THÈSE DE DOCTORAT
DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Présentée par

Monsieur Yu ZHANG

pour obtenir le grade de
DOCTORAT DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Domaine : Informatique

Sujet de la thèse :

Relations logiques cryptographiques

— Qu'est-ce que l'équivalence contextuelle des protocoles cryptographiques et comment la prouver ?

Thèse présentée et soutenue à Cachan le 21 octobre 2005 devant le jury composé de :

Bruno Blanchet	Chargé de recherche	Examinateur
Pierre-Louis Curien	Directeur de recherche	Rapporteur
Thomas Genet	Maître de conférence	Examinateur
Jean Goubault-Larrecq	Professeur	Directeur de thèse
Martin Hyland	Professeur	Rapporteur
Jean-François Monin	Professeur	Président
David Nowak	Chargé de recherche	Co-directeur de thèse

Laboratoire de Spécification et Vérification
ENS CACHAN / CNRS / UMR 8643
61 avenue du Président Wilson, 94235 CACHAN CEDEX (France)

Résumé

Dans le cadre de la vérification des protocoles cryptographiques, une idée importante est d'utiliser l'équivalence contextuelle (aussi appelée l'équivalence observationelle) pour décrire des propriétés de sécurité. Il est difficile de prouver directement l'équivalence contextuelle, mais dans les lambda-calculs typés, on peut souvent la déduire par l'outil dit des relations logiques.

Nous appliquons cette technique à un métalangage cryptographique, qui est une extension du lambda-calcul computationnel de Moggi, et nous utilisons la monade de génération de noms de Stark pour étudier la génération dynamique de clés. La construction de relations logiques pour les types monadiques (par Goubault-Larrecq et al.) nous permet alors de dériver des relations logiques sur le modèle $\mathcal{S}et^{\mathcal{I}}$ de Stark.

Cette étude aboutit à une exploration de ce que doit être la définition de l'équivalence contextuelle pour les protocoles cryptographiques. Nous arguons du fait que l'équivalence contextuelle définie sur le modèle de Stark ne représente pas fidèlement ce que les contextes ou les attaquants peuvent faire. En effet, bien que la catégorie $\mathcal{S}et^{\mathcal{I}}$ soit un modèle parfaitement adéquat de génération de clés, elle est insuffisante par certains aspects lorsqu'on étudie les relations entre programmes du métalangage. Nous montrons que, pour définir l'équivalence contextuelle et les relations logiques dans le métalangage cryptographique, la catégorie $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ est un meilleur choix, où $\mathcal{I}^{\rightarrow}$ est une catégorie que nous définissons. Pourtant, cette catégorie est encore insuffisante par d'autres aspects plus subtils, et nous montrons finalement que la catégorie que l'on doit considérer en est une autre que nous appelons $\mathcal{S}et^{\mathcal{P}\mathcal{I}^{\rightarrow}}$. Nous définissons une notion d'équivalence contextuelle adéquate sur cette catégorie.

Nous montrons ensuite que la relation logique cryptographique définie sur $\mathcal{S}et^{\mathcal{P}\mathcal{I}^{\rightarrow}}$ est correcte, et complète pour certains types du premier ordre. Nous explorons aussi certains cas de la question de la décidabilité des relations logiques cryptographiques reliant deux termes donnés.

Afin d'étendre nos résultats de correction et de complétude à tous les types, nous remplaçons la notion de relations logiques par celle de relations logiques lax, toujours sur la catégorie $\mathcal{S}et^{\mathcal{P}\mathcal{I}^{\rightarrow}}$. Nous définissons donc une relation logique qui est lax sur les types de fonction et de monade, mais stricte (non-lax) sur les autres, et nous montrons qu'elle est correcte et complète pour l'équivalence contextuelle à tous les types.

Abstract

Using contextual equivalence (a.k.a. observational equivalence) to specify security properties is an important idea in the field of formal verification of cryptographic protocols. While contextual equivalence is difficult to prove in general, in typed lambda calculi, one is usually able to deduce it using so-called logical relations.

We apply this technique on the cryptographic metalanguage, an extension of Moggi's computational lambda calculus. To explore the difficult aspect of dynamic key generation, we use Stark's name creation monad. The general construction of logical relations for monadic types (by Goubault-Larrecq et al.) then allows us to derive logical relations on Stark's model $\mathcal{S}et^{\mathcal{I}}$.

This study also leads us to an exploration of what should be the right definition of contextual equivalence for cryptographic protocols. We argue that contextual equivalence defined over Stark's model cannot represent honestly the power of contexts or attackers. Actually, although Stark's category $\mathcal{S}et^{\mathcal{I}}$ is a perfectly adequate model of dynamic key generation, it lacks in some aspects when we study relations between programs in the metalanguage. We show that, to define contextual equivalence and logical relations in the cryptographic metalanguage, a better choice of category is $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$, where $\mathcal{I}^{\rightarrow}$ is a category we define. This category is still lacking in some subtler aspects, and we eventually show that the proper category to consider is one called $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. We find the proper notion of contextual equivalence based on this category.

Next, we show that the cryptographic logical relation defined on $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ is sound, and complete for a certain subset of types up to first order. We explore questions of decidability of cryptographic logical relations relating two given terms in certain cases.

We then extend our soundness and completeness results at all higher-order types. This requires us to shift from logical relations to lax logical relations, still on the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. We then define logical relations which are lax at function types and monadic types but strict (non-lax) at various other types, and show that they are sound and complete for contextual equivalence at all types.

Table des matières

1 Introduction **1**

1.1 Protocoles cryptographiques 2

1.1.1 Cryptographie 2

1.1.2 Protocoles cryptographiques 3

1.2 Méthodes formelles 5

1.2.1 La propriété de secret et l'équivalence contextuelle 5

1.2.2 Relations logiques 7

1.3 Lien avec d'autres travaux 9

1.4 Plan de la thèse 10

2 Le métalangage cryptographique **13**

2.1 Préliminaires 15

2.1.1 Le lambda-calcul simplement typé 15

2.1.2 Le nu-calcul et le lambda-calcul cryptographique 17

2.1.3 Le lambda-calcul computationnel 18

2.2 Le métalangage cryptographique 19

2.3 Formalisation de la propriété de secret 25

2.4 Codage des protocoles 26

2.4.1 Un protocole de l'échange de clés symétriques 26

2.4.2 Le protocole de Needham-Schroeder 29

3 Modèles catégoriques **35**

3.1 Préliminaires de la théorie des catégories 38

3.1.1 Interprétation du lambda-calcul en CCCs 41

3.1.2 Monades et le lambda-calcul computationnel 42

3.2 Le catégorie de foncteurs $Set^{\mathcal{I}}$ 44

3.3 Interprétation du métalangage cryptographique 46

3.3.1 Dénotation de messages 46

3.3.2 Interprétation du métalangage en $\mathcal{S}et^{\mathcal{I}}$ 47
3.4 Formes canoniques 51
3.5 Équivalence contextuelle 54

4 Relations logiques 57
4.1 Relations logiques 60
4.2 Relations logiques pour les types monadiques 64
4.3 Le catégorie $\mathcal{I}^{\rightarrow}$ 70
4.4 Dérivation des relations logiques sur $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ 74
4.5 Une relation logique pour le métalangage 77
4.5.1 La relation entre les messages 78
4.5.2 Une relation logique cryptographique faible 81

5 Relations logiques cryptographiques 87
5.1 La catégorie $\mathcal{PI}^{\rightarrow}$ 89
5.2 Dérivation des relations logiques sur $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ 93
5.3 Relations logiques cryptographiques 96
5.4 Vérification des protocoles à l'aide de relations logiques 99
5.4.1 Le protocole de l'échange de clés symétriques 100
5.4.2 Le protocole de Needham-Schroeder-Lowe 102
5.5 Comparaisons avec les relations logiques du nu-calcul 104

6 Complétude des relations logiques 109
6.1 Équivalence contextuelle des protocoles cryptographiques 113
6.2 Complétude pour les types non-monadiques 116
6.3 Complétude pour les types monadiques 124
6.4 Relations logiques lax complètes 129

7 Décidabilité de l'équivalence contextuelle 135
7.1 Décidabilité dans le cas des fonctions 138
7.2 Décidabilité dans le cas monadique 141
7.3 Décidabilité de l'équivalence contextuelle 146

8 Conclusion 151
8.1 Résumé des résultats 153
8.2 Perspectives 156

A Règles de raisonnement du métalangage cryptographique 159

B Complétude des relations logiques monadiques **163**
B.1 Partial computation 167
B.2 Exceptions 169
B.3 Non-determinism 171
B.4 State transformers 173

Chapitre 1

Introduction

Avec le développement rapide d'Internet et du commerce électronique, les technologies deviennent de plus en plus attachées à notre vie quotidienne. Tandis que les technologies informatiques nous apportent beaucoup de confort, elles causent aussi des problèmes, notamment de sécurité, ce qui est actuellement un sujet critique dans le domaine de l'informatique. En particulier, nous nous intéressons à la sécurité des réseaux, formés de plusieurs terminaux (ordinateurs, téléphones mobiles, etc.) qui se connectent dans un environnement de communication pas sûr et qui collaborent via l'échange de messages.

Pourtant, il est en général difficile de sécuriser les communications sur un réseau ouvert, où les messages envoyés par un terminal passeront devant plusieurs terminaux, avant d'arriver au receveur souhaité. Pendant l'envoi, des terminaux malveillants peuvent donc facilement lire ces messages, les enregistrer, les modifier, les retransmettre à un autre destinataire, ou simplement stopper leur transmission. Essentiellement, les réseaux ouverts ne garantissent pas eux-mêmes la sécurité des communications.

De nombreux protocoles ont été inventés pour sécuriser les communications sur les réseaux, en utilisant la *cryptographie*. Il est universellement considéré que l'appliquer de la cryptographie dans un système complexe est très subtile et il peut y avoir des attaques, même si la cryptographie elle-même est incassable. Alors comment appliquer la cryptographie *correctement* ? Cette thèse a pour but de vérifier les propriétés de sécurité des protocoles cryptographiques.

Nous exposons les connaissances dans ce domaine au premier chapitre de la thèse. Dans la partie 1.1, nous introduirons des concepts de cryptographie et de protocoles cryptographiques, en donnant des exemples. Dans la partie 1.2, nous donnerons une introduction aux méthodes formelles, qui sont de plus en plus utilisées dans le domaine de la vérification des protocoles cryptographiques. En particulier, nous nous concentrons sur la méthode dite des *relations logiques*. Puis dans la partie 1.3, nous résumerons les travaux connexes à ceux présentés dans cette thèse. Enfin, la dernière partie exposera le plan de la thèse.

With the rapid development of Internet and e-business, information technologies become more and more attached to our life. While these technologies offer us much convenience, they cause a lot of problems as well, especially security problems. Computer security is a very critical issue nowadays and catches great attention of the research community in computer science. We are in particular interested in security problems caused by communicating systems, where linked computers in an open environment are inclined to collaborate through message exchanging.

However, secure communications over an open network are in general very hard to achieve. In network communications, messages, sent from one computer to another, will pass by several other computers on route. It is easy for malicious intermediate computers to interfere with these messages passing through. They may read and record messages, change the contents, redirect messages elsewhere, or prevent them arriving at their intended destination. In a word, an open network itself is inherently insecure and does not guarantee any secure communication.

Various protocols have been designed for establishing secure communications over an insecure network, and most protocols adopt the mean of *Cryptography*. A common viewpoint is that applying cryptography in a complex system is very subtle and error-prone, even though the cryptography itself is perfect. How can we apply cryptographic methods *correctly* in network communications? This thesis is devoted to verifying security properties of protocols using cryptography.

1.1 Protocoles cryptographiques

1.1.1 Cryptographie

Cryptography is a fundamental mechanism to achieve security in an open environment. When we apply cryptography in a network communication, we shall disguise messages before sending them, so that only the intended recipients are able to retrieve the original text. These disguised messages are usually called *cipher-texts* and the original texts are *plain-texts*. The operation of disguising a message is known as the *encryption* and the inverse operation, which retrieves the plain-text from the cipher-text, is called *decryption*.

The precise form of a cipher-text, corresponding to certain plain-text m, depends on an additional parameter — the *key*. We shall write the cipher-text as $\{m\}_k$, where k is the key for encryption. An encryption (decryption) is just seen as applying the encryption (decryption) algorithm to a plain-text (cipher-text) and a key. In order to recover the original plain-text from a given cipher-text $\{m\}_k$, one must obtain the correct decryption key (written as k^{-1}). By restricting appropriately who has access to the various keys involved we can limit the ability to form cipher-texts and the ability to derive the corresponding plain-texts.

There are nowadays various cryptographic algorithms [MvOV96, DK02], and most of them can be classified by two schemes: symmetric cryptography and asymmetric cryptography. In symmetric cryptography, keys for encryption and for decryption are identical ($k^{-1} = k$) and it holds

$$dec(\{m\}_k,\ k) = m,$$

where dec denotes the decryption algorithm. In asymmetric cryptography, a.k.a., the public key cryptography, keys always come in pairs, one of which is usually publicly available (known as the *public key*), and the other is kept secret (known as the *private key*). Anyone can get access to the public key and use it to encrypt a message, but only the holder of the private key can decrypt the cipher-texts:

$$dec(\{m\}_{pk},\ sk) = m,$$

where pk and sk denote the corresponding public key and private key respectively. Some asymmetric cryptographic algorithms also allow the private key to be used to encrypt plain-texts with the public key being used for decryption:

$$dec(\{m\}_{sk},\ pk) = m$$

This is mainly used in the digital signature scheme to authenticate the identity of the sender of a message.

1.1.2 Protocoles cryptographiques

There are nowadays several cryptographic algorithms which are considered virtually impossible to crack: the best known ways of cracking the messages would use vast amounts of computer power. However, what is surprising is that even with perfect encryption algorithms, it is still very difficult to achieve secure communication. The question is: how can people agree on a key across network? This was probably the first problem (known as the *key distribution problem*) that protocol designers aimed at solving using cryptography. Of course, there are many cryptographic protocols devoted to solving other problems, such as authenticity of principles, anonymity, and so on.

Cryptographic protocols are protocols that use cryptography to establish secure communications over open networks. A protocol usually involves several participants, called *principles* or *agents*. We can simply regard principles as programs running in parallel. Usually these programs are hosted at different computers in the network, but it is also possible that some of them run at the same computer. The protocol defines the way how these principles exchange messages using cryptography. Here is an example of a symmetric key establishment protocol between two principles, with the help of a trusted server [1] (some notations for protocol specification are listed

[1] This protocol, together with the attack coming after, is from an informal document of Gavin Lowe.

$\{m\}_k$	message m encrypted with key k
$A \to B : m$	principle A sends message m to B
$A \to E(B) : m$	intruder E intercepting message m intended for B
$E(A) \to B : m$	intruder E impersonating A to send message m to B
$pk(A)$	principle A's public key

Figure 1.1: Notations in protocol specifications

in Figure 1.1):

$$\begin{array}{llll} \text{Message } 1\,. & A \to S: & A, B, \{k_{ab}\}_{k_{as}} \\ \text{Message } 2\,. & S \to B: & A, B, \{k_{ab}\}_{k_{bs}} \\ \text{Message } 3\,. & A \to B: & \{i\}_{k_{ab}} \end{array}$$

This protocol uses only symmetric encryption and aims at the distribution of a fresh symmetric key between two principles that are inclined to communicate with each other, through a trusted key server. It is assumed that the server S shares a secret key with every principle (A, B, and others including malicious principles) and it would never be malicious, so that every principle can trust S and communicate with it securely.

Suppose that A wants to talk to B but he does not share a secret key with B, so A generates a fresh symmetric key k_{ab}, encrypts it with the secret key k_{as} that he shares with S. He sends this message (Message 1) to S and asks S to deliver this key to B. Message 1 also contains names of the two intended principles, but they are not encrypted.

Upon receiving Message 1, S knows the fact that A wishes to talk to B, so he retrieves the key k_{ab}, encrypts it with the key k_{bs} that he shares with B, and sends it to B (Message 2).

When B receives Message 2, he retrieves the fresh key k_{ab}, then he can use this key to decrypt messages from A (Message 3). Certainly, he can also use k_{ab} it to encrypt his own messages and send to A.

Such a protocol can be executed for many times, by different principles and with different keys. A complete execution of the protocol as described above is called a *session*.

