistinguishable" to mean $G \equiv_\epsilon^E H$. Secondly, given two B-indexed families of games[2] f and g and a function $h : A \to \mathbb{Q}$, we say "f and g are h-indistinguishable" to mean that for every $a : A$, $f\ a$ and $g\ a$ are $(h\ a)$-indistinguishable.

[1] That is, terms of type CryptoExpr $ST\ A$.
[2] That is, terms of type $B \to$ CryptoExpr $ST\ A$.

Without further ado: for every non-negative $\epsilon : \mathbb{Q}$ and every two types A and ST, let $\equiv^E_\epsilon$ be the least binary relation on CryptoExpr ST A such that the following properties hold:[3]

1. $\equiv^E_\epsilon$ is reflexive and symmetric.
2. If $G \equiv^E_{\epsilon_1} H$ and $H \equiv^E_{\epsilon_2} I$ then $G \equiv^E_{\epsilon_1+\epsilon_2} I$.
3. Every two games are 1-indistinguishable.
4. For every $n : \mathbb{N}$, if f and g are (BitVec n)-indexed families of games and $f\ v \equiv^E_\epsilon g\ v$ for every v : BitVec n, then Uniform n $f \equiv^E_\epsilon$ Uniform n g.
5. If f and g are ST-indexed families of games and $f\ st \equiv^E_\epsilon g\ st$ for every st : ST, then GetState $f \equiv^E_\epsilon$ GetState g.
6. If f and g are $\top$-indexed families of games and f tt $\equiv^E_\epsilon g$ tt then for every st : ST, SetState st $f \equiv^E_\epsilon$ SetState st g.
7. $\equiv^E_\epsilon$ is closed under the state monad laws;
8. $\equiv^E_\epsilon$ is closed under the reordering of uniform and get-state operations;
9. $\equiv^E_\epsilon$ is closed under the reordering of uniform and set-state operations;
10. $\equiv^E_\epsilon$ is closed under the insertion of uniform and get-state operations;
11. $\equiv^E_\epsilon$ is closed under replacing consecutive occurences of uniform n and uniform m by uniform $(n + m)$.
12. $\equiv^E_\epsilon$ is closed under application of bijections to uniform distributions.
13. If two 2^n-indexed families of games f and g are h-indistinguishable, then uniform $\gg\!=$ f and uniform $\gg\!=$ g are $(\sum_{v:2^n} h(v))$-indistinguishable.

Let us write out rule 8 in its entirety. In Agda, the statement is as follows:

```
data _≡ᴱ_ (ST A : Set) : (ce cf : CryptoExpr ST A) → Set where
  ...
  xchg-Uniform-GetState : ∀{n}
                        → (cont : BitVec n → ST → CryptoExpr ST A)
                        → (Uniform n λ v → GetState λ st → cont v st)
                          ≡ᴱ (GetState λ st → Uniform n λ v → cont v st)
  ...
```

Translated to English, this reads: for every n : $\mathbb{N}$ and every continuation *cont* that sends a bit vector and a state to a game, the following two actions are equivalent:

- generate a random bit vector v, retrieve the state st, and call *cont* v st; and
- retrieve the state st, generate a random bit vector v, and call *cont* v st.

[3] c.f. `Syntactic/Logic` and `Syntactic/EpsilonLogic`. We define the case where $\epsilon = 0$ separately as that makes the proofs easier to work with, but this is not an essential difference.

The other rules are similarly simple in meaning and obscure in formal statement.

Let us now go through a number of results that highlight the similarity of $\equiv^E_\epsilon$ to the notion of ϵ-indistinguishability we defined on probability distributions in section 3.1.[4]

Theorem 6. *$\equiv^E_{\epsilon_1}$ is a subrelation of $\equiv^E_{\epsilon_2}$ if $\epsilon_1 \leq \epsilon_2$.*

Proof. The proof is by induction on the derivation of $a \equiv^E_{\epsilon_1} b$. All the cases are trivial: for example, in the last axiom, we can take $h'(v) = h(v) + \epsilon_2 - \epsilon_1$. By induction, f and g are h'-indistinguishable and the desired result follows. □

In section 3.1 we discussed the importance of the fact that the distance between probability distributions remains bounded under monadic binding. These properties can be proved from our axioms.

Theorem 7. *If G is a game with result type A and f and g are A-indexed families of games that are ϵ-indistinguishable, then $G \gg\!= f \equiv^E_\epsilon G \gg\!= h$.*

Proof. This is a straightforward recursion on the structure of G. □

Theorem 8. *If $G \equiv^E_\epsilon H$ and f is a family of games, then $G \gg\!= f \equiv^E_\epsilon H \gg\!= f$.*

Proof. This proof goes by induction on the derivation of $G \equiv^E_\epsilon H$. There are many cases, but they are all straightforward. □

A special case of $\equiv^E_\epsilon$ arises for $\epsilon = 0$. We denote this relation $\equiv^E$ and say that two games G and H satisfying $G \equiv^E H$ are indistinguishable, dropping the ϵ. The $\equiv^E$ relation is of great practical value since it allows us to replace multiple occurences of a subgame at once. This will be particularly important when we look at games that make use of an oracle.

Now that we have investigated the basic properties of this logic, let us look at our first useful result: that every game can be rewritten into a canonical form.[5]

Definition 9. We say that a game G : CryptoExpr ST A is in canonical form if there exist functions $f : ST \to \mathbb{N}$, $g : (st : ST) \to$ BitVec $(f\ st) \to ST$, and $h : (st : ST) \to$ BitVec $(f\ st) \to A$ such that G is provably equal to

```
do
  st ← get-state
  v ← uniform $ f st
  set-state $ g st v
  return $ h st v
```

Theorem 10. *Every G : CryptoExpr ST A is indistinguishable from a game in canonical form.*

Note that we do not need to assume that the state type ST is finite or has decidable equality.

[4] c.f. `Syntactic/LogicDerived`.

[5] c.f. `Syntactic/Reorder`.

Proof. The full proof is in the Agda code, and proceeds in two steps: we first construct f, g, and h and then show indistinguishability. In all cases, the construction is by recursion on the structure of ce.

The key idea of the proof is to explicitly pass around the current state st and, in the case of g and h, a sufficiently long vector of random bits. Since f does not have access to such a bit vector (being the function that determines how much randomness we require), it must enumerate all vectors of the right length when the recursion is on a Uniform constructor. This makes it unfeasible to explicitly compute this canonical form, but we can nevertheless reason with it.

The main difficulty in the proof is showing that the vector v : BitVec k provides a sufficient number of random bits. This involves showing that whenever we recurse on a call of the form Uniform n $cont$, we can prove $n \leqslant k$. Careful manipulation of the indices can show this is indeed the case. □

3.3 Result-Indistinguishability

It is tempting to assume that the notion of ϵ-indistinguishability we have just defined represents ϵ-indistinguishability as we used the term in chapter 1. Unfortunately, the situation is somewhat more nuanced.

In chapter 1, we considered games to be indistinguishable even if they had different effects on the state of the adversary. We did not have this luxury when defining the notion of ϵ-indistinguishability above, since we wanted $\equiv_\epsilon^E$ to be a congruence, and thus closed under bind.

When it comes to bounding the advantage of the adversary, however, we do *not* want to distinguish outcomes based on the state of the adversary: two adversaries that both win the game with probability 0.5 are equivalent for our purposes, even if we can distinguish between them based on the effect they have on the state. As such, we want a weaker notion of indistinguishability which we will call result-indistinguishability.

We do not have an axiomatisation of this relation,[6] like we do of the ϵ-indistinguishability relation, but we present a way of achieving a similar result in chapter 6.

3.4 Indistinguishability with Oracles

In order to reason about games involving oracles, we want to extend the notion of ϵ-indistinguishability to pairs of games involving oracles and oracle definitions for these games. We use the fact that we have the eval funcion, which can combine a game involving an oracle and an oracle definition into a game that makes no mention of oracles. This gives us a direct way of defining ϵ-indistinguishability on oracle game-implementation pairs.

Formally speaking, given G and H of type OracleExpr AST A and ocl_1 and ocl_2 of type Oracle OST, we say that $(G\ ,\ ocl_1) \equiv_\epsilon^{OE} (H\ ,\ ocl_2)$ iff eval ce $ocl_1 \equiv_\epsilon^E$ eval cf ocl_2. We will write $G \equiv_{\epsilon,ocl}^{OE} H$ if ocl is the same on both sides. We will also write $ocl_1 \equiv_\epsilon^E ocl_2$ iff for every ost : OracleState, Init ocl_1 $ost \equiv_\epsilon^E$ Init ocl_2 ost and for every arg : OracleArg, Call ocl_1 $arg \equiv_\epsilon^E$ Call ocl_2 arg.

[6] For one possible approach, c.f. `Syntactic/ResultLogic`.

This is a somewhat unsatisfactory solution, since this does not give us any reasoning principles for $\equiv_\epsilon^{OE}$, making the process of proof extremely manual. We will consider this shortcoming in greater generality in section 5.7.

3.5 Generalised Security

As we have already remarked in section 1.5, in practice we often want to show security only against adversaries that are restricted in the resources they may use. The prime example of this is a restriction to adversaries that run in polynomial time. Restricting the problem in this way allows us to use assumptions about what a polynomial-time algorithm can and cannot do. This allows us to reason about the security of systems that depend on problems like integer factorisation being hard; without such an assumption, since integer factorisation can be performed in Agda, we cannot rule out that the adversary performs the factorisation and thereby defeats our security scheme.

Within our system, we cannot create a type of polynomial-time adversaries, and so we cannot express this restriction directly. However, we can still achieve the desired effect by assuming that certain operations cannot be performed by any adversary. We do this by assuming that certain games are unwinnable, using the generalised notion of games described in section 1.6.

