A Type Theoretic Investigation of the Verification of Voting Protocols

Daniel Gustafsson

Chair : Peter Sestoft
Committe : Peter Dybjer and Peter Ryan

Advisor: Carsten Schürmann
Submitted: 1 April, 2016

IT University of Copenhagen

Abstract

As the world becomes more digital we see a greater push for digital elections, but we need to be cautious when we modernize to make sure we don't introduce issues that affect the election. These issues are related to trust, on the one hand, how can we trust that the result of the election is correct, and on the other hand we can we trust that the secrecy of the vote is preserved. These seemingly contradictory statements have fuelled new developments in cryptography, but the question remain can we trust these developments?

The thesis of my dissertation is that it possible to formalise the proofs used to justify both the correctness and security of such cryptographic constructions inside type theory. This requires a theory of probabilities in order to describe the probabilistic algorithms used. My contributions include, using types and type isomorphisms to simplify both specification and calculation of probabilities, in particular Σ-types are used as they correspond to summation. Furthermore, I extend the type theory with process algebraic constructions in order to capture attack games as defined by semantic security in cryptography. To type these processes I propose a session type system based on based on linear logic, extended with the possibility of depending on the actual messages sent. I prove the consistency of this extension by giving a model of it in Agda. These contributions are combined to form a library, which is named CryptoAgda, to reason about cryptography, and as an example of the approach I prove the receipt freeness of Prêt à Voter using this model.

Acknowledgements

Oh my! Writing a PhD thesis is really tough, but I wish to thank my supervisor, Carsten Schürmann, for teaching me more about dependent types, and for allowing me to research freely into topics. Furthermore I need to thank Nicolas Pouillard, for working with me on formalising cryptography in AGDA, and for always questioning ugly definitions. I also need to thank, the other Frenchmen, Nicolas Guenot, for teaching me more about proof theory and how it relates to concurrency. I must also extend my gratitude towards Lorena Ronquillo, for always answering my questions about how cryptography works.

The interesting discussions about proof theory, in particular about linear logic, that I have had with Taus Brock-Nannestad, have been enlightening, but I would probably have finished my PhD quicker if not for you always bringing new puzzles. I'm grateful to David Christiansen, for our discussions on dependent types and how they will take over the world of programming, one type at the time. Along the same line, discussing foundations of logic and computation, with Peter Brottveit Bock, have left for many late nights.

I wish to thank Jian Wang, for showing me that indeed classical logic can be useful, but I'm more grateful for introducing me to the Chinese cuisine, I hope you have found my Swedish cuisine to be as satisfactory. I wish to apologise to Agata Murawska, for always breaking your logical systems, I suppose I find it easier to break things. In order to relax I have always been able to count on Alexander Faithfull to be there and play silly video games, geography or languages quizzes, this is much appreciated.

Finally, I wish to thank my grandparents for always encouraging me to seek knowledge and approach science. My parents, and siblings, for always supporting me, at the time when that was necessary.

Contents

Introduction

It's not improvise, it's called research, when you do not know what to do.

Magnus & Brasse

Elections based on digital technologies have been used in recent years, which makes this an important topic of study. Since elections are a vital part of democracy, we need to have strong guarantees that the systems provide credible elections, which should be secure and guarantee secrecy of the vote. One reason why this is more difficult to achieve in the area of voting than in for example net banking, is because we wish for privacy of the vote. This is an important principle of democratic systems, that we can not see how someone have voted, in order to protect voters from being coerced into voting in a particular way. In fact, destroying the connection between the voter and the vote, is not limited to only hiding the link to the outside, but even hiding the link for the election officials and the state. In contrast when interacting with a bank on-line, both parties are aware of all the interaction and one wishes only to hide the nature of the interaction for third parties.

Almost all of the proposed digital systems for digital elections use some form of cryptography, in order to achieve privacy. As such it is vital to understand how to reason about cryptographic constructions that are used, which leads to the notion of semantic security proofs [Ste03, GM84]. Semantic security proofs are based on a computational information theory, where we mathematically measure leakage of using the these constructions. A system is deemed to be secure, according to a semantic security notion, if this leakage is minimised. Here leakage is measured against an adversary that is trying to figure out some sort of secret, by interacting with the system, and using the cryptographic constructions, this interaction is called a game. Leakage is then that the adversary is able to find out something which was supposed to be hidden to it. The adversary is in general modelled by a probabilistic polynomial time algorithm, that will be interacting with the system in some way which can be seen as playing a game[BR06, Sho04]. The reason for the adversary to be probabilistic is since some cryptographic construction is using randomness and the adversary must be able to use these constructions.

There is a lot of different types of mathematics that are involved in order to be able to do such proofs, and as such the possibility of mistakes creeping in increases. Since the systems that we wish to prove correct are of such importance, as e.g. a voting system, we need to be able to trust that these proofs have been carried out in a correct manner. As such this thesis formalises the background material needed, and creates a library, in the dependent type theory of AGDA [Nor07], for formally verify semantic security proofs. By formalising the proofs we get a stronger guarantee that the proofs are indeed valid, and furthermore the proof assistant is helpful when doing the proof in pointing out what have been done, and what still needs to be done. The reason for doing this within a type theory is to use to already existing notion of computation that is already resides within type theory, since type theory is based on programming languages, rather than set theory.

In the semantic security proofs there is often two, or more, agents communicating with each other, sending messages to each other using the cryptographic constructions. One of theses agents is the adversary who is trying to find a out anything about the secret. Usual type theories does not by come with a method for describing these kind of communications by themselves, and as such taking inspiration, and taking ideas from process calculus [Mil99, Hon93] is desirable.

Related Work

Canonical big operators [BGOBP08], is a library in the dependently typed language COQ [CH88, dt04, BCHPM04], and is used for reasoning about what are call big operators. These big operators, are indexed versions of normal operators, e.g. the big operator of normal addition $+$ is summation $\sum$. The library defines the big version as a fold on the small version on a list, where the list represents the index set, and this list is supposed to not contain any duplicates, or missing any element. The library gives multiple lemmas and theorems how properties of the small operator can be lifted to the big one.

Reasoning about probabilistic functions in type theory have been done before, for example the ALEA [APM09] library, also for the Coq type theory. This library is axiomatising the real numbers in the interval $[0,1]$, as an ω-cpo, which allows e.g. to take the least upper bound of monotonic sequences. This needs to be axiomatised since taking the lub is not in general a computable operation. On top of this, a layer for defining probability measures are added, which are used to define and reason about probabilistic programs. These measures are given a monadic structure in order to easier combine them.

On top of the ALEA library, the CertiCrypt [BGZB09, Zan10] was created to reason about semantic security proofs. This library defines an imperative programming language, called pWhile, which adds the possibility of asking for randomness from a probability distribution. This language is used to describe the semantic security games, and the adversary is considered to be another pWhile program. The adversary have access to some part of the memory, but not all, and it is in the private part that the secret is stored. To reason about these programs, a machinery based on Relational Hoare-logic [Ben04] is employed, extended to reason about the new probabilistic capabilities added to the pWhile language.

In contrast to CertiCrypt, where the user have to write the proof, tools like ProVerif [Bla09], employ model checkers to automatically verify security. ProVerif tries to find bugs in security protocols. This tools simulate the protocol using a process calculus which is an extended version of the π-calculus [Mil99], where an malicious process interacts with the system. This uses the so-called Dolev-Yao model [DY83], where we assume that a ciphertext can't be decrypted unless the key is known, but it can be duplicated. Notice that that in order to model-check, limits are put on the amount of sessions that can be active, in order to keep the state space finite. This kind of restriction, on the power on what the adversary can do, is one of the major differences between this and semantic security approach. This approach uses a symbolic method, whereas semantic security is more analytical.

Synopsis

The first part of this thesis provides background information needed to understand the rest of this thesis, and includes topics such as:

- Group theory, Section 1.1, since some encryption schemes such as ElGamal uses groups in the construction.
- Probability theory, Section 1.2, and the theory of negligible functions, Section 1.3 is provided since these are vital for semantic security proofs.
- Although not entirely standard, we use communicating processes when we model agents, where communication is the method for which an adversary interacts with the system, an introduction to processes is provided in Section 1.5.
- Semantic security proofs, which are the way we prove security statements, are introduced and discussed in Section 1.4.
- Finally in Chapter 2, the type theory that the rest of the thesis is using is introduced. This chapter is introducing the important constructs that are used for formalising the mathematical constructs in this thesis.

The second part of this thesis contains type theoretic proofs about the background material, all the proofs have been formalised in AGDA, with the exception of semantic security proofs, which are presented in the next part.

- In order to model probabilistic programs, and reason about probabilities, we device exploration functions, which in contrast to ALEA, are based on a computational foundation. Exploration functions, are similar to canonical big operartors [BGOBP08], but are derived from the fold function immediately rather than from a fold of a list. They are defined and studied in Section 3.1, in particular how to define probabilities and use them for probabilistic reasoning, see Section 3.4.
- To model the agents of a semantic security game, we use communicating processes, but since communication is not an inherit notion in type theory, we need to develop a theory for it. This is done in Chapter 4, we also show how such a theory can be embedded into AGDA.

The final part of this thesis, is about the library cryptoAgda, which is combining all the theory of the second part.

- The cryptoAgda library makes it possible to fully model semantic security in the type theory, and is shown in Chapter 5.
- As a bigger case study, of the cryptoAgda library, in Chapter 6, we demonstrate how to prove that ***Prêt à Voter*** [RBH+09, KTR13], which is a end-to-end verifiable voting scheme, is indeed receipt-free. End-to-end verifiable means that the voter can check that their vote have indeed been counted, but can't from this fact prove for whom. This latter property is what is called receipt-freeness.

Part I

Background Material

Chapter 1

Background

We begin our exposition with an introduction of the different mathematical theories used in the process of verifying cryptographic systems, such as a voting system. These theories range from introductory level algebra, such as group theory, a discourse about probabilities and also an introduction to concurrent process calculi and a linear logic based typing of such concurrent processes. In the following sections we will give a mostly standard account of these topics, with the exception of the process calculi which do contains original content. In Chapter 3, most of the results in this chapter are fully formalised, in the type theory of AGDA.

1.1 Group Theory Primer

We begin our presentation with group theory, a short primer is given here to aid the reader with the few ideas that is needed for the rest of this thesis. Groups are an abstract algebraic structure that provide an associative operator together with a neutral element, which is invertible. In this primer we will use a multiplicative reading of a group, i.e. we will often use the $\cdot$ operator standing for the operator in the group.

Definition 1.1.1: Group

A group $(G, 1, \cdot, ^{-1})$ is a set G, an element $1 \in G$, a binary operator $\cdot$ on G and a unary operator $^{-1}$, such that the group axioms are satisfied:

Neutral:	$\forall x \in G.\ 1 \cdot x = x = x \cdot 1$
Associativity:	$\forall x, y, z \in G.\ (x \cdot y) \cdot z = x \cdot (y \cdot z)$
Inverse:	$\forall x \in G.\ x \cdot x^{-1} = 1 = x^{-1} \cdot x$

Furthermore the *order* of a group $(G, 1, \cdot, ^{-1})$ is defined to be the cardinality of G.

By having an inverse for all elements we get cancellation laws for free, these mean that the $\cdot x$ operation is injective for any element x. This is a useful property when using equational reasoning.

Lemma 1.1.2: Group Cancellation

Given a group $(G, 1, \cdot, ^{-1})$ and three elements a, x and y such that $a \cdot x = a \cdot y$ or $x \cdot a = y \cdot a$ then $x = y$.

Proof. If $a \cdot x = a \cdot y$ then the proof goes by equality reasoning:

$$x = 1 \cdot x = (a^{-1} \cdot a) \cdot x = a^{-1} \cdot (a \cdot x) = a^{-1} \cdot (a \cdot y) = (a^{-1} \cdot a) \cdot y = 1 \cdot y = y$$

The other direction, i.e $x \cdot a = y \cdot a$ follows in a similar fashion. □

Furthermore we know that the identity element is unique in every group, and that for every element in the group there is only one other element that is its inverse.

Lemma 1.1.3: Uniqueness of Identity and Inverse

Given a group $(G, 1, \cdot, ^{-1})$ then

1. if x is an element of G such that for all y, $x \cdot y = y$ then $x = 1$,
2. if x is an element of G such that for an element y, $x \cdot y = 1$ then $x = y^{-1}$.

Proof. Both cases follows by equational reasoning:

1. since $x = x \cdot 1 = 1$,
2. we first show that $x \cdot y = 1 = y^{-1} \cdot y$ and by Lemma 1.1.2 we can conclude $x = y^{-1}$.

□

One general operation to construct elements from a group, given an element g is to take the integral power of that g. This is used by for example the ElGamal encryption scheme, which will be explained in Section 1.4.

Definition 1.1.4: Integer Exponentiation

Given a group $(G, 1, \cdot, ^{-1})$, an element $a \in G$ and an integer $n \in \mathbb{Z}$ we define integer exponentiation as follows:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a^{n'} \cdot a & \text{if } n = 1 + n' \text{ and } n > 0 \\ (a^{-1})^{-n} & \text{if } n < 0 \end{cases}$$

The standard laws of exponentiation that one is used to from high school mathematics still holds when working in an arbitrary group.

Theorem 1.1.5: Exponential Laws

Given a group $(G, 1, \cdot, ^{-1})$, an element a and two integers $m, n \in \mathbb{Z}$, then $g^m \cdot g^n = g^{m+n}$ and $(g^m)^n = g^{mn}$

Proof. By induction on m. □

A cyclic group is a group such that all elements are *generated* from some element g, which is called a generator of the group. That is to say that all elements can be constructed from this generator. The ElGamal encryption scheme works inside an cyclic group of some finite order.

Definition 1.1.6: Cyclic Group

A group $(G, 1, \cdot, ^{-1})$ is cyclic if there exists an element g, called a generator, such that for all elements $x \in G$ there exist an $n \in \mathbb{Z}$ such that $g^n = x$.

A consequence of being cyclic is that the operator of the group is commutative.

Theorem 1.1.7: Cyclic Groups are Commutative

Given a cyclic group $(G, 1, \cdot, ^{-1})$ with generator g, then $\cdot$ is commutative i.e. $\forall x, y \in G. x \cdot y = y \cdot x$.

Proof. Since g is a generator there exists $n, m \in \mathbb{Z}$ such that $g^m = x$ and $g^n = y$. So $x \cdot y = g^m \cdot g^n = g^{m+n} = g^n \cdot g^m = y \cdot x$ by Theorem 1.1.5. □

One group that we will pay extra attention towards is the integer modulo q, which is often denoted $\mathbb{Z}/q\mathbb{Z}$ or sometimes $\mathbb{Z}_q$. Informally this is the group of integers between 0 and $(q-1)$ where the operation is addition that loops around. Since we want the equality on this group to be equality modulo q we formally work with congruence classes modulo q, such that $[x]\cdot[y] = [x+y]$. This is a cyclic group with generator $[1]$. Furthermore we extend the integer exponentiation in the following way, which requires that the order of the group is n where x is an element of $\mathbb{Z}/n\mathbb{Z}$ in order to be well defined.

$$a^{[x]} = a^x$$

Definition 1.1.8: Discrete Logarithm

Given a group $(G, 1, \cdot, ^{-1})$ and elements $g, b \in G$, the discrete logarithm is an integer k such that $g^k = b$.

Given two groups, we can ask what kind of mappings exists between them. A function that preserves the structure of the group is called a group homomorphism.

Definition 1.1.9: Group Homomorphism

Given two groups, $(G_A, 1_A, \cdot_A, {}^{-1}_A)$ and $(G_B, 1_B, \cdot_B, {}^{-1}_B)$ a function $f : G_A \to G_B$ is a group homomorphism if for all x, y then $f(x \cdot_A y) = f(x) \cdot_B f(y)$.

Such a function does indeed, as the name implies, preserve all of the structure of the group, not only $\cdot_A$ but the identity and inverse are also preserved as shown by the following lemma:

Lemma 1.1.10: Group Homomorphism

Given a group homomorphism φ between two groups G_A and G_B. Then

1. the identity is preserved $\varphi(1_A) = 1_B$,
2. and inverse is preserved so that for all x then $\varphi(x^{-1}) = \varphi(x)^{-1}$.

Proof.

1. We first show that for all x $\varphi(1_A) \cdot_B \varphi(x) = \varphi(1_A \cdot_A x) = \varphi(x) = 1_B \cdot_B \varphi(x)$ and by lemma 1.1.2 we conclude $\varphi(1_A) = 1_B$.
2. We first show that $\varphi(x^{-1}) \cdot_B \varphi(x) = \varphi(x^{-1} \cdot_A x) = \varphi(1_A) = 1_B$ and by Lemma 1.1.3 we can conclude that $\varphi x^{-1} = \varphi(x)^{-1}$.

$\square$

Integer exponentiation with a value $g \in G$, is a group homomorphism from the group of integers with addition to the group G. The exponential laws, i.e. Theorem 1.1.5, proves that this is indeed a group homomorphism. Furthermore if for the element g, there always exists a unique discrete logarithm, then this integer exponentiation is a group isomorphism. This isomorphism will later be used in the ElGamal encryption scheme.

1.2 Probability Theory

We present here the standard presentation of probability theory, but since the applications we have in mind is for reasoning about randomised programs, we will focus entirely on discrete probabilities later. Furthermore only finite sample spaces will be used, where the sample space is the set of all possible outcomes. A predicate on a sample space is called an event, and is what we wish to assign a probability of, i.e. how likely is it that a particular predicate is true? The structure of events forms a σ-algebra, which is a set theoretical definition, with the idea that a predicate is simply a subset of the sample space. Being an element of the predicate subset means that the predicate is true for that outcome. The formal definition of a σ-algebra is as follows:

Definition 1.2.1: σ-algebra

Let Ω be a set, called sample space, and let $\mathscr{E}$ be a collection of subsets of Ω. Then $\mathscr{E}$ is an σ-algebra over Ω if:

- If $\Omega \in \mathscr{E}$.
- $\mathscr{E}$ is closed under complement, i.e. if $A \in \mathscr{E}$ then so is $\Omega \setminus A \in \mathscr{E}$.
- $\mathscr{E}$ is closed under countable unions, i.e. if $A_1 \in \mathscr{E}, A_2 \in \mathscr{E}, A_3 \in \mathscr{E}, \ldots$ then so is $A = A_1 \cup A_2 \cup A_3 \cup \ldots \in \mathscr{E}$.

We don't need to postulate anything about neither the empty set $\emptyset$, nor how a σ-algebra interacts with intersection. The current axioms are enough to capture this:

Lemma 1.2.2: $\emptyset$ and Intersection with σ-algebra

Let Ω be a set, and $\mathscr{E}$ a σ-algebra of Ω, then:

- The empty set $\emptyset$ is in $\mathscr{E}$.
- $\mathscr{E}$ is closed under intersection, i.e. if $A \in \mathscr{E}$ and $B \in \mathscr{E}$ then so is $A \cap B \in \mathscr{E}$.

Proof. Both properties follows from the axioms.

- Since $\Omega \in \mathscr{E}$ and because $\mathscr{E}$ is closed under complement with Ω then $\emptyset = \Omega \setminus \Omega \in \mathscr{E}$.
- Note that $A \cap B = \Omega \setminus (\Omega \setminus A \cup \Omega \setminus B)$ by De Morgan's laws, which is applicable since $A, B \subset \Omega$. Since $\mathscr{E}$ is closed under union and complement we can conclude that $A \cap B \in \mathscr{E}$.

$\square$

In order to get a probability, i.e. a real number between 0 and 1, for an event we use a probability measure.

Definition 1.2.3: Probability Measure

A probability measure on a σ-algebra $\mathscr{E}$ over a sample space Ω is a function $\mu : \mathscr{E} \to [0,1]$, where $[0,1]$ is the set of real numbers between 0 and 1, such that:

- The measure of the whole sample space is 1, i.e. $\mu(\Omega) = 1$.
- Given a countable collection of pairwise disjoint sets A_i, the measure of the disjoint union is the sum of each individual measure, i.e. $\mu(\bigcup_i A_i) = \sum_i \mu(A_i)$.

The number in the interval $[0,1]$ is the probability and it tells how likely an event is to happen in a given sample space with a particular measure. Since an event will either happen or not, there is a direct way of computing the probability of the complement of an event.

Lemma 1.2.4: Measure the Complement

Let μ be a probability measure for the σ-algebra $\mathscr{E}$ over a sample space Ω, and E be an event, i.e. $E \in \mathscr{E}$, then $\mu(\Omega \setminus E) = 1 - \mu(E)$.

Proof. Because $\mu(\Omega) = 1$ and $\Omega = E \cup (\Omega \setminus E)$, it follows that $1 = \mu(\Omega) = \mu(E \cup \Omega \setminus E) = \mu(E) + \mu(\Omega \setminus E)$. The result is given by subtracting $\mu(E)$ from both sides. $\square$

A simple corollary from this is that the measure of the empty set $\emptyset$ is 0 for any probability measure.

Corollary 1.2.5: Measure of Empty Set

Let μ be a probability measure for the σ-algebra $\mathscr{E}$ over a sample space Ω, then $\mu(\emptyset) = 0$.

Proof. By equational reasoning $\mu(\emptyset) = \mu(\Omega \setminus \Omega) = 1 - \mu(\Omega) = 0$. □

The axioms for probability measures only specify the behaviour of measuring disjoints sets, but nothing about normal union and intersection. We can get a relationship of how the union and intersection interacts with the operands:

Lemma 1.2.6: Measure of Union

Let μ be a probability measure for the σ-algebra $\mathscr{E}$ over a sample space Ω, and A and B be events, i.e. $A \in \mathscr{E}$ and $B \in \mathscr{E}$, then $\mu(A \cup B) + \mu(A \cap B) = \mu(A) + \mu(B)$.

Proof. Note that $A = A \setminus B \cup (A \cap B)$, and that the sets $A \setminus B$ and $A \cap B$ are disjoint, similarly $B = B \setminus A \cup (A \cap B)$. Therefore the union $A \cup B$ is equal to $A \setminus B \cup B \setminus A \cup (A \cap B)$ and the left-hand side of the equation becomes $\mu(A \setminus B) + \mu(B \setminus A) + 2\mu(A \cap B)$. Expanding the right-hand side yields the same expression. □

By packaging in all the components we get a probability space:

Definition 1.2.7: Probability Space

A probability space is $(\Omega, \mathscr{E}, \mu)$ such that Ω is a sample space for the σ-algebra $\mathscr{E}$, and μ is a probability measure for $\mathscr{E}$.

A simple example of a probability space is the roll of a six-sided die. The sample space Ω is simply the value of the die, i.e. a value from 1 to 6. And the possible events is the power set of the sample space. For example one event could be that the die have an even value which is the set $\{2, 4, 6\}$ or that the die have a value of 4 or higher, which would be the set $\{4, 5, 6\}$.

The probability measure will be the uniform distribution, since we wish that the die is fair. This can quite elegantly be described using Iverson bracket, $[P]$, which have the value 1 if the proposition P is true, otherwise is 0.

$$\mu(E) = \frac{\sum_{i=1}^{6}[i \in E]}{6}$$

This is indeed a measure since $\mu(\Omega) = \frac{6}{6} = 1$ and measuring a union of disjoint events is the same a summing up the measure of each event. The general construction of uniform distributions will be the primary measures used in this thesis.

A somewhat confusing term used in probability theory is that of random variable, that is neither a variable, nor random for that matter. Instead it is a probabilistic computation, that depends on the outcome from the sample space. This is easily represented as a function from the sample space to the set of results.

Definition 1.2.8: Random Variable

A random variable X for some sample space Ω is a function $\Omega \quad \rightarrow \quad A$, where A is the set of results.

Random variables will be used throughout this thesis since they capture the concept of probabilistic computation. The events of interest will often be properties about said computations. An example of one such computation could be to compute the sum of two random dice throws, and an event could be that the value of the summation is above 10.

The final concept is about conditional probability, i.e. taking the probability of an event B assuming that we know that another event A is true, we will use the conditional probability measure. This new measure only measures the probability of B being satisfied, when A is satisfied.

Definition 1.2.9: Conditional Probability

Given a probability space $(\Omega, \mathscr{E}, \mu)$, and an event $E \in \mathscr{E}$, such that the event has a positive measure, i.e. $\mu(E) > 0$, then the conditional probability measure can be defined and is $\mu_E(A) = \frac{\mu(A \cap E)}{\mu(E)}$.

The conditional probability measure is indeed a probability measure, as shown by the following lemma.

Lemma 1.2.10: Conditional Probability Measure

Given a probability space $(\Omega, \mathscr{E}, \mu)$, and an event $E \in \mathscr{E}$, such that the event have a positive measure, i.e. $\mu(E) > 0$, then the conditional probability measure μ_E is a measure:

- $\mu_E(\Omega)$ is equal to 1.
- Let A_i be a family of disjoint sets, then $\mu_E(\bigcup_i A_i)$ is equal to $\sum_i \mu_E(A_i)$.

Proof. We need to prove two things:

- $\mu_E(\Omega)$ is equal to $\frac{\mu(\Omega \cap E)}{\mu(E)}$, and we know that $\Omega \cap E = E$, since $E \subseteq \Omega$, we can therefore conclude that $\mu_E(\Omega) = \frac{\mu(E)}{\mu(E)} = 1$.
- $\mu_E(\bigcup_i A_i)$ is equal to $\frac{\mu(\bigcup(A_i \cap E))}{\mu(E)}$ since intersection distributes over unions, and since μ is a probability measure this is equal to $\sum_i \frac{\mu(A_i \cap E)}{\mu(E)}$ which is $\sum_i \mu_E(A_i)$.

$\square$

1.3 Negligible Functions

The theory of negligible functions is used in order to talk about a function that is asymptotically "too small to matter" [Bel97]. This will often be used in order to say that the advantage that an adversary has of breaking a system is too small to matter. Formally this means that the function vanishes faster than any polynomial:[1]

Definition 1.3.1: Negligible Function

A function $\epsilon : \mathbb{N} \to \mathbb{R}_+$ is negligible if for all $c \in \mathbb{N}$ there exists $n_c \in \mathbb{N}$ such that for all $n > n_c \in \mathbb{N}$ it is the case that $\epsilon(n) \leqslant n^{-c}$.

A simple example of a negligible function is the constant zero function, which is trivially negligible.

Theorem 1.3.2: Zero Negligible

The constant zero function $\overline{0}(n) = 0$ is negligible.

Proof. Given c, let $n_c = 0$ then for all $n > 0$ it is the case that $0 \leqslant n^{-c}$. □

On the other hand constant functions are not negligible as demonstrated by this simple lemma:

Lemma 1.3.3: Counterexample of Negligible Function

Given a real number $p \in \mathbb{R}_+$ such that $p > 0$, then the constant function $\overline{p}(n) = p$ is not negligible.

Proof. Assume that $\overline{p}$ is negligible then in particular we know that there exists a n_1 such that for all $n > n_1$ we have $p < n^{-1}$. But this is a contradiction since there exists an natural number $k \in \mathbb{N}$ such that $k > p^{-1} + n_1$. □

Just to make sure that we are not formalising an abstract theory about the constant zero function we here show that there are indeed other functions that are negligible:

[1]Regarding notation, $\mathbb{R}_+$ denotes the set $\{r | r \in \mathbb{R}, 0 \leqslant r\}$.

Lemma 1.3.4: Existence of Negligible Function

The function $f(n) = n^{-n}$ is negligible.

Proof. Given c, let $n_c = c$ then we need to show for all $n > c$ that $n^{-n} \leqslant n^{-c}$ which is trivial. $\square$

A common case is that we wish to scale a negligible function by some constant, it turns out that the resulting function is indeed also negligible.

Theorem 1.3.5: Scalar Multiplication with Negligible Function

Given a real number $r \in \mathbb{R}_+$ and a negligible function ϵ, then $(r \cdot \epsilon)(n) = r \cdot \epsilon(n)$ is negligible.

Proof. Given c then pick n_c to be the maximum of n_{c+1} from ϵ such that $|r| \leqslant n_c$, then for all $n > n_c$, we have $|r\epsilon(n)| \leqslant |r| \cdot |\epsilon(n)| \leqslant |r| n^{-(c+1)} \leqslant n^{-c}$. $\square$

Furthermore addition of two functions preserve the property of being negligible. This is useful when we later will define a relation $\sim$ of functions that are close to each other. Then the fact that negligible functions are closed under addition will be used to prove that $\sim$ is transitive.

Theorem 1.3.6: Addition of Negligible Function

Given two negligible functions ϵ, ν then $\epsilon + \nu$ is negligible.

Proof. Given c then pick n_c to be the maximum n_{c+1} from ϵ and ν and bigger than 2, then: $|\epsilon(n) + \nu(n)| \leqslant |\epsilon(n)| + |\nu(n)| \leqslant n^{-(c+1)} + n^{-(c+1)} = 2n^{-(c+1)} \leqslant n^{-c}$. $\square$

A final proof is that multiplying two negligible functions preserve the property of being negligible.

Theorem 1.3.7: Multiplication of Negligible Functions

Given two negligible functions ϵ, ν then $(\epsilon \cdot \nu)(n) = \epsilon(n) \cdot \nu(n)$ is negligible.

Proof. Given c then pick n_c to be the maximum n_c from ϵ and ν and then for all n, then: $|\epsilon(n) \cdot \nu(n)| = |\epsilon(n)| \cdot |\nu(n)| \leqslant n^{-c} \cdot n^{-c} \leqslant n^{-2c} \leqslant n^{-c}$. $\square$

Before we give the general definition of $\sim$ we will give an indexed version, the namely $f \sim_\epsilon g$ which means that the function $f(x)$ and $g(x)$ are within $\epsilon(x)$ of each other. Put it in another way, the distance between f and g is limited by the function ϵ.

Definition 1.3.8: Limited Distance

Given two functions $f, g : \mathbb{N} \to \mathbb{R}$ and one function $\epsilon : \mathbb{N} \to \mathbb{R}_+$ then the relation $f \sim_\epsilon g$ holds if and only if for all x then $|f(x) - g(x)| \leqslant \epsilon(x)$.

This indexed relation is not really an equivalence relation, since the relation is not transitive. The issue is that the index needs to change, this will later be solved, but first we will prove some indexed version of the equivalence axioms.

Theorem 1.3.9: Indexed Reflexivity

Given a function $f : \mathbb{N} \to \mathbb{R}$ and a function $\epsilon : \mathbb{N} \to \mathbb{R}_+$ then $f \sim_{\overline{0}} f$.

Proof. Given x then $|f(x) - f(x)| = 0 = \overline{0}(x)$. □

Which is a kind of reflexivity, furthermore we can prove that the relation is really symmetric.

Theorem 1.3.10: Indexed Symmetry

Given two functions $f, g : \mathbb{N} \to \mathbb{R}$ and a function $\epsilon : \mathbb{N} \to \mathbb{R}_+$ such that $f \sim_\epsilon g$ then $g \sim_\epsilon f$.

Proof. For all x, $|g(x) - f(x)|$ is equal to $|f(x) - g(x)|$ which is less than or equal to $\epsilon(x)$ by the fact that $f \sim_\epsilon g$. □

As mentioned above we don't get transitivity for the same index, the limiting distance increases. For functions f, g and h, such that the distance between f and g is described by ϵ, and the distance between g and h is described by ν, the distance between f and h is at most $\epsilon + \nu$.

Theorem 1.3.11: Indexed Transitivity

Given three functions $f, g, h : \mathbb{N} \to \mathbb{R}$ and two functions $\epsilon, \nu : \mathbb{N} \to \mathbb{R}_+$ such that $f \sim_\epsilon g$ and $g \sim_\nu h$ then $f \sim_{\epsilon+\nu} h$.