Specifications of cryptographic protocols, like this protocol, are simple and contain only several exchanges of messages. Despite of the simplicity, it is difficult to verify that a protocol can meet its goal. For instance, this symmetric key establishment protocol is flawed: when a principle A initiates a session of this protocol and wishes to talk to another principle B, an intruder E interferes with the execution of the protocol and impersonates B, but the principle A

always believes that he is communicating with B and the texts are kept secret.

$$\begin{array}{lrl} \text{Message } 1.1\ . & A \rightarrow E(S): & A, B, \{k_{ab}\}_{k_{as}} \\ \text{Message } 2.1\ . & E(A) \rightarrow S: & A, E, \{k_{ab}\}_{k_{as}} \\ \text{Message } 2.2\ . & S \rightarrow E: & A, E, \{k_{ab}\}_{k_{es}} \\ \text{Message } 1.3\ . & A \rightarrow E(B): & \{i\}_{k_{ab}} \end{array}$$

When A sends Message 1.1 to S, the intruder E intercepts this message and prevents it reaching S. While he cannot decrypt the message to get the key k_{ab} without knowing k_{as}, the intruder modifies the message by replacing the name B with his own name E, and sends the fake message to S (Message 2.1). What S learns from this fake message is that A wishes to talk with E, so he encrypts the key k_{ab} with the key k_{es} that he shares with E and sends it to E (Message 2.2). The intruder E can then decrypt the cipher-text in Message 2.2 and get the key k_{ab}, hence he can decrypt those encrypted messages sent from A (intended for B).

Clearly, the flaw in this protocol has nothing to do with the involved cryptographic algorithms. It is a flaw of protocol design alone. There were many such flawed protocols which had been thought to be secure [Low96b, CJ97, LSV]. The most classic one is probably the Needham-Schroeder's public key protocol [NS78] and Lowe's well-known attack [Low95, Low96a] (see Chapter 2 for details) .

1.2 Méthodes formelles

Until the middle of 1990s, most work on security protocol analysis was devoted to finding attacks on known protocols and the analysis was quite informal. However, informal analysis is usually prone to errors and not reliable, because security problems are very complex and some flaws are not intuitive at all. On the other hand, analysis of cryptographic protocols appears to be well suited for the application of formal methods. Indeed, a number of formal models for cryptographic protocols have been proposed in the past decade, which opened a way for the use of formal methods and formal analysis of protocols [Mea00, Mea03, CS02]. These formal methods, instead of searching attacks, address mainly proof techniques for protocol correctness. Most formal models are based on the *perfect cryptography* assumption: when we are given a cipher-text $\{m\}_k$, the only way to decrypt it is to get the corresponding decryption key k^{-1}, and it is assumed that there is no way for attackers to guess or forge the key k^{-1}, if it is secret. We shall keep to this assumption in this thesis.

1.2.1 La propriété de secret et l'équivalence contextuelle

While there are many properties that a security protocol may aim to guarantee, we will focus on the *secrecy* property in this thesis. There are many definitions of secrecy, and the relationship between them is not clear [Aba99]. In this thesis, we say that a protocol preserves secrecy of a datum if attackers cannot learn the value by interacting with the protocol within the framework of the traditional Dolev-Yao model [DY83], where attackers are able to eavesdrop on, remove and arbitrarily schedule messages sent on public communication channels, create new messages from pieces of messages they observe and insert them into the channels. The goal of the protocol analysis is then to determine whether there is a protocol trace in which attachers may learn the value of the datum that the protocol aims to protect.

A very popular idea in this area is that the secrecy property can be represented by the notion of *contextual equivalence* (a.k.a. *observational equivalence*) [AG99, SP03]. Contextual equivalence is a notion for programming languages. We say that two programs are contextually equivalent if there is no context that can distinguish them. A context can be simply seen as an operating system, so in order to test the contextual equivalence between two programs, we just execute them and observe the results. If the results are always the same, we can then assert that these two programs are contextually equivalent.

The idea is to consider cryptographic protocols as programs, where secret messages are parameters. A context is the network that may contain attackers. To put it extremely, contexts are attackers. Given a protocol, we shall have different protocol instances for different secret messages, so if two instances of the protocol are proved contextually equivalent, we can then assert that this protocol guarantees the secrecy property, because no attacker can see the difference between different secrets.

This method requires a (formal) language for specifying cryptographic protocols in the first place. There are several existing languages for protocol specification. The most well-known is perhaps the Spi-calculus [AG99]. This is an extension of the pi-calculus — a very simple but powerful system for studying processes and communication channels [MPW92, Mil99, SW01], where communication channels can be created and processes can communicate with each other by sending and receiving messages (even channel names) through channels. While security protocols rely heavily on communication channels with certain security properties like authenticity and privacy, it is very natural to use the pi-calculus (with extensions) for describing and analyzing protocols, at least at an abstract level. The Spi-calculus is basically an extension of the pi-calculus with cryptographic primitives. In the Spi-calculus, a cryptographic protocol is encoded as a process consisting of several child processes running in parallel, each representing a principle involved in this protocol. Message exchanging is naturally modeled by communications through channels shared by these principles. Attackers are also encoded as processes in this language, which are allowed to interact with the protocol process in any possible way. A not so ideal aspect of the pi-calculus is probably the lack of a good denotational model. Reasoning about concurrent processes and channels usually rests on syntax.

A less known formal language for describing protocols is the *cryptographic lambda-calculus*, proposed by Sumii and Pierce [SP01, SP03]. The lambda-calculus has certain advantages. Most obviously, higher-order behaviors are naturally taken into account, which is ignored in other models (although, at the moment, higher order is not perceived as a necessary feature in cryptographic protocols). Furthermore, public keys are encoded as functions in a nice way [SP01] (see Chapter 2 for detail), which requires at least second-order functions in some cases, e.g., the very original version of Needham-Schroeder's public key protocol, where public keys are sent as messages.

Another feature of the lambda-calculus approach is the dynamic generation of fresh keys or nonces. This mechanism plays a crucial rôle in cryptographic protocols. While dealing with fresh key generation is a weak point in most formal models, this has been well studied in a language called the *nu-calculus*, proposed by Pitts and Stark [PS93a]. The nu-calculus is an extension of the simply-typed lambda-calculus, devoted to the study of fresh name creation (where names are seen as syntactically identical to keys). According to Stark's later work on the nu-calculus, dynamic name creation can be nicely modeled in the framework of Moggi's computational lambda-calculus, in which we are allowed to describe various forms of computation, such as exceptions, non-determinism, and so on [Mog89, Mog91]. Denotational models of the computational lambda-calculus must be defined using monads. Stark specializes Moggi's work in the dynamic name creation monad and builds a denotational model — the category $\mathcal{S}et^{\mathcal{I}}$ — for the nu-calculus [Sta94, Sta96].

The model $\mathcal{S}et^{\mathcal{I}}$ is also sufficient for modeling the lambda-calculus with cryptography, since cipher-texts are usually seen as products of plain-texts and keys. However, defining contextual equivalence for cryptographic protocols in such a framework needs more attention. In particular, contexts must represent honestly the power of attackers. It turns out that the category $\mathcal{S}et^{\mathcal{I}}$ is not sufficient for defining a denotational notion for contextual equivalence and we should switch to some subtler category with more information. Detailed discussion on this point will be found in Chapter 4 and Chapter 5.

1.2.2 Relations logiques

Direct proofs of contextual equivalence are unfeasible in general, because its definition involves a universal quantification over an infinite number of contexts. *Logical relations* are an alternative technique for proving contextual equivalence [Mit96], which is the main advantage of the lambda-calculus approach for verifying protocols.

Logical relations are a frequently used technique in lambda-calculi and it has proved useful in proving various properties of typed lambda-calculi [Mit96]. Basically, a logical relation is a family of relations between expressions or elements in semantics, which are indexed by types and defined by induction on their types. For instance, two pairs are related if and only if their components are pairwise related. Crucially, two functions are related if and only if they map related arguments to related results. This technique was first introduced by Plotkin to reason about definable elements in denotational models of the simply-typed lambda-calculus [Plo80]. Later, using the notion of *sconing*, Mitchell and Scedrov obtained a mathematical theory of logical relations [MS93]. In particular, they define a general way of deriving logical relations on categorical models. This is again extended by Goubault-Larrecq, Lasota and Nowak to categories with monads [GLLN02], so that one can naturally derive logical relations for monadic types.

Although logical relations were first developed for the denotational semantics of typed lambda-calculi, they can also be adapted to their term models and this adaptation is sometimes called syntactic logical relations [Pit98, Pit00]. Indeed, the logical relations that Sumii and Pierce defined for the cryptographic lambda-calculus are syntactic [SP01, SP03].

Dealing with fresh key generation is again the main technical difficulty that one must face in defining logical relations for lambda-calculi with cryptography. The work of Sumii and Pierce is inspired in this respect by Pitts and Stark [PS93a, PS93b], who define the notion of *operational logical relation* to establish contextual equivalence of nu-calculus expressions. They also show that this logical relation is complete up to first-order types. Sumii and Pierce's various logical relations are extensions of the operational logical relation.

Stark also defines a denotational logical relation, using the notion of "categories with relations", and shows that such a denotational logical relation identifies their operational logical relation up to second-order types: if two programs are related by the operational logical relation, then their interpretations are related by the denotational one [Sta94, Theorem 4.25]. While his category for deriving denotational logical relations is restricted to the name creation monad, the work by Goubault-Larrecq et al. on logical relations for monadic types provides a more general way to build logical relations for computations [GLLN02]. They define as an example a Kripke logical relation for the dynamic name creation monad, still based on the categorical model $\mathcal{S}et^{\mathcal{I}}$. But compared to Pitts and Stark's logical relations, their logical relation is too weak in the sense that it fails in relating some contextually equivalent programs that are related by Pitts and Stark's [ZN03].

Instead, we shall show that, to define logical relations for lambda-calculi with cryptography, a better choice of category is $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$. However, this category is still lacking in some subtler aspects, and we eventually show that the proper category to consider is $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. These categories will be defined in Chapter 4 and Chapter 5. We then define a cryptographic logical relation over this category, which is proved sound. Logical relations derived over this category are also proved equivalent to Stark's denotational logical relations. Furthermore, we show that the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ is also the right category for defining a proper notion of contextual equivalence for cryptographic protocols.

Completeness is an important concern of logical relations. In general, logical relations are just complete for types up to first order, e.g., Pitts and Stark's operational logical relation. (Sumii and Pierce left this problem unsolved for their logical relations in the cryptographic lambda-calculus. To get the completeness, they switch to the notion of bisimulation [SP04, SP05].) The introduction of monadic types into typed lambda-calculi makes completeness more difficult to achieve. An interesting point is that, in the case of dynamic key generation, if there is no restriction on monadic types, logical relations are not even complete for zero-order types. But this is not specific to name creation monad. Some other monads like the non-determinism monad have similar problems (see Appendix B for details). This is probably because contexts always "flatten" programs to get values that they are able to compare, and information about program structure, which is usually necessary for building logical relations, might be lost during this procedure.

With a restriction on monadic types, we show that logical relations derived over $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ are complete for a certain subset of first-order types. To extend the completeness result to higher-order types, we switch to the notion of *lax logical relations* [PPST00], but the cost is that lax logical relations are not constructed inductively any more. We finally define logical relations which are lax at function types and monadic types, but strict (non-lax) at various other types, still over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. We show that they are sound and complete for contextual equivalence at all types.

1.3 Lien avec d'autres travaux

The basic idea of the logical relation approach is that of formalizing and proving secrecy as equivalences between different instances of a program with secret values. This is usually known as the non-interference approach, which is very popular both in the security community and in the programming language community [RS99, SM03]. One of the main goals of this approach, as Sumii and Pierce claimed, is "to explore how standard techniques for reasoning about type abstraction can be adapted to the task of reasoning about encryption, in particular about security protocols" [SP01]. For this purpose, they choose lambda-calculus as a starting point.

Indeed, various techniques from programming languages have been applied in the (static) analysis of computer security. Some work in this line includes the Spi-calculus (by Abadi and Gordon) [AG97, AG99] and the applied pi-calculus (by Abadi and Fournet) [AF01], which are both based on the pi-calculus and come equipped with useful techniques such as bisimulation for proving behavioral equivalences [AG98]. Also based on the pi-calculus, Gordon and Jeffrey also develop a system called Cryptyc — a type-effect system aiming at proving authenticity through static type checking [GJ02, GJ03a, GJ03b, GJ04]. Another example is Heintze and Riecke's SLam calculus [HR98]. They proposed lambda-calculus with type-based information flow control, and proved a non-interference property — that a value of high security does not leak to contexts of low security — using logical relations.

In addition to logical relations, bisimulations are a main technique for proving contextual equivalence. Bisimulations were originally designed for process calculi and transition systems [Mil80, Mil89, Mil99] and have been studied extensively. In particular, various notions of bisimulation have been proposed in cryptographic process calculi [AG98, BNP99, AF01, BN02]. These are called *environment-sensitive* bisimulations by Borgström and Nestmann [BN02], to stress the fact they explicitly take into account the *knowledge* about a process. However, there is no completeness proof available for these bisimulations: they are either proved incomplete [AG98], or just proved complete for a subset of processes [BNP99].

The notion of bisimulation has also been adapted by Abramsky in lambda-calculi [Abr90]. He calls the adapted notion *applicative bisimulation* and uses it to prove contextual equivalence in untyped lambda-calculus. This was later used by Sumii and Pierce to prove the contextual equivalence in a language called $\lambda_{\texttt{seal}}$, which is basically the untyped version of their cryptographic lambda-calculus [SP04]. Gordon and Rees adapted applicative bisimulations to typed calculi [Gor99, Gor98, GR96], where objects, subtyping, universal polymorphism and recursive types are present, but not existential types. Sumii and Pierce then introduced a notion of bisimulations *annotated with type information* and used it to prove the contextual equivalence in a typed lambda-calculus with full universal, existential, and recursive types [SP05]. These applicative bisimulations are shown to be sound and complete w.r.t. contextual equivalence. However, they are still syntactic (even more than syntactic logical relations) and rely heavily on the operational semantics (e.g., Sumii and Pierce's bisimulations are defined based on a big-step style evaluation), hence are not easy to extend. Furthermore, there is no known mathematical theory supporting applicative bisimulations, as those for logical relations [MS93, GLLN02]. Note that (typed) applicative bisimulations are not essentially logical relations — they have different definitions on relating functions. Furthermore, the point of applicative bisimulations is the ability of dealing with recursions since they are usually built co-inductively.

1.4 Plan de la thèse

The main content (from Chapter 2 to Chapter 7) of this thesis can be divided into the specification part and the verification part.

Specification: This part, consisting of Chapter 2 and Chapter 3, defines a formal model for describing protocols. Precisely, this is a language called the *cryptographic metalanguage*, extended from Moggi's computational lambda-calculus with cryptographic primitives. The syntax of this language is defined in Chapter 2. This language is shown to be adequate for describing cryptographic protocols, through encodings of concrete protocols. Then Chapter 3 defines a denotational semantics of the language. This is based on the category $\mathcal{S}et^{\mathcal{I}}$, which has been used by Stark to interpret the nu-calculus [Sta96]. We are interested in the precise definition of contextual equivalence, since the secrecy property is represented by this notion. In particular, we shall define the contextual equivalence for the cryptographic metalanguage, in the denotational model, so that our reasoning will rest totally on the denotational model. At the end of Chapter 3, we discuss what should be the right notion of contextual equivalence for the cryptographic metalanguage.

Verification: Chapter 4–7 form a verification part. We shall use the technique of logical relations as the verification tool to prove contextual equivalence. Chapter 4 and Chapter 5 are about the construction of logical relations for the cryptographic metalanguage over categories. Constructions of logical relations for lambda-calculus are standard. The metalanguage inherits in particular the monadic type constructor from the computational lambda-calculus and logical relations for this kind of types can be defined (over the category $\mathcal{S}et^{\mathcal{I}}$) by following the general construction of logical relations for monadic types, proposed by Goubault-Larrecq et al. [GLLN02]. However, it is noticed that the category $\mathcal{S}et^{\mathcal{I}}$ is not sufficient for studying relations between programs in the metalanguage. For this purpose, a better category to consider is the category $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$, and eventually the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. We apply to both categories the general construction by Goubault-Larrecq et al.. We finally define a cryptographic logical relation based on the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ and prove some important properties.