For example, let us consider the discrete logarithm problem. Given a cyclic group G and a generator g, the Decisional Diffie-Hellman assumption states that it is hard to distinguish the triple (g^a, g^b, g^c) with a, b, c all uniformly random (with $0 \leq a < |G|$) from (g^a, g^b, g^{ab}), with a, b uniformly random. We can phrase this as follows: let $\mathsf{U_G}$ be the uniform distribution over G. Then there is some ϵ such that the following two games are ϵ-indistinguishable for any adversary adv:

```
do
  st ← get-state
  a ← U_G
  b ← U_G
  c ← U_G
  adv (pow g a , pow g b , pow g c)
  set-state st
do
  st ← get-state
  a ← U_G
  b ← U_G
  adv (pow g a , pow g b , pow g a · b)
  set-state st
```

Notice that we need to explicitly preserve the state due to our lack of a proper notion of result-indistinguishability.

Nevertheless, we can then use this assumption to replace the usage of pow g $(a \cdot b)$ in a proof with pow g c. For example, this is a key step in proving the security of the ElGamal encryption scheme [Sho04]. By introducing this assumption we show that no adversary can find a given pow g a, since otherwise they could take pow (pow g b) a and compare it to the third component of the tuple.

A possible downside to this approach is that our assumptions must all be phrased as statements of indistinguishability of games. Another issue with this approach is that while this allows us to reason about an arbitrary adversary as if it satisfied our assumptions, it does not restrict the class of adversaries that we may use as a counterexample: it is thus necessary to check the validity of any constructed adversaries by hand.

Finally, an assumption of this kind only states that such an ϵ exists, without any bounds on the size of this ϵ. This is unsurprising: for any fixed problem size, a polynomial-time adversary can still achieve arbitrarily good probability, because the property of being polynomial-time is vacuous if the problem size is fixed.

This is not as severe an issue as it may seem: we can assume (without formalising it in Agda) that ϵ is small, and interpret the indistinguishability results we come to in this light. This is not uncommon in existing proofs [Sho04]. However, if we want a more formal resolution to this problem, we can reason about asymptotic indistinguishability.

Instead of attempting to prove that games G and H are ϵ-indistinguishable, we can instead look at families of games G and H parametrised by a security paraeter, and show that they are f-indistinguishable for some vanishing function f.

3.6 Future Work

In this chapter, we have specified the foundations of a theory of indistinguishability of games. An important further step is to develop a collection of lemmas based on this theory that can act as rewrite rules for games. Bellare and Rogaway [BR06] have identified a number of techniques that are commonly used in cryptographical proofs, and a formalisation of these would greatly improve the practical value of this theory.

In particular, the 'identical until bad' technique [Sho04] tells us that if two games X and Y are identical unless some bad event F happens and F has probability ϵ, then X and Y are in fact ϵ-indistiindistinguishable. For example, two games may be indistinguishable unless two uses of uniform n result in the same bitstring, or if the adversary can find a string that causes a hash collision. This technique is very useful, but it is hard to formalise in our context: we may not be able to tell from the final state of the game whether the bad event happened. As such, we need to show that the game is result-indistinguishable from one that adds additional instrumentation to track the bad event, and then use the data provided by this instrumentation to reason that the games can only differ in a minority of cases. This is hard to do even in concrete scenarios, and a general solution would be useful for formalising existing proofs.

We have also been unable to develop the equational theory with oracles to the same point as the theory without them. There does not appear any fundamental reason we could not find comparable results for canonical forms. However, these developments may be better done in the context of section 5.6, where we perform a further generalisation of oracles. The possible interactions between oracles and result-indistinguishability are also an interesting matter of further study.

Finally, not all conseuences of the point-free presentation of indistinguishability we have given here are clear to us. In particular, we may wish to work with the support of a distribution, and we do not yet know how this can be expressed in this system.

Chapter 4

Interpreting Games

With the logic we have developed in hand, we can tackle questions about games being ϵ-indistinguishable. However, if we are to be convinced that our results have any meaning, we must first show that our system is at the very least not trivial: if *every* game G : CryptoExpr ST Bool could be shown to be ϵ-indistinguishable from coin then our proof would have little weight behind it. It would be even better if we could show that our notion of indistinguishability can be expressed as a relation on a type that explicitly models probability distributions, rather than on a purely syntactic description of games.

To this end, we will define the notion of a *model* of our logic and construct a non-trivial model based on the Haskell Dist monad due to Erwig and Kollmansberger [EK06]. Using this model, we can show that our logic does not prove coin $\equiv^R_{st}$ return true or return false $\equiv^R_{st}$ return true.

Although our model will be internal to Agda, we see at present no reason to formalise the model theory: in particular, the notion of a distance relation and the category of models of game logic that we introduce in this chapter are tools for conceptual understanding, not constructions in Agda.

4.1 Distance Relations

Before we set about defining our models, we would like to identify the key properties of $\equiv^E_\epsilon$ and give a name to relations of this kind. This is merely a matter of convenience, to save us the trouble of listing these properties every time we wish to use them.

Our definition of $\equiv^E_\epsilon$ in section 3.2 was inspired by the notion of ϵ-indistinguishability we defined in section 3.1. The latter expressed that an upper bound held on the distance between two elements. In the case of $\equiv^E_\epsilon$, we cannot express this distance function directly, but this is nevertheless the intuition we are attempting to capture. This inspires the following definition:

Definition 11. A family of binary relations $R\_$ indexed by non-negative rationals is a *distance relation* on A if for all $a, b, c : A$ and all non-negative $\epsilon_1, \epsilon_2 : Q$,

- $R_\epsilon(a, a)$;
- if $R_\epsilon(a, b)$ then $R_\epsilon(b, a)$;

- if $R_{\epsilon_1}(a, b)$ and $R_{\epsilon_2}(b, c)$ then $R_{\epsilon_1+\epsilon_2}(a, c)$; and
- if $R_{\epsilon_1}(a, b)$ and $\epsilon_1 \leq \epsilon_2$ then $R_{\epsilon_2}(a, b)$.

Clasically, it is easy to see that if d is a metric on a set A, then $R_\epsilon(a, b) := d(a, b) \leq \epsilon$ defines a distance relation on A. However, the notion of a distance relation is more flexible. Note, in particular, that a binary relation on A is, in Agda, a function $A \to A \to$ Set, meaning that there may be multiple proofs that $R_\epsilon(a, b)$ holds. This is indeed the case with $\equiv^E_\epsilon$, since we can use symmetry twice to obtain a new proof, unequal to the previous.

We will now use the fact that the objects we are working with have more structure than plain sets, or even types: they are in the domain of a monad. In section 3.1, we have seen that there is a certain interaction between ϵ-indistinguishability and the monadic structure. We capture this with the following two definitions:

Definition 12. We say a family of distance relations R^-_- defined on the range of a functor F is *functorial* if whenever $R^{F(A)}_\epsilon(a, b)$ and $f : A \to B$, then $R^{F(B)}_\epsilon(F(f)(a), F(f)(b))$.

For the following definition we will denote monadic binding of $a : M(A)$ with $f : A \to M(B)$ by $a \rhd f := \mu_M(M(f)(a))$.

Definition 13. We say a functorial family of distance relations R^-_- defined on the range of a monad M is *monadic* if the following two conditions hold:

- if $R^{M(A)}(a, b)$ and $f : A \to M(B)$, then $R^{M(B)}(a \rhd f, b \rhd f)$; and
- if $f, g : A \to M(B)$ and for every $x : A$, $R^{M(B)}(f(x), g(x)))$, then for any $a : M(A)$, $R^{M(B)}(a \rhd f, a \rhd g)$.

In other words, a functorial family of distance relations is closed under fmap, while a monadic family of distance relations is closed under bind. Closure under return does not need to be assumed, since distance relations are reflexive.

Once again, we have chosen to present this notion in a mathematical manner, since we feel this gives a better understanding. Translating this formalisation into Agda is a straightforward exercise, but introduces considerable clutter that obstructs the meaning.

4.2 Models of Game Logic

In chapter 2, we defined games in a purely syntactic manner, and we then defined a syntactic distance relation on these games. We can regard this as our first example of a model of game logic. However, this model gives us few tools to show that two games are *not* ϵ-indistinguishable. As such, we would like to define other models, where such a proof is easier to perform.

Since our construction of games was parametrised over a state type ST, so too we will parametrise our construction of models of game logic. However, for convenience, we will assume that ST has decidable equality. We have not encountered existing proofs that relied on a state type with non-decidable equality, so we do not consider this a particularly great limitation.

In the following, we define what it means to be a model, and then construct a category of these models. The categorically disinclined reader may ignore everything except Theorem 14, but we feel that this brief exposition highlights the structure of the matter at hand.

Definition 14. A *model of game logic* is a monad M together with a monadic distance relation $\approx_\epsilon^E$ and a valuation function $[\![\_]\!] : \mathsf{CryptoExpr}\ ST\ A \to M\ A$ such that for any games G and H, if $G \equiv_\epsilon^E H$, then $[\![G]\!] \approx_\epsilon^E [\![H]\!]$.

This definition can be rephrased in categorical terms by considering the syntactic model in a suitable category and taking the coslice:

Definition 15. Let **Pre-MGL** (pre-models of game logic) be the category whose objects are monads M together with a monadic distance relation $\approx_\epsilon^E$ and whose morphisms are monad morphisms that preserve $\approx_\epsilon^E$.

Recall that given a category $\mathcal{C}$ and an object A of $\mathcal{C}$, the coslice category (or under category) $A \downarrow \mathcal{C}$ is the category where the objects are morphisms out of A (in $\mathcal{C}$), and the morphisms from an object $\phi : A \to B$ to $\psi : A \to C$ are morphisms $f : B \to C$ such that $f \circ \phi = \psi$.

Let us now enote the coslice category $CE \downarrow$ **Pre-MGL** by **MGL**.