Proof. For all x, by the triangle inequality property we know $|f(x) - h(x)|$

is less than or equal to $|f(x) - g(x)| + |g(x) - h(x)|$ which by assumption is less than or equal to $\epsilon(x) + \nu(x)$. □

Now we can define the relation $\sim$ that represents that two function eventually will be very close to each other. This relation is going to be an equivalence relation, and is going to be used when reasoning about cryptographic schemes and how increasing the security parameter will eventually be secure enough.

Definition 1.3.12: Negligible Distance

Given two functions $f, g : \mathbb{N} \to \mathbb{R}$ the relation $f \sim g$ holds, if there exists a negligible function $\epsilon : \mathbb{N} \to \mathbb{R}_+$ such that $f \sim_\epsilon g$ holds.

This relation is indeed an equivalence relation as show below:

Theorem 1.3.13: $\sim$ is an Equivalence Relation

The relation $\sim$ is an equivalence relation, i.e. the following properties holds:

- **Reflexive**: Given a function $f : \mathbb{N} \to \mathbb{R}$, then $f \sim f$.
- **Symmetric**: Given two functions $f, g : \mathbb{N} \to \mathbb{R}$, such that $f \sim g$ then $g \sim f$.
- **Transitive**: Given three functions $f, g, h : \mathbb{N} \to \mathbb{R}$, such that $f \sim g$, and $g \sim h$, then $f \sim h$.

Proof. Each property is proved as follows:

- By Theorem 1.3.9 $f \sim_{\overline{0}} f$ holds, and $\overline{0}$ is negligible.
- By assumption there exists ϵ such that $f \sim_\epsilon g$ holds, by Theorem 1.3.10 then $g \sim_\epsilon f$ holds.
- By assumption there exists ϵ and ν such that $f \sim_\epsilon g$ and $g \sim_\nu h$ holds, by Theorem 1.3.11 then $f \sim_{\epsilon+\nu} h$ holds, and by Theorem 1.3.6 we know that $\epsilon + \nu$ is negligible.

□

Finally we prove two theorems about this relation, and how it relates to the arithmetic functions, i.e. addition and multiplication. The first theorem states that the relation is congruent with respect to addition.

Theorem 1.3.14: $\sim$ is Congruent with Addition

Given functions $f, f', g, g' : \mathbb{N} \to \mathbb{R}$, such that $f \sim f'$, and $g \sim g'$, then $f + g \sim f' + g'$ holds.

Proof. For all x, by the triangle inequality $|f(x) + g(x) - (f'(x) - g'(x))|$ is less or equal to $|f(x) - f'(x)| + |g(x) - g'(x)|$, which is less than or equal, by assumption, to some $\epsilon(x) + \nu(x)$, which is negligible by Theorem 1.3.6. □

The final theorem is one can multiply with a negligible function and still be in the relation.

Theorem 1.3.15: Multiply with a Neglible Function

Given functions $f, g : \mathbb{N} \to \mathbb{R}$, and a negligible function $h : \mathbb{N} \to \mathbb{R}_+$, such that $f \sim g$, then $h \cdot f \sim h \cdot g$ holds.

Proof. For all x, we know that $|h(x) \cdot f(x) - h(x) \cdot g(x)|$ is equal to $|h(x)| \cdot |f(x) - g(x)|$, which by assumption is less than or equal to $h(x) \cdot \epsilon(x)$, where ϵ is limited distance between f and g. By Theorem 1.3.7 we know that $h \cdot \epsilon$ is a negligible function, and therefore $h \cdot f \sim h \cdot g$. □

1.4 Semantic Security

Semantic security [GM84, Ste03, Sho04, BR06] have risen to become a major foundation for security proofs. This is also the form of security proof that receipt freeness of ***Prêt à Voter*** [RBH$^+$09, KTR13] is of, so we need a proper understanding of these proofs in order to be able to model them within our type theory. Semantic security derives it's notion of security by working with information theory. One measures how much information is gained, for example about a message, by an adversary when for example it is given a ciphertext of said message. This leads to working with probabilities, and to our desire to minimise the probability, i.e. to make it negligible, that an adversary would gain any information from a ciphertext.

The adversaries used in semantic security, are thought to be unknown probabilistic programs, where we think of a probabilistic program as a normal program that gets one extra input which is random. The reason for using probabilistic programs is because most cryptographic primitives use randomness, and an adversary needs to be able to use these primitives. In most contexts it is possible to detect leakage by some form of brute force enumeration, of for example all ciphertexts of a particular size, which is obviously not a realistic attack. Therefore, one often limits the attention to polynomial time programs, but this thesis will not focus on time complexity of programs.

Semantic security proofs are used since they can in contrast to symbolic proofs [Bla09], prove the security of some cryptographic primitives such as encryption schemes. Symbolic proofs works by axiomatising the properties of the primitives and on top of these build up security protocols, and then prove the security of these by some form of model checking, but there is no guarantee that these axioms are indeed valid. Furthermore, whilst symbolic proofs can find security holes in protocols, there is always a question if one has given enough power to the adversary.

Asymmetric encryption is based on two kinds of keys, one that is used for encryption, usually called the public key, and one for decryption, called private key. Often these keys are bit vectors of some size, and this size is often referred to as the security parameter of the encryption scheme. Symmetric encryption is using the same key for both encryption and decryption, while these may be easier to relate to keys in the real world, it is often much more preferable to split the power of the key. The public key is called public since in most cases it is indeed preferable if everyone have access to this key, for example everyone can encrypt a message, but only the recipient can decrypt. We can be more formal about this with the following definition:

Definition 1.4.1: Asyymetric encryption

An asymmetric encryption scheme for message space M, cipher text space C, public key space PK and private key space SK are three programs (keyGen, enc, dec) such that:

- keyGen : $\mathbb{N} \times R_{KG} \to PK \times SK$ is a program that generates a private and a public key of the given security parameter,
- enc : $M \times PK \times R_E \to C$ is a program that takes a message $m \in M$ and a public key pk and returns a ciphertext,
- dec : $C \times SK \to M$ is a program that decrypts a cipher text to the original message.

It is important to note that the programs are both terminating and probabilistic, i.e. they have access to randomness, which are the extra R_X arguments. For example, in the case keyGen this is by R_{KG} and in the case of enc this represented by the parameter R_E.

In order to be an encryption scheme it furthermore needs to satisfy a law, which we call the functional correctness. That is to say that if one decrypts a message encrypted with keys generated together by the key generator then one gets back the original message.

Definition 1.4.2: Functional Correctness

An asymmetric encryption scheme $(keyGen,\ enc,\ dec)$ is functionally correct if for all security parameter $k \in \mathbb{N}$, and messages $m \in M$, such that $keyGen(k) = (pk, sk)$ and $enc(m, pk) = c$ then $dec(c, sk) = m$.

The example of an asymmetric encryption scheme, that we will use in the following section is the ElGamal encryption system [Elg85]. This encryption system was the first public key encryption scheme based on the Diffie-Hellman key exchange protocol [DH76], and it is using group theory to derive its cryptographic properties.

Definition 1.4.3: ElGamal

The ElGamal encryption scheme is based on a cyclic group $(G, 1, \cdot, ^{-1})$ of order q, for which computing the discrete logarithm is difficult. This set G will be the set of both messages and ciphertexts. The randomness needed by key generation and the encryption are integers in $\mathbb{Z}/q\mathbb{Z}$, notice that the size q is the security parameter.

$$\begin{aligned}
\text{keyGen}(\_, x) &= (g^x, x)\\
\text{enc}(pk, m, y) &= (g^y, pk^y \cdot m)\\
\text{dec}(sk, (r, c)) &= (r^{sk})^{-1} \cdot c
\end{aligned}$$

The ElGamal encryption scheme is indeed an encryption scheme since it is functionally correct according the laws of encryption schemes:

Theorem 1.4.4

The ElGamal encryption scheme satisfy functional correctness, i.e decrypting an encryption with the corresponding private and public keys is the identity.

Proof. By equational reasoning:

$$\begin{aligned}
\text{dec}(x, \text{enc}(g^x, m, y)) &= \text{dec}(x, (g^y, (g^x)^y \cdot m))\\
&= ((g^y)^x)^{-1} \cdot (g^x)^y \cdot m = m
\end{aligned}$$

□

But just functional correctness is not enough to get security, in fact leaving both encryption and decryption be identity function satisfy functional correctness! Therefore we need to introduce some more properties of security that an encryption scheme can provide. The semantic security notion ***IND-CPA*** aims to capture that a ciphertexts hides the underlying message, i.e. that even if we gets access to a ciphertext the only thing we learn about the original message is its length. In fact even if we have some information about the original message we don't learn anything more from the ciphertext.

The intuition for how this is proved is as follows. Assume an adversary A that will play a game with a challenger C. The goal of the game, is for the adversary to guess a secret bit b that is known only to the challenger C. In order to do so, A will receive a public key pk and will then give C two messages m_0 and m_1 (of the same length). The challenger C will return the

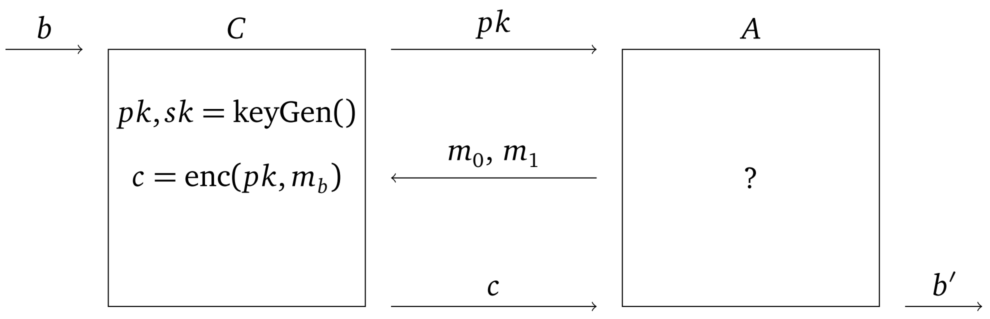

Figure 1.1: The ***IND-CPA*** game

encryption $c = \text{enc}(pk, m_b)$ of one of the messages, depending on the value of the bit b. Given c, and the public key used for the encryption, can the adversary A guess b or not? If the A is not more successful than flipping a coin we can conclude that the encryption scheme is secure, and no information have leaked. The communication between the two parties are summarised in Figure 1.1.

The way we measure how much better an adversary A is from random coin-flipping will be through what is called the advantage $Adv(A)$. This is a real number between 0 and 0.5 which is the distance between the probability of A guessing correctly and a random coin flip guessing correctly.

Definition 1.4.5: Advantage

The advantage for an adversary A is defined to be:

$$Adv(A) = \left| Pr[b = b'] - \frac{1}{2} \right|$$

If there is no adversary that have any significant, i.e. above a negligible amount, advantage, then we deem the encryption scheme to be secure according to the ***IND-CPA*** semantic security notion. In general a semantic security notion is defined in a similar fashion of having an adversary trying to distinguish between two different scenarios.

Definition 1.4.6: Semantic Security Notion

The notion of semantic security comprises of three parts.

Attack Game An attack game of a specified challenger C playing an probabilistic adversary A, were the challenger have a secret bit $b \in \textbf{\textit{Bool}}$ and the final action of A is to output a guess $b' \in \textbf{\textit{Bool}}$.

Advantage The advantage is the distance $\text{Adv}(A) = \left|\mathbf{Pr}[b = b'] - \frac{1}{2}\right|$.

Security If there exists an negligible function ϵ such that for security parameter k and polynomial time adversaries A then $\text{Adv}(A) \leqslant \epsilon(k)$.

As a remark we note that an adversary A' that performs the same interaction as A except guesses $\neg b'$ instead of b' will have the same advantage as A. This shows that we are measuring that an adversary can detect a difference in the two situations, not that they will guess correctly.

As mentioned above ElGamal derives its security from the difficulty of computing the discrete logarithm in the group. Of course the way we should state the difficulty is by stating it in terms of a semantic security notion. The notion is called Decisional Diffie-Hellman, or ***DDH*** for short, and captures that an adversary will not be able to distinguish between a particular exponentiation and an random exponentiation. The ***DDH*** semantic security notion is formally defined using the following attack game.

Definition 1.4.7: *DDH* Attack Game

The decisional Diffie-Hellman attack game is defined as the challenger have a secret bit b, and generates three random integers x, y and z in $\mathbb{Z}/q\mathbb{Z}$. If the bit b is 0 then the challenger will send (g^x, g^y, g^{x+y}), otherwise the challenger sends (g^x, g^y, g^z).

A stronger notion than ***IND-CPA*** is the notion ***IND-CCA1*** (Indistinguishability Under Chosen Ciphertext), this security notion let's the adversary have access to a decryption oracle, before the challenge phase. The decryption oracle can decrypt any encrypted messages, as long as the proper public key have been used, that the adversary sends. If the adversary have access to the decryption oracle after the challenge phase the game is ***IND-CCA2*** (Adaptive

Chosen Ciphertext Attack), of course the oracle is restricted not to decrypt the actual challenge since that would make the game trivial.

Not all ***IND-CPA*** secure encryption schemes are also ***IND-CCA2*** secure, e.g. ElGamal is not secure under ***IND-CCA2***. The fact that ElGamal is not ***IND-CCA2***, follows from the fact that given a ciphertext for M it is possible to construct a ciphertext for $c \cdot M$ for any element c in the group. One simply multiply the second component with c and one gets the new ciphertext, this breaks ***IND-CCA2*** since the new ciphertext can be decrypted by the decryption oracle after the challenge phase. The property that one can manipulate a ciphertext like this is called malleability, and one of the primary reasons of ***IND-CCA2*** is to show non-malleability. It is currently an open problem whether ElGamal is ***IND-CCA1*** or not [Lip11].

A common method that is employed to make ElGamal, or other encryption schemes that is only ***IND-CPA*** secure, be ***IND-CCA1*** secure is the Naor-Yung construction [NY90, DDN91], similar construct exists for ***IND-CCA2*** [NY95, Sah99]. The basic problem is that the adversary should not be allowed to manipulate the received ciphertext into a new, and still valid, ciphertext, and therefore be able to ask the decryption oracle. The Naor-Yung ***IND-CCA2*** construction solves this problem by encrypting the message twice, and then use a sound zero-knowledge proof system [BFM88, GMR89] that shows that both of these encrypted messages are encryption of the correct message. Even if one can manipulate the inner ciphertexts, the zero-knowledge proof will not be correct, and therefore the resulting ciphertext is not a valid ciphertext.

1.5 Processes

Semantic security is defined by different parties communicating with each other, we are therefore working with some sort of communicating processes. These processes are similar to what is usually found in the process calculus community. In particular we will work with a typed π-calculus[Mil99, Hon93].

The terms of the process calculus can be considered to all live in a big soup of processes, where they use channels to send messages to each other. This soup is both one of its strong points, but also it complicates meta-reasoning since now more terms are supposed to be equated. We are going to embed these terms later in our type theory, and use these in order to represent the communicating processes. We start with the basic primitives:

$$\overline{c}\langle M\rangle\, P \qquad\qquad \underline{c}(x:A)\,Q$$

Here $\overline{c}\langle M\rangle\, P$ sends the message M on channel c and then continues as process P. The receiving process, on the channel c, is $\underline{c}(x:A)\,Q$, and it binds the message to the variable x and continue as the process Q which has access to the message.

$$\mathbf{0} \qquad\qquad P \,||\, Q$$

The process $\mathbf{0}$ is the empty process, i.e. the processes that is done. The process $P\,||\,Q$ is the parallel execution of processes P and Q. Semantically, this operation is both commutative and associative and $\mathbf{0}$ is a neutral element.

$$(\nu c d)\, P \qquad\qquad c \longleftrightarrow d$$

The process, $(\nu c d)\,P$, will create two new channels c and d that can be used in P [Vas12]. These channels are linked with each other so sending on one of them will be received on the other. The reason that we are using two linked channels rather than one, as is customary, is because when we are giving types to these processes we wish to have a linear usage of channels. By splitting the creation into two connected channels we can enforce a linear usage on the channels, and still allow for internal communication. The process $c \longleftrightarrow d$ will equate the two channels c and d, by sending any messages on c to d and vice versa.

$$c\{de\}\,P \qquad\qquad c[de]\,P$$

The final two process constructions are for splitting a channel into two new channels. The process $c\{de\}\,P$ and $c[de]\,P$ are in this regard similar to each other, the only difference is that with $c\{de\}P$ the process P can use d and e in the same process, whereas with $c[de]Q$, the process Q must use d and e in parallel, so there can't be any dependencies between d and e.

Linear Logic. The channels are only used once, in a linear fashion, in order for the processes to guarantee deadlock freedom. But only well-typed processes are deadlock free, so how does one type these processes? Looking in the literature we find Session Types[Hon93] as the prominent candidate for typing processes. Typing inspired from Linear Logic[Gir87], were given by Abramsky [Abr93], but the use of Linear Logic did not take off until much later[CP10, Wad14].

Before giving our typing to the processes we will first provide the very basics of Linear Logic. The judgement is $\vDash \Gamma$, where Γ is a context of formulas. A central concept of Linear Logic, is that every formula is associated with a dual formula. The dual of a formula A is denoted $\overline{A}$, and is defined in the meta-theory. The formulas of the multiplicative fragment are as follows:

$$A,B ::= P \mid \overline{P} \mid A \otimes B \mid A \parr B$$

The tensor $\otimes$ is used to combine to formulas, which will be paired up in two separate branches, whereas the par $\parr$ can be in the same process. The atoms come in two forms, P or the dual version $P^{\perp}$, each atom has its own dual. The dual of formulas are defined to satisfy the following equations:

$$\begin{array}{rclcrcl} P^{\perp} & = & \overline{P} & \qquad & \overline{P}^{\perp} & = & P \\ (A \otimes B)^{\perp} & = & A^{\perp} \parr B^{\perp} & & (A \parr B)^{\perp} & = & A^{\perp} \otimes B^{\perp} \end{array}$$

The rules for Linear Logic tries to balance the formulas so that every atom P is matched up with its dual $\overline{P}$. In contrast to ordinary logic, Linear Logic does not have rules for weakening nor contraction. The context is a multiset of formulas, as such swapping formulas in the context is permitted, but one can neither duplicate nor remove a duplicate of a formula.

$$\boxed{\begin{array}{cc} \textit{ax}\,\dfrac{}{\vDash P, \overline{P}} & \textit{cut}_A\,\dfrac{\vDash \Gamma, A \quad \vDash \Delta, A^{\perp}}{\vDash \Gamma, \Delta} \\[2em] \parr\,\dfrac{\vDash \Gamma, A, B}{\vDash \Gamma, A \parr B} & \otimes\,\dfrac{\vDash \Gamma, A \quad \vDash \Delta, B}{\vDash \Gamma, \Delta, A \otimes B} \end{array}}$$

The rules we will use to type processes are variations on the standard use of linear logic as a type system for the π-calculus, described in [Abr93] and [BS94]. This interpretation of linear proofs has been later modified in both an intuitionistic linear [CP10] and a classical linear [Wad14] variant, to follow the principles of *session types* [Hon93], but we return here to the original design of Abramsky.

An important problem in such an interpretation of sequent calculus proofs lies in the handling of parallel composition and the corresponding branching in a proof. Indeed, if a communication operation is associated to a logical rule such as the rule for $\otimes$, we will need to type both the operator and a parallel composition, as done for example following the translation of Abramsky:

$$\frac{\vDash P :: \Gamma, d : S \qquad \vDash Q :: \Delta, e : T}{\vDash c[de](P \parallel Q) :: \Gamma, \Delta, c : S \otimes T}$$

where we have simplified the picture by using a binding output. This leads to a host of problems with the syntactic congruence on processes and the preservation of typing. In order to avoid this, we depart from the usual presentation of linear logic and enrich its sequent calculus with two levels, i.e. using nested contexts, where the typing judgements is of the shape $\vDash P :: \mathscr{J}$, where:

$$\mathscr{I}, \mathscr{J} ::= \cdot \mid [\mathscr{H}] \mid \mathscr{I}, \mathscr{J} \qquad\qquad \mathscr{G}, \mathscr{H} ::= \cdot \mid c : S \mid \mathscr{G}, \mathscr{H}$$

and where c denotes a *channel* name, while S denotes a linear formula seen as the type of this channel. The intuitive interpretation of this two-level approach is simple: comma in $\mathscr{J}$ represents a meta-level $⅋$ while in $\mathscr{H}$ it represents a meta-level $\otimes$. The view of the comma as the $⅋$ is the standard interpretation of sequents in linear logic, but here we can indicate that formulas should be distributed into different sequents, using brackets $[\cdot]$. The purpose of this generalisation is to make branching — and the splitting of the context — independent from the decomposition of the connectives. From a proof-theoretical viewpoint, this is related to the need to decompose the formulas not only at top level, a question adressed partially using *hypersequents* [Avr96] and more thoroughly studied in the setting of *deep inference* [Gug07]. The form of nesting required here is quite limited, as it only offers access to formulas appearing under a $⅋$ and a $\otimes$, when considering the logical interpretation of sequents.

This approach leads to the system described in Figure 1.2, where processes can contain binary inputs and outputs, parallel compositions and scopes, as

$$\frac{\vDash P :: \mathscr{J}, [d : S], [e : T]}{\vDash c\{de\}\, P :: \mathscr{J}, [c : S \parr T]} \qquad \frac{\vDash P :: \mathscr{J}, [d : S, e : T, \mathscr{G}]}{\vDash c[de]\, P :: \mathscr{J}, [c : S \otimes T, \mathscr{G}]} \qquad \frac{\vDash P :: \mathscr{J}, [\mathscr{H}, \mathscr{G}]}{\vDash P :: \mathscr{J}, [\mathscr{H}], [\mathscr{G}]} \qquad \frac{}{\vDash \mathbf{0} :: [\cdot]}$$

$$\frac{\vDash P :: \mathscr{I}, [\mathscr{G}] \quad \vDash Q :: \mathscr{J}, [\mathscr{H}]}{\vDash P \parallel Q :: \mathscr{I}, \mathscr{J}, [\mathscr{G}, \mathscr{H}]} \qquad \frac{\vDash P :: \mathscr{J}, [c : S, d : S^{\perp}]}{\vDash (\nu cd)\, P :: \mathscr{J}} \qquad \frac{}{\vDash c \longleftrightarrow d :: [c : S], [d : S^{\perp}]}$$

Figure 1.2: Typing rules for processes based on linear logic

well as the inactive process and a form of forwarder found in [CP10]. An important observation here is that in the rule for parallel composition, both $\mathscr{G}$ and $\mathscr{H}$ can be $\cdot$ and thus the *mix* rule is present. Although it is not a problematic rule, and in fact is desireable from the process viewpoint[2], it is not part of standard linear logic. Moreover, in order to obtain the expected properties for $\parallel$ we need the rule treating the composition of two blocks $[\cdot]$, which also amounts to the presence of mix, by interpretation of the judgements: the implication $(A \otimes B) \multimap (A \parr B)$ is valid here.

Polyadic communication. We can easily extend the rules for $\parr$ and $\otimes$ to support the n-ary variants of the multiplicative connectives, as follows:

$$\frac{\vDash P :: \mathscr{J}, \overrightarrow{[e : S]}}{\vDash c\{\vec{e}\}\, P :: \mathscr{J}, [c : \parr \vec{S}]} \qquad \frac{\vDash P :: \mathscr{J}, [\overrightarrow{e : S}, \mathscr{H}]}{\vDash c[\vec{e}]\, P :: \mathscr{J}, [c : \bigotimes \vec{S}, \mathscr{H}]}$$

and thereby obtain typing rules for general polyadic communication, as found in the π-calculus. Here, a concise syntax indicates the use of a whole sequence of blocks $[\cdot]$ or types of the shape $c : S$, extracted from the n-ary connective. Note that by doing so, we obtain the multiplicative units of linear logic for free, but we are not interested in the distinction between the two. Beyond accepting the implication induced by mix, we simply collapse the two units, so that the rule for $\mathbf{0}$ can be written:

$$\frac{}{\vDash \mathbf{0} :: \cdot}$$

and the distinction beetween the *nullary* case of $\parr$ and $\otimes$ will not be emphasised in the following, since they are now logically equivalent. Observe that extending linear logic with the mix rule is neither problematic nor surprising, since the concurrent interpretations of linear logic are related to the use

[2]The mix-rule allows for an easy way to make independent processes be in parallel.

of proof-nets [Gir96], as observed in [BS94], where mix simply translates as the juxtaposition of valid nets. The collapse of the two multiplicative units makes sense when considering them as types simply indicating termination of a process.

There is one rule in the system described in Figure 1.2 that is not reflected in the structure of processes, which is not fully satisfactory from a *proofs-as-programs* perspective. However, this could be avoided by using a more complicated variant of the rule for $\|$ where several $[\cdot]$ blocks are split, under some conditions on groupings chosen in the premises.

Finally, note that we use a binary form of scoping, reminiscent of [Vas12] but introducing two distinct channel names c and d. This stems from the design of the *cut* rule chosen here, where no branching is performed: this perspective on cut is consistent with the treatment of $\otimes$ in a system where depth is used and cut is essentially viewed as the implication $(A \otimes A^{\perp}) \multimap \perp$. Indeed, we wish to avoid repetition of channel names inside a sequent, while grouping dual session types of the cut into a $[\cdot]$ block.

Congruence and permutations. The decoupling of $\otimes$ and of cut from branching allows us to clarify the interpretation of each rule as a single construct in the π-calculus, but also provides a natural interpretation of the syntactic congruence $\equiv$ on processes in terms of permutations of inference rules. Indeed, in the sequent calculus and in the generalisation we consider here, many rules can be exchanged without changing the essential structure of the proof. Moreover, such permutations are necessary in the cut elimination process, which proceeds by permuting cut instances upwards in a proof tree.

The acknowledgement of rule permutations has two consequences. First, we can derive the expected equations on $\|$ from permutations of its rule with itself and others:

$$P\|Q \equiv Q\|P \quad P\|(Q\|R) \equiv (P\|Q)\|R \quad P\|\mathbf{0} \equiv P \quad (\nu cd)(P\|Q) \equiv (\nu cd)P\|Q$$

under the condition that neither c nor d appear in Q in the last equation. Then, permutations involving the rules for $\parr$ and $\otimes$ justify the syntax of prefixes where no indication of sequentiality appears, just as in [BS94], since exchange of prefixes is possible. This means that we can validate the equations $\pi\kappa P \equiv \kappa\pi P$ if π and κ *involve fully distinct names*, and $(\nu cd)\,\pi P \equiv \pi(\nu cd)\,P$ if c and d do not appear inside π. Moving prefixes past each other is unusual in the π-calculus, but it is necessary from a logical viewpoint to ensure that no cut can reach a blocked situation preventing its elimination.

Cut elimination and communication. In a type system for π based on the sequent calculus, cut elimination appears at the level of processes as reduction, specified through a rewrite system performing communication steps in appropriate configurations. The multiplicative fragment has a simple cut elimination procedure, with a single principal case involving $\parr$ and its dual $\otimes$. The cut reduction in that case translates on processes as the rewrite rule:

$$(\nu c d)(c[c_0 c_1]\, P \parallel d\{d_0 d_1\}\, Q) \quad\to\quad (\nu c_0 d_0)(\nu c_1 d_1)(P \parallel Q)$$

which corresponds to the usual communication rule of the π-calculus, when considering binary scoping as an alternative to channel renaming. Beyond the simple reorganisation of the channel connections appearing in a process, the integration of this system into type theory will provide the means of communicating functional data, thus moving closer to the setting of a functional programming language extended with communication [GV10].

Chapter 2

Type Theory and Agda

Jag ska be att få ställa upp
med en visa om en tupp.
Som var gammal och utsliten
impotent och väderbiten.

Lyrics from: Hönan Agda
Cornelis Vreeswijk

The Curry-Howard correspondence [Cur34, How80], is a relationship between a logic and a typed programming language, where the formulas of the logic are related to types, and the proofs are related to programs. Through this lens, dependent types can serve a dual purpose, both by providing a logical foundation for reasoning, but also give expressive types to programs. Dependent types allow the types to express properties about terms. For example, quantification, as seen in e.g. first order logic, is generalised to dependent functions, where the return type can mention the input value. This dual viewpoint allows for programs to be used for proofs, but furthermore the types are more expressive since they have the power of logic at their disposal.

The particular type theory we use in this thesis is based on Per Martin-Löf [ML75, MLS84], and is the basis of the type system of AGDA [Nor07], which is the proof assistant, and/or programming language, that has been used to formalise the results of this thesis. Although AGDA was picked, the results are general and can be transferred to other dependently typed programming languages, such as COQ [CH88, dt04, BCHPM04] or IDRIS [Bra13].

2.1 A Dependent Type Theory

In the type theory we are working with, i.e. AGDA, we can define new inductive data types. A standard example of an inductive type, is the natural numbers ℕ, which is defined below using the **data** declaration of AGDA:

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

Here three new constants have been defined, the first is for the type itself namely ℕ : Set, which declares that ℕ is a type. Note that Set is the type of small types in AGDA, and we will come back to this type later. The other two constants are the data constructors, *zero* : ℕ for zero, and *suc* : ℕ → ℕ for the successor function. A function going from natural numbers, can be defined by pattern matching against the different constructors. A simple example of such a function is the addition function _+_, note that in AGDA binary operators are surrounded by underscores, and in general a definition may be mix-fix and have multiple underscores, where each underscore indicates where an argument should be.

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

This definition uses recursion on the first argument. The type theory, AGDA, automatically checks if this function is total, i.e. it is defined for all inputs and it terminates. This is done by a coverage checker and a termination checker, which due to Rice's Theorem [Ric53] can't detect all cases, so some total functions will not be found by the checker to be total. The reasons for checking totality is to guarantee a logical soundness when looking at the types from a logical interpretation.

The next to illustrate here is the List type, which follows a similar scheme as the one for the natural numbers. Lists are polymorphic in the element type, which is achieved by having the type of the elements be a parameter to the list type.

```
data List (A : Set) : Set where
  nil  : List A
  cons : A → List A → List A
```

So far the data types we have shown that are similar to that of functional programming languages like Haskell [P⁺03] or Standard ML [MTM97]. We have not seen any dependent types yet, but the next example will illustrate one, and show how we can use real data dependencies, by using inductive families [Dyb97]. In the dependent types literature, list that are indexed by their length, are usually called Vectors and we will not depart from this tradition and also call them Vector:

```
data Vector (A : Set) : ℕ → Set where
  nil  : Vector A zero
  cons : {n : ℕ} → A → Vector A n → Vector A (suc n)
```

In the *cons* constructor, we use a dependent function type $\{n : \mathbb{N}\} \rightarrow ..$, which shows that *cons* really have three arguments: a natural number for the length, the head and finally the tail of the Vector. The use of curly braces $\{n : \mathbb{N}\} \rightarrow ..$ indicates that we wish this argument to be implicit and let the AGDA system to try to infer this argument.