We then define at the beginning of Chapter 6 the right notion of contextual equivalence, also over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. The cryptographic logical relation is proved sound, but the completeness for logical relations derived over $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ is hard to achieve, even for first-order types. We prove the completeness for a certain subset of first-order types. Then, using the notion of lax logical relation, we define a complete logical relation, which is lax at function types and monadic types. We show that it is sound and complete for all types.

Lastly, in Chapter 7, decidability of the cryptographic logical relation is investigated in certain cases. In particular, relations for types $\mathsf{T}\tau$ and $\mathsf{key} \to \tau$ are shown to be decidable if the relations for type τ are decidable.

Chapter 8 concludes the thesis by summarizing the results and discussing directions for future work.

Chapitre 2

Le métalangage cryptographique

Pour spécifier les protocoles cryptographiques, nous avons actuellement plusieurs langages formels qui permettent de formaliser des propriétés de sécurité par la notion d'équivalence contextuelle, notamment la propriété de secret. Le langage plus connu est probablement le Spi-calcul d'Abadi et Gordon [AG99]. Un autre langage moins connu est le lambda-calcul cryptographique proposé par Sumii et Pierce [SP01, SP03], qui est une extension du lambda-calcul simplement typé avec deux sortes de primitives : les primitives cryptographiques et les primitives de génération de clés. La génération de nouvelles clés (ou nonces) est un mécanisme crucial des protocoles cryptographiques et est une construction de base du π-calcul, mais pas dans le lambda-calcul. Il y a déjà eu des études sur ce mécanisme dans le cadre du lambda-calcul et le plus connu est probablement le *nu-calcul* de Pitts et Stark [PS93a, Sta94]. Le nu-calcul est une extension du lambda-calcul simplement typé avec génération dynamique de noms (que nous verrons ici comme des clés) et s'occupe de l'étude des noms en programmation.

Le codage des protocoles dans le lambda-calcul cryptographique s'effectue comme dans le Spi-calcul : un protocole est codé comme n-uplet des fonctions qui représentent les participants, sauf que dans le Spi-calcul ils sont codés comme des processus en parallèle au lieu de fonctions. La différence remarquable entre les deux modèles est que, dans le lambda-calcul, l'interaction entre les participants est modélisée par l'application de fonctions, pas par les communications sur les canaux comme dans le Spi-calcul. Un attaquant dans le lambda-calcul est alors une fonction qui prend le protocole en paramètre et qui essaye de découvrir le secret.

Nous commençons ce chapitre par une introduction brève aux lambda-calculs typés, dans la partie 2.1. En particulier, nous introduisons des langages fondés sur le lambda-calcul, par exemple le nu-calcul de Pitts et Stark, le lambda-calcul cryptographique de Sumii et Pierce, et le lambda-calcul computationnel de Moggi. Nous définissons ensuite dans la partie 2.2 un langage appelé le *métalangage cryptographique* pour spécifier les protocoles. Ce langage est étendu à partir d'une version spécifique du lambda-calcul computationnel (spécialisé pour la génération de noms) que Stark avait utilisé pour construire un modèle dénotationnel du nu-calcul. La partie 2.3 montre comment formaliser la propriété de secret en utilisant la notion d'équivalence contextuelle. La partie 2.4 expose le codage des protocoles dans le métalangage cryptographique sur deux exemples concrets — le protocole d'établissement de clés symétriques qui a été introduit dans le chapitre 1, et le protocole de Needham-Schroeder [NS78]. Le codage est basé sur la même idée que le codage dans le lambda-calcul cryptographique de Sumii et Pierce, mais nous clarifions leur méthode, en particulier notre codage est plus précis et représente vraiment les traces d'exécution des protocoles.

There are nowadays some formal languages for specifying cryptographic protocols, where security properties like secrecy are formalized by contextual equivalence. The most well-known one is probably Abadi and Gordon's *Spi-calculus* [AG99]. A less known language is Sumii and Pierce's *cryptographic lambda-calculus* [SP01, SP03]. This is an extension of the simply-typed lambda-calculus with two kinds of primitives: cryptographic primitives and the *key generation* primitive. Fresh key (or nonce) generation, which is crucial for cryptographic protocols, is native in the pi-calculus, but not in lambda-calculus. Studying this mechanism within the lambda-calculus framework is not new. The most well-known work on this aspect is probably Pitts and Stark's *nu-calculus* [PS93a, Sta94], which is a language extended from the simply-typed lambda-calculus with fresh name creation, and is devoted to the study of *names* in programming languages (names seen as syntactically identical to keys).

Encoding protocols in the cryptographic lambda-calculus is quite similar as in the Spi-calculus: protocols are encoded as tuples of functions representing principles, except that in the Spi-calculus, they are encoded as parallel processes instead of functions. A notable difference is that, in lambda-calculus, interactions between principles are modeled by function applications, not by communications over channels. An attacker in this model is encoded as a function that takes the protocol program as argument and attempts to reveal the secrets.

This chapter starts with a brief introduction to typed lambda-calculi, in Section 2.1. We give as examples some specific languages based on lambda-calculus, such as Pitts and Stark's nu-calculus, Sumii and Pierce's cryptographic lambda-calculus and Moggi's computational lambda-calculus. We then define in Section 2.2 a language called the *cryptographic metalanguage*, for specifying cryptographic protocols. This language is extended from a specific version of the computational lambda-calculus (specialized for fresh name creation) which has been used by Stark as an intermediate language to build a denotational model for the nu-calculus. Section 2.3 explains how the secrecy property is formalized by the notion of contextual equivalence and Section 2.4 illustrates the encoding of protocols in the cryptographic metalanguage, through two concrete examples — the key establishment protocol which has been introduced in Chapter 1 and Needham-Schroeder's public key protocol [NS78]. The encoding is based on the same idea of the encoding in the cryptographic lambda-calculus, but we clarify Sumii and Pierce's methods by making our encoding based on the protocol execution traces.

2.1 Préliminaires

2.1.1 Le lambda-calcul simplement typé

Lambda-calculus has proved useful in describing and analyzing programming languages [Ten81, Win93, Mit96]. Pure lambda-calculus [Bar80] is a formal system where everything is a *function* and the two basic operations are *abstraction* and *application*: the first one is the way we write function expressions while the second allows us to use the functions we have defined. More formally, the syntax of the (untyped) lambda-calculus consists only of the following three sorts of terms:

$$\begin{array}{rcll} t & ::= & x & \text{variable} \\ & \mid & \lambda x.t & \text{abstraction} \\ & \mid & t\,t & \text{application} \end{array}$$

The lambda-abstraction $\lambda x.t$ binds the variable x in the term t. If a variable is not bound by the λ in a term, then it is a *free variable* of this term. We implicitly identify expressions which only differ in their choice of bound variables (α-equivalence).

In *typed* lambda calculus, terms are assigned types [Bar91]. In particular, we specify the domain of a function by giving a type to the formal argument, that is, if t is some well-formed expression under the assumption that the variable x has type τ, then $\lambda x^{\tau}.t$ defines the function mapping all x in τ to the value given by t. We shall omit the type annotation when the type information is clear from context.

A typed lambda calculus is usually defined based on a collection of *type constants* (or *base types*) and *term constants*. For instance, we may have a type constant for integers with some numerical operations. Such a collection of base types and term constants is called a *signature*, denoted by Σ. We use Σ_b to denote the collection of base types, and Σ_c for term constants. Complex types can be constructed from base types by type constructors. Notably, a typed lambda calculus should contain function types since functions are the very basic primitives of lambda calculus. We present here a simple typed lambda calculus where we have only base types in the signature Σ and one type constructor $\to$:

$$\tau ::= b \mid \tau \to \tau, \qquad b \in \Sigma_b$$

$\tau \to \tau'$ is the type of functions mapping a value of type τ to a value of type τ'. There often exist some other standard type constructors such as products and sums, which we shall see later in a more concrete language. Types may also have type variables, but in this thesis we consider only simply-typed lambda calculus, by which we refer to any version of typed lambda calculus where types do not contain type variables.

The collection of terms then contains term constants and expressions built from these constants using abstraction and application:

$$t ::= x \mid c \mid \lambda x.t \mid t\,t$$

where c is a term constant in the signature Σ. Every term constant in Σ is associated with a unique type. For instance, if we have a base type nat for natural numbers, then a constant $+$ for numerical addition is of type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$. We write $c : \tau \in \Sigma$ if c is a term constant of type τ.

Terms are assigned types via typing assertions and a set of typing rules. Typing assertions are of the form $\Gamma \vdash t : \tau$, where Γ is a typing context — a finite set of typed variables $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ where no x_i ($1 \leq i \leq n$) occurs twice. We write $\Gamma, x : \tau$ for the typing context $\Gamma \cup \{x : \tau\}$. A typing assertion $\Gamma \vdash t : \tau$ says that if variables $x_1, \ldots, x_n$ have types $\tau_1, \ldots, \tau_n$ respectively, then the expression t has type τ.

Types of compound terms are determined through typing rules. A typical typing rule is of the form

$$\frac{\Gamma_1 \vdash t_1 : \tau_1 \quad \ldots \quad \Gamma_n \vdash t_n : \tau_n}{\Gamma \vdash t : \tau}.$$

Intuitively, this rule says that if the typing assertions $\Gamma_1 \vdash t_1 : \tau_1,\ \ldots,\ \Gamma_n \vdash t_n : \tau_n$ hold, then $\Gamma \vdash t : \tau$ holds as well. Notably, the typing rules for variables, abstraction and application in the simply-typed lambda are

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\,(\mathit{Var}) \qquad\qquad \frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau}\,(\mathit{Const})$$

$$\frac{\Gamma, x : \tau \vdash t : \tau'}{\Gamma \vdash \lambda x.t : \tau \to \tau'}\,(\mathit{Abs}) \qquad \frac{\Gamma \vdash t_1 : \tau \to \tau' \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 t_2 : \tau'}\,(\mathit{App})$$

Note that sometimes rules for constants appear in another form. For example, the typing rule for the numerical addition may appear as

$$\frac{\Gamma \vdash t_1 : \mathsf{nat} \quad \Gamma \vdash t_2 : \mathsf{nat}}{\Gamma \vdash t_1 + t_2 : \mathsf{nat}}\,(\mathit{Add})$$

2.1.2 Le nu-calcul et le lambda-calcul cryptographique

While the standard simply-typed lambda calculus is a simple and powerful mathematical model, people usually extend it with more constants and type constructors to describe and analyze complex data and features in real programming languages. The *nu-calculus* [PS93a] and the *cryptographic lambda-calculus* [SP03] are such extensions.

The nu-calculus is a language aiming at the study of dynamic name creation in (functional) programming languages. The nu-calculus has in particular a base type name for *names*, which, not like variables, can be created freshly, compared with others, and passed around. Creating a fresh name in a nu-calculus expression is defined by $\nu n.t$. This term binds the name n in the term t.

Typing assertions are slightly different from those in standard lambda-calculus. A typing assertion in the nu-calculus is of the form $s; \Gamma \vdash t : \tau$, where Γ is a set of typed variables as in standard typed lambda-calculus, and s is a finite set of names, containing all free names in t. The typing rule for the fresh name creation is

$$\frac{s \cup \{n\}; \Gamma \vdash t : \tau}{s; \Gamma \vdash \nu n.t : \tau}\ (New),$$

where $n \notin s$.

The cryptographic lambda-calculus has more primitives than the nu-calculus, in particular those related to cryptography. The two main cryptographic primitives are the encryption — $\{t_1\}_{t_2}$, and the decryption — $\texttt{let}\ \{x\}_{t_1} = t_2\ \texttt{in}\ t_3\ \texttt{else}\ t_4$. The meaning of the encryption is clear. In the decryption expression, the key obtained from t_1 is used to decrypt the cipher-text obtained from t_2. If the decryption succeeds, it binds the plain-text to the variable x and computes t_3, otherwise it computes t_4 directly. There is also fresh key generation, seen syntactically identical as name creation. And the type name in the nu-calculus naturally becomes the type key. In fact, the key type in the cryptographic lambda-calculus is associated with another type τ and $\mathsf{key}[\tau]$ is the type for those keys which can be used to encrypt messages of type τ, but a more natural way is to use a uniform type key. Cipher-texts are of type $\mathsf{bits}[\tau]$, where τ is the type of the corresponding plain-texts. Typing rules of encryption and decryption are as follows

$$\frac{s; \Gamma \vdash t_1 : \tau \quad s; \Gamma \vdash t_2 : \mathsf{key}}{s; \Gamma \vdash \{t_1\}_{t_2} : \mathsf{bits}[\tau]}\ (Enc)$$

$$\frac{s; \Gamma \vdash t_1 : \mathsf{key} \quad s; \Gamma \vdash t_2 : \mathsf{bits}[\tau] \quad s; \Gamma, x : \tau \vdash t_3 : \tau' \quad s; \Gamma \vdash t_4 : \tau'}{s; \Gamma \vdash \texttt{let}\ \{x\}_{t_1} = t_2\ \texttt{in}\ t_3\ \texttt{else}\ t_4 : \tau'}\ (Dec)$$

Operational semantics in a "big-step" style are both defined in the nu-calculus and the cryptographic lambda-calculus:

$$s; \Gamma \vdash t \Downarrow_{\tau}^{s'} u,$$

where s' is a set of fresh names (keys) and $s \cap s' = \emptyset$. This means that in the presence of s and Γ, the term t can be reduced to the canonical term u (a canonical term is either a constant, a variable, a function, a name (key) or a cipher-text), with a set s' of fresh names (keys) being generated during the reduction. For instance, $\emptyset \vdash \nu n.n \Downarrow_{\mathsf{name}}^{\{n\}} n$.

2.1.3 Le lambda-calcul computationnel

Another example of simply-typed lambda-calculus is Moggi's computational lambda-calculus [Mog89, Mog91], which can be used to define a wide range of notions of *computations* or *side-effects*, e.g., exceptions, non-determinism, continuations, etc.. The computational lambda-calculus has in particular a unary type constructor T:

$$\tau ::= \ldots \mid \mathsf{T}\tau.$$

Elements of $\mathsf{T}\tau$ are those computations of type τ. The real contents of $\mathsf{T}\tau$ vary largely for different forms of computations. For instance, when exceptions are concerned, a computation may abort when an exceptions is raised. $\mathsf{T}\tau$ then contains a set of exceptions besides the regular values of τ. While in non-determinism, a computation might return one of several possible values. The result is not deterministic, so an element of $\mathsf{T}\tau$ is actually a set of values.

There are two special constants in the computational lambda-calculus that are useful in expressing most computations:

$$t ::= \ldots \mid \mathtt{val}(t) \mid \mathtt{let}\ x \Leftarrow t\ \mathtt{in}\ t.$$

$\mathtt{val}(t)$ denotes the trivial computation which does nothing but returns t as a value. The $\mathtt{let}$ construction is a sequential computation: $\mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2$ first executes the computation t_1, binds the result to x and then executes the computation t_2. x is a bound variable in t_2. Typing rules for these two constants are

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathtt{val}(t) : \mathsf{T}\tau}\ (\mathit{Val}) \qquad \frac{\Gamma \vdash t_1 : \mathsf{T}\tau \quad \Gamma, x : \tau \vdash t_2 : \mathsf{T}\tau'}{\Gamma \vdash \mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2 : \tau'}\ (\mathit{Let})$$

Moggi's calculus provides a general framework for describing various forms of computation. When dealing with specific computations, we usually need specific constants. For example, in exceptions, it is necessary to have a constant $\mathtt{raise}_\tau(e)$, for raising an exception during a computation; while for describing non-determinism, we shall need a constant like $\mathtt{select}_\tau(t_1, t_2)$, which chooses randomly a value from the two values t_1 and t_2 of type τ.

Dynamic name creation or key generation is another concrete notion of computation. Stark has shown that the nu-calculus can be interpreted in the computational lambda-calculus specialized in name creation [Sta94]. For this, a constant $\mathtt{new}$ for fresh name creation is necessary. $\mathtt{new}$ is of type $\mathsf{T}\mathsf{name}$ (or $\mathsf{T}\mathsf{key}$). It is indeed a computation which generates a fresh name and returns this name as the value. A term of type τ in the nu-calculus is then interpreted as a term of $\mathsf{T}\tau$. For example, the term $\nu n.n$ is interpreted as $\mathtt{let}\ n \Leftarrow \mathtt{new}\ \mathtt{in}\ \mathtt{val}(n)$. In particular, a canonical term u is always interpreted as $\mathtt{val}(u)$.