Theorem 16. *A model of game logic $\mathcal{M}$ is an object in* **MGL**.

Proof. Let $\mathcal{M}$ be a model of game logic. The underlying monad and the distance relations give rise to an object in **Pre-MGL**. The valuation function gives a monad morphism which, by definition of a model of game logic, preserves the distance relation.

On the other hand, let $\mathcal{M}$ be an object in **MGL**. Its codomain is a **Pre-MGL** object. Regarding $\mathcal{M}$ as a valuation function, this gives rise to a model of game logic. □

This result allows us to use standard theorems about coslice categories to analyse the model theory of game logic. In particular, it tells us that the identity function on CE is the initial object in **MGL**, meaning that our syntactic model is initial, as we would expect. Furthermore, since **Pre-MGL** has a terminal object 1 (given by the constant singleton monad), the unique map from CE to 1 gives us a terminal model. In general, limits in **MGL** correspond to the limits of the underlying objects in **Pre-MGL** [Lan98].

4.3 List Model

Let us now regard a specific model based on the Dist monad [EK06], in which we can compute whether two games over a type A with decidable equality are ϵ-indistinguishable. This material has not been fully worked out in Agda, but the claims we make pertain to finite objects (lists of rational numbers) and, as such, can be shown to hold constructively.[1] Furthermore, the construction relies in several places on equality being decidable. This is a serious issue. However, we think that the results we present here are worth stating despite this. For now, we will assume that all types involved have decidable equality, and analyse this assumption at the end of this section.

We represent a probability distribution over a type A as a list of pairs of elements of A and their corresponding probabilities. Our two basic distributions, return a and coin, can thus be represented as follows:

```
return : A → Dist A
return a = (a , 1) :: []
```

[1] For a partial implementation, c.f. `Distribution/List` and the modules it exports.

```
coin : Dist Bool
coin = (false , 1/2) :: (true , 1/2) :: []
```

Any uniform distribution can be constructed by repeated calls to coin. We can define bind as follows:

```
[] >>= f = []
((a , p) :: xs) >>= f = map (λ q → p · q) (f a) ++ (xs >>= f)
```

The resulting Dist monad structure is in fact isomorphic to the $WriterT\ (\mathbb{Q}\ , \cdots)$ List monad. We use the latter in our implementation, as it allows for better separation of concerns; in particular, the monad laws for Dist follow from the monad laws for Writer and List. However, the difference is insignificant to us here, and the direct presentation is clearer.

There is a slight complication that we need to address here. We require that an distance relation on A identify every two elements of A at $\epsilon = 1$. We would like to define the ϵ-indistinguishability relation on distributions with the help of a distance function, much as we did in section 3.1. However, this definition fails if we allow 'distributions' with negative elements, or 'distributions' that sum to more than 1.

In order to deal with this problem properly, we would need to have every distribution carry around a proof of its validity. However, this is very inconvenient from a programming perspective: these proofs must be maintained at all times, which makes it inconvenient to perform recursion on the list structure. As such, it is more convenient to instead make a separate type ValidDist xs that represents a proof that xs is a valid distribution. We can then show that validity is preserved by bind. Our implementation lacks this feature, but as we will see, the reliance on these assumptions is minor.

For the purposes of this section, we will continue to work with the Dist monad but assume that any distribution xs has a corresponding ValidDist xs proof.

The concrete nature of Dist allows us to provide two further operations that are not supported by CryptoExpr ST: computing the support of a distribution and the probability of drawing a specific element.

Since we have assumed that all types we are dealing with have decidable equality, we can define a function uniques : List A → List A that, given a list, returns the list of unique elements it has. We can then define

```
support : Dist A → List A
support xs = uniques (map fst xs)
```

Computing the probability of sampling a particular element is a matter of finding all occurences of this element and summing the corresponding probabilities:

```
sample : Dist A → A → ℚ
sample xs a = sum $ map snd $ filter (isYes ∘ a ==· ∘ fst) xs
```

We can now verify that our definition of bind corresponds to the one defined in section 3.1.

Theorem 17. *For every* xs : *Dist* A, f : A → *Dist* B *and* b : B, *the following expression is equal to* sample $(xs >>= f)\ b$:

```
sum $ map (λ a → ⋯ (sample xs a) (sample (f a) b)) (support xs)
```

This is a result we have been unable to show in Agda. The difficulty lies in finding a suitable value to perform induction on: in our attempts, neither xs nor support xs provided enough structure to carry through the argument.

The monad Dist provides us with a suitable interpretation of probability, but it does not allow us to interpret stateful computations. For this last functionality, we use the StateT ST monad transformer.[2] Since this is a monad transformer, coin lifts into it, and we use the usual get-state and set-state to interpret the corresponding operations. As we have seen in chapter 5, specifying these there base operations extends to a unique monad morphism from CryptoExpr ST to StateT ST Dist.

As above, we assume that given g : StateT ST Dist A, for any st : ST, g st is a valid probability distribution with a corresponding proof ValidDist (g st).

We can now define a notion of distance between distributions. We make use of a union function that merges two lists and removes duplicates.

```
distance : (xs ys : Dist A) → ℚ
distance xs ys = 1/2 · sum (map f sup)
  where sup = union (support xs) (support ys)
    f a = sample xs a − sample ys a
```

We say that $g1 \approx^E_\epsilon g2$ iff for every st : ST, distance ($g1$ st) ($g2$ st) $\leqslant$ ϵ. Unfortunately, we have not shown that this relation is a distance relation within Agda, nor that it is preserved under the valuation. Since these statements can be shown constructively in a classical phrasing of this problem, and the statements we make are about finite objects, we expect the proofs to be formalisable in Agda as well.

Throughout this section, we have assumed that every type has decidable equality. This is, of course, not the case. It is not clear how we can best deal with this. The following trick allows us to nevertheless define the $\approx^E_\epsilon$ relations: for $g1$ and $g2$ in StateT ST Dist A, we say that $g1 \approx^E_\epsilon g2$ iff for every st : ST *and every proof that* A *has decidable equality*, distance ($g1$ st) ($g2$ st) $\leqslant$ ϵ. This is a type that behaves as our earlier definition for decidable A. However, we cannot prove properties such as congruence under fmap if indistinguishability is defined this way.

Another option is to only define indistinguishability for result types that have decidable equality. This, however, means that this is no longer a model of game logic.

Yet another option is to require finiteness of the state type and regard our StateT ST Dist functor as an endofunctor on the category of sets with decidable equality. This requires the additional assumption of functional extensionality. At present, this is the cleanest solution, but it is not clear whether all games we may want to express can be expressed this way.

4.4 Future Work

Throughout this chapter, it is clear that our work on the list model is incomplete. We have intentionally chosen not to prioritise this aspect of the development, since our primary concern was with the development of the logic of games. As we have seen, indistinguishability can be used both when proving that a game does and that a game does not have

[2] c.f. `Utility/State/Normal`.

a winning strategy, so proofs that two games are not indistinguishable are of less interest. However, a completely formalised model would nevertheless put the logic on firmer footing.

In this chapter, we have considered an approach where probability is formalised using lists. This is not the only possibility, and has the considerable drawback that it only works for distributions with finite support. Another approach is to use the continuation monad $(A \to \mathbb{Q}) \to \mathbb{Q}$. Work in this direction has been done by Ramsey and Pfeffer [RP02]. Of course, our general formulation of the notion of a model allows for the posing of questions such as the nature of the product of two models.

Finally, the focus of this chapter has been on models that satisfy soundness properties with respect to our logic. We have not given any attention to questions of completeness. What extensions would have to be made to our axioms to make the list model complete, for example, would be an interesting further research question.

Chapter 5

Command Structures

In chapter 2, we saw the definitions of CryptoExpr and OracleExpr and their corresponding functor and monad instances, and remarked that they contain considerable duplication. In this chapter we will look at how the free monad construction can be performed in a parametrised way, allowing us to automatically generate these types and functions over them by specifying the operations that we want them to support.

This chapter is primarily of interest as a guide to the accompanying code. The ideas presented are not new; they are laid out by McBride [McB15]. We wish to nevertheless present these constructions in some depth, as this serves as a good introduction to the more general case we will consider in section 6.5.

5.1 Definition

We have seen the following pattern in CryptoExpr and OracleExpr: one constructor is a Return, while the others take a command and then a continuation to handle a response to that command. We start by capturing the structure of commands and responses:[1]

```
record CmdStruct : Set₁ where
  field
    Command : Set
    Response : Command → Set
```

5.2 Free Monads

Given a command structure C, the corresponding free monad should have a constructor for Return and a constructor for each command, taking a continuation for its response. We can encode this structure directly in Agda.[2]

```
data FreeMonad : Set → Set where
  Return-FM : A → FreeMonad A
  Invoke-FM : (c : Command C) → (Response c → FreeMonad A) → FreeMonad A
```

[1] c.f. `Syntactic/CommandStructure` and `Interpretation/Complete/InteractionStructure`.
[2] c.f. `Syntactic/CommandStrcuture` again, as well as `Interaction/Complete/FreeMonad`.

The usual catamorphism construction [MFP91] gives us a uniform way to operate on these values:

```
CommandAlgebra : Set → Set
CommandAlgebra R = (c : Command C) → (Response c → R) → R

fold-algebra : CommandAlgebra R → (A → R) → FreeMonad A → R
fold-algebra alg f (Return-FM a)       = f a
fold-algebra alg f (Invoke-FM c cont) = alg c (λ r → fold-algebra alg f (cont r))
```

Note that we can regard Invoke-FM as an algebra with result FreeMonad R. We will denote this algebra by id-Alg. The instances for functor, applicative, and monad now follow immediately:

```
fmap-FM : (A → B) → FreeMonad A → FreeMonad B
fmap-FM f = fold-algebra id-Alg (Return-FM ∘ f)

ap-FM : FreeMonad (A → B) → FreeMonad A → FreeMonad B
ap-FM mf ma = fold-algebra id-Alg (flip fmap ma) mf

bind-FM : FreeMonad A → (A → FreeMonad B) → FreeMonad B
bind-FM m f = fold-algebra id-Alg f m
```

Note that return is already defined by Return-FM.