Now we can write dependent functions, once again we will follow the standard examples of the dependently typed literature, and show the *append* functions for Vector. This function takes two Vectors and combine them into one big Vector, notice that the type of the *append* function shows that the resulting Vector is of the size of the summation of both its arguments.

```
append : {A : Set} {n m : ℕ} → Vector A n → Vector A m
  → Vector A (n + m)
append nil          ys = ys
append (cons x xs) ys = cons x (append xs ys)
```

The AGDA system will infer all the implicit arguments. If we so desired we could have written everything explicitly, but often this gets very cumbersome but for illustrative purposes we show below how the *append* function looks with explicit annotations.

```
append : {A : Set} {n m : ℕ} → Vector A n → Vector A m
  → Vector A (n + m)
append {A} {.zero} {m} nil ys = ys
append {A} {.(suc n)} {m} (cons {n} x xs) ys
  = cons {n + m} x (append {A} {n} {m} xs ys)
```

The curly braces on the left hand side of the equation brings implicit arguments into scope, and can be used on the right hand side. This is done by position, but could also be by name using the syntax $n = .zero$, which is useful if only one of the implicit argument is needed.

The dot makes the pattern a dot pattern and is used to mark that we are not actually matching this argument, instead we statically asserting that the argument *n* must really be equal to *zero*. The reason why *n* is equal to *zero* in this case is that we match the first Vector against *nil* which only makes sense if *n* is indeed *zero*. The arguments to a dot pattern don't have to be patterns, i.e. only constructors or variables, but can be any kind of expression. A good example of when patterns are not enough, and an expression is needed, is when working with the image type, which is a type for defining the image of a function.

```
data Img {A B : Set} (f : A → B) : B → Set where
  im : (a : A) → Img f (f a)
```

When pattern matching on an element of Img *f b* we get access to an element *a* of type A such that *f a* is equal to *b*. A function projecting out this element such as *inv* below needs to use a dot pattern, and the expression inside the dot pattern is not a pattern itself, since it uses an arbitrary function.

```
inv : {A B : Set} {f : A → B} (y : B) → Img f y → A
inv {f = f} .(f a) (im a) = a
```

2.2 Records and Modules

Another construct to create new data types is the **record** declaration, which in contrast to types declared by **data** construct which are defined by the constructors, defines types by projections out from the type. The example we use here is that of Σ-types which are dependent pairs. In contrast to normal pairs, i.e. Cartesian products, the type of the second field of the dependent pair, can depend on the value of the first field.

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    fst  : A
    snd  : B fst
```

The keyword **constructor** introduce a convenient function for creating an element of the record, in this case a pair. There are two ways of constructing pairs, one is to use copatterns [APTS13] as opposed to normal pattern matching, whereas the other is to use the **record** expression. The function that the keyword **constructor** creates could have been defined in one of these ways:

```
_,_ : {A : Set} {B : A → Set} (fst : A) (snd : B fst) → Σ A B
fst (_,_ x y)  = x
snd (_,_ x y) = y


   -- or using the record expression
_,_ x y = record {fst = x; snd = y}
```

Whereas normal pattern matching enables matching on arguments, copatterns allows matching on the result. The informal reading is that _,_ *x y* is a pair such that projecting the first field returns *x*, similar for projecting with *snd*. Using the **record** construct, one lists each field in the record, and assigns a value. The distinction between copatterns and the **record** construct is similar to the difference of defining function by pattern matching or using a λ-expression, i.e. an anonymous function.

Another example using copatterns is the function that swaps the argument of the pair, which requires that there is no type dependencies. We use the type _×_ for pairs that don't have any type dependencies. This function will simply map the first projection to the second and vice versa.

```
_×_ : (A B : Set) → Set
A × B = Σ A (λ _ → B)


swap : {A B : Set} → A × B → B × A
fst  (swap p) = snd p
snd (swap p) = fst p
```

In contrast to records, which are useful in combining data together larger pieces of data, modules are a way of structuring a program definitions together, so that related topics are grouped together in the same file. In contrast to records, modules are only used for name spacing, i.e. control which definitions are in scope, and what names these definitions are supposed to have. Take as an example the Nat module which contains the type of natural numbers as defined above, together with the addition function.

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

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

The contents of the module are shown by being indented under the **module** declaration. The contents of the module can be accessed from the outside by prefixing the name of the identifier with the name of the module and separating with a dot, for example the addition function is written as Nat._+_. By **open**ing a module it is possible, to bring into scope the identifiers of the scope, but this can be restricted by declaring with the **using** modifier which identifiers one is interested in. As an example here is a module for Vector that will use the module of Nat.

```
module Vector where
  open Nat using (ℕ)

  data Vector (A : Set) : ℕ where
    nil  : Vector A zero
    cons : {n : ℕ} → Vector A n → Vector A (suc n)

  append : ∀ {m n A} → Vector A m → Vector A n → Vector A (m Nat.+ n)
  append nil         ys = ys
  append (cons x xs) ys = cons x (append xs ys)
```

Records and modules are closely related, with the difference that records are actual types, and their values can be passed around. Modules on the other hand does not have an affect on the runtime. As a small remark we note that each record will define a module, with the same name as the record, that contains the projection functions.

We will mostly use records to define algebraic theories, like monoids or groups, with their data, such as identities and operators. They may furthermore also contain proofs that the particular laws are satisfied.

2.3 Universes

So far we have used Set to describe types, and one may wonder what is the type of Set? It is known that picking Set : Set leads to inconsistencies [Hur95, Gir72], so instead one introduces a whole hierarchy of types. These are Set, Set_1, Set_2 and so forth, in particular $\text{Set} : \text{Set}_1 : \text{Set}_2$. Although the different levels are important for the consistency, we don't display them in this thesis in order to avoid clutter. Note that this does not mean that we are working in a theory with Set : Set , but simply that we will assign levels as necessary.

The type Set is an open universe, since we can populate it with new types using the **data** or **record** declaration, at any point. But it is useful to sometimes work with a closed universe, which in particular allows to write generic functions over the type in the universe. A closed universe can be defined by first giving a type of codes for the types in the closed universe, and then give an elimination function that computes the actual type for each code. Generic functions can now be defined by looking at the code to learn what the type is.

An example of such a universe would be a universe that guarantees that the inhabitants are functors. This universe contains the identity functor, constant functors, and then functors for disjoint sums and products, where disjoint sum A ⊎ B is a type that will either contain a value of type A or a value of type B. The type of codes Fun, for this universe is defined as follows:

```
data Fun : Set where
  Id' : Fun
  K'  : Set → Fun
  _⊎'_ _×'_ : Fun → Fun → Fun
```

We use the prime as a convention for the codes, also note that AGDA allows to define multiple constructors of the same type by letting them share the type declaration. In order to get the real type of the functor, we use the elimination function El which maps each code to the type it represents, and is defined as follows.

```
El : Fun → (Set → Set)
El Id        X = X
El (K A)     _ = A
El (F ⊎' G) X = El F X ⊎ El G X
El (F ×' G) X = El F X × El G X
```

All types we get by the El functions are functors, which we can demonstrate by giving the functor mapping. This mapping will take a function from A → B and will transform this function to a function over El F, where F is code for the functor. This function is defined by induction on the code F, and by learning which code F is, we learn what kind of structure El F is.

```
map : {A B : Set} (F : Fun) → (A → B) → El F A → El F B
map Id      h x = h x
map (K _) _ k = k
map (F ⊎′ G) h (inl f)  = inl (map F h f)
map (F ⊎′ G) h (inr g)  = inr (map G h g)
fst  (map (F ×′ G) h p) = map F h (fst p)
snd (map (F ×′ G) h p) = map G h (snd p)
```

2.4 Equality

As we saw in a previous section with the Img type, dependent pattern matching can introduce equality constraints on the indices of the type. The canonical type describing these kinds of equality constraints is the equality type, also known as the identity type, which is inductively defined as follows:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

The identity type $a \equiv b$ has only an inhabitant if a and b can be identified as being equal, otherwise this type is uninhabited. This motivates an elimination rule which is traditionally called J, which states that if something is true when the identity is *refl*, it is true for all identifications:

```
J : {A : Set} {x : A} (P : (y : A) → x ≡ y → Set)
  → P x refl → (y : A) (p : x ≡ y) → P y p
J {x = x} P p .x refl = p
```

Traditionally this type was thought to only have at most one inhabitant, but recent models [Voe11, AW09] of dependent type theory have emerged based on homotopy theory [Uni13], in which this is no longer necessary. Instead of thinking of a type as some sort of a set, it is interpreted as a form of space, and the identity type $a \equiv b$ is the type of paths from the point a to the point b. The constructor *refl* represents the identity path between an object and itself. With this interpretation, which Homotopy Type Theory is based on, a type can have multiple paths, but all paths can be transported along the identity path. This justifies the elimination rule, which is still in use in Homotopy Type Theory, but one can't prove that there is only one inhabitant of $x \equiv x$, which is commonly known as axiom K:

```
K : {A : Set} {x : A} (P : x ≡ x → Set)
  → P refl → (p : x ≡ x) → P p
K P p refl = p
```

While we could define this axiom with pattern matching, we will not make use of this power in this thesis. Instead we will only use pattern matching that could be explained by a conversion to the J-eliminator [CDP14]. But since using the J-rule directly is somewhat messy we continue to define by pattern matching. The major problem with using the eliminator is to construct the motive, the value of type P in J. This is hidden in the definitional style, compare for example the proof of transitivity using the eliminator and not using it:

```
transJ : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
transJ {x = x} {_} {z} x≡y y≡z = J (λ y _ → y ≡ z → x ≡ z)
  (λ p → p) x≡y y≡z
trans : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans {x = x} {.x} {z} refl x≡z = x≡z
```

The fact that the identity type was invented to capture equality explains why we could derive transitivity, in fact this relation is an equivalence relation since it is furthermore symmetric.

```
sym : {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl
```

Everything in type theory respects this equality, i.e. it is impossible to distinguish between to equal terms. This is shown by two different kind of proofs, the first called *ap* which shows that all functions will map equal input to equal output. The second proof, called *tr* which is short for transport, shows that for all properties if the property holds for an element, it holds for all equal elements as well.

```
ap : {A B : Set} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
ap f refl = refl
tr : {A : Set} (P : A → Set) {x y : A} → x ≡ y → P x → P y
tr P refl p = p
```

A simple consequence of *tr*, it is possible to get a coercion function between equal types. By simply choosing the property to be the identity function, here specialised to the type Set → Set, the transport will act as an type safe coercion.

```
coe : {A B : Set} → A ≡ B → A → B
coe = tr id
```

In some of the proofs in this thesis, we use axioms that don't have any computational interpretation. These axioms, of which there are two, are sound with respect to the type theory we are working in. The first of this axioms is functional extensionality which states that if two (dependent) functions are equal for all inputs, then they are equal as functions. This statement can't be derived in intentional type theory, but it is consistent to assume it:

```
postulate
  λ= : {A : Set} {B : A → Set} {f₀ f₁ : (x : A) → B x}
    → (∀ x → f₀ x ≡ f₁ x) → f₀ ≡ f₁
```

The second axiom we use when reasoning, is motivated by homotopy type theory, which is a type theory where Identity proofs are seen as paths in topological spaces. The second axiom is the univalence axiom and is one of the most characteristic feature of this homotopy type theory. This axiom asserts what the equality on the universe Set should be, which the intentional type theory of Martin-Löf left open. In a Martin-Löf type theory there is only one way of constructing an element of the identity type, and that is by *refl*. In contrast homotopy type theory states that *homotopically equivalent* types can be made identical, where homotopically equivalent is a type isomorphism, and with a condition on the proofs of the isomorphism, namely that they are natural. In our type theory we capture the fact that a function f : A → B is a homotopical equivalence between the two types A and B, by the type Is-Equiv f:

```
record Is-equiv {A B : Set} (f : A → B) : Set where
  field
    inv         : B → A
    inv-is-linv : ∀ x → inv (f x) ≡ x
    inv-is-rinv : ∀ x → f (inv x) ≡ x
    is-hae      : ∀ x → ap f (inv-is-linv x) ≡ inv-is-rinv (f x)
```

The fact that two types A and B are homotopically equivalent is expressed by the fact that there exists a function that is a homotopical equivalence between the A and B. The collection of such a function and the proof that the function is an equivalence is gathered together in the type _≃_, which is just a Σ-type.

```
_≃_ : Set → Set → Set
A ≃ B = Σ (A → B) Is-equiv
```

The univalence axiom, at least the part that will be used in this thesis, states that if two types A and B are equivalent, i.e. $A \simeq B$, then they are equal, i.e. $A \equiv B$. We add to the type theory an axiom that witnesses this fact, but we will be careful to point out when we are using it.

postulate
  $ua : \{A\,B : \mathrm{Set}\} \rightarrow A \simeq B \rightarrow A \equiv B$

This will be used when proving that Σ-types respect equivalence, but before we can prove that we first show that they respect the equality type.

Lemma 2.4.1: Σ-type Respects ≡

Asuming functional extensionality, let A_0, A_1 be types, such that they are identified $A{=} : A_0 \equiv A_1$. Let B_1, B_1 be type families on A_0 and A_1 respectively. Let B= be a family over A_0 of equivalences between B_0 and B_1. The type of B= is $(x : A_0) \rightarrow B_0\,x \equiv B_1\,((\mathit{coe}\ A{=})\,x)$ and *coe* A= is the identity transport along A=. It is then possible to construct a identity $\Sigma\,A_0\,B_0 \equiv \Sigma\,A_1\,B_1$.

Proof. By induction on A= one has only to consider the case for reflexivity on the base point A_0. Notice that the family B_1 now is on A_0, and that B= is now convertible to $(x : A_0) \rightarrow B_0\,x \equiv B_1\,x$ since *coe* computes to the identity function on the reflexivity path. It remains to show that there is proof of equality between $\Sigma\,A_0\,B_0$ and $\Sigma\,A_1\,B_1$, which amounts to first use function extensionality on B= to get a path $B_0 \equiv B_1$, which can then be applied to the context $\Sigma\,A_0$. □

We can now prove that Σ-types respect type equivalences, this requires the univalence axiom.

Theorem 2.4.2: Σ-type Respects ≃

Let A_0, A_1 be types, and A≃ be a type equivalence : $A_0 \simeq A_1$. Let B_0, B_1 be type families on A_0 and A_1 respectively. Let B= be a family over A_0 of equalities between B_0 and B_1. The type of B= is $(x : A_0) \rightarrow B_0\,x \equiv B_1\,(\mathit{fst}\ A{\simeq}\ x)$ It is then possible to construct an equivalence $\Sigma\,A_0\,B_0 \simeq \Sigma\,A_1\,B_1$.

Proof. The equivalence A≃ is transformed into a path using the univalence axiom *ua*. To use the previous Lemma 2.4.1 it remains to show a family of equalities: $\forall\,x \rightarrow B_0\,x \equiv B_1\,(\mathit{coe}\ (\mathit{ua}\ A{\simeq})\,x)$. Considering such an x : A_0 we first use the equality (B=) x. To show a equality proof between B_1 ($\longrightarrow$ A$\simeq$ x) and B_1 (*coe* (*ua* A$\simeq$) x) we apply the context B_1. Finally we use the β-rule[a] for the univalence axiom which gives a path between $\longrightarrow$ A$\simeq$ x and *coe* (*ua* A$\simeq$) x, which concludes the proof. $\square$

[a]The β-rule is that *coe* (*ua* A$\simeq$) is *fst* A$\simeq$.

A convenient way to construct an equality proof is to use equational reasoning syntax, which is used to make the proof look comparable to mathematical proofs. Although in the type theory this is just a small syntactic trick, for the uses of transitivity and reflexivity, we use it in this thesis to improve the readability of proofs.

As an example we show here a proof that for a commutative and associative operator it is possible to exchange two elements "in the middle" of an expression. This proof assumes that *x*,*y*,*z* and *w* are elements of a type A and that • is the commutative and associative operator, and we will exchange the place of *y* and *z*:

```
example : (x • y) • (z • w) ≡ (x • z) • (y • w)
example = (x • y) • (z • w)
            ≡⟨ •-assoc x y _ ⟩
              x • (y • (z • w))
            ≡⟨ ap (_•_ x) (! •-assoc y z w) ⟩
              x • ((y • z) • w)
            ≡⟨ ap (_•_ x_ (ap (λ p → p • w) (•-comm y z)) ⟩
              x • ((z • y) • w)
            ≡⟨ ap (_•_ x) (•-assoc z y w) ⟩
              x • (z • (y • w))
            ≡⟨ ! •-assoc x z (y • w) ⟩
              (x • z) • (y • w)
            ∎
```

The use of equational equality makes it easier to read of the proof, furthermore we use the unary operator *!_* for symmetry proofs to further reduce clutter and make the proof more concise.

2.5 Relational Parametricity

Polymorphic types satisfies some theorems for free [Wad89], indeed some programming languages have been shown to enjoy a so called abstraction theorem [Rey83, Wad89, BJP10]. The abstraction theorem is called the fundamental theorem in logical relations [Tai67, Sta85]. The theory behind AGDA is known to enjoy this abstraction theorem [BJP10], and the free-theorem statement can be mechanically derived from types, furthermore any well-typed program satisfy the free-theorem arising from its type. While they, the free theorems, are uninformative for monomorphic types they are interesting for polymorphic types. Usually, these theorems are stated using pen and paper proofs for HASKELL programs, but we are in a dependently typed language, i.e. AGDA, where the types, programs, statements and proofs all inhabit a common system. Although these free-theorems are mechanical they are currently not automated by the system.

The high level overview, is that each type $\mathrm{T} : \mathit{Set}$ will induce a (binary) relation, which we will denote by oxford brackets[1] $[\![\mathrm{T}]\!] : \mathrm{T} \to \mathrm{T} \to \mathit{Set}$. The (binary) free-theorem is that this relation, $[\![\mathrm{T}]\!]$, is reflexive, i.e. for all terms $t : \mathrm{T}$ there is a proof term $[\![t]\!] : [\![\mathrm{T}]\!]\, t\, t$. If parametricity was internalised, in our type theory, then this proof, i.e. $[\![t]\!]$, would come for free, but instead we need to prove it in each instance. The $[\![\_]\!]$ relation is defined by induction on the type. For example, functions are in the relation if they map related inputs to related outputs:

$$
\begin{aligned}
&[\![\mathrm{A}{\to}\mathrm{B}]\!] : (\mathrm{A} \to \mathrm{B}) \to (\mathrm{A} \to \mathrm{B}) \to \mathit{Set} \\
&[\![\mathrm{A}{\to}\mathrm{B}]\!]\, f\, f' = (x\, x' : \mathrm{A}) \to [\![\mathrm{A}]\!]\, x\, x' \to [\![\mathrm{B}]\!]\, (f\, x)\, (f'\, x')
\end{aligned}
$$

Since polymorphism is expressed using a universe type *Set*, we need to know what the relation $[\![\mathit{Set}]\!]$ is. Following [BJP10] we pick $[\![\mathit{Set}]\!]$ to be the type of all relations. An intuitive reason for this is that the type of $\mathit{Bool} : \mathit{Set}$ and parametricity will say that $[\![\mathit{Bool}]\!] : [\![\mathit{Set}]\!]\, \mathit{Bool}\, \mathit{Bool}$ and since we already know that $[\![\mathit{Bool}]\!]$ is a binary relation on *Bool* it follows that $[\![\mathit{Set}]\!]$ is the type of relations.

$$
\begin{aligned}
&[\![\mathit{Set}]\!] : \mathit{Set} \to \mathit{Set} \to \mathit{Set} \\
&[\![\mathit{Set}]\!]\, \mathrm{A}\, \mathrm{B} = \mathrm{A} \to \mathrm{B} \to \mathit{Set}
\end{aligned}
$$

Furthermore we need to extend the relation on functions to dependent functions in order to express the type of polymorphic functions. The tricky

[1] Also called double brackets.

part in defining the relation for a dependent function such as $(x : \mathrm{A}) \rightarrow \mathrm{B}\,x$ is that the type of B is indeed dependent, i.e. $\mathrm{B} : \mathrm{A} \rightarrow \mathit{Set}$, which implies that the relation on B is defined to be:

```
⟦B⟧ X X’ = (x x’ : A) → ⟦A⟧ x x’ → X x → X’ x’ → Set
```

Which leaves that dependent functions have the following relation:

```
⟦ (x:A) →B⟧ : ((x : A) → B x) → ((x : A) → B x) → Set
⟦ (x:A) →B⟧ f f’ = (x x’ : A)
  → (x_r : ⟦A⟧ x x’) → ⟦B⟧ x x’ x_r (f x) (f’ x’)
```

For inductive types T, the relation ⟦T⟧ is an indexed inductive type, which has the same number of constructors as T, where each constructor relates each constructor to itself. This is the reason why if the type is monomorphic, the relation just turns out to become an equality type. But if the data type is polymorphic, as for example List A, which can be related to a List B, then list are related as long as the elements in the lists are pair-wise related using a relation between the types A and B.

```
data ⟦List⟧ {A B : Set} (_≈_ : ⟦Set⟧ A B) : List A → List B → Set where
  ⟦[]⟧    : ⟦List⟧ _≈_ [] []
  _⟦::⟧_ : ∀ {x y xs ys} → x ≈ y → ⟦List⟧ _≈_ xs ys
                          → ⟦List⟧ _≈_ (x :: xs) (y :: ys)
```

Part II

Type Theory Formalisation

Chapter 3

Mathematics in Type Theory

Joint work with: Nicolas Pouillard

This chapter introduces the way formalise the mathematics used in Chapter 1, i.e. probabilities, the concept of negligible functions and group theory. In order to do so we utilise the computational content of the type theory we working with, in order to both simplify and specify the correctness of our formalisation. As a consequence of the domain we use probabilities in, namely cryptography, we will only need finite sample spaces, which allows for a more direct computational interpretation of the calculus of probabilities.

When talking about probabilities there is always a sample space Ω which is the discourse of the subject. This space will in the type theory be represented as some type, for example when studying a single coin toss the sample space is $\{0, 1\}$ which we will represent by the types of boolean ***Bool***.

Definition 3.0.1: Event

An event on a sample space Ω is a subset $A \subseteq \Omega$.

We can use the constructive nature of type theory and define a subset by the characteristic function. In other words to define a subset $A \subseteq \Omega$ we give a function $\mathbf{1}_A : \Omega \rightarrow \textbf{\textit{Bool}}$, such that $\mathbf{1}_A(x) = \texttt{true}$ if and only if $x \in A$.

Since we working with uniform distributions, the probability of an event A is how often, if we pick a random element x from our sample space Ω, that this element is in A, i.e. $x \in A$. This can be computed by simply comparing the different cardinalities $\mathbf{Pr}[A] = \frac{|A|}{|\Omega|}$. This leads to our first challenge when trying to model this in type theory since we don't have the cardinality available to us, which we will explore in the following section.

The question of cardinalities leads us to work with functions, that count how many elements there is in of a particular subset. Since we use a characteristic function to identify subsets this means that a counting function for a sample space Ω would have the type $(\Omega \to \mathit{Bool}) \to \mathbb{N}$. The idea is that the function will return for how many elements of type Ω for which the predicate is true. In general there is no such function in the type theory we are working in, so we need to introduce one for every type. The next section will present a general framework for deriving such functions, and furthermore reason about them.

3.1 Exploration Functions

We wish to derive counting functions, i.e. functions of type $(\Omega \to \mathit{Bool}) \to \mathbb{N}$ for some type Ω, but these can't be built in a composeable way. To illustrate this consider the product type $\mathrm{A} \times \mathrm{B}$ for some types A and B. Assuming we have counting functions for these types, $f_A : (\mathrm{A} \to \mathit{Bool}) \to \mathbb{N}$ and $f_B : (\mathrm{B} \to \mathit{Bool}) \to \mathbb{N}$, can we combine them to a function $f_{\mathrm{A}\times\mathrm{B}} : (\mathrm{A} \times \mathrm{B} \to \mathit{Bool}) \to \mathbb{N}$?

A start could be to write $f_{\mathrm{A}\times\mathrm{B}}\ p = f_A\ (\lambda\ (x : \mathrm{A})\ .\ ?)$ but we get stuck, since *?* is supposed to be a *Bool* but we wish to also use f_B which is going to return a $\mathbb{N}$. In fact the problem is that counting functions are too restrictive, we need to generalise the type. The issue is that the type *Bool* and $\mathbb{N}$ are two different types, so instead of giving a predicate we give a measure. We generalises counting functions to summation functions, i.e. $(\Omega \to \mathbb{N}) \to \mathbb{N}$, this makes it possible to compose functions more easily. Let's return to the previous example with the product type. Assume we have $f_A : (\mathrm{A} \to \mathbb{N}) \to \mathbb{N}$ and $f_B : (\mathrm{B} \to \mathbb{N}) \to \mathbb{N}$, we can know define:

$$\begin{aligned} &f_{\mathrm{A}\times\mathrm{B}} : (\mathrm{A} \times \mathrm{B} \to \mathbb{N}) \to \mathbb{N} \\ &f_{\mathrm{A}\times\mathrm{B}}\, p = f_A\, (\lambda\, (x : \mathrm{A})\,.f_B\, (\lambda\, (y : \mathrm{B})\,.p\, (x\,, y))) \end{aligned}$$

There is of course nothing particular about $\mathbb{N}$, and we generalise this to any monoid. We give the name exploration to this generalisation since it is a type that is supposed so explore the elements of the given type, the particular exploration is given by the operations of the monoid. An exploration function is parametric over the given monoid and takes the operations as arguments:

Definition 3.1.1: Exploration Function

An exploration function for a type A is given a type M, a value ϵ of type M, a function _⊕_ of type M → M → M, and function *f* of type A → M. The exploration function finally yields a result of type M:

```
Explore : Set → Set
Explore A = {M : Set} (ϵ : M) (_⊕_ : M → M → M)
   → (A → M) → M
```

For any type A, an exploration function is given a default result ϵ, a binary operator _⊕_ and a function *f* realising the body of the big operator. The function *f* is then called on every value of the type to be explored. All results are combined with the operator _⊕_. If there are no values to explore the default result ϵ is returned.

One viewpoint is that the task of an exploration function is thus to transform any small operator _⊕_ into the corresponding big operator $\bigoplus$ of type (A → M) → M. For instance, if *explore* is an exploration function for a type A, then *explore 0 _+_* is $\sum$ and *explore 1 _*_* is $\prod$, where *0*, *1*, *_+_* and *_*_* are defined on the type $\mathbb{N}$.

Note that the type does not specify that the exploration will be over a monoid, the laws are not given, but only the operations. Though when proving properties about explorations, the monoid laws will be assumed as well. The reason for not having to provide the monoid laws when performing an exploration is to makes it easier to write transformations of exploration functions.

Further note that there is nothing in the type that enforces that every element is applied once, and only once, in the result. These exploration functions are the motivating example but we will first study exploration functions in general. The exploration functions that do explore all values of a type A are said to be *exhaustive*. The name “exploration” is used because these functions are designed to systematically examine every possible value of the type. But it is also useful to study explorations that are not necessary exhaustive. The exhaustiveness of an exploration implies the finiteness of A.

In order to easily define new exploration functions we provide three building blocks inspired by binary trees. These three combinators are defined for any type A and correspond to the constructors *empty*, *leaf*, and *fork* respectively. The exploration function that corresponds with *empty* is *empty-explore*, an exploration function which does not explore anything and just returns the default value ϵ. The function *point-explore* takes a value x of type A and defines an exploration function which explores only this point x using the given exploration body. Finally the function *merge-explore* takes two exploration functions and combines them using the received binary operator $\_\oplus\_$.

$$\begin{aligned}
&\textit{empty-explore} : \forall\ \{\mathrm{A}\} \to \mathrm{Explore\ A}\\
&\textit{empty-explore}\ \epsilon\ \_\oplus\_\ f = \epsilon\\
&\textit{point-explore} : \forall\ \{\mathrm{A}\} \to \mathrm{A} \to \mathrm{Explore\ A}\\
&\textit{point-explore}\ x\ \epsilon\ \_\oplus\_\ f = f\ x\\
&\textit{merge-explore} : \forall\ \{\mathrm{A}\} \to \mathrm{Explore\ A} \to \mathrm{Explore\ A} \to \mathrm{Explore\ A}\\
&\textit{merge-explore}\ e_0\ e_1\ \epsilon\ \_\oplus\_\ f = (e_0\ \epsilon\ \_\oplus\_\ f) \oplus (e_1\ \epsilon\ \_\oplus\_\ f)
\end{aligned}$$

For exhaustively exploring finite types, we have some more specialised combinators. Generally, finite types are a combination of sums and products, and we therefore provide exploration combinators for those. As base case for finite types we have exploration functions for types such as $\bot$, $\mathbb{1}$ and *Bool*. These are defined by the simple exploration functions we defined above:

$$\begin{aligned}
&\textit{explore}_{\bot} : \mathrm{Explore}\ \bot\\
&\textit{explore}_{\bot} = \textit{empty-explore}\\
&\textit{explore}_{\mathbb{1}} : \mathrm{Explore}\ \mathbb{1}\\
&\textit{explore}_{\mathbb{1}} = \textit{point-explore tt}\\
&\textit{explore}_{\textit{Bool}} : \mathrm{Explore}\ \textit{Bool}\\
&\textit{explore}_{\textit{Bool}} = \textit{merge-explore}\ (\textit{point-explore false})\ (\textit{point-explore true})
\end{aligned}$$

For sum types A ⊎ B, the exploration $\textit{explore}_{\uplus}\ e_A\ e_B\ \epsilon\ \_\oplus\_\ f$ combines the two results given by exploring the function f specialised to types A and B using *inl* and *inr* — the injections for the type $\_\uplus\_$. The two results are then combined using $\_\oplus\_$.

$$\begin{aligned}
&\textit{explore}_{\uplus} : \forall\ \{\mathrm{A\ B}\} \to \mathrm{Explore\ A} \to \mathrm{Explore\ B} \to \mathrm{Explore\ (A \uplus B)}\\
&\textit{explore}_{\uplus}\ e_A\ e_B\ \epsilon\ \_\oplus\_\ f = (e_A\ \epsilon\ \_\oplus\_\ (f \circ \textit{inl})) \oplus (e_B\ \epsilon\ \_\oplus\_\ (f \circ \textit{inr}))
\end{aligned}$$

For Cartesian products A × B we define the exploration $\textit{explore}_{\times}\ e_A\ e_B\ \epsilon\ \_\oplus\_\ f$ as we did when only working on $\mathbb{N}$ i.e. by nesting the exploration of B into the function exploring A. Note how this combinator is independent of the operator $\_\oplus\_$. Support for dependent pairs and functions is detailed in Section 3.3.

$$\begin{aligned}
&\textit{explore}_{\times} : \forall\ \{\mathrm{A\ B}\} \to \mathrm{Explore\ A} \to \mathrm{Explore\ B} \to \mathrm{Explore\ (A \times B)}\\
&\textit{explore}_{\times}\ e_A\ e_B\ \epsilon\ \_\oplus\_\ f = e_A\ \epsilon\ \_\oplus\_\ (\lambda\ a \to e_B\ \epsilon\ \_\oplus\_\ (\lambda\ b \to f\ (a\ ,\ b)))
\end{aligned}$$

Derived big operators: We now derive some standard big operators, these are derived from an exploration function $explore_A$ by choosing the appropriate monoid structure. Sums and products are defined using the monoids ($\mathbb{N}$, *0* , _+_) and ($\mathbb{N}$, *1* , _*_) as mentioned earlier.

$$
\begin{aligned}
&sum\ :\ (A \to \mathbb{N}) \to \mathbb{N}\\
&sum\ =\ explore_A\ 0\ \_+\_\\
&product\ :\ (A \to \mathbb{N}) \to \mathbb{N}\\
&product\ =\ explore_A\ 1\ \_*\_
\end{aligned}
$$

From a summation function we derive a counting function *count*, this shows that indeed exploration functions are a generalisation of counting functions. Summing (with *sum* or *count*) using a constant function *1* yields the size of the exploration, which when the exploration function is exhaustive will be the cardinality of the type.

$$
\begin{aligned}
&count\ :\ (A \to Bool) \to \mathbb{N}\\
&count\,f\ =\ sum\ (Bool{\rhd}\mathbb{N} \circ f) \qquad \text{-- } Bool{\rhd}\mathbb{N}\ converts\ Bool\ into\ \mathbb{N}\\
&size\ :\ \mathbb{N}\\
&size\ =\ count\ (const\ true)
\end{aligned}
$$

Finally the functions *all* and *any* test a given predicate to tell whether it holds for all the explored values or any of the explored values, respectively.

$$
\begin{aligned}
&all\ :\ (A \to Bool) \to Bool\\
&all\ =\ explore_A\ true\ \_\wedge\_\\
&any\ :\ (A \to Bool) \to Bool\\
&any\ =\ explore_A\ false\ \_\vee\_
\end{aligned}
$$

As a last example, we show two transformation on exploration functions. The first is that we have a functorial action on explorations, i.e. we can map a function on the exploration.

$$
\begin{aligned}
&map\ :\ (A \to B) \to \text{Explore A} \to \text{Explore B}\\
&map\,f\,e_A\ \epsilon\ \_\oplus\_\ g\ =\ e_A\ \epsilon\ \_\oplus\_\ (g \circ f)
\end{aligned}
$$

Next we consider the monoid of endomorphisms featuring the identity function as the neutral element and function composition as the multiplication operation. Exploring with the monoid of endomorphisms expects a function body that will turn values of type A into functions of type M → M. The body composes the original small operator $\_\oplus\_$ with the original body f. We finally pass in the default value ϵ to the resulting big composition. When $(\epsilon, \_\oplus\_)$ is a monoid, this transformation computes to the same result as the original exploration. Its utility lies in the fact that function composition has an associative computational content which will force all the calls to $\_\oplus\_$ to be associated to the right, finally ending with a single ϵ. This technique, known as *difference lists*, has been used before and is part of the standard toolbox of functional programmers. Its original motivation was to improve the performance, but it is also useful for reasoning since it gives associativity for free. A proof of this technique has been given in [Voi09] and it is our Corollary 3.2.9. Notice that this technique is nicely captured by the following exploration transformer:

$$
\begin{aligned}
&\textit{explore-endo} \;:\; \text{Explore A} \to \text{Explore A}\\
&\textit{explore-endo}\; e_A\; \epsilon\; \_\oplus\_\; f \;=\; e_A\; \textit{id}\; \_\circ\_\; (\_\oplus\_ \circ f)\; \epsilon
\end{aligned}
$$

3.2 Exploration Principle

We now focus our attention how we can reason about exploration functions, and since we will start by looking at what parametricity tells us. The parametricity relation for the Explore type is shown bellow, and while it looks daunting it is fairly straightforward to use. The trick lies in that it is possible to pick any relation for $[\![\text{M}]\!]$.

$$
\begin{aligned}
&[\![\text{Explore}]\!] \;:\; \text{Explore A} \to \text{Explore A} \to \textit{Set}\\
&[\![\text{Explore}]\!]\; e\; e' \;=\; (\text{M M'} \;:\; \textit{Set})\; ([\![\text{M}]\!] \;:\; \text{M} \to \text{M'} \to \textit{Set})\\
&\quad (\epsilon \;:\; \text{M})\; (\epsilon p \;:\; \text{M'})\; (\epsilon_r \;:\; [\![\text{M}]\!]\; \epsilon\; \epsilon p)\\
&\quad (\_\oplus\_ \;:\; \text{M} \to \text{M} \to \text{M})\; (\_\oplus p\_ \;:\; \text{M'} \to \text{M'} \to \text{M'})\\
&\quad (\oplus_r \;:\; \forall\; \{x\, y\}\; \{x'\, y'\} \to [\![\text{M}]\!]\; x\; x' \to [\![\text{M}]\!]\; y\; y'\\
&\qquad \to [\![\text{M}]\!]\; (x \oplus y)\; (x' \oplus' y'))\\
&\quad (f \;:\; \text{A} \to \text{M})\; (f' \;:\; \text{A} \to \text{M'})\\
&\quad (f_r \;:\; \forall\; x \to [\![\text{M}]\!]\; (f\, x)\; (f'\, x))\\
&\quad \to [\![\text{M}]\!]\; (e\; \epsilon\; \_\oplus\_\; f)\; (e'\; \epsilon p\; \_\oplus p\_\; f')
\end{aligned}
$$

As an example of using this relation we work with monoid morphisms, which is the notion of structure preserving maps between monoids.

Definition 3.2.1: Monoid Morphism

A monoid morphism between two monoids m and n is a function $f : m \to n$, such that $f\,\epsilon = \epsilon$ and $f\,(x \oplus y) = f\,x \oplus f\,y$ for all x and y of type m.

We can prove that monoid homomorphisms distributes over explore, and we prove this by instantiating the relation $[\![M]\!]$ appropriately when we use the parametricity relation.

Theorem 3.2.2: Exploring Homomorphism

For any type A, exploration function e_A : Explore A, two monoids: (M, ϵ, _⊕_) and (N, ι, _⊗_), we have a monoid homomorphism h from M to N, and a function g : A → M, then $h\;(e_A\;\epsilon\;\_\oplus\_\;g) \equiv e_A\;\iota\;\_\otimes\_\;(h \circ g)$

Proof. By parametricity of e_A we pick $[\![M]\!]\;x\,y$ to be $h\,x \equiv y$. We need to prove: $h\;\epsilon \equiv \iota$ and for all x, x', y and y' such that $h\,x \equiv x'$, $h\,y \equiv y$ we have $h\;(x \oplus y) \equiv x' \otimes y'$. Both of these requirements follow from the fact that h is a monoid homomorphism. The final requirement is that for all x, $h\;(g\,x) \equiv h\;(g\,x)$ holds, which is trivial. □

As a simple example of this we see that multiplication distributes over summation.

Corollary 3.2.3: Multiplication of a Summation

For any type A, exploration function e_A : Explore A, function f : A → ℕ and constant k : ℕ, we have $k * sum\; e_A f \equiv sum\; e_A\;(\lambda\,x \to k * f\,x)$.

Proof. By Theorem 3.2.2 and the fact that (_*_ k) is a monoid homomorphism, since $k * 0 \equiv 0$ and $k * (x + y) \equiv k * x + k * y$. □

The final property we show with the parametricity relation is that we can find sometimes find an bound on the result of an exploration if the monoid is an pre-order.

Theorem 3.2.4: Monotonicity

For any type A, exploration function e_A : Explore A, a monoid (M, ϵ, _⊕_) equipped with a preorder _⩽_ such that _⊕_ is monotonic, two functions f, g : A → M such that for all x, $f\,x \leqslant g\,x$, we have $e_A\ \epsilon$ _⊕_ $f \leqslant e_A\ \epsilon$ _⊕_ g.

Proof. By parametricity of e_A we pick ⟦M⟧ to be _⩽_, all the requirements follow from assumptions. □

The parametricity relation is a powerful tool but sometimes we want something closer to an induction principle. An induction principle allows the target property (known as ⟦M⟧ in our previous proofs) to be not only a relation between two explorations, but can be an arbitrary predicate on the exploration function itself.

Definition 3.2.5: Exploration Principle

The exploration principle states that any property P on an exploration function e_A holds if: P holds for *empty-explore*; P holds for all points (using *point-explore*); and P is preserved by *merge-explore*.

```
Explore_P : ∀ {A} → Explore A → Set
Explore_P {A} e_A =
  ∀ (P   : Explore A → Set) (ϵ_P : P empty-explore)
    (⊕_P : ∀ {e₀ e₁} → P e₀ → P e₁ → P (merge-explore e₀ e₁))
    (f_P : ∀ x → P (point-explore x)) → P e_A
```

All the exploration functions we have defined so far all come with the principle defined above. This principle is the induction principle on binary trees, but where *empty*, *node*, and *leaf*, respectively become *empty-explore*, *merge-explore* and *point-explore*. Put differently, this property enforces that an exploration function is essentially a binary tree where empty trees are ϵ, nodes are calls to _⊕_, and leaves are calls to f.

Moreover, while the type of the principle, i.e. Explore_P, looks a bit daunting, it is a simple mechanical process to prove it: one mimics what happens in the underlying exploration function. Below is the actual proof term of this principle for our $\textit{explore}_\times$ function. Thanks to implicit parameters, also we hide the motive, the proof term $\textit{explore}^P_\times$ is almost like $\textit{explore}_\times$:

$$\begin{aligned}&\mathit{explore}^P_\times : \mathrm{Explore}_P\ e_A \rightarrow \mathrm{Explore}_P\ e_B \rightarrow \mathrm{Explore}_P\ (\mathit{explore}_\times\ e_A\ e_B)\\ &\mathit{explore}^P_\times\ e^P_A\ e^P_B\ \epsilon\ \_\oplus\_\ f = e^P_A\ \epsilon\ \_\oplus\_\ (\lambda\, a \rightarrow e^P_B\ \epsilon\ \_\oplus\_\ (\lambda\, b \rightarrow f\,(a\,,\,b)))\end{aligned}$$

In an impredicative setting at least the principle is equivalent to the parametricity relation, but so far we have not been able to prove this correspondence in a predicative setting. This causes some duplication in the amount of work one has to do when providing an exploration function, though this work is mostly mechanical.

The first example of when we need to use this stronger principle is that explorations act homomorphically over the monoid operation, if we are exploring a commutative monoid.

Theorem 3.2.6: Exploration over Commutative Operator

For any type A, exploration function e_A : Explore A, a commutative monoid (M, ϵ, $\_\oplus\_$) and two functions f, g : A → M, we have $e_A\ \epsilon\ \_\oplus\_\ (\lambda\, x \rightarrow f\,x \oplus g\,x) \equiv e_A\ \epsilon\ \_\oplus\_\ f \oplus e_A\ \epsilon\ \_\oplus\_\ g$.

Proof. By the principle of e_A and picking the motive[a] P e to be

$$e\ \epsilon\ \_\oplus\_\ (\lambda\, x \rightarrow f\,x \oplus g\,x) \equiv e\ \epsilon\ \_\oplus\_\ f \oplus e\ \epsilon\ \_\oplus\_\ g$$

We need to show P *empty-explore* which is $\epsilon \equiv \epsilon \oplus \epsilon$ which follows by monoid law. The case P (*merge-explore* $e_0\ e_1$) where P e_0 and P e_1 follows by the interchange law (i.e for all a, b, c, and d then $(a \oplus b) \oplus (c \oplus d) \equiv (a \oplus c) \oplus (b \oplus d)$. Finally we need to prove for all x that it holds that P (*point-explore* x) which is $f\,x \oplus g\,x \equiv f\,x \oplus g\,x$ which is trivial. □

[a]It is common to refer as P being the motive for the induction which is a form of elimination. As Conor McBride writes in [Mcb02] "we should give elimination a *motive*".

We can furthermore show how the product monoid compute, which shows that the two monoids are in fact computed independently.

Theorem 3.2.7: Exploring the Product Monoid

For any type A, exploration function e_A : Explore A, two monoids[a] (M, ϵm, $\_\oplus_m\_$) and (N, ϵn, $\_\oplus_n\_$), two functions f_m : A → M and f_n : A → N, we have the exploration of the product monoid is the product of explorations, namely $e_A\ \epsilon\ \_\oplus\_ < f_m \times f_n > \equiv (e_A\ \epsilon_m\ \_\oplus_m\_\ f_m\ ,\ e_A\ \epsilon_n\ \_\oplus_n\_\ f_n)$. Where ((M × N) $,\epsilon\ ,\_\oplus\_$) is the product monoid.

Proof. By the principle of e_A and picking the motive to be P e to be

$$e\ \epsilon\ \_\oplus\_\ {<}f_m \times f_n{>} \equiv (e\ \epsilon_m\ \_\oplus_m\_ f\_m,\ e\ \epsilon_m\ \_\oplus_m\_ f_n))$$

We need to show P *empty-explore* which holds by definition of the product monoid. The case for P (*merge-explore* e_0 e_1) where P e_0 and P e_1 follows by congruence of $\_\oplus\_$ and its definition. Finally we need to prove for all x_m and x_n that P (*point-explore* (x_m, x_n))) which is ${<}f_m \times f_n{>} (x\_m, x_n))$ $\equiv (f_m\ x\_m, f\_n, x_n)$ which holds by definition. □

[a]The monoid laws are actually not used for this theorem

At the end of section 3.1 we introduced *explore-endo* which re-associates the exploration. We can now prove that this indeed only does associate in an other way. First we need to prove a slightly more general theorem:

Theorem 3.2.8: Reassociate the Exploration

For any type A, exploration function e_A : Explore A, monoid (M ,ϵ ,$\_\oplus\_$), function f : A → M, and point z : M, we have that:
$e_A\ \epsilon\ \_\oplus\_ f \oplus z \equiv e_A\ id\ \_\circ\_\ (\_\oplus\_ \circ f)\ z$.

Proof. By the principle of e_A and picking the motive P e to be:

$$\forall\ z \rightarrow e\ \epsilon\ \_\oplus\_ f \oplus z \equiv e\ id\ \_\circ\_\ (\_\oplus\_ \circ f)\ z$$

We need to show P *empty-explore* which is $\forall\ z \rightarrow \epsilon \oplus z \equiv z$ and follows by monoid law. The case for P (*point-explore* x) holds by definition. Finally we need to prove the case for P (*merge-explore* e_0 e_1) where P e_0 and P e_1 hold. By definition it amounts proving that for all z, $(e_0\ \epsilon\ \_\oplus\_ f \oplus e_1\ \epsilon\ \_\oplus\_ f) \oplus z$ equals $e_0\ id\ \_\circ\_\ (\_\oplus\_ \circ f)\ (e_1\ id\ \_\circ\_\ (\_\oplus\_ \circ f)\ z)$. The assumption P e_1 can be used on z and P e_0 can be used on $e_1\ \epsilon\ \_\oplus\_ f \oplus z$. Using the associativity and congruence of $\_\oplus\_$ the proof is complete. □

And now we can prove that *explore-endo* indeed only re-associate:

Corollary 3.2.9: *explore-endo* Preserves Exploration

For any type A, exploration function e_A : Explore A, then any exploration can be re-associated using the monoid on endomorphisms, namely for all monoid (M, ϵ, $\_\oplus\_$) and function f : A → M, we have $e_A\ \epsilon\ \_\oplus\_\ f$ $\equiv$ *explore-endo* $e_A\ \epsilon\ \_\oplus\_ f$.

Proof. Use Theorem 3.2.8 with z being ϵ and conclude by monoid laws. □

The final theorem of this section show that the deep connection with binary trees and exploration functions. With binary trees we mean the following data type:

```
data Tree (A : Set) : Set where
  empty : Tree A
  leaf  : A → Tree A
  fork  : (l r : Tree A) → Tree A
```

We can write functions back and forth between binary trees and exploration functions:

```
  {-Fold over binary trees: -}
foldMap^T : ∀ {A} → Tree A → Explore A
foldMap^T empty      = empty-explore
foldMap^T (leaf x)   = point-explore x
foldMap^T (fork l r) = merge-explore (foldMap^T l)
  (foldMap^T r)


toTree : ∀ {A} → Explore A → Tree A
toTree e_A = e_A empty fork leaf
```

These functions are inverses of each other, which allows us to treat exploration functions as data. We show below one direction, the other is proved by induction on binary trees.

Theorem 3.2.10: Explorations as Trees

For any type A, exploration function e_A : Explore A, monoid (M, ϵ, _⊕_) and function f : A → M, we have $e_A\ \epsilon$ _⊕_ $f \equiv$ *foldMap*^{T} (*toTree* e_A) ϵ _⊕_ f.

Proof. By the principle of e_A and picking the motive to be

$$\mathrm{P}\ e\ =\ e\ \epsilon\ \_\oplus\_\ f \equiv \mathit{foldMap}^T\ (\mathit{toTree}\ e)\ \epsilon\ \_\oplus\_\ f$$

All cases are trivial. □

3.3 Exploring Dependent Types

The function $\mathit{explore}_{\times}$ explores the Cartesian product A × B given explorations for A and B. This construction nicely scales to dependent pairs, in order to explore Σ A B one needs a family of explorations for each B x where x has type A. This implies a small change in comparison to $\mathit{explore}_{\times}$, namely that x is also given to $\mathit{explore}_B$:

$$\begin{array}{l}
\mathit{explore}_{\Sigma} \;:\; \text{Explore A} \to (\forall\, x \to \text{Explore (B}\,x))\\
\quad \to \text{Explore } (\Sigma \text{ A B})\\
\mathit{explore}_{\Sigma}\; \mathit{explore}_A\; \mathit{explore}_B\; \epsilon\; \_{\oplus}\_\; f\\
\quad = \mathit{explore}_A\; \epsilon\; \_{\oplus}\_\; \lambda\, x \to\\
\qquad \mathit{explore}_B\; x\; \epsilon\; \_{\oplus}\_\; \lambda\, y \to f\,(x\,,y)
\end{array}$$

While we found no way to combine two exploration functions to get get an exploration function over the function type, such as Explore (A → B), but there is an attractive workaround. One can use type equivalences on functions to incrementally build such an exploration function. Namely, one decomposes the domain with type equivalences towards simpler types we can explore:

$$\begin{array}{rcl}
\mathbb{1} \to A & = & A\\
A \uplus B \to C & = & (A \to C) \times (B \to C)\\
A \times B \to C & = & A \to (B \to C)
\end{array}$$

In order to prove these type equivalences we are required to use function extensionality. Though we are not required to define the exploration functions themselves using function extensionality, the proofs of these type equivalences are required to prove their adequacy.

As an example of building such an exploration function consider that we wish to build an exploration function e : Explore ($\mathit{Bool} \uplus \mathbb{1} \to$ A) for some type A, for which we have an exploration function e_A : Explore A. By using the above type equivalences we get that $\mathit{Bool} \uplus \mathbb{1} \to$ A is equivalent to (A × A) × A:

$$\begin{array}{l}
\mathit{conv} \;:\; (\text{A} \times \text{A}) \times \text{A} \to \mathit{Bool} \uplus \mathbb{1} \to \text{A}\\
\mathit{conv}\; ((x\,,y)\,,z)\; (\mathit{inl\,true}) \;=\; x\\
\mathit{conv}\; ((x\,,y)\,,z)\; (\mathit{inl\,false}) \;=\; y\\
\mathit{conv}\; ((x\,,y)\,,z)\; (\mathit{inr}\;\_) \quad\;\; =\; z
\end{array}$$

$$\begin{array}{l}
e \;:\; \text{Explore } (\mathit{Bool} \uplus \mathbb{1} \to \text{A})\\
e \;=\; \mathit{map\;conv}\; (\mathit{explore}_{\times}\; (\mathit{explore}_{\times}\; e_A\; e_A)\; e_A)
\end{array}$$

How can we ensure that we have a correct summation function? We need to ensure that an adequate summation function is going to count every value exactly once (i.e. an adequate summation function is not allowed to forget a value or over-count it). In order to guarantee this we use a strong correspondence between the sizes[1] of types in type theory and the act of summing. We use this correspondence as a specification for the summation functions that fully explores a type. It boils down to the observation that Σ A F is acting as a big operator for disjoint union of all F x where x is of type A. Therefore the size of a Σ-type is the summation of the sizes over the type family: $|\mathrm{Fin}\, n| \equiv n$ and $|\Sigma\,(x : \mathrm{A})\,.\,\mathrm{B}| \equiv \Sigma_{x:\mathrm{A}}|\mathrm{B}|$.

Using these size relations we can show that sum_A a summation function is correct, assuming a particular type equivalence exists. Since type equivalences preserve sizes, we argue as follows.

$$\begin{aligned} sum_A\, f &\equiv |\mathrm{Fin}\,(sum_A\, f)| \\ &\equiv |\Sigma\,\mathrm{A}\,(\mathrm{Fin} \circ f)| \\ &\equiv \textstyle\sum_{x \in \mathrm{A}} |\mathrm{Fin}\,(f\, x)| \\ &\equiv \textstyle\sum_{x \in A} f\, x \end{aligned}$$

Definition 3.3.1: Adequate Summation

A function sum_A for a type A is said to be an adequate sum if for all f there is an equivalence between Σ A (Fin $\circ\, f$) and Fin ($sum_A\, f$).

This correspondence can be further extended to products, as Π-types can be seen as the big operator for products. The correctness for product functions can be defined using correspondence similar to the one for summation functions:

$$\begin{aligned} prod_A\, f &\equiv |\mathrm{Fin}\,(prod_A\, f)| \\ &\equiv |\Pi\,\mathrm{A}\,(\mathrm{Fin} \circ f)| \\ &\equiv \textstyle\prod_{x \in \mathrm{A}} |\mathrm{Fin}\,(f\, x)| \\ &\equiv \textstyle\prod_{x \in A} f\, x \end{aligned}$$

[1]We use the notion of size only as an informal guide.

Definition 3.3.2: Adequate Product

A function $prod_A$ for a type A is said to be an adequate product if for all f there is an equivalence between Π A (Fin $\circ f$) and Fin ($prod_A f$).

Not only does the proof of adequacy show that our summation or product function acts like they are supposed to, we can use the property for to prove new equalities. Our first use of adequacy for sums and products is to prove the following equation:

$$\forall(f \in (A \times B) \to \mathbb{N}). \prod_{x \in A} \sum_{y \in B} f(x,y) \equiv \sum_{g \in A \to B} \prod_{x \in A} f(x, g(x)) \tag{3.1}$$

At first we prove a more general result, where B is a family indexed by A and thus dependent functions and dependent pairs are required. The non-dependent version is given as a corollary.

Theorem 3.3.3: Dependent Product of Summation

Let $prod_A$ be an adequate product function for the type A. Let sum_{AB} be an adequate summation function for a type Π A B. Finally let sum_B be a family over A of summation functions on the type B. Then for all function f : (x : A) $\to$ B $x \to \mathbb{N}$, $prod_A$ ($\lambda\, x \to sum_B\, x\, (\lambda\, y \to f\, x\, y)$) is equal to sum_{AB} ($\lambda\, g \to prod_A\, (\lambda\, x \to f\, x\, (g\, x))$).

Proof. Using the adequacy properties together with the type equivalence between Π A ($\lambda\, x \to \Sigma$ (B x) $\lambda\, y \to$ C $x\, y$) and Σ (Π A B) $\lambda\, f \to \Pi$ A $\lambda\, x \to$ C $x\, (f\, x)$[a]. $\square$

[a] The logical interpretation of the forward direction is usually known as the dependent axiom of choice.

The proof of the equation 3.1 follows as an corollary.

Corollary 3.3.4: Product of Summation

Let sum_B and sum_{AB} be adequate summation functions for a type B and A $\to$ B respectively. Furthermore let $prod_A$ be an adequate product function for the type A. Then for all function f : A $\to$ B $\to \mathbb{N}$,
$prod_A$ ($\lambda\, x \to sum_B\, (\lambda\, y \to f\, x\, y)$) is equal to
sum_{AB} ($\lambda\, g \to prod_A\, (\lambda\, x \to f\, x\, (g\, x))$).

Proof. Since non-dependent functions are a particular case of dependent functions one can directly use Theorem 3.3.3. □

Furthermore, we can prove that *size* which is the specialisation of an exploration function to compute the cardinality of an exploration really does compute the cardinality. In fact we can show that for a type A with an adequate summation function, A needs to be a finite type.

Lemma 3.3.5: Adequate Exploration and Cardinality

Having an adequate summation function *sum* over type A, with a derived size $size : \mathbb{N}$, it is possible to construct an equivalence $\mathrm{Fin}\, size \simeq \mathrm{A}$.

Proof. By *sum* being adequate summation and the type equivalence:
$\mathrm{Fin}\, size \simeq \Sigma\, \mathrm{A}\, (\lambda\ \_ \to \mathrm{Fin}\ 1) \simeq \mathrm{A}$. □

The order in which one sums the elements of a type doesn't matter, since all elements are counted once and only once. One way we can show this is to reshuffle the elements in the summation, and we expect that this should not affect the result. One way of performing a reshuffling is to first apply an function that forms a type isomorphism, the following theorem shows that adequacy alone guarantees that an adequate summation function is invariant under shuffling.

Theorem 3.3.6: Adequate Summation and Type Isomorphism

Given two adequate summation functions sum_A and sum_B for types A and B respectively, for all equivalences $\pi : \mathrm{A} \simeq \mathrm{B}$ and functions $f : \mathrm{B} \to \mathbb{N}$ the summation $sum_A\, (f \circ \pi)$ is equal to the summation $sum_B\, f$.

Proof. Using adequacy of the summation functions and the Lemma ref finsize and Theorem 2.4.2 we get an equivalence $thm : \mathrm{Fin}\, (sum_A\, (f \circ \pi))$ $\simeq \mathrm{Fin}\, (sum_B\, f)$. Since Fin is injective (i.e. $\mathrm{Fin}\ m \simeq \mathrm{Fin}\ n \to m \equiv n$) the proof is complete.

$$\begin{tikzcd}
\mathrm{Fin}\,(sum_A\,(f \circ \pi)) \arrow[r, leftrightarrow, "\text{thm}"] \arrow[d, leftrightarrow, "sum_A \text{ adequate}"'] & \mathrm{Fin}\,(sum_B\, f) \arrow[d, leftrightarrow, "sum_B \text{ adequate}"] \\
\Sigma\, \mathrm{A}\,(\mathrm{Fin} \circ f \circ \pi) \arrow[r, leftrightarrow, "\text{lemma 2.4.2}"] & \Sigma\, \mathrm{B}\,(\mathrm{Fin} \circ f)
\end{tikzcd}$$

□

One particular example of such an invariance is that the order of nested summations doesn't matter.

Lemma 3.3.7: Exchanging Summations

Given two summation functions sum_A and sum_B for type A and type B, if both are adequate they satisfy the commutation property that $sum_A\,(\lambda\, a \rightarrow sum_B\,(\lambda\, b \rightarrow f\,(a\,,\,b)))$ is equal to $sum_B\,(\lambda\, b \rightarrow sum_A\,(\lambda\, a \rightarrow f\,(a\,,\,b)))$.

Proof. By adequacy of sum_A and sum_B and the type equivalence between $\Sigma\,\mathrm{A}\,\lambda\,x \rightarrow \Sigma\,\mathrm{B}\,\lambda\,y \rightarrow \mathrm{C}\,x\,y$ and $\Sigma\,\mathrm{B}\,\lambda\,y \rightarrow \Sigma\,\mathrm{A}\,\lambda\,x \rightarrow \mathrm{C}\,x\,y$. □

Finally, we prove that all values are indeed summed once and only once when using an adequate summation function *sum*.

Theorem 3.3.8: Adequate Summation Count Uniquely

Assume for a type A that we have a boolean equality test[a] _==_ such that, for all x and y of type A, the type $(x\ ==\ y) \equiv \mathit{true}$ is equivalent to $x \equiv y$. Furthermore, assume an adequate summation function *sum*, from which we derive a counting function *count*. Then, for all x, the equation $\mathit{count}\,(\lambda\, y \rightarrow x\ ==\ y) \equiv \mathit{1}$ holds.

Proof. Using the fact that *sum* is an adequate summation function together with the type equivalence $\Sigma\,\mathrm{A}\,(\lambda\, y \rightarrow x \equiv y) \simeq \mathbb{1}$. □

[a] Since we have an adequate summation function we know that Fin *size* $\simeq$ A so such an boolean equality test must exist.

3.4 Probabilities and Negligible Distance

Our original motivation was to work with summation functions as a way to compute and reason about uniform discrete probability distributions. Using an exploration function, we can derive a summation function which has stronger properties, by getting the theorems that follows from the principle and adequacy. We can sum the values of a given type, if they belong to a particular event. But before diving into probabilistic reasoning we will first formally define the positive rational numbers.

Any positive rational number can be represented by a pair of natural numbers such that the second one is non-zero, with the idea is that (x, y) is really the number x/y. To get the positive rational numbers from this construction one should perform a quotient such that $(1, 2)$ and $(2, 4)$ are the same numbers. This causes problems since quotient types are not available in our type theory, and we need to emulate them using a setoid construction[BCP00]. We will therefore construct bare rational numbers and then give an explicit equality relation that will equate $(1, 2)$ and $(2, 4)$. I will continue to use $\mathbb{Q}^+$ for these terms, even though they are not of the quotient.

```
record ℚ⁺ : Set where
  constructor _/_[_]
  field
    εN : ℕ
    εD : ℕ
    εD—pos : εD > 0
```

We can define addition on $\mathbb{Q}^+$ in the usual way, i.e. $\frac{a}{b} + \frac{c}{d}$ is equal to $\frac{a*d+c*b}{b*d}$. Furthermore we need to prove that the denominator is larger then 0, which we can do by the fact that * is monotonic in both it's arguments.

```
_+ℚ_ : ℚ⁺ → ℚ⁺ → ℚ⁺
(εN / εD [ εD+ ] +ℚ μN / μD [ μD+ ])
  = (εN * μD + μN * εD) / εD * μD [ εD+ *—mono μD+ ]
```

Instead of giving the equivalence relation we will directly go for $\leqslant$ since this is the relation we are actually interested in for the rational numbers. This is captured by the type $\_\leqslant_{\mathbb{Q}}\_$. We choose here to give this using a record in order to be able to control the unfolding of the definition.

```
record _≤ℚ_ (p q : ℚ⁺) : Set where
  constructor mk
  field
    unfold : ϵN p * ϵD q ≤ ϵN q * ϵD p
```

We get here by taking as definition, the property that $a/b \leqslant c/d$ holds if and only if $a*d \leqslant c*b$. Using this definition we can prove the usual properties we expect such as reflexivity, transitivity. Reflexivity holds by reflexivity on the underlying order $\leqslant$ on natural numbers, whereas getting transitivity requires some more work, the proof below for transitivity uses an extended equational reasoning, that also works for partial orders.