2.2 Le métalangage cryptographique

In order to make use of the general framework of deriving logical relations for monadic types [GLLN02] and approach quickly to the heart of this thesis, we rely on the computational lambda-calculus and define in this section a cryptographic metalanguage language for specifying cryptographic protocols. This is based on Stark's computational metalanguage — the computational lambda-calculus specialized in dynamic name creation and we extend Stark's language with some cryptographic primitives.

Syntaxe

The types of the cryptographic metalanguage are defined by the following grammar:

$$\tau ::= \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{key} \mid \mathsf{msg} \mid \tau \to \tau \mid \tau \times \tau \mid \mathsf{opt}[\tau] \mid \mathsf{T}\tau$$

There are four base types: bool for booleans, nat for integers, key for keys and msg for messages. $\tau \times \tau'$ is the type for cartesian products, where τ and τ' are types of the two components respectively. $\mathsf{opt}[\tau]$ is an option type for τ, which contains a distinguished element besides values of τ, denoting failures of message operations, notably the decryption failure when a wrong key is used. T is the unary type constructor for computation types as in the computational lambda-calculus. In this metalanguage, the difference between computations and values is that a computation may generate some keys before returning a value.

We also define the order and the computation degree of types. The order of a type is defined by

$$\begin{array}{rcl}
\mathtt{ord}(b) & = & 0, \quad \text{for all } b \in \Sigma, \\
\mathtt{ord}(\mathsf{T}\tau) & = & \mathtt{ord}(\tau), \\
\mathtt{ord}(\tau \to \tau') & = & \max(\mathtt{ord}(\tau) + 1, \mathtt{ord}(\tau')), \\
\mathtt{ord}(\tau \times \tau') & = & \max(\mathtt{ord}(\tau), \mathtt{ord}(\tau')), \\
\mathtt{ord}(\mathsf{opt}[\tau]) & = & \mathtt{ord}(\tau).
\end{array}$$

For every well-typed term t of type τ, $\mathtt{ord}(t) = \mathtt{ord}(\tau)$. The computation type constructor T does not change the type order. When we say first-order types, we usually mean types up to first order. The computation degree of a type is defined by

$$\begin{array}{rcl}
\deg(b) & = & 0, \quad \text{for all } b \in \Sigma, \\
\deg(\mathsf{T}\tau) & = & \deg(\tau) + 1, \\
\deg(\tau \to \tau') & = & \max(\deg(\tau), \deg(\tau')), \\
\deg(\tau \times \tau') & = & \max(\deg(\tau), \deg(\tau')), \\
\deg(\mathsf{opt}[\tau]) & = & \deg(\tau).
\end{array}$$

For every well-typed term t of type τ, $\deg(t) = \deg(\tau)$.

$$\frac{x:\tau\in\Gamma}{\Gamma\vdash x:\tau}\,(\mathit{Var})\qquad\frac{}{\Gamma\vdash\mathtt{true}:\mathsf{bool}}\,(\mathit{True})\qquad\frac{}{\Gamma\vdash\mathtt{false}:\mathsf{bool}}\,(\mathit{False})$$

$$\frac{\Gamma\vdash t_1:\mathsf{bool},\quad\Gamma\vdash t_2:\tau,\quad\Gamma\vdash t_3:\tau}{\Gamma\vdash\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3:\tau}\,(\mathit{Cond})$$

$$\frac{}{\Gamma\vdash i:\mathsf{nat}}\,(\mathit{Int})\qquad\frac{\Gamma\vdash t_1:\mathsf{nat}\quad\ldots\quad\Gamma\vdash t_n:\mathsf{nat}}{\Gamma\vdash\mathtt{nat\_op}_n(t_1,\ldots,t_n):\mathsf{nat}}\,(\mathit{Nat\_Op})$$

$$\frac{\Gamma,x:\tau\vdash t:\tau'}{\Gamma\vdash\lambda x.t:\tau\to\tau'}\,(\mathit{Abs})\qquad\frac{\Gamma\vdash t_1:\tau\to\tau',\quad\Gamma\vdash t_2:\tau}{\Gamma\vdash t_1t_2:\tau'}\,(\mathit{App})$$

$$\frac{\Gamma\vdash t_1:\tau_1,\quad\Gamma\vdash t_2:\tau_2}{\Gamma\vdash\langle t_1,t_2\rangle:\tau_1\times\tau_2}\,(\mathit{Pair})\qquad\frac{\Gamma\vdash t:\tau_1\times\tau_2}{\Gamma\vdash\mathtt{proj}_i(t):\tau_i}\,i=1,2\ (\mathit{Proj})$$

$$\frac{}{\Gamma\vdash\mathtt{new}:\mathsf{Tkey}}\,(\mathit{New})\qquad\frac{\Gamma\vdash t:\tau}{\Gamma\vdash\mathtt{some}(t):\mathsf{opt}[\tau]}\,(\mathit{Inj})$$

$$\frac{\Gamma\vdash t_1:\mathsf{opt}[\tau],\quad\Gamma,x:\tau\vdash t_2:\tau'\quad\Gamma\vdash t_3:\tau'}{\Gamma\vdash\mathtt{case}\ t_1\ \mathtt{of}\ \mathtt{some}(x)\ \mathtt{in}\ t_2\ \mathtt{else}\ t_3:\tau'}\,(\mathit{Case})$$

$$\frac{\Gamma\vdash t:\tau}{\Gamma\vdash\mathtt{val}(t):\mathsf{T}\tau}\,(\mathit{Val})\qquad\frac{\Gamma\vdash t_1:\mathsf{T}\tau,\quad\Gamma,x:\tau\vdash t_2:\mathsf{T}\tau'}{\Gamma\vdash\mathtt{let}\ x\Leftarrow t_1\ \mathtt{in}\ t_2:\mathsf{T}\tau'}\,(\mathit{Let})$$

$$\frac{\Gamma\vdash t_1:\mathsf{msg}\quad\Gamma\vdash t_2:\mathsf{key}}{\Gamma\vdash\mathtt{enc}(t_1,t_2):\mathsf{msg}}\,(M.enc)\qquad\frac{\Gamma\vdash t_1:\mathsf{msg}\quad\Gamma\vdash t_2:\mathsf{key}}{\Gamma\vdash\mathtt{dec}(t_1,t_2):\mathsf{opt}[\mathsf{msg}]}\,(M.dec)$$

$$\frac{\Gamma\vdash t_1:\mathsf{msg}\quad\Gamma\vdash t_2:\mathsf{msg}}{\Gamma\vdash\mathtt{p}(t_1,t_2):\mathsf{msg}}\,(M.pair)$$

$$\frac{\Gamma\vdash t:\mathsf{msg}}{\Gamma\vdash\mathtt{fst}(t):\mathsf{opt}[\mathsf{msg}]}\,(M.fst)\qquad\frac{\Gamma\vdash t:\mathsf{msg}}{\Gamma\vdash\mathtt{snd}(t):\mathsf{opt}[\mathsf{msg}]}\,(M.snd)$$

$$\frac{\Gamma\vdash t:\mathsf{nat}}{\Gamma\vdash\mathtt{n}(t):\mathsf{msg}}\,(M.r)\qquad\frac{\Gamma\vdash t:\mathsf{msg}}{\Gamma\vdash\mathtt{getnum}(t):\mathsf{opt}[\mathsf{nat}]}\,(M.mr)$$

$$\frac{\Gamma\vdash t:\mathsf{key}}{\Gamma\vdash\mathtt{k}(t):\mathsf{msg}}\,(M.k)\qquad\frac{\Gamma\vdash t:\mathsf{msg}}{\Gamma\vdash\mathtt{getkey}(t):\mathsf{opt}[\mathsf{key}]}\,(M.mk)$$

Figure 2.1: Typing rules for the metalanguage

Expressions in the cryptographic metalanguage are defined by

t	$::=$	x	variables
	$\mid$	$\mathtt{true} \mid \mathtt{false}$	boolean values
	$\mid$	$\mathtt{if}\ t\ \mathtt{then}\ t\ \mathtt{else}\ t$	conditional
	$\mid$	i	integer constants $0, 1, 2, \ldots$
	$\mid$	$\mathtt{nat\_op}_n(t, \cdots, t)$	integer operation
	$\mid$	$\lambda x.t$	abstraction
	$\mid$	$t\ t$	application
	$\mid$	$\langle t, t \rangle$	pairing
	$\mid$	$\mathtt{proj}_1(t) \mid \mathtt{proj}_2(t)$	projections
	$\mid$	$\mathtt{some}(t)$	option injection
	$\mid$	$\mathtt{case}\ t\ \mathtt{of}\ \mathtt{some}(x)\ \mathtt{in}\ t\ \mathtt{else}\ t$	option case
	$\mid$	$\mathtt{new}$	fresh key generation
	$\mid$	$\mathtt{val}(t)$	trivial computation
	$\mid$	$\mathtt{let}\ x \Leftarrow t\ \mathtt{in}\ t$	sequential computation
	$\mid$	$\mathtt{enc}(t, t)$	encryption
	$\mid$	$\mathtt{dec}(t, t)$	decryption
	$\mid$	$\mathtt{p}(t, t)$	pairing of messages
	$\mid$	$\mathtt{fst}(t) \mid \mathtt{snd}(t)$	message projection
	$\mid$	$\mathtt{n}(t) \mid \mathtt{getnum}(t)$	integer as/from messages
	$\mid$	$\mathtt{k}(t) \mid \mathtt{getkey}(t)$	keys as/from messages

Typing rules for these terms are given in Figure 2.1. Most rules are standard, such as pairing, projection, injection, case and so on. The option injection $\mathtt{some}(t)$ and the option case $\mathtt{case}\ t_1\ \mathtt{of}\ \mathtt{some}(x)\ \mathtt{in}\ t_2\ \mathtt{else}\ t_3$ are abbreviated versions of standard injection and case operations for sum types: $\mathsf{opt}[\tau]$ is seen as the sum type of $\tau + \mathsf{unit}$ with unit a type containing a single (dummy) value, $\mathtt{some}(\_)$ is the left injection and the option case abbreviates $\mathtt{case}\ t_1\ \mathtt{of}\ \mathtt{in}_1(x) \triangleright t_2;\ \mathtt{in}_2(\_) \triangleright t_3$. The constant $\mathtt{new}$ stands for fresh key generation, which generates a fresh key and returns this key as a value. It is a computation of type $T\mathsf{key}$.

There are several term constants related to the msg type. $\mathtt{enc}$ and $\mathtt{dec}$ are two primitives for encryption and decryption: $\mathtt{enc}(t_1, t_2)$ uses the key obtained from t_2 to encrypt the message obtained from t_1 and returns the encrypted message; $\mathtt{dec}(t_1, t_2)$ uses the key obtained from t_2 to decrypt the message obtained from t_1, and returns the corresponding plain-text (if the decryption succeeds) or an error (if the decryption fails), so it is of type $\mathsf{opt}[\mathsf{msg}]$. One can take an integer or a key as a message, by the operations $\mathtt{n}$ and $\mathtt{k}$. Constants $\mathtt{getnum}$ and $\mathtt{getkey}$ are the inverse operations, which attempt to retrieve an integer or a key from a message. Messages can also be paired by a particular pairing operation $\mathtt{p}$ for messages. The difference between message pairing and the normal pairing ($\langle\_,\_\rangle$) is that the pairing $\mathtt{p}$ of two messages is still of type msg, not $\mathsf{msg} \times \mathsf{msg}$, as shown by the typing rule $(M.pair)$. Correspondingly, we have projections $\mathtt{fst}$ and $\mathtt{snd}$ that return the components of a pairing message.

Compared with the cryptographic lambda-calculus, the main change in this metalanguage is on the treating of messages. Types for cipher-texts are no longer associated with the types of corresponding plain-texts as in Sumii and Pierce's language. Instead, we use a more uniform type msg and all cryptographic operations such as encryption and decryption can only be applied to messages. By doing so we are able to encode "typing attacks" in the metalanguage. A typing attack is an attack where attackers trick principles to accept some fake messages with confused types. For example, if a principle is waiting for an encrypted message which should be composed by two keys, e.g., $\{k_1, k_2\}_k$, then an attacker can send him a message $\{k_1, E\}_k$, where E is a principle name and is usually represented as an integer, e.g., an IP address. In practice, every messages is just a bit string. If both principle names and keys are bit strings of the same length, when the principle receives the message $\{k_1, E\}_k$, he has no way to tell that E is not a key. This will become vital if later they use E as a key to exchange secrets. Type flaw does exist in real protocols [CJ97, HLS03]. To represent these attacks in Sumii and Pierce's language, we have to introduce dynamic type checking, because the fake message is not of the expected type and will be rejected by static type checking (for instance, the message $\{k_1, E\}_k$ is of type $\mathsf{bits}[\mathsf{key} \times \mathsf{nat}]$ while a message of type $\mathsf{bits}[\mathsf{key} \times \mathsf{key}]$ is expected). but in the cryptographic metalanguage, we can do this with no cost (messages are all of type msg).

Furthermore, because msg is a base type, the language can be easily extended with other cryptographic primitives with particular algebraic properties [CDL05], e.g., in RSA encoding [RSA78], encryption $\mathtt{enc}$ is implemented as modular exponentiation, which obeys various associativity, commutativity and distributivity laws. To give an example that remains in the framework of symmetric encryption, DES [DES] obeys the property that $\mathtt{not}\ \mathtt{enc}(v, k) = \mathtt{enc}(\mathtt{not}\ v, \mathtt{not}\ k)$, where $\mathtt{not}$ is bitwise logical not.

In Figure 2.2 are syntax abbreviations which we shall use in the sequel.

Equational semantics

Stark defines an equational logic of Horn clauses for reasoning about terms of his computational metalanguage [Sta94]. If the typing assertions $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \tau$ hold, then $\Gamma \vdash t_1 = t_2 : \tau$ is an *equation in context* Γ. We shall omit the type and write $\Gamma \vdash t_1 = t_2$. A *sequent* is a judgement $\Gamma; \Phi \vdash \phi$ where Γ is a typing context, Φ is a finite set of equations in context Γ and ϕ is a single equation in context Γ. We write $\Phi, t_1 = t_2$ for $\Phi \cup \{t_1 = t_2 : \tau\}$ and we may omit Φ when it is empty.