Put together, this allows us to extend our games with new operations without having to define the monadic structure each time. This is of little theoretical interest, but makes for a more straightforward implementation.

5.3 Example: Games

Let us now consider how we can express our constructions from chapter 2 using this approach. We will start by taking a simplistic but straightforward approach, and then refine it in section 5.5 to allow for greater flexibility. To begin, we can define the type of commands a CryptoExpr supports, and the corresponding responses, to get the CryptoExpr monad:[3]

```
data CryptoCmd : Set where
  Uniform  : ℕ  → CryptoCmd
  GetState :      CryptoCmd
  SetState : ST → CryptoCmd

CryptoCS : CmdStruct
Command CryptoCS = CryptoCmd
Response CryptoCS (Uniform n) = BitVec n
Response CryptoCS GetState    = ST
Response CryptoCS (SetState _) = ⊤

CryptoExpr : Set → Set
CryptoExpr = FreeMonad CryptoCS
```

Defined this way, we get the functor and monad instances of CryptoExpr for free. We can repeat this construction for OracleExpr by adding two new commands:[4]

[3] c.f. `Syntactic/CryptoExpr`.

[4] c.f. `Syntactic/OracleExpr`, but note that this is not the approach taken there.

```
data OracleCmd : Set where
  Uniform   : ℕ           → OracleCmd
  GetState  :               OracleCmd
  SetState  : ST          → OracleCmd
  InitOracle : OracleState → OracleCmd
  CallOracle : OracleArg  → OracleCmd

OracleCS : CmdStruct
Command OracleCS = OracleCmd
Response OracleCS (Uniform n)    = BitVec n
Response OracleCS GetState       = ST
Response OracleCS (SetState _)   = ⊤
Response OracleCS (InitOracle _) = ⊤
Response OracleCS (CallOracle _) = OracleResult

OracleExpr : Set → Set
OracleExpr = FreeMonad OracleCS
```

Being able to define OracleExpr this way already saves us a considerable amount of duplication compared to our construction in chapter 2, and we could choose to stop here and define the gluing of oracles by hand, as we had done earlier. However, we may note that OracleCmd is an extension of CryptoCmd. This can be used to not only obtain a yet more compact presentation, but also give a more elegant definition of the oracle implementation type Oracle.

5.4 Combining Command Structures

There are two questions we need to tackle in order to define OracleExpr in a more composable way: how can we combine two command structures in one, and how can we express the implementation of one command structure in another?

We define a binary operation $\uplus^{CS}$ on command structures that represents taking the disjoint union of their commands, with the responses to each command staying unchanged. We can express this directly in Agda:[5]

```
_⊎ᶜˢ_ : (C₁ C₂ : CmdStruct) → CmdStruct
Command (C₁ ⊎ᶜˢ C₂) = Command C₁ ⊎ Command C₂
Response (C₁ ⊎ᶜˢ C₂) (left  c) = Response C₁ c
Response (C₁ ⊎ᶜˢ C₂) (right c) = Response C₂ c
```

If we regard FreeMonad C_1 A as a term with result A that may use commands from C_1, then FreeMonad $(C_1 \uplus^{CS} C_2)$ A is a term with result A that may use commands from both C_1 and C_2. This, therefore, solves the first half of our problem. For the future, we remark that $\_\uplus^{CS}\_$ has an identity, namely the command structure with no commands:

```
0ᶜˢ : CmdStruct
Command 0ᶜˢ = ⊥
Response 0ᶜˢ ()
```

[5] c.f. `Syntactic/CSConstructs` and `Interaction/Complete/InteractionStructure`.

For the second half, we need to ask what an interpretation of C_1 in C_2 means. A simple approach would be to map the commands of C_1 to the commands of C_2 and map the responses in the other direction,[6] but this would be too restrictive: an implementation could consist of only a single command, rather than an arbitrary monadic term. Instead, to implement a command structure C_1 in terms of a command structure C_2, we send every command c of C_1 with response type $R\ c$ to a term in the free monad of C_2 with result $R\ c$. In code, this is given as[7]

```
Impl : (C₁ C₂ : CmdStruct) → Set
Impl C₁ C₂ = (c : Command C₁) → FreeMonad C₂ (Response C₁ c)
```

This choice is in line with the development performed by McBride [McB15] on free monads, and in fact every such implementation of C_1 in terms of C_2 gives rise to a unique monad morphism from FreeMonad C_1 to FreeMonad C_2. Moreover, we can define a function of the following type:

```
parallel-impl : Impl C₁ C₂
              → Impl D1 D2
              → Impl (C₁ ⊎ᶜˢ D1) (C₂ ⊎ᶜˢ D2)
```

We invite the reader to go back and check that the Oracle type defined earlier is in fact, up to isomorphism, an example of an implementation. In the next section, we will make this explicit.

5.5 Example: Oracle Implementations

Previously, we defined CryptoCS and OracleCS entirely independently. However, the latter is an extension of the former. Using the $\_\uplus^{CS}\_$ operation defined above, we can make this explicit in our Agda code:[8]

```
data OracleCmd : Set where
  InitOracle : OracleState → OracleCmd
  CallOracle : OracleArg   → OracleCmd

OracleBaseCS : CmdStruct
Command OracleBaseCS = OracleCmd
Response OracleBaseCS (InitOracle _) = OracleResult
Response OracleBaseCS (CallOracle _) = ⊤

OracleCS : CmdStruct
OracleCS = CryptoCS ⊎ᶜˢ OracleBaseCS

OracleExpr : Set → Set
OracleExpr = FreeMonad OracleCS
```

[6] Although we skim over this idea, this is in fact a very useful notion, as it gives rise to a category of command structures. Unfortunately, while being undeniably *interesting*, this category has not proven to be useful in any concrete way.

[7] c.f. `Syntactic/CommandStructure` and `Interaction/Complete/Implementation`. The latter is more faithful to this presentation.

[8] c.f. `Syntactic/OracleExpr`.

Defining OracleExpr this way gives us more information than our initial definition: we have now made explicit that in order to implement an oracle in terms of our base language, we must define a term of type Impl OracleBaseCS CryptoCS. By using parallel-impl above, we can extend this to an Impl OracleCS CryptoCS, giving rise to a function OracleExpr $A \to$ CryptoExpr A.

There is a problem with this approach, namely in the usage of state. In section 2.2, we had allowed for two state types, AST and OST, for the adversary and oracle state respectively. However, here we have used the same type ST, and the implementation of OracleCS in terms of CryptoCS proposed above would merge the operations. This is not the expected behaviour.

We can rectify this by parametrising CryptoCS by the type of the state, and then showing that there is an implementation of CryptoCS $AST \uplus^{CS}$ CryptoCS OST in terms of CryptoCS $(AST \times OST)$. With this change in place, the composition of implementations corresponds to the eval function defined in section 2.2. However, unlike in the earlier development, we have been left with no ad-hoc choices once we specified what the permitted commands were. We could use this technique to add further players, for example to specify the adversary as a player explicitly, or to modify the oracle interface, and the effects of these changes would be propagated automatically.

5.6 Multiplayer Systems

In the example above, we have seen how we can express the relationship between the adversary and the oracle explicitly. We would like to generalise this notion to be able to represent an arbitrary N-player system, where player i may make use of the public interface of player j iff $i < j$. This makes it possible to specify the games in a more modular manner. Moreover, we conjecture that this approach can give rise to a uniform way of extending the notion of ϵ-indistinguishability on games to a similar notion on such larger systems.

Let us start by considering the following example: how would we represent the IND-CPA game as a four-player system, the four players being the challenger, encryption scheme, adversary, and oracle? It may seem strange to include "encryption scheme" in this list, but being able to do so easily is an important benefit of this approach. By phrasing the problem this way, we can perform all the gluing of players at once, simplifying the system as a whole.

Each of the four players has the basic core language CryptoCS, parametrised by their state type. The challenger and encryption scheme are stateless, while the adversary has state AST and the oracle has state OST. The interfaces are as follows:[9]

```
Command ChallengerCS = ⊤
Response ChallengerCS tt = Bool

data EncSchemeCmd : Set where
  keygen  : EncSchemeCmd
  encrypt : K → PT → EncSchemeCmd

Command EncSchemeCS = EncSchemeCmd
Response EncSchemeCS keygen      = K
```

[9] c.f. `Interaction/Complete/Example`.

```
Response EncSchemeCS (encrypt _ _) = CT
data AdversaryCmd : Set where
  gen-messages : AdversaryCmd
  guess-coin   : CT → AdversaryCmd
Command AdversaryCS = AdversaryCmd
Response AdversaryCS gen-messages = PT × PT
Response AdversaryCS (guess-coin _) = Bool
```

We have already seen the definition of OracleBaseCS, which defines the oracle. We will refer to it as OracleCS in this section for the sake of uniformity.

The command structures are as we would expect: the possible actions and their arguments are captured in the command data structures, and the responses are the result types of these action. Having phrased it this way, we see that the types EncScheme and Adversary ST we defined in chapter 1 are in fact equivalent to

```
∀{st} → Impl EncSchemeCS (CryptoExpr st)
```

and Impl AdversaryCS (CryptoExpr ST) respectively. This supports our four-player approach: types that we had to write out explicitly in the past are now generated for us.

Note that the command structure for the challenger has a single command that represents running the entire game. Represented this way, the game itself is an implementation, rather than simply a CryptoExpr term. This suggests that the right notion of ϵ-indistinguishability should be in terms of implementations.