```
≤ℚ-trans : ∀ {f g h} → f ≤ℚ g → g ≤ℚ h → f ≤ℚ h
≤ℚ unfold (≤ℚ-trans {fN / fD [ fD-pos ]} {gN / gD [ gD-pos ]}
                    {hN / hD [ hD-pos ]} (mk fg) (mk gh))
  = ≤-*-cancel gD-pos lemma
  where
    open ≤-Reasoning
    lemma : gD * (fN * hD) ≤ gD * (hN * fD)
    lemma = gD * (fN * hD)
              ≡⟨ ! ℕ°.*-assoc gD fN hD • ap (flip _*_ hD) (ℕ°.*-comm gD fN) ⟩
                (fN * gD) * hD
              ≤⟨ fg *-mono ℕ≤ refl ⟩
                (gN * fD) * hD
              ≡⟨ ℕ°.*-assoc gN fD hD • ap (_*_ gN) (ℕ°.*-comm fD hD)
                • ! ℕ°.*-assoc gN hD fD ⟩
                (gN * hD) * fD
              ≤⟨ gh *-mono ℕ≤ refl ⟩
                (hN * gD) * fD
              ≡⟨ ap (flip _*_fD) (ℕ°.*-comm hN gD) • ℕ°.*-assoc gD hN fD ⟩
                gD * (hN * fD)
              ∎
```

Remark 3.4.1

As a consequence we get equality by defining that $\leqslant_{\mathbb{Q}}$ should hold in both directions, which is equivalent to the standard definition namely $(a,b) = (c,d)$ if $a*d = c*b$.

A probabilistic value, also called random variable, of type A over some sample space Ω is a function $\Omega \to A$. We will use adequate exploration functions to automatically derive probability distributions, these distributions are finite, discrete and uniform by construction. Of particular interest is probabilistic values of the Boolean type, i.e. the type A is *Bool*, which is means that the probabilistic value is an event. It is of these values that we calculate the probability of them being *true*, either concretely or abstractly, by simply dividing the number of times the event returns *true* by the total size of the sample space. This puts in the extra requirement that the sample space can't be empty, which we represents by an extra assumption that the size is strictly bigger than 0. Types that have such an adequate exploration function with the non-zero size is called a RSpace:

```
record RSpace (A : Set) : Set where
  constructor mk
  field
    explore : Explore A
    adq-exp : Adequate explore
    nz-size : 0 < size explore
```

For every RSpace we can define the probability function Prob:

```
Prob : {A : Set} → RSpace A → (A → Bool) → ℚ⁺
Prob RS E = sum (explore RS) (Bool▹ℕ ∘ E) / size (explore RS) [ nz-size RS ]
syntax Prob e (λ x → E) = Pr[ e ≫ x || E ]
```

In the rest of this thesis we use a syntactic short-cut to call the Prob function, by using Pr[e $\gg$ x || E] we use the sample space explained by the exploration function e and bind the sample x in the event E. The function *b2*▹ℕ is the simple conversion function from Booleans to natural numbers, that maps *false* to 0 and *true* to 1.

Since summation functions are invariant under type isomorphism, so is probabilities which is shown by the following theorem.

Theorem 3.4.2

Given two types A and B, each with adequate exploration functions e_A and e_B respectively, a type isomorphism $\pi : A \simeq B$, and an event on B namely $F : B \to$ *Bool*, then Pr[$e_B \gg x$ || F x] is equal to Pr[$e_A \gg x$ || F $(\pi\, x)$].

Proof. Follows by Theorem 3.3.6. □

A function is negligible if after some point will tend towards zero, another way of phrasing this is that the function is bounded by a $\frac{1}{x^c}$, for some c. Using mathematical notation the latter criterion can be characterised by the following inequality should hold for ε to be negligible.

$$\forall c . \exists n_c . \forall n > n_c . \varepsilon(n) \leqslant n^{-c}$$

We push out the existential and instead use a function to compute at which point the function is bound by the polynomial for each input. This function is called c_n in the following predicate for functions of type $\mathbb{N} \to \mathbb{Q}^+$:

```
record Is-Neg (ϵ : ℕ → ℚ⁺) : Set where
  constructor mk
  field
    cₙ : (c : ℕ) → ℕ
    prf : ∀ (c n : ℕ) → n > cₙ n → n  c * ϵN (ϵ n)  ⩽  ϵD (ϵ n)
```

We can of course add two functions of such at type, i.e. $\mathbb{N} \to \mathbb{Q}^+$, by point-wise addition, i.e. $(f + g)\,(n)$ is $f\,(n) + g\,(n)$. This point-wise addition is called _+ℕℚ_ and we can indeed prove that adding two negligible functions results in a negligible function. We omit for presentation purpose the lemma the helper lemma, and here only gives the type.

```
+ℕℚ—neg : {ϵ μ : ℕ→ℚ} → Is-Neg ϵ → Is-Neg μ → Is-Neg (ϵ +ℕℚ μ)
cₙ (+ℕℚ—neg ϵ μ) n = 1 + cₙ ϵ n + cₙ μ n
prf (+ℕℚ—neg {ϵM} {μM} ϵ μ) c n n>nc
   = ⩽—*—cancel {x = n} (ℕ⩽.trans (s⩽s z⩽n) n>nc) lemma
  where
  open module DUMMY x = ℚ⁺ (ϵM x)
  open module DUMMY2 x = ℚ⁺ (μM x)
    renaming (ϵN to μN; ϵD to μD; ϵD—pos to μD—pos)
  lemma : n * (n  c * (ϵN n * μD n + μN n * ϵD n))  ⩽  n * (ϵD n * μD n)
```

So with that done, we can now move on to limit a function by another function, this is just lifting point-wise the partial relation $\leqslant$ on $\mathbb{Q}$. We have here in-lined the definitions.

```
infix 4 _≤→_
record _≤→_ (f g : ℕ → ℚ⁺) : Set where
  constructor mk
  field
      -- fN k / fD k ≤ gN k / gD k
    unfold : ∀ k → f k ≤ℚ g k
```

Since this is just lifting a known partial order, it is easy to show that the result is also a partial order. We show here the proof of transitivity and that point-wise addition is monotonic, the other follows by a similar construction. The latter uses the proof that addition on $\mathbb{Q}^+$ is monotonic, which is witnessed by the term *+ℚ-mono*.

```
≤→–trans : ∀ {f g h} → f ≤→ g → g ≤→ h → f ≤→ h
≤→.unfold (≤→–trans (mk fg) (mk gh)) k = ≤ℚ–trans (fg k) (gh k)

+ℕℚ–mono : ∀ {f f' g g'} → f ≤→ f' → g ≤→ g' → f +ℕℚ g ≤→ f' +ℕℚ g'
≤→.unfold (+ℕℚ–mono (mk ff) (mk gg)) k = +ℚ-mono (ff k) (gg k)
```

We make use of NegBounded, which is a predicate that a function is less than some negligible function.

```
record NegBounded (f : ℕ → ℚ⁺) : Set where
  constructor mk
  field
    ϵ : ℕ → ℚ⁺
    ϵ–neg : Is-Neg ϵ
    bounded : f ≤→ ϵ
```

Here comes three simple properties about NegBounded functions. The first one is that a negligible function is trivially NegBounded by itself.

```
fromNeg : {f : ℕ→ℚ} → Is-Neg f → NegBounded f
ϵ (fromNeg f-neg) = _
ϵ–neg (fromNeg f-neg) = f-neg
bounded (fromNeg f-neg) = ≤→–refl
```

The second property, that a smaller function than a NegBounded function is also NegBounded. This is a simple way to prove that a function is negligible bounded.

```
≤–NB : {f g : ℕ→ℚ} → f ≤→ g → NegBounded g → NegBounded f
ϵ (≤–NB le nb) = ϵ nb
ϵ–neg (≤–NB le nb) = ϵ–neg nb
bounded (≤–NB le nb) = ≤→–trans le (bounded nb)
```

The third property is that point-wise addition of NegBounded functions is still NegBounded.

$$\begin{aligned}
&\_{+NB}\_ \;:\; \{f\,g \;:\; \mathbb{N}{\to}\mathbb{Q}\} \to \text{NegBounded}\,f \to \text{NegBounded}\,g \\
&\qquad\qquad\qquad\qquad \to \text{NegBounded}\,(f +\mathbb{N}\mathbb{Q}\,g) \\
&\epsilon\;(fNB +NB\,gNB) \;=\; \epsilon\,fNB +\mathbb{N}\mathbb{Q}\;\epsilon\,gNB \\
&\epsilon\text{—}neg\;(fNB +NB\,gNB) \;=\; +\mathbb{N}\mathbb{Q}\text{—}neg\;(\epsilon\text{—}neg\,fNB)\,(\epsilon\text{—}neg\,gNB) \\
&bounded\,(fNB +NB\,gNB) \;=\; +\mathbb{N}\mathbb{Q}\text{—}mono\;(bounded\,fNB)\,(bounded\,gNB)
\end{aligned}$$

The final thing we need to define about rational numbers, is the concept of distance between two rational numbers. This captures the mathematical concept of $|x - y|$ which we denote by *dist*$\mathbb{Q}$, this shows up when we later define the concept of advantage. This uses a function to compute the distance on natural numbers called *dist*$\mathbb{N}$:

$$\begin{aligned}
&dist\mathbb{N} \;:\; \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&dist\mathbb{N}\;0\;n \;=\; n \\
&dist\mathbb{N}\;m\;0 \;=\; m \\
&dist\mathbb{N}\;(suc\;m)\;(suc\;n) \;=\; dist\mathbb{N}\;m\;n \\
&dist\mathbb{Q} \;:\; \mathbb{Q}^+ \to \mathbb{Q}^+ \to \mathbb{Q}^+ \\
&dist\mathbb{Q}\;(\epsilon\text{N} \;/\; \epsilon\text{D}\,[\;\epsilon\text{D}+\;])\,(\mu\text{N} \;/\; \mu\text{D}\,[\;\mu\text{D}+\;]) \\
&\quad =\; dist\;(\epsilon\text{N} * \mu\text{D})\,(\mu\text{N} * \epsilon\text{D}) \;/\; \epsilon\text{D} * \mu\text{D}\,[\;\epsilon\text{D}+ *\text{—}mono\;\mu\text{D}+\;]
\end{aligned}$$

This function is congruent over the equality of $\mathbb{Q}^+$ and furthermore satisfy the triangle inequality. This property is very useful when doing proofs, and we will use it.

Theorem 3.4.3

Given three rational numbers, p,q and r, then the distance $dist\mathbb{Q}\;p\;q$ is less than $dist\mathbb{Q}\,p\,r +_{\mathbb{Q}}\; dist\;r\;q$.

Proof. This proof uses two facts of the *dist*$\mathbb{N}$ function, one that it also satisfy the triangle inequality property, and two that one can factor out multiplication, i.e. for all a,b and c, then $c * dist\mathbb{N}\;a\;b$ is equal to $dist\mathbb{N}\;(c * a)\;(c * b)$. □

3.5 Group Theory in Type Theory

In this section we present type theoretic proofs of the group theory presented in the background material. In general an algebraic structure is represented by using records, since this provides with clearer names for both operators and laws. Furthermore we split of all the operations, which we will call the "data", and the proofs, i.e. the laws. The first component for groups is the type Group-Ops, which contain only the data, the operators, of the group of a given type G.

```
record Group-Ops (G : Set) : Set where
  field
    _•_  : G → G → G
    ϵ    : G
    _⁻¹  : G → G
  _/_ : G → G → G
  x / y = x • y ⁻¹
```

The type G represents the carrier set, and the fields are the operators, namely the binary operation _•_, the identity element ϵ and finally an inverse operator $\_^{-1}$. Furthermore we have a general definition of $x\ /\ y$ which is defined for every group to be $x \bullet y^{-1}$. As mentioned above this record does not contain any of the group axioms, they are instead captured by the following record Group-Struct which is indexed by the data of Group-Ops which contains the operators of interest.

```
record Group-Struct {G : Set} (grp-ops : Group-Ops G) : Set where
  open Group-Ops grp-ops
  field
    assoc : Associative _•_
    identity : Identity ϵ _•_
    inverse  : Inverse ϵ _⁻¹ _•_
  ϵ•-identity : LeftIdentity ϵ _•_
  ϵ•-identity = fst identity
  ⁻¹•-identity : LeftInverse ϵ _⁻¹ _•_
  ⁻¹•-identity = fst inverse
  •= : ∀ {a b a' b'} → a ≡ a' → b ≡ b' → a • b ≡ a • b'
  •= = ap₂ _•_
```

There are three properties, and each are represented by one field in

Group-Struct[2]. The field *assocs* is that the operator of the group _•_ is associative, the field *identity* is stating that ϵ is indeed the identity element for _•_, and *inverse* which states that $\_^{-1}$ is indeed an inverse for _•_. Later in the record there are definitions to project out particular equality proofs, e.g. *ϵ•-identity* states that ϵ is a left inverse. In this definition only the first projection is given for each proof, the second projection is similar, e.g. and the second *•ϵ-identity* that it is a right identity. Furthermore we have a short cut for the congruence proof of _•_ in •=. The combination of both of these records, constructs a proper Group:

```
record Group (G : Set) : Set where
  field
    grp-ops    : Group-Ops G
    grp-struct : Group-Struct grp-ops
  open Group-Ops grp-ops public
  open Group-Struct grp-struct public
```

Now that we have a proper type theoretic definition Group we can prove the theorems stated in the preliminaries in the introduction. The first of these will be the for a given group we can cancel elements see Lemma 1.1.2.

```
cancels-•-left : LeftCancel _•_
cancels-•-left {c} {x} {y} e
  = x           ≡⟨ ! ϵ•-identity ⟩
    ϵ • x       ≡⟨ •= (! ⁻¹•-inverse) refl ⟩
    c ⁻¹ • c • x ≡⟨ !assoc= e ⟩
    c ⁻¹ • c • y ≡⟨ •= ⁻¹•-inverse refl ⟩
    ϵ • y       ≡⟨ ϵ•-identity ⟩
    y           ■
```

As a second example of how to transcribe a proof, we give the proof that the identity of the group is unique, which is Lemma 1.1.3.

```
unique—ϵ—right : ∀ {x y} → x • y ≡ x → y ≡ ϵ
unique—ϵ—right eq
  = y             ≡⟨ ! is—ϵ—left ⁻¹•-inverse ⟩
    x ⁻¹ • x • y   ≡⟨ assoc ⟩
    x ⁻¹ • (x • y) ≡⟨ •= refl eq ⟩
    x ⁻¹ • x       ≡⟨ ⁻¹•-inverse ⟩
    ϵ             ■
```

[2]The definition of the properties can be found in Figure 3.1

```
module Properties where
  Op₁ : Set → Set
  Op₁ A = A → A

  Op₂ : Set → Set
  Op₂ A = A → A → A

  Associative : ∀ {A} → Op₂ A → Set
  Associative _•_ = ∀ {x y z} → (x • y) • z ≡ x • (y • z)

  Commutative : ∀ {A} → Op₂ A → Set
  Commutative _•_ = ∀ {x y} → x • y ≡ y • x

  LeftIdentity : ∀ {A} → A → Op₂ A → Set
  LeftIdentity ϵ _•_ = ∀ {x} → ϵ • x ≡ x

  RightIdentity : ∀ {A} → A → Op₂ A → Set
  RightIdentity ϵ _•_ = LeftIdentity ϵ (flip _•_)

  Identity : ∀ {A} → A → Op₂ A → Set
  Identity ϵ _•_ = LeftIdentity ϵ _•_ × RightIdentity ϵ _•_

  LeftInverse : ∀ {A} → A → Op₁ A → Op₂ A → Set
  LeftInverse ϵ _⁻¹ _•_ = ∀ {x} → x ⁻¹ • x ≡ ϵ

  RightInverse : ∀ {A} → A → Op₁ A → Op₂ A → Set
  RightInverse ϵ _⁻¹ _•_ = LeftInverse ϵ _⁻¹ (flip _•_)

  Inverse : ∀ {A} → A → Op₁ A → Op₂ A → Set
  Inverse ϵ _⁻¹ _•_ = LeftInverse ϵ _⁻¹ _•_ × RightInverse ϵ _⁻¹ _•_

  Interchange : ∀ {A} → Op₂ A → Op₂ A → Set
  Interchange _•_ _⊕_ = ∀ {x y z t} → (x • y) ⊕ (z • t) ≡ (x ⊕ z) • (y ⊕ t)

  LeftCancel : ∀ {A} → Op₂ A → Set
  LeftCancel _•_ = ∀ {c x y} → c • x ≡ c • y → x ≡ y

  RightCancel : ∀ {A} → Op₂ A → Set
  RightCancel _•_ = LeftCancel (flip _•_)