Rules for deriving sequents are also given in [Sta94] for the computational metalanguage for

$$
\begin{array}{rcl}
\{t_1\}_{t_2} & \equiv & \mathtt{enc}(t_1, t_2). \\
[t_1, t_2, \ldots, t_n] & \equiv & \mathtt{p}(t_1, \ldots, \mathtt{p}(t_{n-1}, t_n)\cdots). \\
\pi_i^n(t) & \equiv & \begin{cases} \mathtt{fst}(\underbrace{\mathtt{snd}\cdots\mathtt{snd}}_{i-1}(t)\cdots) & \text{if } i < n, \\ \underbrace{\mathtt{snd}(\cdots\mathtt{snd}}_{i-1}(t)\cdots) & \text{if } i = n. \end{cases} \\
\langle t_1, t_2, \ldots, t_n\rangle & \equiv & \langle t_1, \ldots, \langle t_{n-1}, t_n\rangle\cdots\rangle. \\
\Pi_i^n(t) & \equiv & \begin{cases} \mathtt{proj}_1(\underbrace{\mathtt{proj}_2\cdots\mathtt{proj}_2}_{i-1}(t)\cdots) & \text{if } i < n, \\ \underbrace{\mathtt{proj}_2(\cdots\mathtt{proj}_2}_{i-1}(t)\cdots) & \text{if } i = n. \end{cases} \\
\mathtt{error} & \equiv & \mathtt{dec}(\{1\}_{k_1}, k_2), \qquad (k_1 \neq k_2). \\
\mathtt{letopt}\ x \Leftarrow t_1\ \mathtt{in}\ t_2 & \equiv & \mathtt{case}\ t_1\ \mathtt{of}\ \mathtt{some}(x)\ \mathtt{in}\ t_2\ \mathtt{else}\ \mathtt{error}, \\
 & & \qquad \text{where } t_2 \text{ must be of type } \mathsf{opt}[\mathsf{msg}]. \\
\lambda\langle x_1, \ldots, x_n\rangle.t & \equiv & \lambda y.(\lambda x_1.\cdots\lambda x_n.t)\mathtt{proj}_1^n(y)\cdots\mathtt{proj}_n^n(y). \\
\lambda\{x\}_k.t & \equiv & \lambda y.\mathtt{letopt}\ x \Leftarrow \mathtt{dec}(y, k)\ \mathtt{in}\ t. \\
\lambda\{x_1, \ldots, x_n\}_k.t & \equiv & \lambda\{y\}_k.(\lambda x_1.\cdots\lambda x_n.t)\mathtt{proj}_1^n(y)\cdots\mathtt{proj}_n^n(y). \\
\nu(x_1, \ldots, x_n).t & \equiv & \mathtt{let}\ x_1 \Leftarrow \mathtt{new}\ \mathtt{in}\cdots\mathtt{let}\ x_n \Leftarrow \mathtt{new}\ \mathtt{in}\ \mathtt{val}(t). \\
\mathtt{let}\ \langle x_1, \ldots, x_n\rangle \Leftarrow t\ \mathtt{in}\ t' & \equiv & \mathtt{let}\ y \Leftarrow t\ \mathtt{in} \\
 & & \qquad (\lambda x_1.\cdots\lambda x_n.t)\mathtt{proj}_1^n(y)\cdots\mathtt{proj}_n^n(y).
\end{array}
$$

Figure 2.2: Syntax abbreviations

name creation. In particular, rules for computations are as follows:

$$\frac{\Gamma \vdash t : \mathsf{T}\tau}{\Gamma \vdash \mathtt{let}\ x \Leftarrow t\ \mathtt{in}\ \mathtt{val}(x) = t} \qquad \frac{\Gamma \vdash t_1, t_2 : \tau}{\Gamma; \mathtt{val}(t_1) = \mathtt{val}(t_2) \vdash t_1 = t_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : \tau \vdash t_2 : \mathsf{T}\tau_2}{\Gamma \vdash \mathtt{let}\ x \Leftarrow \mathtt{val}(t_1)\ \mathtt{in}\ t_2 = t_2[t_1/x]}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{T}\tau_1 \quad \Gamma, x_1 : \tau \vdash t_2 : \mathsf{T}\tau_2 \quad \Gamma, x_2 : \tau \vdash t_3 : \mathsf{T}\tau_3}{\Gamma \vdash \mathtt{let}\ x_2 \Leftarrow (\mathtt{let}\ x_1 \Leftarrow t_1\ \mathtt{in}\ t_2)\ \mathtt{in}\ t_3 = \mathtt{let}\ x_1 \Leftarrow t_1\ \mathtt{in}\ \mathtt{let}\ x_2 \Leftarrow t_2\ \mathtt{in}\ t_3}.$$

Because the cryptographic metalanguage is an extension of Stark's computational metalanguage, the equational logic for the computational metalanguage can be easily extended here for giving the semantics of the cryptographic metalanguage and reasoning about terms. The detailed rules for deriving sequents are given in Appendix A.

2.3 Formalisation de la propriété de secret

Protocols usually involve several principles running in parallel and interacting with each other. It is somehow surprising at first glance that they can be encoded in lambda-calculus, which is natively sequential. The basic idea is that every principle is encoded as a function and interactions between principles can be modeled by function applications. Consider the following naive protocol:

$$\begin{array}{lll} \text{Message } 1\,. & A \to B : & \{i\}_k \\ \text{Message } 2\,. & B \to A : & i \bmod 2 \end{array}$$

In this protocol, k is a secret key shared only by A and B (to guarantee this, we let k be *freshly* generated in the program). A encrypts an integer i with k and sends it to B (Message 1), then B answers by the value $i \bmod 2$. The protocol is encoded as:

$$\begin{array}{rcl} p(i) & = & \mathtt{let}\ k \Leftarrow \mathtt{new}\ \mathtt{in} \\ & & \quad \mathtt{val}(\langle \mathtt{enc}(i,k), \\ & & \qquad \lambda x.\ \mathtt{case}\ \mathtt{dec}(x,k)\ \mathtt{of}\ \mathtt{some}(y)\ \mathtt{in}\ (y\ \mathtt{mod}\ 2)\ \mathtt{else}\ -1\rangle) \end{array}$$

whose type is $\mathsf{T}(\mathsf{msg} \times (\mathsf{msg} \to \mathsf{nat}))$. The first component of the pair, seen as a constant function $\lambda\_.\mathtt{enc}(i,k)$, represents the principle A. The second component represents the principle B, a function which receives a message and returns another message.

Interactions between principles are scheduled by the network and every scheduler is encoded as a function taking the protocol program as an argument, in the same language. A scheduler has full control of the network and it is possibly malicious and tries to attack protocols. A protocol is a sequence of message exchanging operations and the source and destination of every message are explicitly defined, so a "good" scheduler simply forwards every message to its intended receiver, and it is easy to check whether the system accomplishes its goal under such a "good" scheduler. The main property that we like to prove for a protocol is the secrecy property: the secret does not leak under any schedule. This difficult to prove directly — enumerating all possible schedulers is not possible. Suppose a protocol is secure in the sense that no execution of this protocol leaks the secret to attackers. Then it is clear that attackers cannot see the difference between any two instances of the protocol, hence are not able to distinguish them. For example, in the above protocol, if the principle A only sends encrypted even numbers to B, then for any secret integers i and j, the two instances $p(2 * i)$ and $p(2 * j)$ are equivalent, since the reply from B is always 0.

Such an indistinguishable property can be formalized by a notion in programming languages, called *contextual equivalence*, a.k.a. *observational equivalence*. We say that two programs are contextually equivalent, if we always get the same results when running them in all contexts. Given a protocol, consider its encoding in the metalanguage, i.e., a program with the secret message as an argument. If we can prove that two instances of this program are contextually equivalent, then we can assert that this protocol satisfies the secrecy property. Note that this requires that both contexts (attackers) and protocols are encoded in the same language.

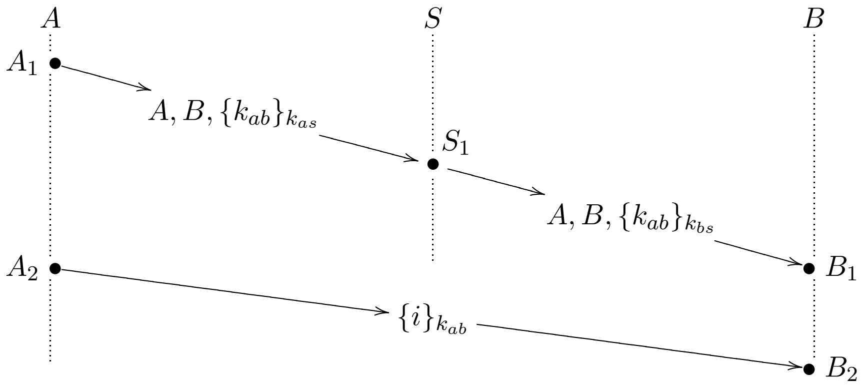
The rest of our work is to encode protocols in the cryptographic metalanguage and to develop methods for proving contextual equivalences. In lambda-calculus, there are standard definitions of contextual equivalences [Mor68, PS93a]. However, it turns out that contextual equivalence for cryptographic protocols is a very subtle notion, especially when it is defined in a denotational way. Standard definitions must be adapted carefully in this case. In particular, contexts are supposed to represent honestly the power of attackers. We shall have more discussion on this at the end of Chapter 3 and in the Chapter 6.

2.4 Codage des protocoles

Following the scheme given in last section, we can encode concrete protocols in the cryptographic metalanguage. Two examples are given in this section. Both are flawed protocols, and we show that these protocols (and their corrected versions), as well as attacks, can be encoded in the metalanguage.

2.4.1 Un protocole de l'échange de clés symétriques

The first example is the protocol that we have introduced in Chapter 1, which aims at establishing a fresh symmetric key between two principles, with the help of a trusted key server S. It uses only the symmetric encryption.



$$
\begin{array}{lcl}
f_{A_1} & \equiv & \langle[\mathtt{n}(A), \mathtt{n}(B), \{\mathtt{k}(k_{ab})\}_{k_{as}}],\ f_{A_2}\rangle \\
f_{A_2} & \equiv & \{i\}_{k_{ab}} \\
f_{S_1} & \equiv & \lambda x.\mathtt{letopt}\ x_a \Leftarrow \mathtt{getnum}(\pi_1^3(x))\ \mathtt{in} \\
 & & \quad \mathtt{letopt}\ x_b \Leftarrow \mathtt{getnum}(\pi_2^3(x))\ \mathtt{in} \\
 & & \quad \mathtt{letopt}\ k \Leftarrow \mathtt{dec}(\pi_3^3(x), \mathtt{K}(x_a, s))\ \mathtt{in} \\
 & & \quad \mathtt{some}([\mathtt{n}(x_a), \mathtt{n}(x_b), \{k\}_{\mathtt{K}(x_b,s)}]) \\
f_{B_1} & \equiv & \lambda x.\langle *, f_{B_2}\rangle \\
f_{B_2} & \equiv & \lambda x.*
\end{array}
$$

where $\mathtt{K}(x, s)$ abbreviates

$$\mathtt{if}\ x = A\ \mathtt{then}\ k_{as}\ \mathtt{else}\ \mathtt{if}\ x = B\ \mathtt{then}\ k_{bs}\ \mathtt{else}\ k_{es}$$

Figure 2.3: A symmetric key establishment protocol and its encoding

As shown previously, the encoding of a protocol in the cryptographic metalanguage consists of the encoding of every principle, as a term in the metalanguage. The protocol is reformulated by the diagram in Figure 2.3, where every principle is a sequence of "bullets", representing the "states" of the program. A bullet waits for messages in a certain expected format, does some checking on the formats and contents upon receiving a message, then sends back another message and passes the control to the next bullet. There is no secret channel and every message is published on the public network. Such a bullet is encoded as a function, which takes the incoming message as argument, and returns a pair of the out-coming message and another function representing the "next state". For example, the bullet A_1 in the diagram of Figure 2.3 is encoded as the function:

$$f_{A_1} \equiv \lambda\_.\nu(k_{ab}).\langle[\mathtt{n}(A), \mathtt{n}(B), \{\mathtt{k}(k_{ab})\}_{k_{as}}],\ f_{A_2}\rangle$$

Because A_1 does not wait for any message, this is just a constant function (we omit the lambda binder at the beginning). A_1 publishes a message $\langle A, B, \{k_{ab}\}_{k_{as}}\rangle$ and passes the control to the bullet A_2, denoted by the function f_{A_2}. Note that A_1 generates a fresh key k_{ab}, but since this is the first step of the protocol, we can assume that k_{ab} is generated by the whole program and we encode A_1 as

$$f_{A_1} \equiv \langle[\mathtt{n}(A), \mathtt{n}(B), \{\mathtt{k}(k_{ab})\}_{k_{as}}],\ f_{A_2}\rangle.$$

By doing so we avoid programs with types of the form $\mathsf{T}(\cdots \times \mathsf{T}\_ \times \cdots)$, because completeness of logical relations for this kind of types is hard to deal with. We shall explain this difficulty in Chapter 6.

Encodings of other bullets are given in Figure 2.3. The whole protocol is then encoded as a 4-tuple:

$$P \equiv \nu(k_{as}, k_{bs}, k_{es}, k_{ab}).\langle k_{es},\ f_{A_1},\ f_{S_1},\ f_{B_1}\rangle$$

where k_{es} is the key shared by the server and a third (malicious) principle. The trivial function f_{B_1} can be ignored, so we write simply

$$P \equiv \nu(k_{as}, k_{bs}, k_{es}, k_{ab}).\langle k_{es},\ f_{A_1},\ f_{S_1}\rangle, \tag{2.1}$$

According to the typing rules in Figure 2.1, the type of this program is:

$$\emptyset \vdash P : \mathsf{T}(\mathsf{key} \times (\mathsf{msg} \times \mathsf{msg}) \times (\mathsf{msg} \to \mathsf{opt}[\mathsf{msg}]))$$

As shown in Chapter 1, this protocol is flawed. In the cryptographic metalanguage, an attack is simply a "bad" scheduler, which can be encoded as a function that takes the program (2.1) as argument and tries to reveal the secret text i:

$$\begin{array}{rcl} Attack(P) & \equiv & \texttt{let}\ \langle k_e, \langle m_a^1, m_a^2\rangle, p_s\rangle \Leftarrow P\ \texttt{in} \\ & & \texttt{val(letopt}\ m_e \Leftarrow p_s([\texttt{n}(A), \texttt{n}(E), \pi_3^3(m_a^1)])\ \texttt{in} \\ & & \quad \texttt{letopt}\ m_e' \Leftarrow \texttt{dec}(\pi_3^3(m_e), k_e)\ \texttt{in} \\ & & \quad \texttt{letopt}\ k \Leftarrow \texttt{getkey}(m_e')\ \texttt{in} \\ & & \quad \texttt{letopt}\ i \Leftarrow \texttt{dec}(m_a^2, k)\ \texttt{in} \\ & & \quad \texttt{some}(i)) \end{array}$$

Since the attack is caused by some replacement of principle identities in messages, to fix this flaw and prevent the attack, it is sufficient to encrypt those critical identities, so that intruders are not able to modify them:

$$\begin{array}{lll} \text{Message } 1\,. & A \to S: & A, B, \{B, k_{ab}\}_{k_{as}} \\ \text{Message } 2\,. & S \to B: & A, B, \{A, k_{ab}\}_{k_{bs}} \\ \text{Message } 3\,. & A \to B: & \{i\}_{k_{ab}} \end{array}$$

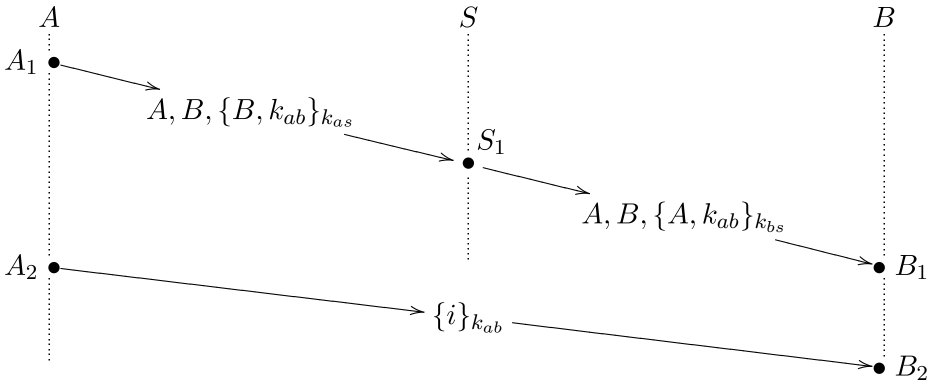
The diagram and the encoding of the fixed protocol are given in Figure 2.4. The type of the protocol program does not change.