The implementations we need to have a complete description of the game are as follows:

- an implementation of the oracle in terms of the base language;
- an implementation of the adversary in terms of the oracle and the base language;
- an implementation of the encryption scheme in terms of the base language; and
- an implementation of the challenger in terms of the encryption scheme, adversary, and the oracle.

Since we choose the encryption scheme, it is not a problem if the encryption scheme also has access to the adversary and the oracle: we can simply consider encryption schemes that make no use of this access. In code, the above thus becomes

```
Impl OracleCS       (CryptoCS OST)
Impl AdversaryCS    (CryptoCS AST ⊎CS OracleCS)
Impl EncSchemeCS    (CryptoCS ⊤   ⊎CS AdversaryCS
                                  ⊎CS OracleCS)
Impl ChallengerCS   (CryptoCS ⊤   ⊎CS EncSchemeCS
                                  ⊎CS AdversaryCS
                                  ⊎CS OracleCS)
```

We will now define a structure that, given a list of interfaces and their corresponding base languages, represents a list of implementations like the above. Such an "N-player

implementation" is either an empty list, or an implementation of the first interface in terms of the first base language and the other interfaces, together with an N-player[10] implementation of the tails.[11]

```
sum^CS : List CmdStruct → CmdStruct
sum^CS = foldr _⊎^CS_ 0^CS
data N-Impl : (infc base : List CmdStruct) → Set where
  []    : N-Impl [] []
  _::_ : {C1 D1 : CmdStruct} {CS DS : List CmdStruct}
       → (Impl C1 (D1 ⊎^CS sumCS CS))
       → N-Impl CS DS
       → N-Impl (C1 :: CS) (D1 :: DS)
```

Given an N-Impl CS DS, we can construct an Impl (sum^{CS} CS) (sum^{CS} DS), exactly as we wanted: in the recursive case, when CS is of the form C_1 :: CS' and DS is of the form $D1$ :: DS', this can be done by implementing C_1 $\uplus^{CS}$ CS' in terms of $D1$ $\uplus^{CS}$ CS' $\uplus^{CS}$ CS' using the first implementation, then merging the two occurences of CS', and finally implementing CS' in terms of DS' by recursion.

With this result, we have developed a general method of constructing types such as EncScheme, Adversary, and Oracle in a manner that is independent of the number of players involved. While not essential for any specific example, we feel this would be an important part of any larger system that made use of command structures. This is a direct generalisation of the constructs in chapter 2.

5.7 Future Work

While the notion of an N-player implementation is helpful in constructing games, by itself it does not solve our problem of reasoning about games. For this, a comparable generalisation of the material of chapter 3 would be needed. Specifically, a way to automatically generalise a notion of ϵ-indistinguishability from the base language to an N-player implementation, would be invaluable.

The primary difficulty here lies in the fact that a player may make use of a later player's interface multiple times, making a slight change in the implementation of the latter compound. As we have seen in section 2.3, limiting the number of times an operation is used is possible but involves constructing appropriate proof terms, which would make the proofs even more verbose.

[10] Or, perhaps more correctly, $(N-1)$-player.

[11] The full implementation of this can be found in `Interaction/Complete/Combine`. Due to the verbosity, we only present the key ideas here.

Chapter 6

Indexed Monads

Let us return to the problem of enforcing an upper bound on the number of times the adversary may query the oracle. In section 2.3 we have seen a way of imposing such a bound by defining an additional data structure which can only be constructed if the bound is respected. In this chapter, we will explore an alternative approach that makes use of McBride's notion of an indexed monad [McB11].

A major frustration with the usage of an additional datatype for imposing a boundary on the adversary is that we must explicitly construct the proof, despite its structure being determined by the structure of the game it refers to. We can avoid this by encoding the same proof information in the OracleExpr type itself. Fixing an ST type for the (adversary) state, we may imagine the following datatype to work:[1]

```
data OracleExpr : ℕ → Set → Set where
  Return    : A                                  → OracleExpr k       A
  Uniform   : ... → (... → OracleExpr k A)       → OracleExpr k       A
  GetState  : ... → (... → OracleExpr k A)       → OracleExpr k       A
  SetState  : ... → (... → OracleExpr k A)       → OracleExpr k       A
  InitOracle : ... → (... → OracleExpr k A)      → OracleExpr k       A
  CallOracle : ... → (... → OracleExpr k A)      → OracleExpr (suc k) A
```

Indeed, a term of type OracleExpr k A can make at most k calls to the oracle. However, this naïve attempt fails to be a monad: the CallOracle case of bind does not go through. This is to be expected, since binding may change the number of oracle calls performed. As such, we need to extend our notion of a monad to allow terms to have an index, and allow the bind to modify this index.

6.1 Definition

In functional programming, we are used to the term monad referring specifically to monads on the category of types and terms of the language we are using. However, the

[1] We have not expressed this type explicitly, but `Interaction/Indexed/Memory` gives a simpler example based on these ideas.

mathematical definition of monad can refer to endofunctors on any category. The following is simply a specialisation of this definition to the category of types and terms indexed by some type S.

Definition 18. Given any type S, the category $SetS$ is the category with functions $S \to \mathsf{Set}$ as its objects and S-indexed families of functions as its morphisms.

We define the morphisms in Agda as follows:

```
_⇝_ : (S → Set) → (S → Set) → Set
A ⇝ B = ∀ {s} → A s → B s
```

Given a function $(S \to \mathsf{Set}) \to (S \to \mathsf{Set})$, we can regard it as a functor if it has a corresponding action on morphisms. This gives rise to the notion of an S-indexed functor on Set, or endofunctor on $SetS$, as follows:[2]

```
record IxFunctor {S : Set} (F : (S → Set) → (S → Set)) : Set₁ where
  field
    fmapⁱ : (A ⇝ B) → (F A ⇝ F B)
```

Together with the usual notion of a natural transformation, this gives rise to the category of endofunctors and natural transformations between them. The notion of an indexed monad now arises naturally:[3]

```
record IxMonad {S : Set} (M : (S → Set) → (S → Set)) : Set₁ where
  field
    overlap {{ixfunctor-super}} : IxFunctor M
    returnⁱ : A ⇝ M A
    joinⁱ : M (M A) ⇝ M A
```

Just as with normal monads, the bind operation is of more use if our goal is to write programs. It can be defined in terms of fmap^i and join^i just as in the non-indexed case:

```
_>>=ⁱ_ : M A s → (A ⇝ M B) → M B s
m >>=ⁱ cont = joinⁱ (fmapⁱ cont m)
```

This type signature looks a lot more familiar if we flip the order of the arguments to get $(A \rightsquigarrow M\ B) \to (M\ A \to M\ B)$.

A useful intuition is that a term $m : M\ A\ s$ is a computation that starts at state s and ends at some (unknown) state s'. Since s' is not known, the continuation must work for *any* s'. This is essential to avoid the problem we encountered when we tried to implement bind for OracleExpr at the begining of the chapter. However, on the face of it, this seems very restrictive: in our example, this means that the continuation must work no matter how many oracle queries may still be made. Can it query the oracle at all?

Fortunately, the Atkey construction [McB11] can be used to work around this limitation.

[2] c.f. `Algebra/Indexed/Functor`.

[3] c.f. `Algebra/Indexed/Monad`.

6.2 The Atkey Construction

An important feature of indexed monads is that the type of the result may depend on the index. That is, unlike our definition of OracleExpr above, the type of an indexed monad is $(S \to \mathsf{Set}) \to S \to \mathsf{Set}$ rather than $S \to \mathsf{Set} \to \mathsf{Set}$. This means that in the call $m \mathbin{>\!\!>\!\!=^{\mathsf{i}}} f$, the type of values f operates on depends on the index at which it is used.

The original Atkey construction [McB11] makes use of this by constructing a type that is empty except at one selected index. This can be defined as follows:[4]

```
data Atkey (A : Set) : S → S → Set where
  V : A → Atkey A s s
```

Suppose we have an indexed monad M, a term $m : M\ (\mathsf{Atkey}\ A\ s')\ s$, and a continuation $cont : A \to M\ B\ s'$. We can use bind as follows:

```
m >>=ⁱ λ { (V a) → cont a }
```

In the context of the lambda, we know that the only value $\mathsf{Atkey}\ A\ s'\ s''$ can attain is $\mathsf{V}\ a$, and thus $s' \equiv s''$. The case we provided typechecks, since $f\ a : M\ B\ s'$. Since all other cases are empty, we do not have to write them out. The expression as a whole has type $M\ B\ s$.

There are a number of variations on this construction. We will explore one generalisation in particular that will come in useful later.

Suppose that we have a monadic computation that starts at an index s and that yields a value of type A. How can we express that given the value $a : A$ returned, we know the state s' that the computation ended at? For example, suppose that we have an oracle that may refuse queries: it returns a value of type Maybe OracleResult. If the query was refused, the number of uses does not go down. We know that if we query the oracle at state $\mathsf{suc}\ k$, we end up in either state k (if we get $\mathsf{just}\ x$) or $\mathsf{suc}\ k$ (if we get nothing). We thus want an indexed type which contains values of the form $\mathsf{just}\ x$ at index k and the value nothing at index $\mathsf{suc}\ k$, and is empty elsewhere.

We can solve this using a dependent Atkey construction as follows, where $f : A \to S$ is the function that indicates the ending state for a given value $a : A$.

```
data DepAtkey (A : Set) (f : A → S) : S → Set where
  DepV : (a : A) {s : S} → (pf : s ≡ f a) → DepAtkey A f s
```

Consider the expression $m \mathbin{>\!\!>\!\!=^{\mathsf{i}}} \lambda\ \{(\mathsf{DepV}\ a\ \mathsf{refl}) \to ?\}$. Matching on the pf argument lets us rewrite the type of the hole to $M\ B\ (f\ a)$. It follows that a continuation to a term $m : M\ (\mathsf{DepAtkey}\ A\ f)\ s$ corresponds to a function $(a : A) \to M\ B\ (f\ a)$.