  HomoMorphism : ∀ {A B} → (A → B) → Op₂ A → Op₂ B → Set
  HomoMorphism f _⊞_ _•_ = ∀ {x y} → f (x ⊞ y) ≡ f x • f y
```

Figure 3.1: Definition of Properties

Other theorems from Section 1.1 is proved in a similar way, as the theorems shown above. Furthermore it is easy to define integer exponentiation, first by defining it on the natural numbers:

$$
\begin{aligned}
&\_{}^{\wedge}_{\mathbb{N}}\_ : \mathrm{G} \rightarrow \mathbb{N} \rightarrow \mathrm{G} \\
&x \,{}^{\wedge}_{\mathbb{N}}\, \mathit{0} = \epsilon \\
&x \,{}^{\wedge}_{\mathbb{N}}\, (\mathit{suc}\; n) = x \bullet x \,{}^{\wedge}_{\mathbb{N}}\, n
\end{aligned}
$$

To finish the integer exponentiation definition, the type $\mathbb{Z}$ that is defined inductively to be either a negative number, excluding 0, or being a natural number is used. The definition of integer exponentiation is defined as follows:

$$
\begin{aligned}
&\_{}^{\wedge}_{\mathbb{Z}}\_ : \mathrm{G} \rightarrow \mathbb{Z} \rightarrow \mathrm{G} \\
&x \,{}^{\wedge}_{\mathbb{Z}}\, \text{-}[\mathit{1}+n\,] = (x^{-1}) \,{}^{\wedge}_{\mathbb{N}}\, (\mathit{1}+n) \\
&x \,{}^{\wedge}_{\mathbb{Z}}\, (+n) = x \,{}^{\wedge}_{\mathbb{N}}\, n
\end{aligned}
$$

Finally, we give a theorem that is useful for proving the ElGamal encryption scheme correct. But before that we need to note of a small lemma that multiplying with the binary operator and an element of the group is an type isomorphism.

Lemma 3.5.1

For all groups G, with binary operator $\_\bullet\_$, for all k : G the function $\phi\ x = x \bullet k$ is a type isomorphism.

Proof. The inverse is the function $\phi^{-1}\ x = x \bullet k^{-1}$, the equalities are trivial. □

This lemma is useful when one is drawing randomness from a group, i.e. calculating probabilities with a sample space that is a group. In these cases, since probabilities are preserved under type isomorphism, multiplying with an element of the group everywhere don't affect the probability. A useful theorem when proving ElGamal secure, that uses this fact is the following:

Theorem 3.5.2

For all types A and B, with group structure GA : Group A and GB : Group B, let $\_\bullet\_$ be the operator of GB, with group isomorphism ϕ : A → B, furthermore A is an explorable RA : RSpace A, for all events O : B → *Bool*, then for all m : B, the probability Pr[RA || $x \gg$ O ($\phi\ x \bullet m$)] is equal to Pr[RA || $x \gg$ O ($\phi\ x$)].

Proof. Since probabilities respects type isomorphisms, by Theorem 3.4.2, and Lemma 3.5.1, it is possible to add to x in the probability. By picking the -ϕ^{-1} m, the event sees O (ϕ (x + -ϕ^{-1} (m)) $\bullet$ m), which is equal to O (ϕ x $\bullet$ ϕ (-ϕ^{-1} (m)) $\bullet$ m), since ϕ is an group homomorphism, simplifying further one ends up with O (ϕ x) since m $^{-1}$ $\bullet$ m is the identity. □

Chapter 4

Dependent Communication

Joint work with: Nicolas Guenot and Nicolas Pouillard

The representation of communication in programs has been intensively investigated over the last few decades, in particular with the development of process algebras [Hoa85] and the π-calculus [Mil99]. This has lead to progress in the area of types for processes, in particular with the introduction of session types [Hon93] and later with their connection to linear logic [CP10, Wad14]. However, the theory of typed communicating programs is not yet as developed as the theory of functional programs, and this leads naturally to the question of how one might integrate one into the other.

Among the possible combinations of the functional and communication-centric paradigms, we are interested here in the idea of extending our type theory, with primitives for communication, as done for example in the study of session types for functional programs [GV10]. However, the system we consider integrates communication of functional data as well as channel names, building on the connection between the π-calculus and linear logic and going further than previous integrations based on a similar scheme [TCP13]. In particular, we use full dependent type theory as our host functional language. The difficulty lies in the smooth integration of the delicate linear typing of processes in a system where dependencies can appear between non-linear terms and types. The reward for such an integration is high, as it provides the means of defining communication protocols, in particular semantic security protocols, which can depend on the data transmitted, and ensure that these protocols are implemented by well-behaved programs.

The association of communicating processes and dependent types has two complementary sides: it extends the expressive power of session types as used for variants of the π-calculus, and it simplifies the task of reasoning about communicating programs. The purpose of the here is to introduce a minimal extension to type theory: it shall support enough communication to avoid extending further the language, but rather encode the types necessary to express more complex interactions.

Linear types for processes. The cornerstone of this investigation is the introduction of the multiplicative fragment of linear logic [Gir87], as presented in the sequent calculus, into Martin-Löf's type theory. In order to maintain the properties of linear typing of processes, we isolate processes from functional terms but define an interface to allow these two forms of programs to mix. Such a separation is essential, and is enforced by using two different typing judgements.

A notable difference between our system and related ones [TCP13] based on the interpretation of linear formulas as sessions [CP10, Wad14] is that the linear connectives $\parr$ and $\otimes$ are used to organise channels rather than indicate input/output behaviour. The strictly behavioural part of the types we use consists in the standard input and output types from session types [Vas12]. As a result, the processes typed in this system are those obtained from proofs through the translation described by Abramsky [Abr93], extended with special inputs and outputs for functional terms. The particular use of linear rules to type processes and are described in the preliminaries in Section 1.5 and their integration in type theory are described in Section 4.1 — note that basic knowledge of linear logic [Gir87] is assumed.

Encoding processes in AGDA. On a more practical level, one of the interests of the extension to the Martin-Löf's type theory proposed here is that it can be encoded in AGDA by introducing a new universe for *protocols* — collections of session types associated to channel names — and defining processes in the language of type theory. We describe in Section 4.2 such an encoding, and discuss the subtleties involved in the handling of linear proofs. This development also contains a proof of cut elimination in AGDA, see Section 4.3, ensuring that well-typed terms containing processes reduce correctly and without deadlock nor livelock. Furthermore, it forms a basis for future investigations into the use of processes inside type theory, or the mechanised verification of protocol specifications expressed in the language of type theory.

Representing complex protocols. Only a small fragment of linear logic is introduced into type theory here, where rules are purely multiplicative and thus contain no duplication, even in a restricted or controlled way. An interesting observation is that this, combined with the interface between intuitionistic type theory and linear logic, is enough to recover the specification of more complex behaviours, using for example controlled forms of duplication.

Limits to linearity in type theory. One should note that although the type theory we present here allows one to exploit the expressivity of dependent types in a language supporting processes and sessions, defining a dependent form of session types is a different matter entirely. Indeed, dependencies are handled at the level of intuitionistic types and we cannot specify a type *linearly depending* on a term, as it would require defining a precise meaning for such a dependency. The investigations conducted towards this end all face the difficulty of this question, and so far only limited forms of linearity have been mixed with dependent types, in the sense that the dependent product is disconnected from the linear decomposition of implication [CP02, KPB15]. Even though the modalities of such an integration are better understood now, the limits to the use of dependencies in a linear type theory have not yet been overcome.

4.1 Dependent Types with Communication

We now turn to the extendend type theory, incorporating processes into dependent type theory. This system is based on the version of Martin-Löf type theory underlying Agda [Nor07]. Our methodology is to extend the syntactic categories of terms involved with both a term level for communicating programs, using a syntax coming from the π-calculus [Mil99], and a type level of *sessions* inspired by both linear logic [Gir87] and session types [Vas12]. The resulting type theory, including processes, will be called ***PTT*** in the following.

Definition 4.1.1

The *terms, processes, types* and *sessions* of the ***PTT*** type theory are defined by t, P, A and S respectively in the following grammar:

$$\begin{array}{rcl} t,u & ::= & x \mid a \mid \lambda(x:A).t \mid t\,u \mid \langle t,u\rangle \mid \bullet \mid \pi_1\,t \mid \pi_2\,t \mid \vec{c}.P \mid A \mid S \\ P,Q & ::= & \mathbf{0} \mid P \parallel Q \mid (\nu cd)P \mid c\{\vec{e}\}\,P \mid c[\vec{e}]\,P \mid \underline{c}(x:A)\,P \mid \overline{c}\langle t\rangle\,P \mid t \,@\, \vec{c} \end{array}$$

$$\begin{array}{rcl} A,B & ::= & t \mid \Pi(x:A).B \mid \Sigma(x:A).B \mid \mathbb{1} \mid \{\!|\vec{S}|\!\} \mid \boldsymbol{Set}_i \mid \boldsymbol{Session}_i \\ S,T & ::= & t \mid ?(x:A).S \mid !(x:A).S \mid ⅋\vec{S} \mid \bigotimes\vec{S} \end{array}$$

where letters such as x, y and z denote term *variables* and a denotes a *constant*, while c, d and e denote *channels* used in processes.

A large part of this syntax is standard for type theory, but functional terms are extended with the binding construct $\vec{c}.P$ representing the process P interacting on the channels recorded in the sequence $\vec{c}$. This can be viewed as the *packaging* of a process in a functional form, where no free channel name is allowed, as this would correspond to *dangling* connections from a programming perspective, and lead to problems during reduction.

The part of the syntax intuitively representing types is also extended, with a construct $\{\!|\vec{S}\,|\!\}$ in which a sequence of sessions is mentioned. Note that order matters in this operator, since these sessions will describe the channels recorded in the sequence $\vec{c}$ in the term $\vec{c}.P$. Moreover, we add the type $\boldsymbol{Session}_i$ representing the universe of sessions: this type, just as $\boldsymbol{Set}_i$, is annotated with a *level* in order to distinguish the levels of the type hierarchy, and we will consider an additive treatment of levels.

The fragment of the ***PTT*** system dealing with functional terms and types is essentially the standard type theory from Agda [Nor07] with additional rules for the extra universes. The typing rules are shown in Figure 4.1, where $\Gamma \vdash t : A$ denotes a typing judgement associating type A to the term t in a context Γ, which is a list of typing hypotheses. The alternate judgement $\Gamma \vdash \cdot$ corresponds to the verification of well-formation for Γ, ensuring that each assumption mentions a type valid at that position in the context.

In Figure 4.1, we denote by $\mathscr{S}$ the *signature*, containing typed constants, that we leave implicit in all rules since it is specified externally. The rules for dependent products and sums involve the substitution of a term t for a variable x in a type A, denoted by $A\{t/x\}$. Moreover, the *conversion* rule allowing to consider normal forms of types is based on a relation defined externally.

Definition 4.1.2

The *conversion* relation $\simeq$ between ***PTT*** terms t and u under a given Γ, written $\Gamma \vdash t \simeq u$, is defined as the smallest congruence such that:

$$\frac{t \longrightarrow u}{\Gamma \vdash t \simeq u} \qquad \frac{\Gamma \vdash t : \Pi(x : A).B}{\Gamma \vdash t \simeq \lambda(x : A).t\ x} \qquad \frac{\Gamma \vdash t : \Sigma(x : A).B}{\Gamma \vdash t \simeq \langle \pi_1\, t, \pi_2\, t\rangle} \qquad \frac{\Gamma \vdash t : \mathbb{1}}{\Gamma \vdash t \simeq \bullet}$$

where $\longrightarrow$ denotes the reduction relation that we will define on ***PTT*** terms.

The verification of well-formedness of types, shown at the top of Figure 4.1, deals with levels of types using the notation $i \sqcup j$ for the maximum of i

$$\frac{\Gamma \vdash \cdot \quad \Gamma \vdash A : \boldsymbol{Set}_i}{\Gamma, x : A \vdash \cdot} \qquad \frac{\Gamma \vdash \cdot}{\Gamma \vdash \boldsymbol{Set}_i : \boldsymbol{Set}_{i+1}} \qquad \frac{\Gamma \vdash A : \boldsymbol{Set}_i \quad \Gamma, x : A \vdash B : \boldsymbol{Set}_j}{\Gamma \vdash \Pi(x : A).B : \boldsymbol{Set}_{i \sqcup j}}$$

$$\frac{}{\cdot \vdash \cdot} \qquad \frac{\Gamma \vdash \cdot}{\Gamma \vdash \mathbb{1} : \boldsymbol{Set}_0} \qquad \frac{\Gamma \vdash A : \boldsymbol{Set}_i \quad \Gamma, x : A \vdash B : \boldsymbol{Set}_j}{\Gamma \vdash \Sigma(x : A).B : \boldsymbol{Set}_{i \sqcup j}}$$

$$\frac{\Gamma \vdash \cdot \quad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash A \simeq B}{\Gamma \vdash t : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).t : \Pi(x : A).B}$$

$$\frac{\Gamma \vdash \cdot \quad a : A \in \mathscr{S}}{\Gamma \vdash a : A} \qquad \frac{\Gamma \vdash t : \Sigma(x : A).B}{\Gamma \vdash \pi_1\, t : A} \qquad \frac{\Gamma \vdash t : \Pi(x : A).B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B\{u/x\}}$$

$$\frac{\Gamma \vdash \cdot}{\Gamma \vdash \bullet : \mathbb{1}} \qquad \frac{\Gamma \vdash t : \Sigma(x : A).B}{\Gamma \vdash \pi_2\, t : B\{\pi_1\, t/x\}} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B\{t/x\}}{\Gamma \vdash \langle t, u \rangle : \Sigma(x : A).B}$$

Figure 4.1: Functional terms and types fragment of the ***PTT*** system

and j, and has a rule stating that the type $\boldsymbol{Session}_i$ has itself type $\boldsymbol{Set}_{i+1}$ so that this type can itself be used in the theory. In particular, one can write terms computing sessions, for which typing requires knowing the relation between the levels of ***Session*** and ***Set***.

Finally, the verification that $\{\!|\vec{S}\,|\!\}$ is a valid type relies on the rule checking that every single session S_k it contains is a valid session. In this rule, the notation $\sqcup\vec{i}$ denotes the maximum of a sequence of levels — the sequence obtained from the premises verifying each of the S_k sessions.

Processes and protocols. The typing rules for the fragment of ***PTT*** where processes, sessions and protocols — collections of sessions — appear are shown in Figure 4.2. Most rules are defining a judgement $\Gamma \vDash P :: \mathscr{J}$, where $\mathscr{J}$ is a two-level structure as defined in the previous section:

$$\mathscr{I}, \mathscr{J} ::= \cdot \mid [\mathscr{H}] \mid \mathscr{I}, \mathscr{J} \qquad\qquad \mathscr{G}, \mathscr{H} ::= \cdot \mid \mathrm{c} : S \mid \mathscr{G}, \mathscr{H}$$

considered under the equations making $\cdot$ a unit for comma, and ensuring the commutativity and associativity of commas. Note that this structure is entirely at the meta-level in the type system, and is not part of the syntax of ***PTT***.

The rules for polyadic input and (bound) output from the π-calculus, parallel composition and scoping are directly imported from the system we considered in Section 1.5. However, the treatment of duality is more subtle here, and performed through a special relation.

Definition 4.1.3

The *duality* relation $\simeq_\perp$ on sessions of ***PTT*** is the smallest symmetric relation containing the relation defined by the following rules:

$$\frac{\Gamma, x : A \vdash S \simeq_\perp T}{\Gamma \vdash ?(x:A).\,S \simeq_\perp !(x:A).\,T} \qquad \frac{(\Gamma \vdash S_k \simeq_\perp T_k)_k}{\Gamma \vdash \parr \vec{S} \simeq_\perp \bigotimes \vec{T}} \qquad \frac{\Gamma \vdash S \simeq S' \quad \Gamma \vdash S' \simeq_\perp T}{\Gamma \vdash S \simeq_\perp T}$$

Notice that this relation is invariant under conversion, by the last rule.

The core π-calculus is extended in ***PTT*** with operators for the input and output of functional data, and we write $\underline{c}(x : A)$ and $\overline{c}\langle t \rangle$ for a reception on a channel c of a value of type A named x in the continuation, and an emission of t on c, respectively. The corresponding typing rules are best compared to those of the functional language from [Wad14], inspired by [GV10], but incorporate the additional treatment of dependent types.

Beyond the interaction between functional terms and processes in data input and output, the interface between judgments $\vdash$ and $\vDash$ offers the possibility of combining these paradigms. From one to the other, *packaging* processes as terms and *unpackaging* terms to processes is trivial but for one question: that of the treatment of channel names in a protocol $\mathscr{J}$. The solution adopted here is to select an order among the free channel names when packaging, and to preserve that order in the functional layer until it is unpackaged following the same order. Notice that typing $t \mathbin{@} \vec{c}$ requires that blocks in the protocol contain a single session type, implying that compound blocks handled by cut and the $\otimes$ rule must be split within the phase where the corresponding rule appears. This allows the functional type $\{\!\!|\, \vec{S} \,|\!\!\}$ to remain a simple sequence of sessions.

Congruence and reduction. As mentioned in Section 1.5, the correspondence of reductions in a process calculus to the dynamics of cut elimination crucially relies on the identification of a number of syntactically distinct processes that are, for all intents and purposes, the same. The language of ***PTT*** is therefore equipped with an equivalence relation dealing with basic properties of $||$ as well as the exchange of unrelated prefixes. This last notion is

$$\frac{\Gamma \vdash \cdot}{\Gamma \vdash \boldsymbol{Session}_i : \boldsymbol{Set}_{i+1}} \qquad \frac{(\Gamma \vdash S_k : \boldsymbol{Session}_{i_k})_k}{\Gamma \vdash \parr \vec{S} : \boldsymbol{Session}_{\sqcup \vec{i}}} \qquad \frac{\Gamma \vdash A : \boldsymbol{Set}_i \quad \Gamma \vdash S : \boldsymbol{Session}_j}{\Gamma \vdash ?(x : A).S : \boldsymbol{Session}_{i \sqcup j}}$$

$$\frac{(\Gamma \vdash S_k : \boldsymbol{Session}_{i_k})_k}{\Gamma \vdash \{\!\!\{ \vec{S} \}\!\!\} : \boldsymbol{Set}_{\sqcup \vec{i}}} \qquad \frac{(\Gamma \vdash S_k : \boldsymbol{Session}_{i_k})_k}{\Gamma \vdash \bigotimes \vec{S} : \boldsymbol{Session}_{\sqcup \vec{i}}} \qquad \frac{\Gamma \vdash A : \boldsymbol{Set}_i \quad \Gamma \vdash S : \boldsymbol{Session}_j}{\Gamma \vdash !(x : A).S : \boldsymbol{Session}_{i \sqcup j}}$$

$$\frac{\Gamma \vDash P :: \mathscr{I},[\mathscr{G}] \quad \Gamma \vDash Q :: \mathscr{J},[\mathscr{H}]}{\Gamma \vDash P \parallel Q :: \mathscr{I},\mathscr{J},[\mathscr{G},\mathscr{H}]} \qquad \frac{\Gamma \vDash P :: \mathscr{J},\overrightarrow{[e : S]}}{\Gamma \vDash c\{\vec{e}\}\, P :: \mathscr{J},[c : \parr \vec{S}]} \qquad \frac{\Gamma \vdash \cdot}{\Gamma \vDash \mathbf{0} :: \cdot}$$

$$\frac{\Gamma \vDash P :: \mathscr{J},[c : S, d : T] \quad \Gamma \vdash T \simeq_\perp S}{\Gamma \vDash (\nu cd)\, P :: \mathscr{J}} \qquad \frac{\Gamma \vDash P :: \mathscr{J},[\overrightarrow{e : S}, \mathscr{H}]}{\Gamma \vDash c[\vec{e}]\, P :: \mathscr{J},[c : \otimes \vec{S}, \mathscr{H}]} \qquad \frac{\Gamma \vDash P :: \mathscr{J},[\mathscr{H},\mathscr{G}]}{\Gamma \vDash P :: \mathscr{J},[\mathscr{H}],[\mathscr{G}]}$$

$$\frac{\Gamma \vdash A : \boldsymbol{Set}_i \quad \Gamma, x : A \vDash P :: \mathscr{J},[c : S]}{\Gamma \vDash \underline{c}(x : A)\, P :: \mathscr{J},[c : ?(x : A).S]} \qquad \frac{\Gamma \vdash t : \{\!\!\{ \vec{S} \}\!\!\}}{\Gamma \vDash t \mathbin{@} \vec{c} :: \overrightarrow{[c : S]}}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vDash P :: \mathscr{J},[c : S\{t/x\}]}{\Gamma \vDash \overline{c}\langle t \rangle\, P :: \mathscr{J},[c : !(x : A).S]} \qquad \frac{\Gamma \vDash P :: \overrightarrow{[c : S]}}{\Gamma \vdash \vec{c}.P : \{\!\!\{ \vec{S} \}\!\!\}}$$

Figure 4.2: Process fragment of the ***PTT*** system

made precise by considering channel names in prefixes, and stating that input/output prefixes π and κ are *orthogonal*, which is denoted by $\pi \perp \kappa$, if and only if *they have no channel name in common*.

Definition 4.1.4

The equivalence $\equiv$ on terms and processes of ***PTT*** is the smallest congruence relation satisfying the equations below:

$$\begin{array}{rclrcll}
P \parallel \mathbf{0} & \equiv & P & (\nu cd)(P \parallel Q) & \equiv & (\nu cd)P \parallel Q & (c,d \notin Q) \\
P \parallel Q & \equiv & Q \parallel P & (\nu cd)\,\pi P & \equiv & \pi(\nu cd)P & (c,d \perp \pi) \\
P \parallel (Q \parallel R) & \equiv & (P \parallel Q) \parallel R & \pi\kappa P & \equiv & \kappa\pi P & (\pi \perp \kappa)
\end{array}$$

Although the equivalence considered here is close to the one used in an untyped π-calculus, it has no equation to remove unnecessary scope restrictions. Indeed, the logical correspondence of this construct to the cut rule prevents this, as a cut cannot be introduced or eliminated silently — because the definition of sessions prevents cuts on a symbol like $\cdot$ or an equivalent.

The next step is to equip the language of ***PTT*** with a reduction relation, defined as a rewrite system. The semantics given to the standard part of the type theory is as usual β-reduction, while the reductions involving communication operators correspond to the reduction of processes in the π-calculus. We have two additional reductions that treat the packaging and unpackaging of processes inside terms or of terms inside processes, but these are simple unboxing operations.

Definition 4.1.5

The reduction relation $\rightarrow$ on ***PTT*** terms and processes is the contextual closure of the following set of rewrite rules:

$$\begin{array}{rcl rcl}
(\lambda x.t)\,u & \rightarrow & t\{u/x\} & (\vec{c}.P)\,@\,\vec{e} & \rightarrow & P\{\vec{e}/\vec{c}\} \quad \textit{if } |\vec{c}| = |\vec{e}| \\
\pi_1\,\langle t,u\rangle & \rightarrow & t & (\nu c d)(c[\vec{e}]\,P \parallel d\{\vec{g}\}Q) & \rightarrow & (\nu \vec{e}\vec{g})(P \parallel Q) \\
\pi_2\,\langle t,u\rangle & \rightarrow & u & (\nu c d)(\overline{c}\langle t\rangle\,P \parallel \underline{d}(x)Q) & \rightarrow & (\nu c d)(P \parallel Q\{t/x\})
\end{array}$$

Notice that the two reduction rules for processes correspond to synchronisation with the $\parr$ and $\otimes$, or message communication. In the case of message communication the process Q will receive the message t from the process P. In both of these cases the cut will continue to reduce a new smaller cut.

We now have a complete system, and we turn to the properties it satisfies. First, we consider the stability of typing under the syntactic congruence defined to identify processes. This result is rather straightforward here, but it does not generally holds in session type systems based on linear logic [CP10, Wad14], where the grouping of outputs and parallel composition is problematic, for example, and restricts the shape of well-typed processes.

Theorem 4.1.6

For any terms t and u of ***PTT***, if $\Gamma \vdash t : A$ and $t \equiv u$ then $\Gamma \vdash u : A$.

Proof. By inspection of the equations for $\equiv$. In each case, we reorganise the typing derivation to obtain a new derivation for the equivalent term. Note that $\equiv$ is homomorphic on the functional fragment of the theory, and modifies only the parts of derivations corresponding to linear logic rule instances, which admit permutations with other independent rule instances. □

Then, we show that the system has subject reduction, so that typing is stable under reduction:

Theorem 4.1.7

For any terms t and u of ***PTT***, if $\Gamma \vdash t : A$ and $t \rightarrow u$ then $\Gamma \vdash u : A$.

Proof. By induction on the structure of the reduction from t to u, and by analysis of the shape of the typing derivation for t. In the case where the subterm reduced in t appears at top level, a case analysis shows that the reduction to u corresponds to a case from either normalisation in the type theory, or cut elimination in the linear system, based on the sequent calculus. □

More details on the normalisation result, in particular concerning the linear system where cut elimination corresponds to reduction, can be found in the next section, described from the viewpoint of the AGDA formalisation of ***PTT***.

4.2 Embedding in Type Theory

The type theory presented in the previous section is based on the idea that a layer of processes can be introduced into a type theory *à la* Martin-Löf, thus allowing a calculus modelling concurrent programming to use dependent types. We now consider the realisation of this idea on a more practical level, in a dependently typed programming language: a part of the ***PTT*** language has been implemented in AGDA [Nor07], using a partial *shallow embedding* where the functional layer of ***PTT*** is directly translated as the corresponding part of the AGDA language, on the other hand the concurrency part into a *deep embedding* with a corresponding data type.

Embedding the process layer in a functional language requires to define a representation that supports the properties expected from processes, but we have designed rules exploiting a rather general syntax to allow for more flexibility. In the AGDA representation, we simplify the theory by considering binary forms of $\parr$ and $\otimes$, and replace the cut and $\otimes$ rules by the branching variants:

$$\frac{\Gamma \vDash P :: \mathscr{I},[c:S] \quad \Gamma \vDash Q :: \mathscr{J},[d:T] \quad T \simeq_{\perp} S}{\Gamma \vDash (\nu cd)(P \parallel Q) :: \mathscr{I},\mathscr{J}}$$

$$\frac{\Gamma \vDash P :: \mathscr{J},[d:S] \quad \Gamma \vDash Q :: \mathscr{J},[e:T]}{\Gamma \vDash c[de](P \parallel Q) :: \mathscr{I},\mathscr{J},[c:S \otimes T]}$$

Moreover, we drop the *pure mix* rule and retain only, beyond the two rules above, the binary $\parr$ rule, the axiom typing **0**, the mix rule typing $\parallel$ by branching and the rules for data input and output as well as the interfaces to the functional layer. The resulting system is equivalent to the original one on a logical level, except for the units. Indeed, n-ary connectives can be recovered in all cases where $n \geqslant 1$ and the pure mix rule was used because of the congruence only.

Data input and output. Interestingly, the rules introducing dependencies in a session type can be interpreted in a straightforward way inside pure type theory, building on the similarity of ? and ! to Π and Σ respectively. A product $\Pi(x:A).B$ can indeed interact with a sum $\Sigma(x:A).C$ in a way similar to input and output: the sum provides a proof of A and B assumes the existence of a proof x of A. These constructions are thus simple to handle in the AGDA representation.

Cut elimination in linear logic. The most important property of the ***PTT*** system is that any term can be reduced to a normal form. However, this requires the normalisation of terms containing processes and therefore to showing that the linear typing implies that communication can be properly executed in processes. This amounts to a proof of cut elimination in the multiplicative fragment of linear logic extended with mix, message communication, and accommodating the rules interfacing functional terms and processes. We discuss now the proof of cut admissibility used in the AGDA representation.

The proof proceeds by induction on the cut formula, and relies on *splitting lemmas*, in which a continuation term is inserted in a process to modify the subprocess involving the channel on which cut was performed. There is a splitting lemma for each possible shape of the session, but we show here only the case of $\otimes$:

$$\text{(splitting lemma for } \otimes\text{)}\quad \dfrac{\vDash P :: \mathscr{I},[c : S \otimes T] \qquad \begin{array}{c} \vDash Q_0 :: \mathscr{I}_0,[d : S] \quad \vDash Q_1 :: \mathscr{I}_1,[e : T] \\ \vdots \\ \vDash R :: \mathscr{I}_0,\mathscr{I}_1,\mathscr{J} \end{array}}{\vDash P\{R /\!\!/ c[de](Q_0 \parallel Q_1)\} :: \mathscr{I},\mathscr{J}}$$

where the notation indicates that we modify some process P using a second process R *parametric* in the structures $\mathscr{I}_0$ and $\mathscr{I}_1$. The key idea is that inside P, there exists a subprocess of the shape $c[de](Q_0 \parallel Q_1)$ corresponding to the decomposition of $[c : S \otimes T]$. The notation $\{\cdot /\!\!/ \cdot\}$, in the style of a substitution, denotes the replacement of this output process by R where Q_0 and Q_1 are plugged. This splitting rule commutes upwards with all other rules until the adequate $\otimes$ rule is met — and this rule is unique by linearity.

The splitting lemma for $\parr$, the dual of $\otimes$, is expressed by the following scheme, where R is parametric in a single structure $\mathscr{I}_0$ and contains only one open premise Q_0:

$$\text{(splitting lemma for } \parr\text{)}\quad \dfrac{\vDash P :: \mathscr{I},[c : S \parr T] \qquad \begin{array}{c} \vDash Q_0 :: \mathscr{I}_0,[d : S],[e : T] \\ \vdots \\ \vDash R :: \mathscr{I}_0,\mathscr{J} \end{array}}{\vDash P\{R /\!\!/ c\{de\}Q\} :: \mathscr{I},\mathscr{J}}$$

and the combination of these two lemmas yields the case of cut reduction involving a formula $S \otimes T$ and its dual. Notice that no functional context Γ is mentioned here, since by the time normalisation requires eliminating a cut, this context is empty — as a consequence of reduction not being performed under abstractions in type theory. Consider a cut on $\otimes$ and $\parr$:

$$\dfrac{\vDash P :: \mathscr{I},[c : S \otimes T] \quad \vDash P' :: \mathscr{J},[d : S^\perp \parr T^\perp]}{\vDash (\nu cd)(P \parallel P') :: \mathscr{I},\mathscr{J}}$$

that we rewrite into two cuts, on the two subformulas of these connectives, where the rightmost premise is an unknown process Q' typed using an unknown structure $\mathscr{J}'$ that we introduce in the conclusion:

$$\dfrac{\vDash Q_0 :: \mathscr{I}_0,[e : S] \qquad \dfrac{\vDash Q_1 :: \mathscr{I}_1,[f : T] \quad \vDash Q' :: \mathscr{J}',[g : S^\perp],[h : T^\perp]}{\vDash (\nu fh)(Q_1 \parallel Q') :: \mathscr{I}_1,\mathscr{J}',[g : S^\perp]}}{\vDash R \triangleq (\nu eg)(Q_0 \parallel (\nu fh)(Q_1 \parallel Q')) :: \mathscr{I}_0,\mathscr{I}_1,\mathscr{J}'}$$

to obtain a process that we name R, parametric in the unknowns Q' and $\mathscr{J}'$. We can now use this R to produce the expected proof, using a combination of the splitting lemmas for $\otimes$ and $⅋$:

$$\dfrac{\vDash P :: \mathscr{I}, [c : S \otimes T] \qquad \dfrac{\vDash P' :: \mathscr{J}, [d : S^{\perp} ⅋ T^{\perp}] \quad \vDash R :: \mathscr{I}_0, \mathscr{I}_1, \mathscr{J}'}{\vDash P'\{R \mathbin{/\!/} d\{gh\}Q'\} :: \mathscr{I}_0, \mathscr{I}_1, \mathscr{J}}}{\vDash P\{P'\{R \mathbin{/\!/} d\{gh\}Q'\} \mathbin{/\!/} c[ef](Q_0 \parallel Q_1)\} :: \mathscr{I}, \mathscr{J}}$$

This procedure can be applied the other rules, and we can therefore prove that all cuts can be eliminated from a typing derivation of ***PTT***. As a consequence, functional reductions blocked by some process constructs can be performed once the process has been reduced. Note that proving normalisation in ***PTT*** requires proving normalisation in the functional layer and cut elimination in the process layer simultaneously.

Forwarders. The process layer of the ***PTT*** system has no correspondent to the identity axiom, which was interpreted as a forwarder in Section 1.5, following [CP10]. However, we can express this process as follows:

$$\begin{array}{lcl}
\mathit{fwd} : \Pi\,(\mathrm{S} : \mathrm{Session}).\,\{\!\mid \mathrm{S}; \mathrm{S}^{\perp} \mid\!\} & & \\
\mathit{fwd}\,(?\,(x{:}A).\,\mathrm{S}\,x) & = & i\,o.\ \underline{i}(x : \mathrm{A})\ \overline{o}\langle x\rangle\ \mathit{fwd}\,(\mathrm{S}\,x)\ @\ (i;o) \\
\mathit{fwd}\,(!\,(x{:}A).\,\mathrm{S}\,x) & = & o\,i.\ \underline{i}(x : \mathrm{A})\ \overline{o}\langle x\rangle\ \mathit{fwd}\,(\mathrm{S}\,x)\ @\ (o;i) \\
\mathit{fwd}\,(\mathrm{S}_0 ⅋ \mathrm{S}_1) & = & i\,o.\ i\,\{i_0,i_1\}\ o[o_0,o_1](\mathit{fwd}\,\mathrm{S}_0\ @\ (i_0;o_0) \mid \mathit{fwd}\,\mathrm{S}_1\ @\ (i_1;o_1)) \\
\mathit{fwd}\,(\mathrm{S}_0 \otimes \mathrm{S}_1) & = & o\,i.\ i\,\{i_0,i_1\}\ o[o_0,o_1](\mathit{fwd}\,\mathrm{S}_0\ @\ (o_0;i_0) \mid \mathit{fwd}\,\mathrm{S}_1\ @\ (o_1;i_1))
\end{array}$$

By doing induction on the Session we can find a process that will perform the communication necessary. This communication will always input messages before sending them to the other channel, similarly it will always break a $⅋$ before a $\otimes$ and then continue in the two independent branches.

4.3 The Model

The result in the previous section was proven by giving a model inside AGDA. This model is defined by introducing a type for the session types and an indexed type for the process terms. The type of the sessions are defined by the following type in AGDA:

```
data Com : Set where IN OUT : Com

data Session : Set where
  act : Com → {M : Set} (P : M → Session) → Session
  _⅋_ _⊗_ : Session → Session → Session
  𝟙' ⊥' : Session
```

The *act* constructor, represents either a $?(x : M).\,P$ or a $!(x : M).\,P$ depending on the value of the Com field. The remainder of the protocol is described by the function P, which is allowed to inspect the value that is sent between the two parties. The other constructors are rather self explanatory of what formula they corresponds to. Notice that since we are embedding directly into AGDA we get for free the possibility of computing the session.

The duality relation is defined as an indexed type, once again we get the conversion rule for free since we are embedding into the type theory of AGDA.

```
data DualC : Com → Com → Set₁ where
  ?! : DualC IN OUT
  !? : DualC OUT IN

data Dual : (P Q : Session) → Set₁ where
  𝟙⊥ : Dual 𝟙’ ⊥’
  ⊥𝟙 : Dual ⊥’ 𝟙’
  act : ∀ {C C’} {M : Set} {F G : M → Session}
    → DualC C C’
    → (∀ m → Dual (F m) (G m))
    → (∀ m → Dual (G m) (F m))
    → Dual (act (com C F)) (act (com C’ G))
  ⊗⅋ : ∀ {A A’ B B’}
    → Dual A A’ → Dual A’ A
    → Dual B B’ → Dual B’ B
    → Dual (A ⊗ B) (A’ ⅋ B’)
  ⅋⊗ : ∀ {A A’ B B’}
    → Dual A A’ → Dual A’ A
    → Dual B B’ → Dual B’ B
    → Dual (A ⅋ B) (A’ ⊗ B’)
```

There might seem to be some redundancy in the definition of the duality, but this is to make induction simpler. When splitting we are not only getting that the substructures are in dual relation, but we get the symmetric version as well. Another way of defining this would be to use sized types[Abe06], prove that the duality relation is symmetric and that the symmetric version is of the same size.

Theorem 4.3.1

The duality relation Dual is symmetric, i.e. for all sessions: P and Q such that *d* : Dual P Q there exists a term of type Dual Q P.

Proof. By inversion on d, it is immediate by re-arranging the sub-terms. □

Before we type processes, we need to explain how we will represent the linear context. Since it is nested we can't just use a list of channels and Session's immediately. We could just have contexts be List (List (URI × Session)), where URI is the type we give to channels, but we give a special purpose type for this, and do this for two reasons. One is to document our intention, the second is to index the type over which channels that will occur in the context. On the inner level we define the type Dom as the list of channels that will appear, morally this is just a snoc-list of URI.

```
data Dom : Set where
  ε : Dom
  _,_↦* : (δ : Dom) (c : URI) → Dom
```

The actual inner context will be indexed over a Dom for which channels that occur in the context. Once again we work with snoc-list since this is more natural when describing contexts. The definition here is parametric over what will be stored, as a context it will be used with Session.

```
data Map {a} (A : Set a) : Dom → Set a where
  ε : Map A ε
  _,_↦_ : ∀ {δ} (E : Map A δ) c (v : A) → Map A (δ , c ↦*)
```

The membership predicate, i.e. the relation that relates a particular channel with a value, is split up in two steps. The first is where in the Dom we can find the channel, which looks like the normal snoc-list membership predicate:

```
data _∈D_ (c : URI) : Dom → Set where
  here : ∀ {δ} → c ∈D (δ , c ↦*)
  there : ∀ {δ d} (p : c ∈D δ) → c ∈D (δ , d ↦*)
```

The second part, is what value is associated with the particular channel. This will be computed by a lookup function _!!_, this also works in a similar way to ordinary lists.

```
infix 7 _!!_
_!!_ : ∀ {a} {A : Set a} {c δ} → Map A δ → c ∈D δ → A
(M , c ↦ v)  !! here       = v
(M , c₁ ↦ v) !! (there l)  = M !! l
```

By combining these two we get the membership predicate. This split of having an inductive type for finding the channel, and a computational interpretation to find the value will make certain, is useful when doing proofs. This makes it possible to delay the unification of indexes, which sometimes is a major source of problems.

```
record _↦_∈_ {a} {A : Set a} (d : URI)
                    (S : A) {δ} (M : Map A δ) : Set a where
  constructor ⟨_,_⟩
  field
    lA : d ∈D δ
    ↦A : M !! lA ≡ S
module ↦∈ = _↦_∈_   -- shorter name when using projections
```

That was the inner context, the outer context will follow a similar approach, just one level up. We once again split the information about where we can find the particular Map and the actual Map. The type Doms is a snoclist of Dom, i.e. a list of list of channels.

```
infixl 3 _,[_]
data Doms : Set where
  · : Doms
  _,[_] : Doms → Dom → Doms
```

And the contents will be stored in Maps which will store a Map inside. This type is also polymorphic what is stored in the inner Map's.

```
data Maps {a} (A : Set a) : Doms → Set a where
  · : Maps A ·
  _,[_] : ∀ {δs δ} (I : Maps A δs) (Δ : Map A δ) → Maps A (δs ,[ δ ])
```

When using this type as a context we will use the type Proto which will be indexed over the information about which channels are in the context. Our particular encoding is going not remove old channels, instead it will mark them as ended. This is achieved by using a particular kind of option type MSession which is either an active Session or an ended one. Furthermore we use Env as a name for tensor blocks.

```
data MSession : Set where
  ≪_≫ : (S : Session) → MSession
  end : MSession

Env Proto : Doms → Set where
Env   = Map  MSession
Proto = Maps MSession
```

As before we split the membership predicate into an inductive definition where one can find a particular snoc-list of channels.

```
infix 3 [_]∈D_
data [_]∈D_ (δ : Dom) : Doms → Set where
  here : ∀ {δs} → [ δ ]∈D δs ,[ δ ]
  there : ∀ {δs δ'} → [ δ ]∈D δs → [ δ ]∈D δs ,[ δ' ]
```

And a computational *lookup* function that computes what Map one would find at that particular place.

```
lookup : ∀ {a δs δ} {A : Set a} → Maps A δs → [ δ ]∈D δs → Map A δ
lookup (M ,[ Δ ]) here = Δ
lookup (M ,[ Δ ]) (there l) = lookup M l
```

Both of which are combined to find be that a particular Δ : Map can be found in the outer context M : Maps.

```
record [_]∈_ {a} {A : Set a} {δ} (Δ : Map A δ)
                {δs} (M : Maps A δs) : Set a where
  constructor ⟨_,_⟩
  field
    lΔ : [ δ ]∈D δs
    ↦Δ : lookup M lΔ ≡ Δ
module []∈ = [_]∈_
```

So from then outer level if we wish to prove that a particular Session is in a particular Proto one needs to use two membership predicates. One for the outer level, and one for the inner level. It is therefore useful to combine these, basically doing a composition of the two relations, which is shown in Figure 4.3.

```
infix 0 [_↦_...]∈_
record [_↦_...]∈_ {δs} (c : URI)
  (S : Session) (I : Proto δs) : Set
  where
  constructor mk
  field
    {δE} : Dom
    {E}  : Map MSession δE
    lI   : [ E ]∈ I
    lE   : c Env.↦ ≪ S ≫ ∈ E
  open [_]∈_ lI public
  open _↦_∈_ lE public
  E/ : Map MSession δE
  E/ = E Env./' lE
module [↦...]∈ = [_↦_...]∈_
```

```
infix 0 [_↦_]∈_
record [_↦_]∈_ {δs} (c : URI)
  (S : Session) (I : Proto δs) : Set
  where
  constructor mk
  field
    l... : [ c ↦ S ...]∈ I
  open [↦...]∈ l... public
  field
    E/c : Env.Ended E/
module [↦]∈ = [_↦_]∈_
```

Figure 4.3: Channel membership in Proto, and a version for which the channel is the only active in the tensor block.

The reason for making Map and Maps polymorphic, is so that we can reuse the structure to describe how to split the context. As mentioned before we are using MSession, so some channels are in the context but not active. This makes splitting easier, both branches will still use the same Doms but every active channel will only be active in one of the branches. So by instead of picking MSession if we pick *Bool* we get a structure that describes a split.

```
Selections : Doms → Set
Selections = Maps Bool
```

To actually perform the split, we use a *zipWith* function that, which will preserve all channels. This operation becomes simple due to the fact that we made both Maps and Map indexed over the channels. So *zipWiths* can be given a descriptive enough type to only zip when the two Maps have the same structure. Furthermore we define Map.zipWith that works on the inner level.

$$\begin{aligned}
&\mathit{zipWith} : \forall \{\mathrm{A}\ \mathrm{B}\ \mathrm{C}\ \delta s\}\ (f : \forall \{\delta\} \to \mathrm{Map}\ \mathrm{A}\ \delta \to \mathrm{Map}\ \mathrm{B}\ \delta \to \mathrm{Map}\ \mathrm{C}\ \delta)\\
&\qquad\qquad \to \mathrm{Maps}\ \mathrm{A}\ \delta s \to \mathrm{Maps}\ \mathrm{B}\ \delta s \to \mathrm{Maps}\ \mathrm{C}\ \delta s\\
&\mathit{zipWith}\ f \cdot \qquad \cdot \qquad = \cdot\\
&\mathit{zipWith}\ f\ (\mathrm{I}\ ,[\ \Delta\ ])\ (\sigma s\ ,[\ \sigma\ ]) = \mathit{zipWith}\ f\ \mathrm{I}\ \sigma s\ ,[\ f\ \Delta\ \sigma\ ]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Map.zipWith} : \forall \{\mathrm{A}\ \mathrm{B}\ \mathrm{C}\ \delta\}\ (f : \mathrm{A} \to \mathrm{B} \to \mathrm{C})\\
&\qquad\qquad \to \mathrm{Map}\ \mathrm{A}\ \delta \to \mathrm{Map}\ \mathrm{B}\ \delta \to \mathrm{Map}\ \mathrm{C}\ \delta\\
&\mathrm{Map.zipWith}\ f\ \epsilon\ \epsilon = \epsilon\\
&\mathrm{Map.zipWith}\ f\ (\Delta\ ,\ c \mapsto v_0)\ (\sigma\ ,\ .c \mapsto v_1) = \mathrm{Map.zipWith}\ f\ \Delta\ \sigma\ ,\ c \mapsto f\ v_0\ v_1
\end{aligned}$$

Performing a particular selection is a special case of *zipWith*. Now a selection gives two Maps, one for each branch. We select which of the two we want with the function *_[]/[_]_* which takes a boolean as the second argument, which represents which of the two cases that is desired.

$$\begin{aligned}
&\mathit{selectProj} : \mathit{Bool} \to (\mathrm{MSession} \to (\mathit{Bool} \to \mathrm{MSession}))\\
&\mathit{selectProj}\ 0\ {}_2\ v = [0{:}\ v\ 1\ :\ \mathit{end}\ ]\\
&\mathit{selectProj}\ 1\ {}_2\ v = [0{:}\ \mathit{end}\ 1\ :\ v\ ]
\end{aligned}$$

$$\begin{aligned}
&\_\mathit{Map./[\_]\_} : \forall \{\delta\}\ (\Delta : \mathrm{Env}\ \delta)\ (b : \mathit{Bool})\ (\sigma : \mathrm{Selection}\ \delta) \to \mathrm{Env}\ \delta\\
&\Delta\ \mathrm{Map./[}\ b\ ]\ \sigma = \mathrm{Map.zipWith}\ (\mathit{selectProj}\ b)\ \Delta\ \sigma
\end{aligned}$$

$$\begin{aligned}
&\_\mathit{sel[\_]\_} : \forall \{\delta s\}\ (\mathrm{I} : \mathrm{Proto}\ \delta s)\ (b : \mathit{Bool})\ (\sigma s : \mathrm{Selections}\ \delta s) \to \mathrm{Proto}\ \delta s\\
&\mathrm{I}\ \mathit{sel[}\ b\ ]\ \sigma s = \mathit{zipWith}\ (\lambda\ \mathrm{E}\ \sigma \to \mathrm{E}\ \mathrm{Map./[}\ b\ ]\ \sigma)\ \mathrm{I}\ \sigma s
\end{aligned}$$

We can furthermore introduce a few more operations on contexts, in particular we have the are using the operations I / *l* and I /... *l* which are deactivating all sessions in the same tensor block, in I, as session which the membership predicate *l* is living in. The difference is that _/_ is for membership proofs that have a unique active session, whereas _/..._ have no such requirement.

One can also perform substitution, which will be used when we are updating a channel in a continuation. The substitution [I / *l*]:= S will replace what the membership predicate *l* points to in I with S.

The final thing we need before we can give the process derivations are a predicate that a particular Selections is only splitting at most one tensor block. In fact it is easier to make a predicate that at most n tensor blocks have been split where n is a natural number. We use the relation SelAtMost n E σ m to tell if σ is splitting E, in which case $m = n + 1$, and if σ does not split E, i.e. σ sends everything to the same side, then $m = n$.