2.4.2 Le protocole de Needham-Schroeder

Another example is a famous protocol of asymmetric cryptography — the Needham-Schroeder's public key protocol [NS78]. This protocol consists of two principles aiming at authenticating with each other and generating a session key for later communication. The exchange of messages is specified as follows:

$$\begin{array}{lll} \text{Message } 1\,. & A \to B: & A \\ \text{Message } 2\,. & B \to A: & \{N_b, B\}_{pk(A)} \\ \text{Message } 3\,. & A \to B: & \{N_a, N_b\}_{pk(B)} \\ \text{Message } 4\,. & B \to A: & \{N_a\}_{pk(A)} \\ \text{Message } 5\,. & A \to B: & \{i\}_{N_a} \end{array}$$

Principle A initiates a session by sending its own identity to another principle B. B wants A to prove A's identity first. It generates a nonce N_b (a fresh random number) and encrypts it with A's public key (Message 2) so that only A can decrypt this message. A must send back this nonce N_b to convince B that it is A who is talking with it. At the same time, A also wants B to prove B's identity. A then generates another nonce N_a and sends it to B by encrypting it with B's public key (Message 3). Since only B can decrypt this message, B should send back the nonce

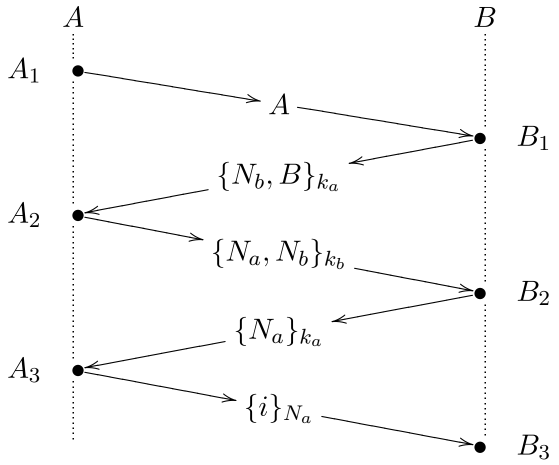


$$
\begin{array}{lcl}
f_{A_1} & \equiv & \langle [\mathtt{n}(A), \mathtt{n}(B), \{[\mathtt{n}(B), \mathtt{k}(k_{ab})]\}_{k_{as}}],\ f_{A_2} \rangle \\
f_{A_2} & \equiv & \{i\}_{k_{ab}} \\
f_{S_1} & \equiv & \lambda x.\mathtt{letopt}\ x_a \Leftarrow \mathtt{getnum}(\pi^3_1(x))\ \mathtt{in} \\
& & \qquad \mathtt{letopt}\ x_b \Leftarrow \mathtt{getnum}(\pi^3_2(x))\ \mathtt{in} \\
& & \qquad \mathtt{letopt}\ y \Leftarrow \mathtt{dec}(\pi^3_3(x), \mathtt{K}(x_a, s))\ \mathtt{in} \\
& & \qquad \mathtt{letopt}\ x'_b \Leftarrow \mathtt{getnum}(\mathtt{fst}(y))\ \mathtt{in} \\
& & \qquad \mathtt{if}\ x_b = x'_b\ \mathtt{then}\ \mathtt{some}([\mathtt{n}(x_a), \mathtt{n}(x_b), \{\mathtt{snd}(y)\}_{\mathtt{K}(x_b,s)}]) \\
& & \qquad\qquad\qquad \mathtt{else}\ \mathtt{error}
\end{array}
$$

where $\mathtt{K}(x, s)$ abbreviates

$$\mathtt{if}\ x = A\ \mathtt{then}\ k_{as}\ \mathtt{else}\ \mathtt{if}\ x = B\ \mathtt{then}\ k_{bs}\ \mathtt{else}\ k_{es}$$

Figure 2.4: Fixed symmetric key establishment protocol and its encoding



$$
\begin{array}{lcl}
f_{A_1} & \equiv & \langle \mathtt{n}[A], f_{A_2} \rangle \\
f_{A_2} & \equiv & \lambda\{x\}_{k_a}.\mathtt{letopt}\ x' \Leftarrow \mathtt{getnum}(\mathtt{snd}(x))\ \mathtt{in} \\
 & & \qquad\qquad \mathtt{some}(\nu(N_a).\langle\{[\mathtt{k}(N_a), \mathtt{fst}(x)]\}_{\mathtt{K}(x')},\ f_{A_3}\rangle) \\
f_{A_3} & \equiv & \lambda\{x''\}_{k_a}.\mathtt{letopt}\ x''' \Leftarrow \mathtt{getkey}(x'')\ \mathtt{in} \\
 & & \qquad\qquad \mathtt{if}\ x''' = N_a\ \mathtt{then}\ \mathtt{some}(\{i\}_{N_a})\ \mathtt{else}\ \mathtt{error} \\
f_{B_1} & \equiv & \lambda y.\mathtt{letopt}\ y' \Leftarrow \mathtt{getnum}(y)\ \mathtt{in} \\
 & & \qquad\qquad \mathtt{some}(\nu(N_b).\langle\{[\mathtt{k}(N_b), \mathtt{n}(B)]\}_{\mathtt{K}(y')},\ f_{B_2}\rangle) \\
f_{B_2} & \equiv & \lambda\{y''\}_{k_b}.\mathtt{letopt}\ y''' \Leftarrow \mathtt{getkey}(\mathtt{snd}(y''))\ \mathtt{in} \\
 & & \qquad\qquad \mathtt{if}\ y''' = N_b\ \mathtt{then}\ \mathtt{some}(\{\mathtt{fst}(y'')\}_{\mathtt{K}(y')})\ \mathtt{else}\ \mathtt{error}
\end{array}
$$

where $\mathtt{K}(x)$ abbreviates for

$$\mathtt{if}\ x = A\ \mathtt{then}\ k_a\ \mathtt{else}\ \mathtt{if}\ x = B\ \mathtt{then}\ k_b\ \mathtt{else}\ k_e$$

Figure 2.5: Needham-Schroeder's public key protocol

N_a (Message 4) to convince A that it is B who is talking. Finally, the two principles authenticate with each other and the nonce N_a is agreed by them as the session key.

Encoding this protocol in the cryptographic metalanguage is similar as for the previous symmetric key establishment protocol. We first reformulate the protocol as the diagram in Figure 2.5, then encode every bullet function. Note that we have only symmetric cryptography in the metalanguage, so $pk(A)$, $pk(B)$ are represented by two functions $\lambda x.\texttt{enc}(x, k_a)$ and $\lambda x.\texttt{enc}(x, k_b)$, where k_a and k_b are private keys of A and B. Finally, we merge these bullet functions in a proper way and we get the encoding of the whole protocol

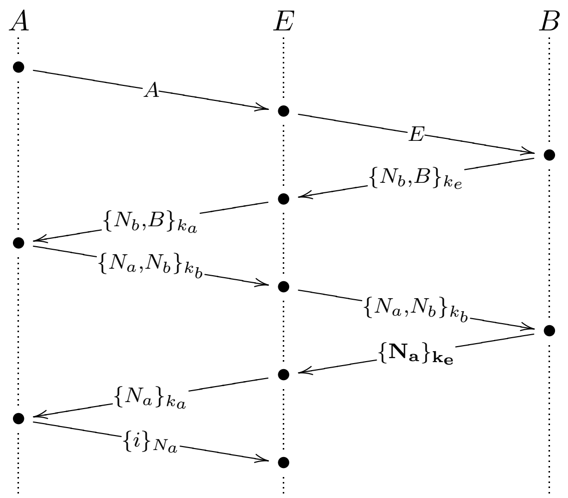
$$P \equiv \nu(k_a, k_b, k_e).\langle \lambda x.\{x\}_{k_a},\ \lambda x.\{x\}_{k_b},\ k_e,\ f_{A_1},\ f_{B_1}\rangle.$$

The type of this program can be deduced using the typing rules:

$$\begin{aligned} \emptyset \ \vdash\ & \mathsf{T}((\mathsf{msg} \to \mathsf{msg}) \times (\mathsf{msg} \to \mathsf{msg}) \times \mathsf{key} \\ & \times(\mathsf{msg} \times (\mathsf{msg} \to \mathsf{opt}[\mathsf{T}(\mathsf{msg} \times (\mathsf{msg} \to \mathsf{opt}[\mathsf{msg}]))])) \\ & \times(\mathsf{msg} \to \mathsf{opt}[\mathsf{T}(\mathsf{msg} \times (\mathsf{msg} \to \mathsf{opt}[\mathsf{msg}]))])) \end{aligned}$$

For a very long time (around 17 years), this protocol had been regarded as a perfect way to solve the key distribution and authentication problems, using the public key cryptography scheme. However, in 1995, a serious flaw was found by Gavin Lowe in this protocol [Low95]. The attack to the protocol is very subtle and it has nothing to do with the adopted cryptographic algorithms. This is completely a flaw of protocol design.

We demonstrate the attack by the following diagram:



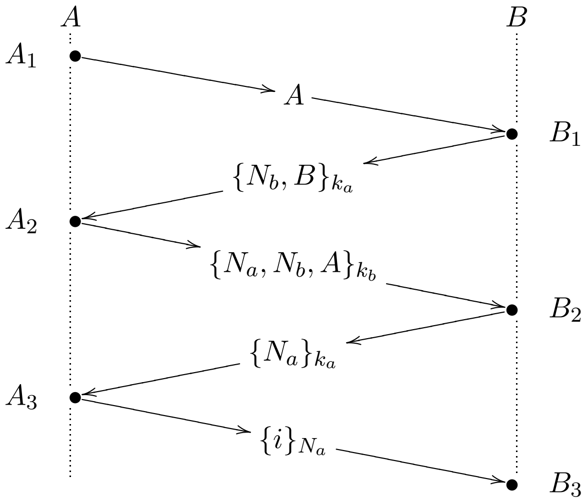
This is a typical "man-in-the-middle" attack. A is trying to talk to B, but all messages are intercepted by another principle E. Instead of forwarding A's messages to B, E starts another session with B. Thus there are two sessions of the protocol running in parallel — one between A and E and the other between E and B. E impersonates B in the session with A by reusing messages that he gets from B in the other session, so that A believes that he is talking with B. Clearly, both sessions follow exactly the protocol specification, but messages from B are encrypted by E's public key because B believes that he is talking with E, so E can decrypt these messages. For instance, E can decrypt the fourth message in the right session, and obtain the critical information — the session key. This attack can be encoded as the following function in the metalanguage:

$$
\begin{aligned}
Attack(P) \quad \equiv \quad & \texttt{let}\ \langle pk_a, pk_b, k_e, \langle A, m_a\rangle, m_b\rangle \Leftarrow P\ \texttt{in} \\
& \texttt{let}\ \langle m_1, m_b^2\rangle \Leftarrow m_b(E)\ \texttt{in} \\
& \texttt{letopt}\ m_2 \Leftarrow (\lambda\{x\}_{k_e}.x)m_1\ \texttt{in} \\
& \texttt{letopt}\ \langle m_3, m_a^3\rangle \Leftarrow m_a(pk_a(m_2))\ \texttt{in} \\
& \texttt{letopt}\ m_4 \Leftarrow m_b^2(m_3)\ \texttt{in} \\
& \texttt{letopt}\ m_5 \Leftarrow (\lambda\{x\}_{k_e}.x)m_4\ \texttt{in} \\
& \texttt{letopt}\ m_6 \Leftarrow m_a^3(pk_a(m_5))\ \texttt{in} \\
& \texttt{letopt}\ i \Leftarrow (\lambda\{x\}_{m_5}.x)m_6\ \texttt{in} \\
& \texttt{some}(i)
\end{aligned}
$$

The point of this attack is that the intruder E is involved in two sessions of the same protocol at the same time, and he can make use of some messages from one session in the other session, without breaking the protocol specification. In particular, the third message in the right session is exactly the third one in the left session. Hence, we can prevent this attack by adding the identity of the sender into Message 3 of the protocol:

$$\text{Message } 3\ . \quad A \rightarrow B : \quad \{N_a, N_b, A\}_{pk(B)}$$

Both the attack and the fix to this protocol were proposed by Gavin Lowe [Low96a], and the fixed version of Needham-Schroeder's public key protocol, called Needham-Schroeder-Lowe's protocol, is given in Figure 2.6, together with its encoding.



$$
\begin{array}{lcl}
f_{A_1} & \equiv & \langle \mathtt{n}(A), f_{A_2} \rangle \\
f_{A_2} & \equiv & \lambda\{x\}_{k_a}.\mathtt{letopt}\ x' \Leftarrow \mathtt{getnum}(\mathtt{snd}(x))\ \mathtt{in} \\
 & & \qquad\qquad \mathtt{some}(\nu(N_a).\langle\{[\mathtt{k}(N_a), \mathtt{fst}(x), A]\}_{\mathtt{K}_a(x')},\ f_{A_3}\rangle) \\
f_{A_3} & \equiv & \lambda\{x''\}_{k_a}.\mathtt{letopt}\ x''' \Leftarrow \mathtt{getkey}(x'')\ \mathtt{in} \\
 & & \qquad\qquad \mathtt{if}\ x''' = N_a\ \mathtt{then}\ \mathtt{some}(\{i\}_{N_a})\ \mathtt{else}\ \mathtt{error} \\
f_{B_1} & \equiv & \lambda y.\mathtt{letopt}\ y' \Leftarrow \mathtt{getnum}(y)\ \mathtt{in} \\
 & & \qquad \mathtt{some}(\nu(N_b).\langle\{[\mathtt{k}(N_b), \mathtt{n}(B)]\}_{\mathtt{K}_b(y')},\ f_{B_2}\rangle) \\
f_{B_2} & \equiv & \lambda\{y''\}_{k_b}.\mathtt{letopt}\ y''_2 \Leftarrow \mathtt{getkey}(\pi^3_2(y''))\ \mathtt{in} \\
 & & \qquad\qquad \mathtt{letopt}\ y''_3 \Leftarrow \mathtt{getnum}(\pi^3_3(y''))\ \mathtt{in} \\
 & & \qquad\qquad \mathtt{if}\ (y''_2 = N_b\ \mathtt{and}\ y' = y''_3) \\
 & & \qquad\qquad\quad \mathtt{then}\ \mathtt{some}(\{\pi^3_1(y'')\}_{\mathtt{K}_b(y')})\ \mathtt{else}\ \mathtt{error}
\end{array}
$$

where $\mathtt{K}_a(x)$ and $\mathtt{K}_b(x)$ abbreviate for

$$
\begin{array}{lcl}
\mathtt{K}_a(x) & \equiv & \mathtt{if}\ x = B\ \mathtt{then}\ k_b\ \mathtt{else}\ k_e \\
\mathtt{K}_b(x) & \equiv & \mathtt{if}\ x = A\ \mathtt{then}\ k_a\ \mathtt{else}\ k_e
\end{array}
$$

Figure 2.6: Needham-Schroeder-Lowe's public key protocol and its encoding

Chapitre 3

Modèles catégoriques

La syntaxe du métalangage cryptographique et le codage des protocoles ont été vus au chapitre 2. Maintenant, nous allons définir dans ce chapitre la sémantique dénotationnelle du métalangage. En général, le lambda-calcul typé pourrait être interprété dans les *catégories cartésiennes fermées* (*CCC* en abrégé) [LS86, AL91]. Nous suivons cette convention et nous construisons un modèle catégorique du métalangage cryptographique. Pour cela, nous devons interpréter deux sortes de primitives : les primitives cryptographiques et la génération de clés. Pour la première, nous adaptons ici des stratégies standard dans la plupart des modèles de protocoles cryptographiques.

Il est plus difficile de traiter de la génération dynamique de clés. Grâce à Moggi [Mog89, Mog90, Mog91], ce mécanisme est considéré comme un *effet de bord* et par conséquent, elle peut être formalisée par la notion de *monade*. Stark précise cette idée par une monade spécifique de génération de noms et il montre qu'un modèle catégorique correct de la génération de noms doit satisfaire une liste de propriétés. En particulier, il définit un modèle catégorique basé sur la catégorie de foncteurs appelée $\mathcal{S}et^{\mathcal{I}}$, qui satisfait les propriétés. Le métalangage computationnel de la génération de noms, ainsi que le nu-calcul, peut donc être interprété dans ce modèle [Sta94, Sta96].

Puisque le métalangage cryptographique est une extension du métalangage computationnel de Stark, nous pouvons naturellement prendre la catégorie $\mathcal{S}et^{\mathcal{I}}$ comme modèle de notre langage. Pour cela, nous devons définir d'abord un objet dans cette catégorie pour le type msg. Nous définissons aussi la forme canonique du métalangage cryptographique et nous prouvons qu'il existe un terme en forme canonique (avec la même sémantique) pour chaque terme du métalangage.

Formaliser la propriété de secret à l'aide de la notion d'équivalence contextuelle est un point crucial de notre modélisation. Alors qu'est-ce que l'équivalence contextuelle pour les protocoles cryptographiques ? Nous verrons que la définition standard d'équivalence contextuelle du lambda-calcul ne s'applique pas dans notre métalangage. En nous inspirant de la notion d'équivalence contextuelle du nu-calcul définie par Pitts et Stark, nous déciderons d'adapter leur définition à notre métalangage et à notre modèle. Dans ce chapitre, nous n'arriverons pas encore à une définition finale de l'équivalence contextuelle (qui sera donnée au chapitre 6), mais la discussion que nous mènerons montra que définir une notion correcte d'équivalence contextuelle dans le métalangage cryptographique demande réflexion — il faut considérer plusieurs points subtils. En effet, nous allons montrer dans les chapitres suivants que la catégorie $\mathcal{S}et^{\mathcal{I}}$ n'est pas suffisante pour étudier les relations entre les programmes du métalangage, y compris l'équivalence contextuelle.