Interestingly, this type arises as the left Kan extension of the functor $\mathsf{const}\ A : A \to \mathsf{Set}$ along $f : A \to S$, where A and S are seen as discrete categories.

6.3 Oracle Query Bounds

Let us return to our motivating example. We can define the OracleExpr type from above as follows:

[4] c.f. `Algebra/Indexed/Atkey`.

```
data OracleExpr (A : ℕ → Set) : ℕ → Set where
  Return    : A k                                  → OracleExpr A k
  Uniform   : ... → (... → OracleExpr A k)  → OracleExpr A k
  GetState  : ... → (... → OracleExpr A k)  → OracleExpr A k
  SetState  : ... → (... → OracleExpr A k)  → OracleExpr A k
  InitOracle : ... → (... → OracleExpr A k)  → OracleExpr A k
  CallOracle : ... → (... → OracleExpr A k)  → OracleExpr A (suc k)
```

The definitions of fmap, bind, and return all proceed like their non-indexed counterparts. However, the smart constructors need a change, since Return has the wrong type to be a continuation for the other constructors. Instead, we use the Atkey construction:

```
uniform : ℕ → OracleExpr (Atkey ℕ k) k
uniform n = Uniform n λ v → Return (V v)
```

The other smart constructors are similar, except that call-oracle has type OracleArg → OracleExpr (Atkey OracleResult k) (suc k) to indicate that it uses up one call to the oracle.

Effectively, we have now merged the call-counting portion of BoundedOracleUse from section 2.3 into the OracleExpr type. One big benefit of this merge is that we retain this information under binding without having to separately manipulate a proof term. We can also extend this system to track other properties, such as how many bits of randomness are necessary, whether the state has been accessed, and whether the oracle has been initialised.

Unfortunately, this solution also has some considerable drawbacks. While we *can* keep track of many things in this way, we cannot easily choose to track some things and ignore others, and the increasing number of type parameters makes the code hard to read. We will expand on what might be done about this in section 6.7 and chapter 7.

6.4 Player State Types

Indexed monads can play another role in the development of games. Recall that we had to introduce separate notions of indistinguishability and result-indistinguishability, since the initial and final state of the game are not of interest to us when deciding who won. This is rather unfortunate, as the logic of indistinguishability is far more straightforward.

Using indexed monads we can avoid this problem by allowing the players to modify the type of their state during execution. This allows us to specify that a game starts and ends with state type ⊤, making the notions of indistinguishability and result-indistinguishability coincide. Within it, the adversary may switch to a different type of state to store whatever information it needs, as long as it sets the type back to ⊤ when it is done.

We would like to define CryptoExpr as follows:

```
data CryptoExpr : (Set → Set) → Set → Set where
  Return   : A s                                          → CryptoExpr A s
  Uniform  : (n : ℕ) → (BitVec n → CryptoExpr A s)  → CryptoExpr A s
  GetState :           (s → CryptoExpr A s)          → CryptoExpr A s
  SetState : s'      → (⊤ → CryptoExpr A s')         → CryptoExpr A s
```

Unfortunately, this definition does not type-check: the Return constructor is polymorphic in s : Set, and so CryptoExpr A s has type Set_1, not Set. This problem can be remedied by giving CryptoExpr the type (Set → Set_1) → Set → Set_1, but this is very impractical. For one, since Agda does not have universe cumulativity, such an approach requires our Atkey constructions to *also* be in Set_1 (or universe-polymorphic), and the code quickly becomes unmaintainable.

A more manageable solution is to define a universe U and an evaluation function eval : U → Set, and let CryptoExpr be a U-indexed monad. This has the added benefit of letting us impose constraints on state types by our choice of U: for example, we can ensure that all state types have decidable equality.

A natural question to ask is whether the state monad transformer we used in order to define the list interpretation can be generalised to this context. This is indeed the case: although it has the same universe size issues as CryptoExpr, fixing an S-indexed monad M, some custom universe U, and a function eval : U → Set we can define:[5]

```
IxStateT : (U × S → Set) → U × S → Set
IxStateT A (u , s) = eval u → M (λ s' → Σ U λ u' → eval u' × A (u' , s')) s
```

This code can be a little confusing due to the overloaded meaning of 'state': on the one hand, it refers to the pair u , s that we have on the type level, and on the other to the values of type eval u that we pass around on the value level. To disambiguate, we will refer to the former as the 'index' and to the latter as the 'message'.

The definition of IxStateT can then be expressed as follows: a term of IxStateT A (u , s) is a function that takes a message of type u and returns a monadic action in M at index s which has, as result value at index s', a new message type u', a message of that type, and a value of type A (u' , s'). If you squint, this resembles the type U → M (U , A) of a non-indexed state monad transformer.

The definition of modify is less complicated and makes it clear why this approach works:

```
modify : (eval u → eval u') → IxStateT (Atkey (eval u') (u' , s)) (u , s)
modify f v = return (u' , fv , V fv))
  where fv = f v
```

Here, v is the message. We apply the function f to it, and then store the new message type, the new message, and give as result the new message wrapped in an Atkey. The definitions of get and set can, fortunately, be derived from this, and the definitions of bind and return are straightforward.

With this development in place, we believe that the interpretations described in chapter 4 can be replicated in this context, while resolving entirely the question of result-indistinguishability in a straightforward manner.

6.5 Interaction Structures

In chapter 5, we saw a way of encoding the command-response structure in a way that allowed us to generate the corresponding free monad automatically. Interaction structures allow us to do the same in the indexed context.

[5] c.f. `Utility/State/Indexed/FromUniverseTransformer` and other files in that directory.

Let S be our index type. An interaction structure consists of a type of commands Command s for each s : S, a type of responses Response c for each command c : Command s, and a next state next c r for each c : Command s and r : Response c.

We can implement this in Agda:[6]

```
record IntStruct (S : Set) : Set where
  field
    Command  : S → Set
    Response : {s : S} → Command s → Set
    next     : {s : S} (c : Command s) (r : Response c) → S
```

An interaction structure over a state type S gives rise to a free indexed monad over S as follows:[7]

```
data FreeMonad (IS : IntStruct S) : (S → Set) → (S → Set) where
  Return-FM : A s → FreeMonad IS A s
  Invoke-FM : (c : Command IS s) → ((r : Response IS c)
            → FreeMonad IS A (next IS c r))
            → FreeMonad IS A s
```

This follows the usual pattern we have seen before: to invoke a particular operation, we specify it (with all parameters) and then we provide a continuation that handles the possible responses. The fmap and bind functions are also straightforward generalisations:

```
fmap-FM : A ⇝ B → FreeMonad IS A ⇝ FreeMonad IS B
fmap-FM f (Return-FM a)      = Return-FM (f a)
fmap-FM f (Invoke-FM c cont) = Invoke-FM c λ r → fmap-FM f (cont r)
bind-FM : A ⇝ FreeMonad IS B → FreeMonad IS A ⇝ FreeMonad IS B
bind-FM f (Return-FM a)      = f a
bind-FM f (Invoke-FM c cont) = Invoke-FM c λ r → bind-FM f (cont r)
```

We have flipped the arguments of bind-FM to emphasise the indexed structure.

6.6 Multiplayer Systems

Just like in the non-indexed case, we can consider the question of how to create an N-player implementation. The essential construction does not change, but we have to redefine the implementation and the $\uplus^{CS}$ operation in this context.

For the definition of Impl, we need the DepAtkey construction we defined earlier. An important note here is that the structure we are interpreting may be indexed over a different set than the structure we are interpreting it in. For this purpose we take the *Sf* map, which relates states in the one to states in the other.[8]

```
Impl : (IS : IntStruct S₁) (M : (S₂ → Set) → S₂ → Set) (Sf : S₁ → S₂)
     → Set
```

[6] c.f. `Interaction/Indexed/InteractionStructure`.
[7] c.f. `Interaction/Indexed/FreeMonad`.
[8] c.f. `Interaction/Indexed/Implementation`.

```
Impl IS M Sf
  = (c : Command IS s) → M (DepAtkey (Response IS c) (Sf ∘ next IS c)) (Sf s)
```

Note that if both S_1 and S_2 are singletons then this definition is equivalent to the definition of Impl from chapter 5.

The definition of $\uplus^{CS}$ in this context is more complicated. The problem is that in general, while $C \ \uplus^{CS} \ C$ can be implemented in terms of C, this does not carry over into the indexed case. This can be resolved by defining two different constructions on interaction structures, one of which is used for combining interfaces and the other for combining base languages.