```
data SelAtMost (n : ℕ) {δ : Dom} (E : Env δ) (σ : Sel δ) : ℕ → Set where
  01 : ∀ b → EnvSelectionAll≡ b E σ → SelAtMost n E σ n
  m  : SelAtMost n E σ (suc n)
data AtMost : ℕ → ∀ {δs} → Proto δs → Selections δs → Set where
  · : ∀ {n} → AtMost n · ·
  _,[_] : ∀ {n m δ δs} {E : Env δ} {I : Proto δs} {σs σ}
    → AtMost n I σs → SelAtMost n E σ m → AtMost m (I ,[ E ]) (σs ,[ σ ])
```

With all the machinery set up we can define well typed processes, i.e. we are using a intrinsic typing approach, is given in Figure 4.4. The type TC'⟨_⟩, represents cut free derivations indexed by a type Proto which is the current protocol that the process is following. Each constructor of this type is one action the process can perform. Most of these are going to have continuations for what to do after the action have been performed.

The way the multiplicative connectives are modelled is deactivate all channels and creating new tensor blocks at the end. The reason for this is that performing a complicated substitution were one would replace a channel with potentially several tensor blocks is a complicated one. Furthermore notice that the rule TC-⊗-out is also splitting the context, so it have two parallel continuations. The splitting is performed by the σs : Selections, and since we are splitting with the rule we don't allow any further splitting inside tensor blocks by the AtMost predicate, and setting the allowed number of splits to 0.

Notice that we are reusing the type theory in for example the rule TC-?-inp where we are putting the continuation under a function in the type theory. This gives an embedding such that we don't have to worry about boxing the channels when moving between the type theory part and the process layers.

The final two actions are TC-end which is the process that has finished, which requires that all sessions are not active, this check is performed by the predicate Ended. The other action is TC-split which is (potentially[1]) performing a splitting on a tensor block. This action is also using a Selections and the AtMost predicate, but this time we allow 1 split.

[1]It is possible to use a $_m$ constructor but still send everything to the same side

```
data TC'⟨_⟩ {δI} (I : Proto δI) : Set₁ where
  TC-⊗-out : ∀ {c S₀ S₁}
    (l : [ c ↦ S₀ ⊗ S₁ ]∈ I)
    (σs : Selections δI)
    (A0 : AtMost 0 (I / l) σs)
    (P₀ : ∀ c₀ → TC'⟨ I / l sel[ 0 ₂ ] σs ,[ c₀ ↦ S₀ ] ⟩)
    (P₁ : ∀ c₁ → TC'⟨ I / l sel[ 1 ₂ ] σs ,[ c₁ ↦ S₁ ] ⟩)
    → TC'⟨ I ⟩
  TC-⅋-inp : ∀ {c S₀ S₁}
    (l : [ c ↦ S₀ ⅋ S₁ ]∈ I)
    (P : ∀ c₀ c₁ → TC'⟨ I / l ,[ c₀ ↦ S₀ ] ,[ c₁ ↦ S₁ ] ⟩)
    → TC'⟨ I ⟩
  TC-𝟙-out : ∀ {c}
    (l : [ c ↦ 𝟙' ...]∈ I)
    (P : TC'⟨ I /... l ⟩)
    → TC'⟨ I ⟩
  TC-⊥-inp : ∀ {c}
    (l : [ c ↦ ⊥' ]∈ I)
    (P : TC'⟨ I / l ⟩)
    → TC'⟨ I ⟩
  TC-?-inp : ∀ {c A S₁}
    (l : [ c ↦ act IN {A} S₁ ]∈ I)
    (P : (m : A) → TC'⟨ [ I / l ]:= ≪ S₁ m ≫ ⟩)
    → TC'⟨ I ⟩
  TC-!-out : ∀ {c A S₁}
    (l : [ c ↦ act OUT {A} S₁ ]∈ I)
    (m : A) (P : TC'⟨ [ I / l ]:= ≪ S₁ m ≫ ⟩)
    → TC'⟨ I ⟩
  TC-end : ∀ (E : Ended I) → TC'⟨ I ⟩
  TC-split :
    (σs : Selections δI)
    (A1 : AtMost 1 I σs)
    (P₀ : TC'⟨ I sel[ 0 ₂ ] σs ⟩)
    (P₁ : TC'⟨ I sel[ 1 ₂ ] σs ⟩)
    → TC'⟨ I ⟩
```

Figure 4.4: Agda model of ***PTT***

The type Proto, does not have the proper definitional equalities as we want, and therefore we need to once again use a setoid construction. We first define a relation for equivalent inner contexts, named _∼_, which represents the equality. Notice that we don't reqire the Dom for the two sides to be equal, this because non-active channels can be removed with this representation.

```
infix 0 _∼_
data _∼_ : ∀ {δE δF} (E : Env δE) (F : Env δF) → Set₁ where
  ∼-refl : ∀ {δE} {E : Env δE}
    → E ∼ E
  ∼-trans : ∀ {δE δF δG} {E : Env δE} {F : Env δF} {G : Env δG}
    → E ∼ F → F ∼ G → E ∼ G
  ∼,↦ : ∀ {δE δF} {E : Env δE} {F : Env δF} {c S}
    → E ∼ F → E , c ↦ S ∼ F , c ↦ S
  ∼,↦end : ∀ {δE} {E : Env δE} {c}
    → E , c ↦ end ∼ E
  ∼,↦end' : ∀ {δE} {E : Env δE} {c}
    → E ∼ E , c ↦ end
  ∼,[swap] : ∀ {δE c d A B} {E : Env δE}
    → E , c ↦ A , d ↦ B ∼ E , d ↦ B , c ↦ A
```

In order to simplify proofs using this equality we don't have a constructor for symmetry directly, which complicates induction. Instead we have an external proof that the given relation is indeed symmetric. In order to be able to prove this, we need to be able to push down the symmetry, and all the leafs need to have the symmetric version as well, which is the reason we have the symmetric version of ∼-↦*end*.

Lemma 4.3.2

The relation _∼_ is symmetric.

Proof. Follows immediately by induction, notice that ∼-*refl* and ∼-*[swap]* are self symmetric. □

For the outer level, i.e. equality on Proto, we also define an external equivalence relation, named _≈_. This relation is very similar to _∼_ with the exception that equality of the inner context is _∼_ instead of the equality of the type theory.

```
infix 0 _≈_
data _≈_ : ∀ {δI δJ} (I : Proto δI) (J : Proto δJ) → Set₁ where
  ≈-refl : ∀ {δI} {I : Proto δI}
    → I ≈ I
  ≈-trans : ∀ {δI δJ δK} {I : Proto δI} {J : Proto δJ} {K : Proto δK}
    → I ≈ J → J ≈ K → I ≈ K
  ≈,[] : ∀ {δE δF δI δJ} {E : Env δE} {F : Env δF}
           {I : Proto δI} {J : Proto δJ}
    → I ≈ J → E ∼ F → I ,[ E ] ≈ J ,[ F ]
  ≈,[ϵ] : ∀ {δI} {I : Proto δI}
    → I ,[ ϵ ] ≈ I
  ≈,[ϵ]! : ∀ {δI} {I : Proto δI}
    → I ≈ I ,[ ϵ ]
  ≈,[swap] : ∀ {δE δF δI} {I : Proto δI} {E : Env δE} {F : Env δF}
    → I ,[ E ] ,[ F ] ≈ I ,[ F ] ,[ E ]
```

Lemma 4.3.3

The relation _≈_ is symmetric.

Proof. Follows by induction, and that _∼_ is symmetric 4.3.2. □

To use the definitional equality we use TC-conv which transports a process of some protocol to an equivalent one. This transformation is homomorphic in the sense that it will map every action to an equivalent action.

Theorem 4.3.4

For all Proto, I and J such that eq : I ≈ J, and process p : TC'⟨ I ⟩ there exists a process TC-conv eq p : TC'⟨ J ⟩.

Proof. By induction on the process p, use the equality eq to map all membership proofs. □

Cut Elimination. Currently in syntax of the actions for processes there is no action for cut. Rather cut will be implemented in the type theory as a computation that will perform the inner communication. As such there will be a function of the following type:

$$\begin{aligned}
&\text{TC-cut} : \forall \{c_0\, c_1\, \mathrm{S}_0\, \mathrm{S}_1\, \delta_0\, \delta_1\}\, \{\mathrm{I}_0 : \text{Proto}\, \delta_0\}\, \{\mathrm{I}_1 : \text{Proto}\, \delta_1\} \\
&\quad (\mathrm{D} : \text{Dual}\, \mathrm{S}_0\, \mathrm{S}_1) \\
&\quad (l_0 : [\, c_0 \mapsto \mathrm{S}_0\,]\!\in \mathrm{I}_0)\, (l_1 : [\, c_1 \mapsto \mathrm{S}_1\,]\!\in \mathrm{I}_1) \\
&\quad (\mathrm{P}_0 : \text{TC'}\langle\, \mathrm{I}_0\, \rangle)\, (\mathrm{P}_1 : \text{TC'}\langle\, \mathrm{I}_1\, \rangle) \\
&\quad \to \text{TC'}\langle\, (\mathrm{I}_0 \,/\, l_0)\, \blacklozenge\mathit{Proto}\, (\mathrm{I}_1 \,/\, l_1)\, \rangle
\end{aligned}$$

Where $\blacklozenge$*Proto* is the append function for two Proto. So the type of TC-cut takes two processes P_0 and P_1, that contains sessions S_0 and S_1 respectively, and these sessions are dual to each other. The result is the concatenation of the two protocols, where we have removed the cut session in both cases. The TC-cut function works by induction on the Dual predicate, this will tell us what is the cut formula. In each case we can perform commuting conversions using the splitting lemma, until we reach the principal case. These splitting lemma takes a bunch of continuations on what do when the principal case have been found. Which of the continuation to use depends on what the split formula is.

Part III

CryptoAgda

Chapter 5

Semantic Security in Type Theory

Joint work with: Nicolas Pouillard

In this chapter we use the type theoretical definitions defined in the previous chapters, to create CRYPTOAGDA[1]. This is a library in AGDA for reasoning about semantic security, as presented in Section 1.4, about various cryptographic constructions. Although this library have been implemented in AGDA, we stress that the constructions could be implemented in other type theories. Furthermore since we are using the communicating version of type theory presented in Chapter 4 the programs presented here, are not valid AGDA, but instead they are presented in a version of AGDA with communication primitives. It is important to note that all the formalised proofs in AGDA of this chapter uses the model for the proofs, since no implementation of this version of AGDA exists. This makes it easier to understand the proofs, since the details of the model is hidden, but all the proofs are still type checked by AGDA.

5.1 Definitions

We begin by combining the formalisations from the previous chapters, to formally define semantic security. This will use the definitions of probabilistic processes, for example the challenger and adversary will be represented by such processes, and defining and reasoning about advantage makes use of probabilistic reasoning. Furthermore the interactions between the challenger and the adversary is dictated by a protocol, defined by a dependent session type. The communication in the game will be represented by communication

[1]The url is: `https://github.com/crypto-agda/`

by the processes. This leads to the following formal definition of a Game that we will use in CRYPTOAGDA.

Definition 5.1.1: Crypto-Game

A Crypto-Game in CRYPTOAGDA is defined to be:

- A protocol, P, describing the behaviour interaction between the adversary and its environment, which includes the interaction with the challenger, but also possible oracles.
- A type, R_η, for the sample space for the environment.
- An implementation, i.e. a process, of the probabilistic environment, that is parametrised by a Boolean. In CRYPTOAGDA:

```
record Crypto-Game : Set where
  constructor mk
  field
    P : Protocol
    {Rη} : Set
    Rη-fin : RSpace Rη
    η : Bool → Rη → ⟦ P ⟧
```

The adversary is a probabilistic function communicating with the challenger of the Crypto-Game. We use the ability that processes can communicate over multiple channels, in order to describe the powers of the adversary. By making the final guess on a different channel, we get a simpler explanation of the adversary, since we don't have to change the protocol of the challenger. Furthermore by having a separate channel for the result, gives us a convenient definition of advantage, and also the possibility of making reductions easier.

Definition 5.1.2: Adversary for Crypto-Game

An adversary for a Crypto-Game in CRYPTOAGDA is defined to be:

- A type, RA, for the sample space.

- A probabilistic process over the sample space RA that follows two protocols: the first is $\mathrm{P}^{\perp}$ and the second one is the trivial protocol of just outputting a boolean, $!(x : \mathit{Bool}).\perp$. In CRYPTOAGDA:

```
record Adversary (G : Crypto-Game) : Set where
  constructor mk
  field
    {RA} : Set
    RA-fin : RSpace RA
    Adv : RA → ⦃P⊥; ! (x : Bool) . ⊥⦄
```

By connecting an environment with an adversary, i.e. by utilising a cut, the result will be a process of type $!\ (x : \mathit{Bool}) . \perp$. Such a process is extensionally equal to $\bar{c}\langle b\rangle\ \mathbf{0}$, where c is the channel for the protocol, and b is a Boolean value. It is therefore possible to extract the value, i.e. b, from the process. We will use the function *run* : {A : Set} → ⦃! (_ : A) . ⊥⦄ → A, to extract such a value. This is used to define the notion of advantage, by extracting the value of the final Boolean after cutting the environment and the adversary. Since the processes are probabilistic, the result is a probability.

Definition 5.1.3: Advantage

The advantage of an adversary for a Crypto-Game is defined to be:

```
Advantage : (G : Crypto-Game) → Adversary G → ℚ⁺
Advantage (mk P Rη-f η) (mk RA-f A)
  = distℚ Pr[ Bool-f ×f Rη-f ×f RA-f || (b , r_η , ra)
               ≫ run (TC-cut (η b r_η) (A ra) == b) ]
          1/2
```

Here we compute the probability of the fact that after performing the cut between the environment $\eta\ b$ and the adversary A, the resulting process will output the answer b. The distance between this probability and $\frac{1}{2}$ is computed by the *dist*$\mathbb{Q}$ function. Notice that I have hidden the channels used for the communication.

So far we have not seen the security parameter, all the definitions above have used an implicit, and fixed, security parameter. To really make it a parameter, all we need to do is consider not Crypto-Game directly but functions of the security parameter returning the Crypto-Game. As mentioned in 1.4, we use $\mathbb{N}$ for the security parameter. The notion of semantic security for such a family of games is that for every adversary, their advantage is negligible. Formally the semantic security is defined as follows:

Definition 5.1.4: Semantic Security

A family G of Crypto-Games, is semantically secure if for all families of adversaries Adv the Advantage is Negligible. In CRYPTOAGDA:

$$\begin{array}{l} \text{Semantic-Security} : (\mathbb{N} \to \text{Crypto-Game}) \to \text{Set} \\ \text{Semantic-Security}\,\text{G} = \forall\,(\text{Adv} : (n : \mathbb{N}) \to \text{Adversary}\,(\text{G}\,n)) \\ \quad \to \text{Negligible}\,(\lambda\, n \to \text{Advantage}\,(\text{G}\,n)\,(\text{Adv}\,n)) \end{array}$$

We can now return to the example of ***IND-CPA*** as described in Section 1.4, and provide the corresponding definition in CRYPTOAGDA. Since ***IND-CPA*** depends on an encryption system, this definition will be parametrised over functions for key-generation, encryption and decryption. We make no assumptions about the types of these functions, other than the type of randomness for encryption and key generation, is an sample space, i.e. an RSpace.

Example 5.1.5: *IND-CPA*

The signature of the encryption system is described below, notice that we don't provide the laws here:

$$\begin{array}{l} \textit{key-gen} : \text{R}_\text{G} \to \text{Public} \times \text{Secret} \\ \textit{enc} : \text{R}_\text{E} \to \text{Public} \to \text{M} \to \text{C} \\ \textit{dec} : \text{Secret} \to \text{C} \to \text{M} \end{array}$$

The environment is the following process:

$$\begin{array}{l} \eta_{\textit{\textbf{IND-CPA}}} : \textit{Bool} \to \text{R}_\text{G} \times \text{R}_\text{E} \to \{\!\!|\ !\ (\textit{pk} : \text{Public})\,.\,?\,(\textit{ms} : \text{M} \times \text{M})\,. \\ \qquad\qquad\qquad\qquad\qquad\quad\ !\ (\textit{c} : \text{C})\,.\,\bot |\!\!\} \\ \eta_{\textit{\textbf{IND-CPA}}}\, b\,(\textit{rg}\,,\textit{re}) = \textbf{let}\,\textit{pk}\,,\textit{sk} = \textit{key-gen}\;\textit{rg} \\ \quad \textbf{in}\; p.\ \overline{p}\langle \textit{pk}\rangle\ \underline{p}(\textit{ms})\ \overline{p}\langle \textit{enc}\;\textit{re}\;\textit{pk}\;(\textit{select}\;\textit{ms}\;b)\rangle\ \mathbf{0} \end{array}$$

This process for ***IND-CPA*** first generates the public and private keys, i.e. *pk* and *sk*, and then will use channel *p* for communication. It first sends the public key, *pk*, and then receives the two messages *ms*. Finally the process sends the encrypted message that corresponds to *b* back to the adversary.

5.2 ElGamal revisited

This section will use the definitions seen in the previous sections to prove that El Gamal is semantic secure under ***IND-CPA***, assuming the group is ***DDH***. First we provide the type theoretic version of ElGamal that we will work with. This encryption system works over a group G of order q, this group have a generator g, i.e. any element in the group can be represented as g^n for some n. Of course the idea is that it is difficult to invert, which is what the ***DDH*** property of the group tells.

Definition 5.2.1: ElGamal - Encryption System

The type of public keys, PubKey is from the group G, whereas the secret key, SecKey is a number in $\mathbb{Z}/q\mathbb{Z}$. Message are in the also in the group, and the ciphertexts, CipherText are a pair of elements in the group. Finally the randomness needed by both encryption and key generation, R_e and R_k respectively, is an element in $\mathbb{Z}/q\mathbb{Z}$. In the code below, the multiplication in the group is $\bullet$, the integer exponentiation is ^, and the division, i.e. multiplication with inverse, is $/$.

```
KeyGen : $\mathrm{R}_k \to \mathrm{PubKey} \times \mathrm{SecKey}$
KeyGen $x = (g \wedge x\,,\, x)$

Enc : $\mathrm{PubKey} \to \mathrm{Message} \to \mathrm{R}_e \to \mathrm{CipherText}$
Enc $g^x\ m\ y = g^y\,,\, \zeta$ **where**
  $g^y = g \wedge y$
  $\delta = g^x \wedge y$
  $\zeta = \delta \bullet m$

Dec : $\mathrm{SecKey} \to \mathrm{CipherText} \to \mathrm{Message}$
Dec $x\,(g^y\,,\, \zeta) = \zeta\ /\ (g^y \wedge x)$
```

This is indeed a valid encryption system if we are really working with a group as shown by the following theorem.

Theorem 5.2.2: ElGamal Functional Correctness

ElGamal satisfies the functional correctness, i.e. decrypting with a secret key a message that have been encrypted with the corresponding public key is the identity operation.

Proof. The proof have two assumptions that the type G is really a group.

```
module FunctionalCorrectness
  (/-• : ∀ {α m} → (α • m) / α ≡ m)
  (comm-^ : ∀ {α x y} → (α ^ x) ^ y ≡ (α ^ y) ^ x) where
  functional-correctness : ∀ x y m → Dec x (Enc (g ^ x) m y) ≡ m
  functional-correctness x y m = trans (ap Ctx comm-^) /-•
    where Ctx = λ z → (z • m) / ((g ^ y) ^ x)
```

□

We now return to the ***DDH*** and define this game in our type theory. The formal statement we are going to prove below is that if the group is semantically secure according to the ***DDH*** Crypto-Game then ElGamal is semantically secure according to ***IND-CPA*** Crypto-Game. The Crypto-Game for ***DDH*** tells that an adversary can't find a discrete logarithm in the group, and does so by showing that the adversary cannot distinguish between the multiplication of two elements and random:

Definition 5.2.3: *DDH* Crypto-Game

The environment takes three numbers from $\mathbb{Z}/q\mathbb{Z}$ as randomness, i.e. $\mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q$, and follows the protocol of sending three elements in the group G:

$$\eta_{DDH} : Bool \to \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \to \{\!\!\{\ !\ \mathrm{G}\ .\ !\ \mathrm{G}\ .\ !\ \mathrm{G}\ .\ \bot\ \}\!\!\}$$
$$\eta_{DDH}\, false\ (x, y, z) = p.\ \overline{p}\langle g \mathbin{\char`^} x\rangle\ \overline{p}\langle g \mathbin{\char`^} y\rangle\ \overline{p}\langle (g \mathbin{\char`^} x) \mathbin{\char`^} y\rangle\ \mathbf{0}$$
$$\eta_{DDH}\, true\ (x, y, z) = p.\ \overline{p}\langle g \mathbin{\char`^} x\rangle\ \overline{p}\langle g \mathbin{\char`^} y\rangle\ \overline{p}\langle g \mathbin{\char`^} z\rangle\ \mathbf{0}$$

We now prove that ElGamal is semantically secure according to the ***IND-CPA*** Crypto-Game, under the ***DDH*** assumption. By ***DDH*** assumption we refer to the fact that the ***DDH*** Crypto-Game is semantic secure, which depends on the group. This is done by giving a *simulator* that will connect the two games and act as the challenger in one, and as an adversary in the other. The goal is to transform an ***IND-CPA*** adversary into a ***DDH*** adversary, that will be as effective, or at least negligible close, in playing both games. Therefore if an adversary is good, i.e. can break, the ***IND-CPA*** game, then it would also be good against ***DDH*** which would break our assumption.

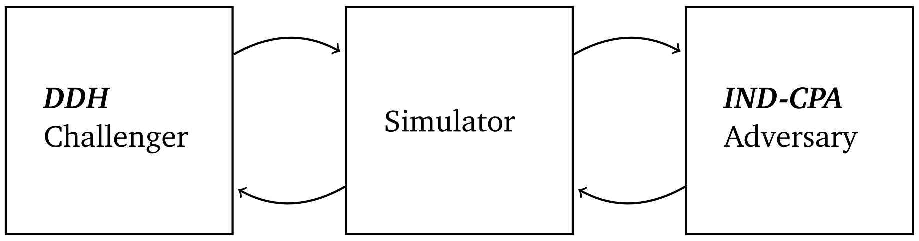

Figure 5.1: The reduction between ***IND-CPA*** and ***DDH***

As depicted in Figure 5.1, the simulator will be situated in-between the challenger of the ***DDH*** game, and the adversary of the ***IND-CPA*** game. As such it is implemented as a process that communicates with both parties at the same time. The simulator is also going to change the guess of the adversary, and will therefore have the type of a tensor for the communication with the ***IND-CPA*** adversary. In other words the simulator have three active channels, one for ***DDH*** challenger, one for the adversary and the final one for the final guess. The simulator, is also probabilistic, and as randomness takes one boolean.

$$\begin{aligned} &\mathit{sim} \;:\; \mathit{Bool} \rightarrow \{\!\mid ?G.?G.?G.0; (!\mathit{Public}.\ ?M{\times}M.!C.\bot \otimes ?\mathit{Bool}.0); !\mathit{Bool}.\bot \mid\!\} \\ &\mathit{sim}\; b \;=\; i\; ar\; r.\ \underline{ar}(a\; r')\ (\underline{r'}(b')\ \overline{r}\langle b \;==\; b'\rangle\ \mathbf{0})\ \mid \\ &\quad (\underline{i}(g^{x})\ \underline{i}(g^{y})\ \underline{i}(g^{z})\ \overline{a}\langle g^{x}\rangle\ \underline{a}(mb)\ \overline{a}\langle g^{y}\,,\, g^{z} \bullet \mathit{sel}\; b\; mb\rangle\,) \end{aligned}$$

The simulator receives from the ***DDH*** game three elements from the group: g^x, g^y and g^z. The first element will act as the public key, the second as the randomness for encryption and the final is the one that is used in the multiplication in the encryption. In order to make the final proof we use two lemmas, the first one equates the probability of the ***IND-CPA*** game, and the *false* case of ***DDH*** with the simulator.

Lemma 5.2.4

$\mathbf{Pr}[b,r_e,r_a; \mathit{run}\ (\text{TC-cut}\ (\eta_{\textbf{\textit{IND-CPA}}}\ b\ r_e)\ (\text{A}\ r_a))]$ is equal to
$\mathbf{Pr}[\gamma,r_e,r_a; \mathit{run}\ (\text{TC-cut}\ (\eta_{\textbf{\textit{DDH}}}\ \mathit{false}\ r_e)\ (\text{TC-cut}\ (\mathit{sim}\ \gamma)\ (\text{A}\ r_a)))]$

Proof. Follows by definitional equality. □

The second lemma, is for the *true* case with the simulator, the probability here will be $\frac{1}{2}$.

Lemma 5.2.5

$\mathbf{Pr}[\gamma, r_e, r_a; \mathit{run}\ (\text{TC-cut}\ (\eta_{\mathbf{DDH}}\ \mathit{true}\ r_e)\ (\text{TC-cut}\ (\mathit{sim}\ \gamma)\ (\text{A}\ r_a)))]$ is equal to $\frac{1}{2}$

Proof. This proof uses Theorem 3.5.2, to show that the adversary is unaware of which message it receives, a situation that is similar to the one in one time pad [Sha49]. Therefore the probability is $\frac{1}{2}$. □

Combining these two facts we can prove the security of ElGamal, this proof uses a small mathematical fact about the advantage of an event, with an sample space that is a product with a boolean. If the event is $b\ ==\ \text{X}$ where b is the boolean, then the advantage is half of $\left|\text{Pr}X_{b=true} - \text{Pr}X_{b=false}\right|$. Which can be proved by reasoning with exploration functions. Therefore the following equation holds:

Theorem 5.2.6: ElGamal *IND-CPA* Security

For all ***IND-CPA*** adversaries A, the following is true:

$$\begin{aligned}&\mathit{2} * \text{Advantage DDH}\ (\lambda\gamma\ r_a \to \text{TC-cut}\ (\mathit{sim}\ \gamma)\ (\text{A}\ r_a))\\ &\equiv \text{Advantage CPA A}\end{aligned}$$

Which shows that if all ***DDH*** adversaries had negligible advantage then all ***IND-CPA*** adversaries have it as well, since multiplying with a constant is still negligible.

5.3 *IND-CCA1* and *IND-CCA2*

There is a big zoo of Crypto-Games [BHK09], and in this section we introduce some more variants of ***IND-CPA***. These games will be stronger, i.e. provide more security, and the big feature is that these schemes protect against malleability. Which is the property that one can manipulate a ciphertext, for example creating a new one based on an old one, in order to carry out a man in the middle attack. In order to protect against such attacks, the extensions ***IND-CCA1*** and ***IND-CCA2*** were invented.

The major difference between ***IND-CCA1*** and ***IND-CPA*** is that the former allows for oracle calls before the Challenge phase. These oracle calls, allows the adversary to ask for decryption of certain encrypted messages. There is no limit on the number of calls the adversary makes, but since the adversary is always terminating there will be a finite amount of them. But this poses a problem for us, how does one describe a loop with the protocols? There is no looping session type, in what was presented in Chapter 4.

There are several options available to us, one is to add a form of looping construct. One way of guaranteeing that the loop always terminates is to base it on a similar construct as the W-type. But instead of adding a new construct we encode the loop within the current system, by using the fact that we can compute protocols. The protocol Server n Q R C is one way of doing so, this protocol loops n-times. In each cycle it receives a message q of type Q, and it respond with a message of type R q. By computing the protocol, it is possible to shift the recursion from the protocol to use the recursion already existing in the type theory.

$$\begin{aligned}
&\mathrm{Server} : \mathbb{N} \to (\mathrm{Q} : \mathrm{Set})\,(\mathrm{Resp} : \mathrm{Q} \to \mathrm{Set}) \to \mathrm{Session} \to \mathrm{Session}\\
&\mathrm{Server}\; \mathit{zero} \quad \mathrm{Q}\;\mathrm{Resp}\;\mathrm{Cont} = \mathrm{Cont}\\
&\mathrm{Server}\; (\mathit{suc}\; n)\; \mathrm{Q}\;\mathrm{Resp}\;\mathrm{Cont} = \mathit{?}\,(q : \mathrm{Q})\,.\,!\;(\mathrm{Resp}\; q)\,.\,\mathrm{Server}\; n\; \mathrm{Q}\;\mathrm{Resp}\;\mathrm{Cont}
\end{aligned}$$

The dual of this type is called Client n Q R C and one can show that *dual* (Server n Q R C) is equal to Client n Q R (*dual* C). The reason for the name is that a Client asks queries and gets responses from the Server, similar to the client-server model. The amount of queries to answer is given by the type, of course since the session can be dependent the number can come from communication. We define Server? as a special version of the server that will first receive the number of queries, and then act as a Server for the allotted amount of queries.

$$\begin{aligned}
&\mathrm{Server?} : (\mathrm{Q} : \mathrm{Set})\,(\mathrm{Resp} : \mathrm{Q} \to \mathrm{Set}) \to \mathrm{Session} \to \mathrm{Session}\\
&\mathrm{Server?}\; \mathrm{Q}\;\mathrm{Resp}\;\mathrm{Cont} = \mathit{?}\,(n : \mathbb{N})\,.\,\mathrm{Server}\; n\; \mathrm{Q}\;\mathrm{Resp}\;\mathrm{Cont}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Client!} : (\mathrm{Q} : \mathrm{Set})\,(\mathrm{Resp} : \mathrm{Q} \to \mathrm{Set}) \to \mathrm{Session} \to \mathrm{Session}\\
&\mathrm{Client!}\; \mathrm{Q}\;\mathrm{Resp}\;\mathrm{Cont} = !\,(n : \mathbb{N})\,.\,\mathrm{Client}\; n\; \mathrm{Q}\;\mathrm{Resp}\;\mathrm{Cont}
\end{aligned}$$

Dually we define Client! to be a Client that before asking queries, first sends the amount of queries that the Client will send. The definition of ***IND-CCA1*** is similar to ***IND-CPA*** with the exception that before the challenge phase the Challenger acts as an oracle answering decryption requests. The Session is formally described as follows:

Definition 5.3.1: *IND-CCA1*

The ***IND-CCA1*** (Indistinguishability Under Chosen Ciphertext Attack) is similar to ***IND-CPA*** but with an Server? round before the challenge phase. The environment is implemented by the following process.

$$
\begin{array}{l}
\eta CCA \;:\; Bool \to \mathrm{R_G} \times \mathrm{R_E} \to \{\!|\; !\; (pk \;:\; \mathrm{Public})\,.\,\mathrm{Server?}\;\mathrm{C}\;\mathrm{M} \\
\qquad\qquad\qquad\qquad\qquad\qquad (?\;(ms \;:\; \mathrm{M} \times \mathrm{M})\,.\;!\;(c \;:\; \mathrm{C})\,.\,\bot |\!\} \\
\eta CCA\; b\; (rg\,,\, re) \;=\; \\
\quad \mathbf{let}\; pk\,,\, sk \;=\; \textit{key-gen}\; rg \\
\qquad oracle \;:\; (n \;:\; \mathbb{N}) \to \{\!|\; \mathrm{Server}\; n\; \mathrm{C}\;\mathrm{M} \\
\qquad\qquad\qquad\qquad\qquad (?\;(ms \;:\; \mathrm{M} \times \mathrm{M})\,.\;!\;(c \;:\; \mathrm{C})\,.\,\bot\; |\!\} \\
\qquad oracle\; zero \qquad\; = \; p.\; \underline{p}(ms)\; \overline{p}\langle enc\; re\; pk\; (select\; ms\; b)\rangle\;\; \mathbf{0} \\
\qquad oracle\; (suc\; n) \;=\; p.\; \underline{p}(c)\; \overline{p}\langle dec\; sk\; c\rangle\;\; (oracle\; n\; @\; p) \\
\quad \mathbf{in}\; p.\; \overline{p}\langle pk\rangle\;\; \underline{p}(n)\;\; (oracle\; n\; @\; p)
\end{array}
$$

In order to be ***IND-CCA1*** secure, the encryption scheme needs to be ***IND-CPA*** secure. This fact can easily be demonstrated by the following lemma that shows every ***IND-CCA1*** secure encryption scheme is also ***IND-CPA*** secure.

Lemma 5.3.2: *IND-CCA1* Implies *IND-CPA*

An encryption scheme (key-gen, enc, dec) that is ***IND-CCA1*** secure is also ***IND-CPA*** secure.

Proof. A ***IND-CPA*** adversary is a ***IND-CCA1*** adversary that performs 0 oracle requests, so if the encryption scheme is secure against all ***IND-CCA1*** adversary then it is also secure against the ***IND-CPA*** ones. □

The ***IND-CCA1*** game is considered to be non-adaptive, since the adversary can't ask queries based on the encrypted message he receives. When performing reductions for proofs it is often necessary to be able to adapt the oracle request, and as such an adaptive version of ***IND-CCA1*** was created. This version is called ***IND-CCA2*** (Adaptive Chosen Ciphertext Attack), and is similar to ***IND-CCA1*** with the exception that another oracle round is happening after the challenge phase. Of course the adversary is not allowed to ask for the decryption of the encrypted text that was received as part of the challenge, since it would make the game trivial then.

Chapter 6

Prêt à Voter

Joint work with: Nicolas Pouillard

In this chapter, as a case study, we will prove that the voting system Prêt à Voter have the Receipt Freeness property. This proof, and the mathematical definition of Prêt à Voter is based on the paper [KTR13].

The voting system Prêt à Voter seeks to achieve voter verifiability and full audibility. This means that every voter is provided with a method to confirm that the vote has been cast correctly, while at the same time still provide ballot privacy. This is in general quite difficult, since providing both of these notions seems contradictionary, verifiability seeks transparency of the voting process, whereas privacy seeks the opposite.

The solution employed by Prêt à Voter, is that the voter will be provided a receipt that can be used to verify that the vote has been cast. But this receipt can't be used to prove for what the vote is and therefore provides secrecy of the ballot.

A ballot comes in two parts, a left hand side(*lhs*) which provides a randomised candidate order in clear text. The other part is the right hand side, historically called the onion, which provides an area to mark the vote. The ballot is represented in our type theory by the following data type:

```
record Ballot : Set where
  field
    lhs : CandidateOrder
    rhs : Onion
```

Bob	
Alice	
Eve	
Charlie	
David	
TBD	*SN* : 49209 *XXXXXX*

Bob	
Alice	✗
Eve	
Charlie	
David	
TBD	*SN* : 49209 *XXXXXX*

Bob
Alice
Eve
Charlie
David
TBD

✗
SN : 49209 *XXXXXX*

Figure 6.1: Example of process of voting starting with an empty ballot, that gets marked and then finally separated.

The onion contains an encrypted representation of the candidate order and a unique serial number. The serial number is used for administrative purposes, it is of course important, for the privacy of the vote, that the link between the voter and the serial number is not stored. The mark is a type that identifies which box has been marked, if the ballot haven't been marked yet the *none* constructor of the Maybe type is used.