Ce chapitre commence par une introduction élémentaire de la théorie des catégories dans la partie 3.1, où nous décrivons en particulier l'interprétation du lambda-calcul dans une catégorie cartésienne fermée et l'interprétation des effets de bord en utilisant les monades. Nous introduisons le modèle de Stark dans la partie 3.2 et nous définissons ensuite dans la partie 3.3 une sémantique dénotationnelle du métalangage cryptographique, basée sur la catégorie $\mathcal{S}et^{\mathcal{I}}$. La partie 3.4 parle de la forme canonique du métalangage. Le chapitre se termine par la partie 3.5, qui consiste en une discussion sur la notion d'équivalence contextuelle des protocoles cryptographiques.

While Chapter 2 is mainly on the syntax of the cryptographic metalanguage and the encoding of protocols, we shall define in this chapter its semantics, in a denotational way. It is standard that typed lambda-calculus can be interpreted by *cartesian closed categories* (*CCC* for short) [LS86, AL91]. We follow this convention to construct a categorical model for the cryptographic metalanguage. For this purpose, we have to deal with cryptographic primitives and key generation. Encryption and decryption are usually modeled by products in most formal models for cryptographic protocols and such a strategy is also adopted here.

Dealing with dynamic key generation is more difficult. Thanks to Moggi's work on the computational lambda-calculus [Mog89, Mog90, Mog91], this is seen as some kind of *side effect* and can be modeled by a *monad*. Stark specializes Moggi's work in the name creation monad and shows that a proper categorical model for name creation must satisfy certain properties. He defines in particular a categorical model based on the functor category $\mathcal{S}et^{\mathcal{I}}$, which satisfies those properties, so that the computational metalanguage for name creation (consequently the nu-calculus) can be naturally interpreted in it [Sta94, Sta96].

Since the cryptographic metalanguage is an extension of Stark's computational metalanguage, it is natural to take the category $\mathcal{S}et^{\mathcal{I}}$ as model. We then use this category to define the denotational semantics of our language, by first defining an object for the specific base type msg. We also define the canonical forms for the cryptographic metalanguage, and prove that every expression is equivalent to a canonical term (with the same semantics), provided that it does not return any error (decryption failure for example).

The very essential point of our method is using contextual equivalence to formalize secrecy property of protocols, but what should be the right notion of contextual equivalence for cryptographic protocols? Indeed, we find that standard definitions for lambda-calculi do not fit in our case. Inspired by Pitts and Stark's notion of contextual equivalence for the nu-calculus [PS93a], we try to adapt their definition to the cryptographic metalanguage and the denotational model. Note that this kind of equivalence, which states that two values (or terms) a_1 and a_2 are equivalent provided every context of type bool must give identical results on a_1 and on a_2, is also called *observational equivalence*. We must stress that it should not be confused with observational equivalence as it is defined for data refinement [Mit96], where *models* are related, not *values* in the same model as here.

Although this chapter does not arrive finally at a correct definition of contextual equivalence (the final one will be given in Chapter 6), it indeed shows that defining this notion of equivalence in the cryptographic metalanguage is not straightforward at all. Several subtle points have to be taken into account. In fact, as will be shown in later chapters, the categorical model $\mathcal{S}et^{\mathcal{I}}$ is not sufficient for studying relations, including contextual equivalence, between programs in the metalanguage.

We start this chapter by a basic introduction to category theory (Section 3.1), where we show in particular how to use cartesian closed categories to model lambda-calculus (Section 3.1.1) and monads to model computations (Section 3.1.2). We then introduce Stark's model in Section 3.2. By defining objects for some special types like the msg type, we obtain in Section 3.3 a denotational semantics for the cryptographic metalanguage based on $\mathcal{S}et^{\mathcal{I}}$. Section 3.4 is about the canonical forms of the metalanguage and Section 3.5 is a discussion on what should be the right notion of contextual equivalence for cryptographic protocols. We end this chapter by a conclusion on our language — the specification part of this thesis.

3.1 Préliminaires de la théorie des catégories

This section is provided here as an introductory text on categories. We shall introduce some basic concepts in category theory, like categories, functors, cartesian closed categories, and so on. These are necessary for understanding the rest of the thesis. Most definitions in this section are from [BW90, LS86, AL91, Pie91].

A *category* $\mathcal{C}$ comprises:

- a collection $\mathbf{Obj}(\mathcal{C})$ of *objects*;
- a collection $\mathbf{Mor}(\mathcal{C})$ of *morphisms* (*arrows*);
- two operations $\mathbf{dom}, \mathbf{cod}$ assigning to each morphism f two objects respectively called *domain* and *codomain* of f (we write $f : A \to B$ to show that $\mathbf{dom}(f) = A$ and $\mathbf{cod}(f) = B$; the collection of all morphisms with domain A and codomain B is written $\mathcal{C}[A, B]$);
- an operator $\circ$ (*composition*) assigning to each pair of morphisms f and g, with $\mathbf{cod}(f) = \mathbf{dom}(g)$, a morphism $g \circ f : \mathbf{dom}(f) \to \mathbf{cod}(g)$, satisfying the associative law: for any morphisms $f : A \to B$, $g : B \to C$ and $h : C \to D$, $h \circ (g \circ f) = (h \circ g) \circ f$;
- for each object A, an *identity* morphism $\mathbf{id}_A : A \to A$ satisfying the identity law: for any morphism $f : A \to A$, $\mathbf{id}_A \circ f = f$ and $f \circ \mathbf{id}_A = f$.

Example 3.1. The category $\mathcal{S}et$ has sets as objects and total functions between sets as morphisms. Composition of morphisms is set-theoretic function composition. Identity morphisms are identity functions. $\mathcal{P}fun$ is the category with sets as objects and partial functions as morphisms.

Let $f : B \to C$ be a morphism in a category $\mathcal{C}$, then f is said *monic* (or a *monomorphism*) if, for any two morphisms $g, h \in \mathcal{C}[A, B]$, the equality $f \circ g = f \circ h$ implies that $g = h$, and it is *epic* (or a *epimorphism*) if, for any two morphisms $g', h' \in \mathcal{C}[C, D]$, the equality $g' \circ f = h' \circ f$ implies that $g' = h'$. f is a *isomorphism* if there is a morphism $f^{-1} : C \to B$ such that $f^{-1} \circ f = \mathbf{id}_A$ and $f \circ f^{-1} = \mathbf{id}_B$. The objects B and C are said to be *isomorphic* if there is an isomorphism between them. For example, in the category $\mathcal{S}et$, the monomorphisms are just the injective functions, the epimorphisms are the surjective functions and the isomorphisms are the bijective functions.

A *diagram* in a category $\mathcal{C}$ is a collection of vertices and directed edges labeled with objects and morphisms of $\mathcal{C}$ such that if an edge in the diagram is labeled with a morphism f and f has domain A and codomain B, then the endpoints of this edge must be labeled with A and B. A diagram is said to *commute* if, for every pair of vertices X and Y, all the paths in the diagram from X to Y are equal, in the sense that each path in the diagram determines a morphism and these morphisms are identical in $\mathcal{C}$.

Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A *functor* $F : \mathcal{C} \to \mathcal{D}$ is a pair of operations $F_{obj} : \mathbf{Obj}(\mathcal{C}) \to \mathbf{Obj}(\mathcal{D})$ and $F_{mor} : \mathbf{Mor}(\mathcal{C}) \to \mathbf{Mor}(\mathcal{D})$ such that, for each $f : A \to B, g : B \to C$ in $\mathcal{C}$:

- $F_{mor}(f) : F_{obj}(A) \to F_{obj}(B)$ is a morphism in $\mathcal{D}$;
- $F_{mor}(g \circ f) = F_{mor}(g) \circ F_{mor}(f)$;
- $F_{mor}(\mathbf{id}_A) = \mathbf{id}_{F_{obj}(A)}$.

Given functors $F, G : \mathcal{C} \to \mathcal{D}$, a *natural transformation* $\delta : F \to G$ is a family of morphisms in the category $\mathcal{D}$ such that

- for any object $A \in \mathcal{C}$, $\delta_A \in \mathcal{D}[F(A), G(A)]$, and
- for any morphism $f \in \mathcal{C}[A, B]$, $\delta_B \circ F(f) = G(f) \circ \delta_A$, i.e., the following square commutes:

$$\begin{tikzcd} F(A) \arrow[r, "\delta_A"] \arrow[d, "F(f)"'] & G(A) \arrow[d, "G(f)"] \\ F(B) \arrow[r, "\delta_B"] & G(B) \end{tikzcd} .$$

Given a functor $F : \mathcal{C} \to \mathcal{D}$, we write FA for $F_{obj}(A)$ and Ff for $F_{mor}(f)$, for any object A and any morphism f in $\mathcal{C}$. We define for any category $\mathcal{C}$ an identity functor $Id_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ mapping every object to itself and every morphism to itself. If $F : \mathcal{C} \to \mathcal{C}$ is a functor over the category $\mathcal{C}$, then $F^2 : \mathcal{C} \to \mathcal{C}$ is another functor over $\mathcal{C}$ such that $F^2A = F(FA)$ and $F^2f = F(Ff)$ for any object A and any morphism f in $\mathcal{C}$. This can be generalized to define $F^n : \mathcal{C} \to \mathcal{C}$ for any finite number n.

An object 0 is called an *initial object* if, for every object A, there is exactly one morphism from 0 to A. Dually, an object 1 is called a *terminal object* if, for every object A, there is exactly one morphism from A to 1. In the category $\mathcal{S}et$, the empty set $\{\}$ is the only initial object; for every set A, the empty function is the unique function from $\{\}$ to A. Moreover, each one-element set S is a terminal object, since for every non-empty set A there is only one total function from A to S which maps every element of A to that single element of S, and for the empty set, there is the unique empty function.

A *product* of two objects A and B is an object $A \times B$, together with two projection morphisms $\mathbf{proj}_1 : A \times B \to A$ and $\mathbf{proj}_2 : A \times B \to B$, such that for any object C and pair of morphisms $f : C \to A$ and $g : C \to B$, there is exactly one mediating morphism $\langle f, g\rangle : C \to A \times B$ making the following diagram commute:

$$\begin{tikzcd} & C \arrow[dl, "f"'] \arrow[d, dashed, "\langle f,g\rangle"] \arrow[dr, "g"] & \\ A & A \times B \arrow[l, "\mathbf{proj}_1"'] \arrow[r, "\mathbf{proj}_2"] & B \end{tikzcd},$$

i.e., $\mathbf{proj}_1 \circ \langle f, g\rangle = f$ and $\mathbf{proj}_2 \circ \langle f, g\rangle = g$. If $A \times C$ and $B \times D$ are product objects, then for every pair of morphisms $f : A \to B$ and $g : C \to D$, the product morphism $f \times g : A \times C \to B \times D$ is the morphism $\langle f \circ \mathbf{proj}_1, g \circ \mathbf{proj}_2\rangle$.

A *coproduct* of two objects A and B is an object $A + B$, together with two injection morphisms $\mathbf{inl}_{A,B} : A \to A + B$ and $\mathbf{inr}_{A,B} : B \to A + B$, such that for any object C and pair of morphisms $f : A \to C$ and $g : B \to C$, there is exactly one morphism $\langle f \mid g\rangle : A + B \to C$ making the following diagram commute:

$$\begin{tikzcd} A \arrow[r, "\mathbf{inl}_{A,B}"] \arrow[dr, "f"'] & A + B \arrow[d, dashed, "\langle f|g\rangle"] & B \arrow[l, "\mathbf{inr}_{A,B}"'] \arrow[dl, "g"] \\ & C & \end{tikzcd}.$$

If a category $\mathcal{C}$ has a product (coproduct) for every pair of objects, we say that $\mathcal{C}$ has (binary) products (coproducts). The category $\mathcal{S}et$ has products and coproducts. The product of two sets A and B is their cartesian product:

$$A \times B = \{(a, b) \mid a \in A \ \ \& \ \ b \in B\}.$$

The coproduct is the disjoint union of the two sets:

$$A + B = \{(1, a) \mid a \in A\} \cup \{(2, b) \mid b \in B\}.$$

We shall omit the indices 1 and 2 if there is no confusing element, that is, when A and B are disjoint.

Let $\mathcal{C}$ be a category with all binary products and let A and B be objects of $\mathcal{C}$. An object B^A is an *exponential object* if there is a morphism $\mathbf{eval}_{AB} : (B^A \times A) \to B$ such that for any object C and morphism $f : C \times A \to B$ there is a unique morphism $\mathbf{curry}(f) : C \to B^A$ making the following triangle commute:

$$\begin{tikzcd} C \times A \arrow[d, "\mathbf{curry}(f)\times\mathbf{id}_A"'] \arrow[dr, "f"] & \\ B^A \times A \arrow[r, "\mathbf{eval}_{AB}"] & B \end{tikzcd}.$$

A category C is said to have exponentiation if it has an exponential B^A for every pair of objects A and B. The category $\mathcal{S}et$ has exponentiation: the exponential B^A of two sets A and B is the set of all functions from A to B, i.e., $\mathcal{S}et[A, B]$. A *cartesian closed category* (usually abbreviated as CCC) is a category with a terminal object, binary products and exponentiation.

3.1.1 Interprétation du lambda-calcul en CCCs

It is standard that the simply-typed lambda calculus can be modeled in a cartesian closed category [LS86, AL91], with types as objects and terms as morphisms. Let Σ be the signature of the intended lambda calculus and $\mathcal{C}$ be a CCC where we choose an object $b_{\mathcal{C}}$ for each base type b and a morphism $c_{\mathcal{C}} : 1 \to [\![\tau]\!]$ for each term constant $c : \tau \in \Sigma$. $[\![\_]\!]$ denotes the interpretation in $\mathcal{C}$ of the lambda calculus. A CCC $\mathcal{C}$ together with interpretations for base types and term constants in Σ, is called a Σ-*CCC* and denoted by $\mathcal{C}_\Sigma$ (we may omit Σ when it is clear from the context).

Function types are interpreted using exponentiation, i.e., the interpretation of type $\tau \to \tau'$ is the exponential object $[\![\tau']\!]^{[\![\tau]\!]}$. Product types and sum types are naturally interpreted as products and coproducts of $\mathcal{C}$.

We interpret a typing context as a finite product: if $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, then $[\![\Gamma]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$. The denotation of a well-typed term $\Gamma \vdash t : \tau$ is a morphism $[\![\Gamma \vdash t : \tau]\!]$ from $[\![\Gamma]\!]$ to $[\![\tau]\!]$. Given a typing derivation π of the judgment $\Gamma \vdash t : \tau$, we define $[\![\pi]\!]$ by structural induction on π:

$$\begin{aligned}
[\![\Gamma \vdash x_i : \tau_i]\!] &= [\![\Gamma]\!] \xrightarrow{\mathbf{proj}_i} [\![\tau_i]\!], \\
[\![\Gamma \vdash c : \tau]\!] &= [\![\Gamma]\!] \xrightarrow{!} 1 \xrightarrow{f_c} [\![\tau]\!], \\
[\![\Gamma \vdash \lambda x.t : \tau \to \tau']\!] &= [\![\Gamma]\!] \xrightarrow{\mathbf{curry}([\![\Gamma, x:\tau \vdash t:\tau']\!])} [\![\tau']\!]^{[\![\tau]\!]}, \\
[\![\Gamma \vdash t_1 t_2 : \tau']\!] &= [\![\Gamma]\!] \xrightarrow{\langle [\![\Gamma \vdash t_1 : \tau \to \tau']\!],\, [\![\Gamma \vdash t_2 : \tau]\!] \rangle} [\![\tau']\!]^{[\![\tau]\!]} \times [\![\tau]\!] \xrightarrow{\mathbf{eval}} [\![\tau']\!].
\end{aligned}$$

We shall write $\Gamma \vdash t : \tau$ or even just t in place of π, since typing derivations are (almost) isomorphic to the terms t themselves.