Let us start with the construction for base languages. The essential property we use is that our base languages do not in any way influence each other's state. This allows us to use the following definition, which appears to be a straightforward generalisation of $\uplus^{CS}$:[9]

```
_⊕_ : IntStruct S₁ → IntStruct S₂ → IntStruct (S₁ × S₂)
Command (IS₁ ⊕ IS₂) (s₁ , s₂) = Command IS₁ s₁ ⊎ Command IS₂ s₂
Response (IS₁ ⊕ IS₂) {s₁ , s₂} (left c)  = Response IS₁ c
Response (IS₁ ⊕ IS₂) {s₁ , s₂} (right c) = Response IS₂ c
next     (IS₁ ⊕ IS₂) {s₁ , s₂} (left c) r  = next IS₁ c r , s₂
next     (IS₁ ⊕ IS₂) {s₁ , s₂} (right c) r = s₁ , next IS₂ c r
```

Just like the $\uplus^{CS}$ construction, this construction has a unit, and we can fold over this construction. We are cheating slightly here: in reality, the argument to sum^{IS} is not simply a list, since it may store elements of type IntStruct S for any S. We correct this in the code, but the difference is not essential here.

```
1^IS : IntStruct ⊤
Command 1^IS tt = ⊥
Response 1^IS {tt} ()
next     1^IS {tt} ()
sum^IS = foldr _⊕_ 1^IS
```

The $\_\oplus\_$ construction defined above can be seen as taking two interaction structures and combining their state orthogonally, so that commands that influence one state component do not influence the other. However, this is not how players interact: if the oracle may be queried n times and an adversary command uses k of these queries, the challenger must be aware of this. Essentially, the state of every player must include the state of all players that they can issue commands to. To capture this notion, we introduce a second operation on interaction structures denoted $\_\oplus_{/\sim}\_$. It can be seen as the $\_\oplus\_$ operation from above with a quotient applied to the state space. Again, we define a fold over this operation as well.[10]

```
_⊕/∼_ : IntStruct (S₁ × S₂) → IntStruct S₂ → IntStruct (S₁ × S₂)
Command (IS₁ ⊕/∼ IS₂) (s₁ , s₂) = Command IS₁ (s₁ , s₂) ⊎ Command IS₂ s₂
Response (IS₁ ⊕/∼ IS₂) {s₁ , s₂} (left c)  = Response IS₁ c
```

[9] c.f. `Interaction/Indexed/InteractionStructure`.

[10] c.f. `Interaction/Indexed/QuotientTensor`.

$$
\begin{aligned}
&\mathsf{Response}\ (IS_1 \oplus_{/\sim} IS_2)\ \{s_1\ ,\ s_2\}\ (\mathsf{right}\ c)\ =\ \mathsf{Response}\ IS_2\ c\\
&\mathsf{next}\ \ \ \ \ (IS_1 \oplus_{/\sim} IS_2)\ \{s_1\ ,\ s_2\}\ (\mathsf{left}\ c)\ r\ =\ \mathsf{next}\ IS_1\ c\ r\\
&\mathsf{next}\ \ \ \ \ (IS_1 \oplus_{/\sim} IS_2)\ \{s_1\ ,\ s_2\}\ (\mathsf{right}\ c)\ r\ =\ s_1\ ,\ \mathsf{next}\ IS_2\ c\ r\\
&\mathsf{sum}^{IS}_{/\sim}\ =\ foldr\ \_\oplus_{/\sim}\_\ 1^{IS}
\end{aligned}
$$

With these choices in place, we can construct N-player implementations as we did before. The Agda formulation is not enlightening, so we will not present it here.[11] The essential point is that just as we could use an implementation of C_1 in terms of $D1$ and $\mathsf{sum}^{CS}\ CS$ and an implementation of $\mathsf{sum}^{CS}\ CS$ in terms of $\mathsf{sum}^{CS}\ DS$ to obtain an implementation of $C_1\ \uplus^{CS}\ \mathsf{sum}^{CS}\ CS$ in terms of $D1\ \uplus^{CS}\ \mathsf{sum}^{CS}\ DS$ in the command structure case, here we achieve the same result, with the important distinction that we now have two ways of combining interaction structures. The choice that has worked for our purposes is as follows: an implementatino of C_1 in terms of $D1 \oplus \mathsf{sum}^{IS}_{/\sim}\ CS$ and an implementation of $\mathsf{sum}^{IS}_{/\sim}\ CS$ in terms of $\mathsf{sum}^{IS}\ DS$ gives an implementation of $C_1 \oplus_{/\sim} \mathsf{sum}^{IS}_{/\sim}\ CS$ in terms of $D1 \oplus \mathsf{sum}^{IS}\ DS$.

This gives rise to the same kind of N-player implementation that we already discussed in the case of command structures in chapter 5. In fact, this saves us a considerable amount of work: the generalisation of simple games to games with oracles follows in its entirety from this construction. However, we also run into the same issue: we do not yet know how to generalise ϵ-indistinguishability from a definition on the base language to a general definition. As such, just like in the previous case, the proofs of equality will necessarily be low-level compared to the expressions of the games themselves.

6.7 Future Work

Indexed monads are a very powerful tool, but their verbosity and lack of modularity makes them unappealing to use directly. In section 6.3 we have seen how they can be used to restrict what games we consider well-formed. If we have two such constraints, we have no way to express them separately and then enable one or both depending on our present needs. It would be interesting to see whether such a system could be developed. This could also lead to a more concise formulation of what effect a monadic action has on the index.

Also in the oracle example, we glossed over the fact that the number of permitted calls to the oracle does not ever increase. As such, given an m : OracleExpr $A\ k$ we only need an $f\ :\ A\ i\ \rightarrow$ OracleExpr $A\ i$ that works for $i \leq k$, not one that works for every i. This is explored by Visser *et al.* [BPRT+17], and appears to rely on a categorical structure on the index set. Expressing the conditions and resulting definition of bind explicitly could be of interest.

[11] c.f. `Interaction/Indexed/Telescope`.

Chapter 7

A Language for Cryptography

In the course of the last chapters, we have developed a way of representing cryptographic games as monadic action, developed an equational theory of indistinguishability between games, and shown how we can extend and simplify this with the help of indexed monads. We have also seen that this construction can be done in a modular manner using command or interaction structures. One question remains: what must be done to turn these developments into a practical system for reasoning about cryptography?

7.1 A Complete System

There are a number of questions that fall within the scope of this thesis, but that have not been able to resolve due to time constraints. We have already discussed them in greater depth at the end of the relevant chapters, but let us recap the key points.

In chapter 3, we defined the notions of an ϵ-normal form and an (ϵ, st)-normal form for CryptoExprs. However, we did not prove a similar result for OracleExprs. This is unfortunate, as many game-playing proofs rely on a transformation to this kind of normal form [Sho04].

A planned topic for chapter 3 was the "identical until bad" proof technique [Sho04], which states that if games X and Y are different only in the case of a bad event F happening, then X and Y are ϵ-indistinguishable, with ϵ being the probability of F. This is a very useful proof step, but we have not found a way to express it in Agda. The problem is that the bad event F is often implicit in the game: for example, the event may be "two calls to uniform n return the same bitstring" or "the adversary finds a hash collision." Instrumenting the game code to state when F occurs can be a non-trivial problem even in concrete cases; a general solution would be of great value.

In chapter 5, we developed the N-player implementation construction and showed how we can systematically fold such an implementation into a single game. If an ϵ-relation like indistinguishability was defined on this game, then this fold gave rise to an ϵ-relation on the whole implementation. However, this induced relation is inconvenient to work with: we would like to be able to express our indistinguishability proofs in terms of the implementations themselves, and then show via a lemma that the indistinguishability of their folds follows. The difficulty lies in striking a balance between what can be shown and how much bookkeeping this requires.

In chapter 4, we present a list-based model of game logic. We have constructed the carrier of this model in Agda, but we have not verified all of the properties we require of it. The proofs involved are typically not hard conceptually, but require many steps to formulate in Agda. This suggests that a proof system for arguments about lists may be an interesting project in its own right.

Another important issue with this model is the treatment of types that lack decidable equality. This issue is made particularly frustrating by the fact that in cryptography, our problem domain, every value we may want to discuss can be represented as a bit-string; as such, the question of types without decidable equality does not arise in practice. Nevertheless, it is a blemish in our theory.

The last issue worth mentioning is that our model theory is specified in the non-indexed case. We see no fundamental obstacles to generalising the argument to the indexed case, but we have not done so ourselves.

7.2 Further Possibilities

The greatest obstacle in using the ideas we have presented is the verbosity. Even in the non-indexed case, expressing the indistinguishability of two games will typically involve multiple applications of the fact that indistinguishability is a congruence under bind. In the indexed case, the situation is even worse, as the indices may also need to be explicitly specified.

We expect that reflection could be used to tackle these issues. Ulf Norell's `agda-prelude`[1] library features tactics: these allow the user to write *by pf*, where *pf* is some equality proof, and allow the library to find a proof of the type expected by the context. A similar technique could be used for proofs of ϵ-indistinguishability to automatically find the applications of congruence needed to prove a theorem. The pinnacle of this approach would be to allow the user to invoke an external SMT solver.

Reflection could also be used to take non-indexed descriptions of games and generate corresponding indexed versions. This would free up the syntactic burden from the programmer, while retaining the extra safety. However, it is not clear whether this can be done without considerable loss of expressive power.

The use of command and interaction structures to simplify the defintion of games also induces some syntactic burden. In particular, specifying implementations and their folds is done in a style that is unfriendly to the user. A language with dedicated syntactic constructs for defining an N-player implementation could make the process significantly easier, while still benefiting from our abstract approach under the hood.

Such a language would benefit from the development of appropriate tooling. In our experiments in Agda, we found ourselves repeating the game we were operating on many times as we made minor changes to it. These games could (at least partially) be generated automatically based on the rewrite rules we applied. However, we were unable to do this within Agda. This is even more important in the indexed case, where we do not want to make the user write out every index, but do want to display this index if the user asks for it.

[1] `https://github.com/UlfNorell/agda-prelude`

7.3 Conclusions

In this thesis, we have laid a foundation for a system for reasoning about indistinguishability in cryptography. There is still considerable engineerign work to be done before practical proofs can be expressed, but we nevertheless consider this a successful proof of concept that shows the viability of a syntactic approach to this problem.

The primary drawback we have discovered when using Agda is that automating even very simple proofs is difficult. In particular, it is not sufficient to specify what rewrite step we wish to apply, but also precisely where. This causes significant duplication, since often almost the entire game has to be written out as part of the rewrite step.

Another problem arises when we introduce indexing to our games: the correctness of the code with respect to indices must be specified inline with the code itself. For example, if we want to pass a BitVec n to a function that takes a BitVec k, we must first provide a proof that $n \equiv k$ and then use a rewrite rule or transport. This obscures the primary logic of the code; in an ideal system, we could write the code first and then prove it correct separately.