```
record Onion : Set where
  field
    mark?       : Maybe Mark
    permuation  : CipherText
    sn          : SerialNumber
```

The voter will be given a random ballot and will proceed to mark the ballot for the preferred candidate. Afterwards the *lhs* will be destroyed such that there is no visible way to see what the mark is for. It is the *rhs* that is cast and a signed copy of this is given to the voter such that the voter can verify later that her vote has been accounted for. This since all the onions used in the tally are published together with the result of the tally.

Since the voter can't see that the encryption of the candidate order on the onion, she can't check if it is the same as the one on the *lhs*. Therefore the voter is given the power of auditing the ballot, in doing so the encrypted order will be decrypted and can therefore be compared against the visible candidate order. This will spoil the ballot and can therefore not be used to vote, and the voter will be given a new one. This process can be repeated multiple times, and since the ballots are picked at random it is likely that if the order was fraudulent, i.e. doesn't match, this would be detected.

6.1 ReceiptFreeness Game

The security notion we wish to establish, is that the voter can't prove how she voted. This is formalised by a somewhat complicated Crypto-Game, which is essentially the same game as in [KTR13], in which there is an election, of only two candidates, and we allow the adversary to corrupt as many votes as he pleases. There is going to be two more votes, for different candidates, but the adversary only sees one of the receipts. He should try to figure out for whom this receipt is, i.e. trying to figure out how the person whose receipt he received voted. The intent of the receipt is to verify that a person has voted and not how one voted, so if the adversary is able to figure out from the receipt then something is wrong.

In this game we assume that there are only two candidates, which are named Alice and Bob. The goal of the adversary is to guess for which candidate a particular Receipt is from. The Adversary is given access to an oracle, where he can query for certain powers. The powers is represented by the type Q, which is the type of queries the adversary can ask.

```
data Q : Set where
  REB RBB RTally : Q
  RCO  : Receipt → Q
  Vote : Receipt → Q
```

For every query q : Q the oracle will respond with a message of type Resp q, which is a type defined using large elimination. The type Accept?, is a type equivalent to the boolean with two constructors *accept* and *reject*. The type BB is for the bulletin board, which is just a list of onions. Finally the Tally is represented by $\mathbb{N} \times \mathbb{N}$ where the first number is the total of votes for Alice, and the second number of votes for Bob.

```
Resp : Q → Set
Resp REB i      = Ballot
Resp (RCO x)    = CandidateOrder
Resp (Vote x)   = Accept?
Resp RBB        = BB
Resp RTally     = Tally
```

The intended semantic of the queries that adversary A is sending is as follows:

REB Retrieve Empty Ballot: a new blank ballot, i.e. a ballot with both LHS and RHS, is generated and given to the adversary.

RCO x Reveal Candidate Order: given an RHS x of a ballot the oracle will reveal the candidate order.

Vote x A will send a vote, i.e. a marked RHS, this vote is validated and if it is valid put on the bulletin board, otherwise the vote is rejected.

RBB Reveal Bulletin Board: Returns the bulletin board in its current state.

RTally Reveal Tally: Returns the current tally for each candidate.

Remark 6.1.1

We note here that for aesthetic reasons we have chosen to deviate a bit from the original definition. In the paper the adversary always have read access to the bulletin board and therefore RBB would return only the tally. But it fits better with the rest of the system if access to the bulletin board goes through an oracle call, therefore we separated this into two separate requests.

The attack game, used by the receipt freeness property is using the following steps. The steps are similar to that of ***IND-CCA2***†, in that oracle calls happens before and after a Challenge.

Setup The challenger $\mathscr{C}$ sets up the system, and creates an empty bulletin board. The public parameters are sent to the adversary $\mathscr{A}$.

Phase I In this phase, the adversary $\mathscr{A}$ performs queries to the oracle.

Challenge The challenger $\mathscr{C}$ sends two receipts i and j, one is a vote for Alice and one for Bob. These receipts are in a random order, and have been freshly generated, i.e. not used in any oracle call before.

Phase II The adversary $\mathscr{A}$ can perform more oracle queries, although not allowed to use the receipts i and j.

Guess $\mathscr{A}$ returns the guess if i is a vote for Alice.

As usual, we can define advantage of an adversary as:

Definition 6.1.2: Advantage

A Prêt à Voter voting scheme is receipt free if for all polynomial time adversaries $\mathscr{A}$ the advantage $\mathrm{Adv}_{RF}(\mathscr{A}) = \left|\mathbf{Pr}[Exp_{RF} = 1] - \frac{1}{2}\right|$ is an negligible function.

Formally we can describe the attack game using the following protocol with types. We describe the protocol from the perspective of the challenger, this is the reason why the Round which is used during each phase is using the Server construct.

```
Round : Session → Session
Round Next = Server #q Q Resp Next

Exchange : Session → Session
Exchange Next = ? SerialNumber². ! Receipt ². Next

ReceiptFreeness : Session
ReceiptFreeness = ! PubKey. Round (Exchange (Round ⊥))
```

So a challenger in the Receipt Freeness game will first output a public key (PubKey). It will then have a round of answering oracle queries before doing the challenge exchange. Here it receives two serial numbers and will output two receipts before finally doing another round of answering oracle queries.

We now focus our attention on the actual implementation of the challenger, i.e. a process that follows the aforementioned protocol. The first step is the code that implements the oracle. This code has access to both the private and public key of the challenger. Furthermore it has read access to the bulletin board, write updates will happen in a separate function later on. The randomness *rgb* is used for generating the ballots, in the case of a REB call.

```
module Oracle (sk : SecKey) (pk : PubKey) (rgb : Rgb) (bb : BB) where
  resp : (q : Q) → Resp q
  resp REB = genBallot pk rgb
  resp RBB = bb
  resp RTally = tally sk bb
  resp (RCO receipt) = DecReceipt sk receipt
  resp (Vote v) = if Check bb v then accept else reject
```

Here Check is a function that checks if the given vote is acceptable given the current bulletin board *bb*. The module Chal is used for defining the challenger, this uses similar arguments as the Oracle. But furthermore has access to two sets of randomness, one for each phase, and each phase has #q number of Rgb elements. The code to update the bulletin board is *newBB*, which will only update if the request is a Vote, and then updates only if the ballot is valid.

```
module Chal (b : Bool) (uri : URI) (pk : PubKey) (sk : SecKey)
            (v : PhaseNumber → Vec Rgb #q) ($r_e$ : $R_e{}^2$) where
  newBB : BB → Q → BB
  newBB bb (Vote v) = if Check bb v then v :: bb else bb
  newBB bb _ = bb
 {-... -}
```

Still inside the Chal module we can now define the process for each step of the protocol. The *service* code is shared in each loop of the oracle phase, and it simply calls the oracle for the response and decreases the loop variable i. During the Challenge phase the challenger encrypts the two votes in some order and returns.

```
module _ {X} p# (cont : BB → Proc uri X) where
  service : (i : ℕ) → BB → Fin #q → Proc uri (Round i X)
  service zero     bb i = cont bb
  service (suc ri) bb i =
    $\underline{uri}$(q) $\overline{uri}$⟨Oracle.resp sk pk (lookup i (v p#)) bb q⟩
    service ri (newBB bb q) (pred i)
phase2 : BB → Fin #q → Proc uri (Round #q ⊥)
phase2 = service $1_2$ (λ _ → $\underline{uri}$() (0)) #q
exc : BB → Proc uri (Exchange (Round #q ⊥))
exc bb = $\underline{uri}$(sn)
  let $r_2$ = EncReceipts pk $r_e$ sn b in
  $\overline{uri}$⟨$r_2$⟩ phase2 ($r_2$ $::^2$ bb) max#q
phase1 : BB → Fin #q → Proc uri (Round #q (Exchange (Round #q ⊥)))
phase1 = service $0_2$ exc #q
```

Finally outside the Chal module we can now declare RF—C as the actual process that implements the protocol. This code receives the secret bit b and three pieces of randomness.

```
RF—C : ($b$ : $Bool$) ($uri$ : URI) ($r_k$ : $R_k$) ($v$ : PhaseNumber → Vec Rgb $\#q$) ($r_e$ : $R_e{}^2$)
  → Proc $uri$ ReceiptFreeness
RF—C $b$ $uri$ $r_k$ $v$ $r_e$ =
  let $pk$ , $sk$ = KeyGen $r_k$
      BBsetup = $[]$
  in $\overline{uri}\langle pk\rangle$ Chal.phase1 $b$ $uri$ $pk$ $sk$ $v$ $r_e$ BBsetup $max\#q$
```

We will develop a simulator showing that if the encryption used is ***IND-CCA2***†, a variation of ***IND-CCA2***, then Prêt à Voter is receipt free. But before that we will introduce this new variant of ***IND-CCA2*** named ***IND-CCA2***†.

6.2 The *IND-CCA2*† game

The difference between ***IND-CCA2***† and ***IND-CCA2*** is very small, the only difference is that in the challenge phase, the adversary receives encrypted versions of both messages sent, but in an order that depends on the secret bit b. Of course in the oracle phase after the challenge the adversary is not allowed to ask for the decryption of any of the two cipher texts received. This minimal change will be useful later. Now any encryption scheme that is ***IND-CCA2*** secure will also be ***IND-CCA2***† secure, and vice versa. This result is the main topic of this section.

Theorem 6.2.1: *IND-CCA2*† implies *IND-CCA2*

If a public-key encryption scheme $(KeyGen, Enc, Dec)$ is ***IND-CCA2***† secure then it is ***IND-CCA2*** secure.

The simulator will behave just like the ***IND-CCA2*** adversary, but during the challenge phase it will only forward the first cipher text. This is achieved by using the *mapStrategy* function that will behave similarly during the first round, but will then apply the transformation function for the challenge phase.

```
A-t' = Map.A* $id$ ($\lambda f \rightarrow f\, false$) $id$

A-transform : ($adv$ : CCA2.Adversary) → CCA2d.Adversary
A-transform $adv$ $r_a$ $pk$ = $mapStrategy$ A-t' ($adv$ $r_a$ $pk$)
```

We can prove that the transformed process, i.e the one that is communicating with the simulator, will have the same probability playing the ***IND-CCA2***† game as the adversary have playing the ***IND-CCA2*** game:

```
correct : ∀ {r_e r'_e r_k r_a} b adv
    → CCA2.EXP b adv (r_a , r_k , r_e)
    ≡ CCA2d.EXP b (A-transform adv) (r_a , r_k , r_e , r'_e)
correct {r_e} {r'_e} {r_k} {r_a} b m with KeyGen r_k
... | pk , sk =
  rs (put—resp rm (Enc pk (get—chal rm b) r_e))
    ≡⟨ ap (λ x → rs (put—resp rm (Enc pk x r_e))) rmrmd ⟩
  rs (put—resp (A-t' (runStrategy (Dec sk) (m r_a pk))) kd)
    ≡⟨ ap (λ x  →  rs (put—resp x kd)) !rm ⟩
  rs (put—resp rmd kd) ∎
  where open ≡—Reasoning
    rs = runStrategy (Dec sk)
    md = A-transform m r_a pk
    rmd = rs md
    rm = rs (m r_a pk)
    !rm = ! run—map (Dec sk) A-t' (m r_a pk)
    rmrmd = ap (λ x → get—chal x b) !rm
    kd = λ x → Enc pk (get—chal rmd (x xor b)) ([0: r_e 1 : r'_e ] x)
```

Proof. Since the probability of the two processes are the same we have that the advantages are the same as well. □

The other direction is more interesting, the proof is also based on the one found in the original paper [KTR13], but with a slight change, since the original makes a slight mistake and confuses advantage and probability. These two concept are similar but not equal, and therefore the original proof can't be reconstructed as it was in the paper. The original proof makes a case distinction on the secret bit, and in the first case calculates an advantage to be ϵ, and in the second case an advantage of ν. Then the proof conclude that the overall advantage is the average of these $\frac{\epsilon+\nu}{2}$, but this is not in general true for advantages, although it is true for probabilities. In other words the advantage is $\left|\frac{\epsilon+\nu}{2} - \frac{1}{2}\right|$, which could be different from $\frac{\left|\epsilon-\frac{1}{2}\right|+\left|\nu-\frac{1}{2}\right|}{2}$ depending on the values of ϵ and ν. This shows the value of doing mechanised proofs, we can't conflate similar concepts and as such simple mistakes like these are not possible. The theorem is still true, and can be proven.

Theorem 6.2.2: *IND-CCA2* implies *IND-CCA2*†

If a public-key encryption scheme $(KeyGen, Enc, Dec)$ is ***IND-CCA2*** secure then it is ***IND-CCA2***† secure.

Proof. There are two simulators which act similar to the one in [KTR13], which are simulators that receive only one ciphertext from the ***IND-CCA2*** challenger, therefore the simulator needs to create a new ciphertext for the ***IND-CCA2***† adversary. The simulator will encrypt at random one of the previous messages received. The difference between the two is which of the ciphertexts is the real one.

The proof of correctness has to jump between these two simulators, which makes one more appeal to the negligible assumption of ***IND-CCA2***, than the original proof. $\square$

6.3 The Simulator for the Receipt Freeness Game

The simulator between the RF-Game and the ***IND-CCA2***† is here implemented as a process between the RF-Game adversary and the ***IND-CCA2***† challenger. This means that this is a process that can communicate with two other processes, one following the ReceiptFreeness protocol and one following CCA2—*dagger*$^\perp$. Since the simulator are in the middle communicating with both the protocols in question are combined with the $\parr$ connective. For the reader with a more intuitionist's mind this could be represented by CCA2—*dagger* $\multimap$ ReceiptFreeness.

The simulator will store its own version of the ballot box, and the current tally. The ballot box will be set up using the public key from the ***IND-CCA2***† challenger, this means that the simulator can't decrypt ballots by itself. So whenever this is needed the simulator will have to be in the oracle phase from ***IND-CCA2***† game. Luckily we only need to decrypt during the oracle phase of the Receipts Freeness game, and these will line up with each other. Furthermore we will always be able to know what the current tally is.

Similar to how the processes for both the challengers of the two different attack games have used one function to describe them, so will the simulator. This code which is shown in Figure 6.2 will be called *service*, it assumes that we have a process to handle the continuation *cont*. The continuation gets access to the current ballot box and its current tally. The channel c is used to communicate with ***IND-CCA2***† challenger and the channel d is used for communication with the adversary of the RF-Game.

```
module _ {A B} (p# : Bool) (cont : BB → Tally → A ⊸ B) where
  service : (r : ℕ) → BB → Fin #q → Tally
    → Server r CipherText (const Bool) A ⊸ Round r B
  service zero    bb i ta = cont bb ta
  service (suc r) bb i ta =
    d (REB → d̄⟨ballot p# i⟩ ignore r (service r bb (pred i) ta)
      ; RBB → d̄⟨bb⟩    ignore r (service r bb (pred i) ta)
      ; RTally → d̄⟨ta⟩ ignore r (service r bb (pred i) ta)
      ; (RCO x) → c̄⟨permutation x⟩ c (co) d̄⟨co⟩ service r bb (pred i) ta)
      ; (Vote x) → c̄⟨permutation x⟩ d (co)
          if Check bb x
            then d̄⟨accept⟩ (service r (x :: bb) (pred i)
              (tallyMarkedReceipt? co (m? x) +,+ ta))
            else d̄⟨reject⟩ (service r bb (pred i) ta))
      )
```

Figure 6.2: Simulator between two oracle rounds of the ***IND-CCA2***† and Receipt Freeness games

Notice that we here are using a pattern matching receive, i.e. $d\ (p \to t; q \to u)$ is here used to represent an input on the channel d. The input are then matched against the patterns p and q, and the resulting process will be either t or u depending on which pattern match. Furthermore we use the *ignore* combinator which will perform a dummy run of the oracle in ***IND-CCA2***† game. It will simply send a dummy ciphertext and then throw away the result. Let us discuss more general how each request is handled, we read a request from the Receipt Freeness adversary and then case on this request. The first argument to *service* is how many rounds there are left in this oracle round, for both games, and this will decrease by each round. The Fin $\#q$ is used to select correct randomness for each round.

REB This requests an empty ballot for which we use the *ballot* function, the definition of which is not shown, and then we *ignore* the ***IND-CCA2***† side.

RBB This requests the current ballot box, so the current ballot box is given to the adversary, and then we *ignore* the ***IND-CCA2***† side.

RTAlly This requests the current tally, so the current tally is given to the adversary, and then we *ignore* the ***IND-CCA2***† side.

RCO x This requests to reveal the candidate order of a particular onion, here we can ask the ***IND-CCA2***† oracle for decryption since and return what it returns.

Vote x The adversary wishes to vote the onion x. We first check if this onion is valid. This check is possible to do without knowing the secret key of the ballot box, since we only need to check that the amount of marks is correct according to the voting scheme, and that this ballot have not been cast before. If the vote is valid then we *accept* the vote and ask the oracle to reveal the candidate order and we update the ballot box and tally accordingly, otherwise we *reject* the vote.

The actual simulator can now be combined in a similar fashion as the challengers have so far. The interesting bit is the code for the challenge phase, where use the *receipts* function that votes on the first mark on both ballots. Since we have picked the candidate order to be either Alice first, or Bob first, this will result in one vote for Alice and one for Bob. The simulator don't know the order of course, but that doesn't matter, finally since there is one vote for each we can increase each tally by 1.

$$
\begin{aligned}
&\mathit{sim{-}phase2} \;:\; \mathrm{BB} \to \mathrm{Fin}\ \#q \to \mathrm{Tally} \to \mathrm{CCARound}\ \bot \multimap \mathrm{Round}\ \#q\ \bot \\
&\mathit{sim{-}phase2} \;=\; \mathit{service}\ 1_2\ (\lambda\ \_\ \_ \to \bar{c}\langle\rangle\ d\ ()\ \mathbf{0})\ \#q \\
&\mathit{sim{-}chal} \;:\; \mathrm{BB} \to \mathrm{Tally} \to \mathrm{CCAChal}\ (\mathrm{CCARound}\ \bot) \multimap \mathrm{Exchange}\ (\mathrm{Round}\ \#q\ \bot) \\
&\mathit{sim{-}chal}\ \mathit{bb}\ \mathit{ta} \;=\; \underline{d}(\mathit{sn})\ \bar{c}\langle(0_2\,,\,1_2)\rangle\ d\ (\mathit{ct}) \\
&\quad \mathbf{let}\ r \;=\; \mathit{receipts}\ \mathit{sn}\ \mathit{ct}\ \mathbf{in}\ \bar{d}\langle r\rangle\ \mathit{sim{-}phase2}\ (r\ ::^{2}\ \mathit{bb})\ \mathit{max\#q}\ (\mathit{1}\ ,\mathit{1}\ +,+\ \mathit{ta})) \\
&\mathit{sim{-}phase1} \;:\; \mathrm{BB} \to \mathrm{Fin}\ \#q \to \mathrm{Tally} \\
&\quad \to \mathrm{CCARound}\ (\mathrm{CCAChal}\ (\mathrm{CCARound}\ \bot)) \multimap \mathrm{Round}\ \#q\ (\mathrm{Exchange}\ (\mathrm{Round}\ \#q\ \bot)) \\
&\mathit{sim{-}phase1} \;=\; \mathit{service}\ 0_2\ \mathit{sim{-}chal}\ \#q
\end{aligned}
$$

The final code for the simulator will first send the public key pk from the ***IND-CCA2***† challenger to the RF-Game adversary. The randomness for *simulator* is here made explicit, it was hidden in the previous code, the only code that relies on the randomness is the code for *ballot* for generating a new ballot. Furthermore we make the channels explicit.

$$
\begin{aligned}
&\mathit{simulator} \;:\; (\mathrm{Vec}\ \mathrm{Rgb}\ \#q)^{\,2} \to \mathrm{CCA2}{-}\mathit{dagger} \multimap \mathrm{ReceiptFreeness} \\
&\mathit{simulator}\ r \;=\; c\ d.\ \underline{c}(\mathit{pk})\ \bar{d}\langle \mathit{pk}\rangle\ \mathit{sim{-}phase1}\ c\ d\ r\ \mathit{pk}\ [\,]\ \mathit{max\#q}\ \mathit{0}\ ,\mathit{0}
\end{aligned}
$$

The simulator starts off with an empty ballot box and empty tally, both of which are easy to without knowledge of the secret key. This concludes the simulator between these two games . In the next section we will prove the security property, that this simulator provides a proper reduction between the two games.

The reduction is proved as follows, we first prove that the *simulator* when talking with the ***IND-CCA2***† challenger will be bisimilar to the Receipt Freeness challenger. Therefore any Receipt Freeness adversary RF—A will not be able to distinguish them. But here we can use the associativity of the cut for the processes and establish that the advantage of RF—A in the RF-Game will the same as the advantage of (νcd) *simulator rgb* $\|$ RF—A in the ***IND-CCA2***† game.

Theorem 6.3.1: Receipt Freeness for Prêt à Voter

For all Receipt Freeness adversaries RF—A, the final guess when communicating with the challenger of the Receipt Freeness challenger, is the same as when communicating with the *simulator* communicating with the ***IND-CCA2***† challenger.

Proof. The proof goes by induction on the messages that the adversary RF—A sends, in each step the behaviour of the challenger and simulator is the same. □

With this proof complete, we have proved that Prêt à Voter has the Receipt Freeness property assuming that the underlying encryption system is ***IND-CCA2***†. Since all ***IND-CCA2*** encryption systems are also ***IND-CCA2***†, this shows that one can indeed realise Prêt à Voter as a voting system.

Conclusion

As a stronger push to move to electronic voting is happening around the world, it becomes increasingly important to build these systems in a way that induces trust. For this endeavour to work, the security of the systems need to be based on solid arguments. As such, we look into the use of semantic security proofs to verify the proposed cryptographic constructions used to build the voting systems. These mathematical proofs are rigorous and have been used by the security community for decades. The added benefit of using mathematical construction that are built on top of logical arguments is that the proofs can be verified with proof assistants, as have been carried out with this work.

We have worked within a type theory, since these formalisations work make it easier to reason about computational systems. The agents that we are modelling, and by that we mean not only the algorithms used for the cryptographic construction, but also the adversaries that want to break the system, can all be modelled by constructive functions. Furthermore it is important that these functions are probabilistic, which can be modelled as deterministic functions with randomness as an extra argument. Since the functions are just normal functions, but with an extra argument, one can reason about them in the present type theory as one would do with normal terms. This work has introduced some new techniques to work with such functions though, in particular since the notion of equality is changed from the normal. Instead probabilistic values are equal if they have the same distribution. By using types effectively, and in particular type isomorphisms, it is easier to reason about such equivalences.

The agents can apart from using computation, also communicate with the environment. This is important to correctly capture the notion of games that are used in semantic security proofs. The use of communication and concurrency is not standard in type theory, as such, this work has proposed a new type theory which includes communication and concurrency primitives. The agents using communication, also called processes, can be typed with behaviour types that can be included in the normal type theory. These behaviour types share a commonality with linear logic, and have been heavily inspired by recent work in the area of using linear logic to type communicating processes. The type theory has then been modelled in a current type theory to prove it's consistency.

The reasoning principles, and the model of communication, have been put together in this work into a library by the name of CRYPTOAGDA. With this library it is possible to formally verify semantic security notions, for proposed constructions. In this thesis, as a case study the proof of receipt freeness for the voting system ***Prêt à Voter***, has been formally verified using this library, and is a testament to the feasibility of the method for providing guarantees about cryptographic constructions.

Bibliography

[Abe06] Andreas Abel. *A polymorphic lambda-calculus with sized higher-order types*. PhD thesis, PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[APM09] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in coq. *Science of Computer Programming*, 74(8):568 – 589, 2009.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 27–38, New York, NY, USA, 2013. ACM.

[Avr96] Arnon Avron. The method of hypersequents in the proof theory of propositional non-classical logics. In *Logic: from foundations to applications*, pages 1–32. 1996.

[AW09] Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146:45–55, 1 2009.

[BCHPM04] Yves Bertot, Pierre Castéran, Gérard (informaticien) Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. Données complémentaires http://coq.inria.fr.

[BCP00] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory, 2000.

[Bel97] Mihir Bellare. A note on negligible functions. 1997.

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 14–25, New York, NY, USA, 2004. ACM.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 103–112, New York, NY, USA, 1988. ACM.

[BGOBP08] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In OtmaneAit Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 86–101. Springer Berlin Heidelberg, 2008.

[BGZB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009.

[BHK09] Mihir Bellare, Dennis Hofheinz, and Eike Kiltz. Subtleties in the definition of ind-cca: When and how should challenge-decryption be disallowed? Cryptology ePrint Archive, Report 2009/418, 2009. `http://eprint.iacr.org/`.

[BJP10] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 345–356, New York, NY, USA, 2010. ACM.

[Bla09] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, July 2009.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer Berlin Heidelberg, 2006.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[BS94] Gianluigi Bellin and Philip Scott. On the π-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994.

[CDP14] Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without k. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 257–268, New York, NY, USA, 2014. ACM.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[CP02] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, 2002.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, editors, *CONCUR'10*, volume 6269 of *LNCS*, pages 222–236, 2010.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):pp. 584–590, 1934.

[DDN91] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-malleable cryptography. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 542–552, New York, NY, USA, 1991. ACM.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[dt04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198 – 208, mar 1983.

[Dyb97] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.

[Elg85] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *Information Theory, IEEE Transactions on*, 31(4):469 – 472, jul 1985.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, University of Paris VII, 1972.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir96] Jean-Yves Girard. Proof-nets : the parallel syntax for proof-theory. In A. Ursini and P. Agliano, editors, *Logic and Algebra*. M. Dekker, New York, 1996.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and Systems Sciences*, (28):270–299, 1984.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, February 1989.

[Gug07] Alessio Guglielmi. A system of interaction and structure. *Transactions on Computational Logic (ACM)*, 8(1):1–64, 2007.

[GV10] Simon Gay and Vasco Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

[Hoa85] Tony Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hon93] Kohei Honda. Types for dyadic interaction. In E. Best, editor, *CONCUR'93*, volume 715 of *LNCS*, pages 509–523, 1993.

[How80] W. A. Howard. *The formulae-as-types notion of construction*, pages 480–490. Academic Press, London-New York, 1980.

[Hur95] AntoniusJ.C. Hurkens. A simplification of girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 266–278. 1995.

[KPB15] Neelakantan Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *POPL'15*, pages 17–30, 2015.

[KTR13] Dalia Khader, Qiang Tang, and Peter Y.A. Ryan. Proving prêt à voter receipt free using computational security models. In *Presented as part of the 2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*, Berkeley, CA, 2013. USENIX.

[Lip11] Helger Lipmaa. On the cca1-security of elgamal and damgård's elgamal. In Xuejia Lai, Moti Yung, and Dongdai Lin, editors, *Information Security and Cryptology*, volume 6584 of *Lecture Notes in Computer Science*, pages 18–35. Springer Berlin Heidelberg, 2011.

[Mcb02] Conor Mcbride. Elimination with a motive. In *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00), volume 2277 of LNCS*, pages 197–216. Springer-Verlag, 2002.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Proceedings of the Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*. 1975.

[MLS84] P. Martin-Löf and G. Sambin. *Intuitionistic type theory*. Studies in proof theory. Bibliopolis, 1984.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[NY90] Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 427–437. ACM, 1990.

[NY95] Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *In Proc. of the 22nd STOC*, pages 427–437. ACM Press, 1995.

[P$^+$03] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

[RBH$^+$09] P.Y.A. Ryan, D. Bismark, J. Heather, S. Schneider, and Zhe Xia. Prêt à voter: a voter-verifiable voting system. *Information Forensics and Security, IEEE Transactions on*, 4(4):662 –673, dec. 2009.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523, 1983.

[Ric53] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 543–, Washington, DC, USA, 1999. IEEE Computer Society.

[Sha49] Claude E Shannon. Communication theory of secrecy systems*. *Bell system technical journal*, 28(4):656–715, 1949.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs, 2004. shoup@cs.nyu.edu 13166 received 30 Nov 2004, last revised 18 Jan 2006.

[Sta85] R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65(2–3):85 – 97, 1985.

[Ste03] Jacques Stern. Why provable security matters? In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 449–461. Springer Berlin Heidelberg, 2003.

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):pp. 198–212, 1967.

[TCP13] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In M. Felleisen and P. Gardner, editors, *ESOP'13*, volume 7792 of *LNCS*, pages 350–369, 2013.

[Uni13] T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013.

[Vas12] Vasco Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.

[Voe11] Vladimir Voevodsky. Univalent foundations of mathematics, 2011. Invited talk at WoLLIC 2011 18th Workshop on Logic, Language, Informaiton and Computation.

[Voi09] Janis Voigtländer. Free theorems involving type constructor classes. In Andrew Tolmach, editor, *14th International Conference on Functional Programming, Edinburgh, Scotland, Proceedings*, volume 44 of *SIGPLAN Notices*, pages 173–184. ACM Press, September 2009.

[Wad89] Philip Wadler. Theorems for free! pages 347–359, September 1989.

[Wad14] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.

[Zan10] Santiago Zanella Béguelin. *Formal Certification of Game-Based Cryptographic Proofs*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2010.