However, we feel that these issues are primarily ones of presentation, and not inherent to the approach we are taking. The usage of N-player implementations to represent the interactions between the challenger, adversary, and oracle makes it possible to statically enforce constraints on what the players can do, that are hard to express otherwise. As far as we know, this usage of command and interaction structures is novel, and we consider this the primary contribution of this work.

Appendix A

Notation

For the reader's convenience, we include an overview of the Agda features that we use in this text. This is by no means a comprehensive introduction to Agda, nor do we include all features used in the accompanying code. Our purpose is primarily to make the text accessible to readers with a purely mathematical background.

A more thorough overview can be found at `https://agda.readthedocs.io/en/v2.5.4.1/`.

A.1 Built-in Types

Agda can be regarded as an implementation of the dependently typed λ-calculus $\lambda\Pi$, corresponding to the internal language of locally (bi)Cartesian closed categories. In particular, finite product and and coproduct types exist, as do function types, dependent function types, and dependent product types. The notation for these types is as follows:

- The empty type is denoted $\bot$.
- The unit type is denoted $\top$ and has a single element $\mathsf{tt} : \top$.
- The product of types A and B is denoted $A \times B$. Elements of this product are pairs $a\ ,\ b$ with $a : A$ and $b : B$.
- The coproduct of types A and B is denoted $A \uplus B$. Elements of this coproduct are of the form $\mathsf{left}\ a$ with $a : A$ or $\mathsf{right}\ b$ with $b : B$.
- The type of functions from A to B is denoted $A \to B$. We shall give several examples of function definitions in section A.3. Functions are associated to the right, so $A \to B \to C$ denotes $A \to (B \to C)$.
- The type of dependent functions that take an $a : A$ and return a $B\ a$ is denoted $(a : A) \to B$. They are defined in the same way as non-dependent functions.
- The type of dependent pairs that contain an $a : A$ in the first component and a $B\ a$ in the second are denoted $\Sigma\ A\ B$.

Additionally, Agda has an ω-indexed hierarchy of universes: all basic types are of type Set, which itself is of type Set_1, and so on. We also assume that the types $\mathbb{N}$ of natural numbers, Bool of Booleans, and $\mathbb{Q}$ of rational numbers are already available, together with the usual arithmetic and comparison operations on them.

Finally, equality in Agda is expressed by means of an equality type $a \equiv b$. We will discuss this type in section A.5.

A.2 Expressions

Since Agda is based on the λ-calculus, the two most important kinds of expressions it provides are uses of variables and function application. The first is what it would appear. The second is denoted by juxtaposition, so $a\ b$ denotes the application of a to b. Function application associates to the left, so $a\ b\ c$ denotes $(a\ b)\ c$.

Since operations such as addition are better written in infix form, Agda allows definitions of names to choose a different syntax for their use. This is done by denoting the locations of the arguments by underscores. We will see this usage in the definition of $\_+\_$ below, which defines addition as an infix operator. Another common usage is for the if-then-else construction:

```
if_then_else_ : Bool → A → A → A
if true then e1 else e2 = e1
if false then e1 else e2 = e2
```

Most syntax we use will be based on mathematical notation, and so will be familiar to the reader. One important exception is the $ operator, which is another way of denoting function application. The expression $a\ \$\ b$ simply means $(a)\ (b)$, but $ has extremely low precedence, making it suitable for writing terms such as $\mathsf{square}\ \$\ 2 \uplus 1$, which evaluates to 9, rather 5, as it would if we wrote $\mathsf{square}\ 2 \uplus 1$.

A.3 Value Definitions

In order to introduce a name in Agda, we specify its type and then give its definition. The syntax is as follows, with the first line giving the type, and the second giving the definition. A definition of the form $f\ x\ y\ =\ \ldots$ defines a function f that takes parameters x and y.

```
five : ℕ
five = 2 + 3
square : ℕ → ℕ
square x = x · x
```

In some cases, Agda can deduce the type of a name without it being explicitly specified. We will avoid using this for the sake of clarity.

Functions can also be defined with the use of lambda expressions. The following is equivalent to the definition of square above:

```
square : ℕ → ℕ
square = λ x → x · x
```

A.4 Type Definitions

Agda supports two ways of defining new data types: data and record definition.

A data definition inductively defines a type by specifying all of the ways in which it can be constructed. For example, the following defines the type of natural numbers:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

We call zero and suc *constructors*. Give such a definition, we can define functions by induction on the constructors:

```
_+_ : ℕ → ℕ → ℕ
zero + m = m
suc n + m = suc (n + m)
```

Agda allows recursive definitions like the above, as long as it can prove that the recursion is well-founded. If the set of possible cases is empty, we can replace the argument by () and omit the body, such as in the following example:

```
absurd : ⊥ → A
absurd ()
```

In order to do pattern-matching in lambda expressions, the body of the lambda must be enclosed in curly braces. We can, for example, define:

```
pred : ℕ → ℕ
pred = λ {(suc k) → k; zero → zero}
```

The other way of defining a new data type is a record definition, which defines a type by the fields it must provide. For example, the following defines the type of monoids over a type A:[1]

```
record Monoid (A : Set) : Set where
  field
    op : A → A → A
    e  : A
```

We can define the addition monoid on the natural numbers as follows:

```
Addition : Monoid ℕ
op Addition n m = n ⊎ m
e  Addition = 0
```

As a final note, we remark that Agda allows for implicit arguments in definitions of functions and types. These are placed within curly braces and do not need to be specified when the corresponding name is used, as long as Agda can deduce the value from other arguments. For example, the identity function is written as follows:

[1] Technically, a proper definition would include the monoid laws, but these are typically omitted.

```
id : ∀{A} → A → A
id x = x
```

Since in a usage such as `id 5`, Agda knows that A must be $\mathbb{N}$ since that is the type of 5, we can omit the argument. In this text we will generally omit implicit arguments entirely, trusting the user to replace them; this allows us to focus on the substance of the matter.

A.5 Equality

In Agda, we can express equality of terms on the type level using the $\equiv$ type. An element of type $a \equiv b$ is a proof that a and b are equal. If that is the case, there is a single constructor `refl : a ≡ a`; otherwise, the type is empty. The Agda type system is aware of this, so pattern matching on `refl` simplifies type goals. For example, the following expresses the symmetry of equality:

```
sym : ∀ {a b} → a ≡ b → b ≡ a
sym pf = ?
```

We call ? a 'hole', and use it to denote a piece of the program we have not yet written. Here, the type of the hole is $b \equiv a$. However, if we pattern match on pf, then this type changes:

```
sym : ∀ {a b} → a ≡ b → b ≡ a
sym refl = ?
```

Here, the type of ? is $a \equiv a$, since we know that a and b are equal, and may thus replace b by a in the goal. We can thus complete the definition:

```
sym : ∀ {a b} → a ≡ b → b ≡ a
sym refl = refl
```

This is how equality is handled in proofs. Equality of values at runtime is handled separately: essentially, when we ask whether two natural numbers n and m are equal, we want a proof of $n \equiv m$ or a proof of `neg` $(n \equiv m)$. This is performed using the equality comparison operator $n == m$.

While these proofs are very useful in the code, having to convert from a proof to a Boolean explicitly clutters up the code and does not aid in understanding. As such, we will (especially in chapter 1) understand $n == m$ as `true` if n and m are equal and `false` otherwise. This is purely a matter of syntactic convenience, but makes the code read much more naturally. The proper way of writing this would be `isYes` $n == m$.

A.6 Monads

In a purely functional programming language such as Agda, functions are pure: they produce a single output based on their input, deterministically. However, sometimes we wish for a function to do more: for example, to also make use of some state. An ingenious approach to this problem is to make use of the structure of monads [Wad95].

The situation is best illustrated by an example. The product-exponent adjunction gives rise to a monad $\mathsf{State}\ S\ A\ =\ S\ \to\ (S\ \times\ A)$. In addition to the action on morphisms (denoted fmap), unit (denoted return) and multiplication (denoted *join*), we can construct the following monadic actions:

```
_>>=_ : State S A → (A → State S B) → State S B
a >>= f = join (fmap f a)
_>>_ : State S A → State S B → State S B
a >> b = a >>= const b
get : State S S
get = λ s → s , s
set : S → State S ⊤
set s = λ _ → s , tt
```

This allows us to write code like the following:

```
counter : State ℕ ℕ
counter = get >>= λ n
          set (n + 1) >>
          return (n + 1)
```

We can use this to encode stateful programs in a functional programming language. However, the notation is somewhat unfortunate. Agda therefore provides us with a syntactic sugar, using which we can write the above as follows:

```
counter : State ℕ ℕ
counter = do
  n ← get
  set $ n + 1
  return $ n + 1
```

Apart from the state monad, numerous others have been found to be useful. A number of examples are covered by Moggi [Mog91].

Bibliography

[BPRT+17] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL):16:1–16:34, December 2017.

[BR06] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. *Eurocrypt 2006, LNCS*, pages 409–426, 2006.

[EK06] Martin Erwig and Steve Kollmansberger. Functional Pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16, 2006.

[Lan98] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1998.

[McB11] Conor McBride. Kleisli arrows of outrageous fortune. 2011.

[McB15] Conor McBride. Turing-completeness totally free. In *MPC*, volume 9129 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2015.

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Nor09] Ulf Norell. Dependently typed programming in agda. In *TLDI*, pages 1–2. ACM, 2009.

[PM14] Adam Petcher and Greg Morrisett. The foundational cryptography framework. *CoRR*, abs/1410.3735, 2014.

[RP02] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. 37(1):154–165, January 2002.

[Sho04] Victor Shoup. Sequences of games: A tool for taming complexity in security proofs. *IACR Cryptology ePrint Archive*, 2004, 2004.

[Swi08] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4), 2008.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.