Let the intended category $\mathcal{C}_\Sigma$ be $\mathcal{S}et$, then the denotation of a type is a set. By selecting a proper element f_c in $[\![\tau]\!]$ for each term constant $c : \tau \in \Sigma$, we shall easily get a set-theoretical model for the simply-typed lambda calculus, where $[\![\tau \to \tau']\!]$ is the set of all total functions from $[\![\tau]\!]$ to $[\![\tau']\!]$. We describe the value $[\![t]\!]\rho$ of the term t in the environment ρ by structural induction on t:

$$\begin{array}{rcl} [\![x]\!]\rho & = & \rho(x), \text{ where } x : \tau \in \Gamma, \\ [\![\lambda x.t]\!]\rho & = & \text{the unique } f \in [\![\tau \to \tau']\!] \text{ such that} \\ & & \qquad \text{for any } a \in [\![\tau]\!],\ f(a) = [\![t]\!]\rho[x \mapsto a], \\ [\![t_1 t_2]\!]\rho & = & [\![t_1]\!]\rho([\![t_2]\!]\rho). \end{array}$$

More formally, for any typing context Γ, a Γ-*environment* ρ, is a map such that for every $x : \tau \in \Gamma$, $\rho(x)$ is an element of $[\![\tau]\!]$. This is isomorphic to an element in $[\![\Gamma]\!]$. We write $\rho[x \mapsto a]$ for the environment mapping x to a and every other variable y to $\rho(y)$, and $[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$ for environment mapping each x_i to a_i. We also write $[\![t]\!]$ instead of $[\![t]\!]\rho$ when the environment ρ is irrelevant, e.g., t is a closed term.

Given a signature Σ, we define a Σ-CCC $\boldsymbol{\lambda}(\Sigma)$, for the simply-typed lambda-calculus, as follows: the objects of $\boldsymbol{\lambda}(\Sigma)$ are typing contexts Γ, a morphism from $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ to $\Delta = \{y_1 : \tau'_1, \ldots, y_m : \tau'_m\}$ is a substitution $[y_1 := t_1, \ldots, y_m := t_m]$, where $\Gamma \vdash t_i : \tau_i$ $(1 \leq i \leq m)$, modulo $\beta\eta$-conversion. We often abbreviate a context $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ to $\{\overline{x_i : \tau_i}\}^n$ and a substitution $[x_1 := t_1, \ldots, x_n := t_n]$ to $[\overline{x_i := t_i}]^n$. The composition of two morphisms $[\overline{y_i := t_i}]^m : \{\overline{x_i : \tau_i}\}^n \to \{\overline{y_i : \tau'_i}\}^m$ and $[\overline{z_i := u_i}]^l : \{\overline{y_i : \tau_i}\}^m \to \{\overline{z_i : \tau''_i}\}^l$ is the substitution $[z_1 := u_1[\overline{y_i := t_i}]^m, \ldots, z_l := u_l[\overline{y_i := t_i}]^m]$. It is easy to check that $\boldsymbol{\lambda}(\Sigma)$ is indeed a Σ-CCC: the terminal object is the empty context ϵ, products are disjoint unions, the exponential is defined by $\Delta^\Gamma = \{z_1 : \sigma_1 \times \cdots \times \sigma_m \to \tau_1, \ldots z_n : \sigma_1 \times \cdots \sigma_m \to \tau_n\}$ and for every constant $c : \tau \in \Sigma$, there is a unique morphism $[x := c] : \epsilon \to \{x : \tau\}$ defining the interpretation of c. In particular, Γ-environments are exactly morphisms from the empty context ϵ, to Γ. $\boldsymbol{\lambda}(\Sigma)$ is a *free* Σ-CCC [LS86], which means that, for every Σ-CCC $\mathcal{C}$, there is a unique representation $[\![\_]\!]$ of Σ-CCCs from $\lambda(\Sigma)$ to $\mathcal{C}$. A representation of Σ-CCCs is a functor that preserves products, exponentials and interpretations of each base type and each term constant in Σ.

3.1.2 Monades et le lambda-calcul computationnel

Moggi's computational lambda-calculus, as presented in the Section 2.1.3 of Chapter 2, is basically a simply-typed lambda calculus with some special language primitives, hence a categorical model for this language is necessarily cartesian closed. But this is not sufficient. According to

Moggi, a sound categorical model for the computational lambda-calculus must be a cartesian closed category together with a *strong monad*.

A **monad** [Mac71] over a category $\mathcal{C}$ is a triple (T, η, μ), where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\eta : Id_{\mathcal{C}} \to T$ and $\mu : T^2 \to T$ are natural transformations and the following diagrams commute:

$$\begin{tikzcd} T^3A \arrow[r, "\mu_{TA}"] \arrow[d, "T\mu_A"'] & T^2A \arrow[d, "\mu_A"] \\ T^2A \arrow[r, "\mu_A"] & TA \end{tikzcd} , \qquad \begin{tikzcd} TA \arrow[r, "\eta_{TA}"] \arrow[dr, "\mathbf{id}_{TA}"'] & T^2A \arrow[d, "\mu_A"] & TA \arrow[l, "T\eta_A"'] \arrow[dl, "\mathbf{id}_{TA}"] \\ & TA & \end{tikzcd} .$$

If $\mathcal{C}$ has products, then a monad over $\mathcal{C}$ is said to be a *strong monad* if there is a natural transformation $\mathrm{t}_{A,B} : A \times TB \to T(A \times B)$ for any pair of objects A and B [Mog89, Mog91].

As in the construction of categorical models of simply-typed lambda calculus, we can interpret the computational lambda-calculus in a cartesian closed category $\mathcal{C}$ with a strong monad $(T, \eta, \mu, \mathrm{t})$, with types as objects and terms as morphisms. Notably, the functor T is used to model the unary type constructor T: $[\![\mathsf{T}\tau]\!] = T[\![\tau]\!]$. The interpretation of the constants $\mathtt{val}(\_)$ and the $\mathtt{let}$ construction is defined as follows:

$$\begin{aligned} &[\![\Gamma \vdash \mathtt{val}(t) : \mathsf{T}\tau]\!] = \\ &\qquad [\![\Gamma]\!] \xrightarrow{[\![\Gamma\vdash t:\tau]\!]} [\![\tau]\!] \xrightarrow{\eta_{[\![\tau]\!]}} T[\![\tau]\!], \\ &[\![\Gamma \vdash \mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2 : \mathsf{T}\tau']\!] = \\ &\qquad [\![\Gamma]\!] \xrightarrow{\langle \mathbf{id}_{[\![\Gamma]\!]}, [\![\Gamma\vdash t_1:\mathsf{T}\tau]\!]\rangle} [\![\Gamma]\!] \times T[\![\tau]\!] \xrightarrow{\mathrm{t}_{[\![\Gamma]\!],[\![\tau]\!]}} T([\![\Gamma]\!] \times [\![\tau]\!]) \\ &\qquad\qquad \xrightarrow{T[\![\Gamma, x:\tau\vdash t_2:\mathsf{T}\tau']\!]} TT[\![\tau']\!] \xrightarrow{\mu_{[\![\tau']\!]}} T[\![\tau']\!]. \end{aligned}$$

A list of monads for concrete forms of computation is given in [Mog91]. We cite here two of them, both being defined in $\mathcal{S}et$.

Example 3.2. *Exceptions:* $TA = A + E$, where E is the set of exceptions:

$$\begin{aligned} [\![\mathsf{T}\tau]\!] &= [\![\tau]\!] \cup E, \\ [\![\mathtt{val}t]\!]\rho &= [\![t]\!]\rho, \\ [\![\mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2]\!]\rho &= \begin{cases} [\![t_2]\!]\rho[x := [\![t_1]\!]\rho], & \text{if } [\![t_1]\!]\rho \notin E; \\ [\![t_1]\!]\rho, & \text{if } [\![t_1]\!]\rho \in E. \end{cases} \end{aligned}$$

Example 3.3. *Non-determinism:* $TA = \mathbb{P}_{\mathit{fin}}(A)$, also known as the powerset monad:

$$\begin{aligned} [\![\mathsf{T}\tau]\!] &= \mathbb{P}_{\mathit{fin}}([\![\tau]\!]), \\ [\![\mathtt{val}t]\!]\rho &= \{[\![t]\!]\rho\}, \\ [\![\mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2]\!]\rho &= \bigcup_{a \in [\![t_1]\!]\rho} [\![t_2]\!]\rho[x := a]. \end{aligned}$$

As the simply-typed lambda calculus gives rise to a free Σ-CCC $\boldsymbol{\lambda}(\Sigma)$, the computational lambda-calculus gives rise to a *free* Σ*-let-CCC* $\boldsymbol{Comp}(\Sigma)$. We define $\boldsymbol{Comp}(\Sigma)$ in a similar way as we define the free Σ-CCC $\boldsymbol{\lambda}(\Sigma)$, with typing contexts as objects and substitutions as morphisms. $\boldsymbol{Comp}(\Sigma)$ is also equipped with a strong monad $(\boldsymbol{T}, \boldsymbol{\eta}, \boldsymbol{\mu}, \mathbf{t})$, which is defined as follows:

- $\boldsymbol{T}\Gamma = \{\bar{x}_1 : \mathsf{T}\tau_1, \ldots, \bar{x}_n : \mathsf{T}\tau_n\}$;
- $\boldsymbol{T}[\overline{y_i := t_i}]^m = [\overline{\bar{y}_i := \mathtt{let}\ x_1 \Leftarrow \bar{x}_1\ \mathtt{in} \cdots \mathtt{let}\ x_n \Leftarrow \bar{x}_n\ \mathtt{in}\ \mathtt{val}(t_i)}]^m$;
- $\boldsymbol{\eta}_\Gamma = [\overline{\bar{x}_i := \mathtt{val}(x_i)}]^n : \Gamma \to \boldsymbol{T}\Gamma$;
- $\boldsymbol{\mu}_\Gamma = [\overline{\bar{x}_i := \mathtt{let}\ x'_i \Leftarrow \bar{\bar{x}}_i\ \mathtt{in}\ \mathtt{let}\ x''_i \Leftarrow x'_i\ \mathtt{in}\ \mathtt{val}(x''_i)}]^n : \boldsymbol{T}^2\Gamma \to \boldsymbol{T}\Gamma$;
- $\mathbf{t}_{\Gamma,\Delta} = [\overline{\bar{x}_i := \mathtt{val}(x_i)}]^n \cup [\overline{\bar{y}_j := \bar{y}_j}]^m : \Gamma \times \boldsymbol{T}\Delta \to \boldsymbol{T}(\Gamma \times \Delta)$,

where $\Gamma = \{\overline{x_i : \tau_i}\}^n$ and $\Delta = \{\overline{y_i : \tau'_i}\}^m$ are arbitrary contexts and $[\overline{y_i := t_i}]^m$ is seen as a morphism from Γ to Δ in $\boldsymbol{Comp}(\Sigma)$.

3.2 Le catégorie de foncteurs $Set^{\mathcal{I}}$

Stark specializes Moggi's computational lambda-calculus to the specific computation of dynamic name creation, and he defines a computational metalanguage to interpret the nu-calculus [Sta96, Sta94]. According to Stark, if a category satisfies certain requirements then its *internal language* will include the metalanguage for name creation. As an example, Stark defines a functor category $\mathcal{S}et^{\mathcal{I}}$ equipped with a strong monad T and shows that it satisfies those requirements, hence it is sufficient to model the metalanguage.

The cryptographic metalanguage is indeed an extension of Stark's computational metalanguage. All the requirements that are needed for modeling his language are also necessary for constructing a categorical model for the cryptographic metalanguage. Besides, an option type opt$[\tau]$ in our language is naturally interpreted as a coproduct $[\![\tau]\!] + 1$, where 1 is the terminal object. This requires that coproducts exist for every object and the terminal object, not as in Stark's categories where the existence of the coproduct of terminal objects is enough. Fortunately, coproducts of any two objects exist in the category $\mathcal{S}et^{\mathcal{I}}$.

The category $\mathcal{S}et^{\mathcal{I}}$ is a functor category where objects are functors from $\mathcal{I}$ to $\mathcal{S}et$ and morphisms are natural transformations between these functors. Here $\mathcal{I}$ is the category of finite sets and injective functions. Intuitively, objects of $\mathcal{I}$ represent *computation stages*, since an object contains keys (or names) that have been generated at a certain stage. For any functor $A : \mathcal{I} \to \mathcal{S}et$, the set As is composed of values defined over the keys in s. Morphisms in $\mathcal{I}$ and their images in $\mathcal{S}et$ correspond to substitutions: if $i : s \to s'$ is a morphism in $\mathcal{I}$ and $a \in As$, then $Ai(a)$ is the value obtained by substituting every name $n \in s$ with $i(n)$ in the value a.

This category is cartesian closed. Let A, B be two functors from $\mathcal{I}$ to $\mathcal{S}et$. Products and coproducts [1] are taken pointwise:

$$\begin{array}{rcll} (A \times B)s & = & As \times Bs, & s \in \mathcal{I}, \\ (A + B)s & = & As + Bs, & \\ (A \times B)i(a,b) & = & (Ai(a), Bi(b)), & i : s \to s' \in \mathcal{I}, \\ (A + B)i(m,x) & = & \begin{cases} Ai(x), & \text{if } m = 1; \\ Bi(x), & \text{if } m = 2. \end{cases} & \end{array}$$

Exponentials are defined by the standard construction in covariant presheaves [LS86]:

$$\begin{array}{rcl} B^A s & = & \mathcal{S}et^{\mathcal{I}}(\mathcal{I}(s,-) \times A, B), \\ B^A i f s'' \langle j, a'' \rangle & = & f s'' \langle j \circ i, a'' \rangle, \end{array}$$

where $i : s \to s', j : s' \to s'' \in \mathcal{I}$ and $f \in B^A s$, $a'' \in As''$. An equivalent way to define the exponential in $\mathcal{S}et^{\mathcal{I}}$ is:

$$B^A s = \mathcal{S}et^{\mathcal{I}}(A(s + \_), B(s + \_)).$$

This definition says that a function from A to B defined at stage s includes information on how it behaves at all later stages.

We then consider a strong monad $(T, \eta, \mu, \mathrm{t})$ on $\mathcal{S}et^{\mathcal{I}}$ defined in [Sta96, GLLN02]:

- $TA = \mathbf{colim}_{s'} A(\_ + s') : \mathcal{I} \to \mathcal{S}et$. On objects, $TAs = \mathbf{colim}_{s'} A(s + s')$ is the set of all equivalence classes of pairs (s', a), with $s' \in \mathcal{I}$ and $a \in A(s + s')$, modulo the smallest equivalence relation $\simeq$ such that $(s', a) \simeq (s'', A(\mathbf{id}_s + j)a)$ for every morphism $j : s' \to s''$ in $\mathcal{I}$. We write $[s', a]$ for the equivalence class of (s', a). On morphisms, TAi with $i : s \to s_1 \in \mathcal{I}$, maps the equivalence class $[s', a]$ to the equivalence class $[s', A(i + \mathbf{id}_{s'})a]$.

- For any $f : A \to B$ in $\mathcal{S}et^{\mathcal{I}}$, $Tfs : TAs \to TBs$ is defined by $Tfs[s', a] = [s', f(s+s')a]$. This is compatible with $\simeq$ because f is natural.

- $\eta_A s : As \to TAs$ is defined by $\eta_A s a = [\emptyset, a]$.

- $\mu_A s : T^2 As \to TAs$ is defined by $\mu_A s[s', [s'', a]] = [s' + s'', a]$.

- $\mathrm{t}_{A,B} s : As \times TBs \to T(A \times B)s$ is defined by $\mathrm{t}_{A,B} s(a, [s', b]) = [s', (Ai_{s,s'} a, b)]$ where $i_{s,s'} : s \to s + s'$ is the canonical injection.

[1] Note that $+$ is not a coproduct in $\mathcal{I}$. In fact, $\mathcal{I}$ does not have a coproduct. However $+$ is functorial in both components, associative, and has a neutral element.

A set TA is meant to be the denotation of a computation type. The semantics of a computation of key generation consists of a set of fresh keys generated during the computation, and a final value. These are exactly the intuitive meaning of s' and a of a pair (s', a) in TA. In particular, if a set s is given before the execution of such a computation, the final value a must be defined over the disjoint sum $s + s'$, that is, it just uses keys from $s + s'$.

The equivalence $\simeq$ means that we care only whether those new keys are *fresh*, not what exactly they are. In other words, all the keys in s' are bound in a but those in s are free, and by renaming those bound keys we get equivalent values. Furthermore, if a computation generates some fresh keys but does not use them, then it is equivalent to the one which does not generate those keys, i.e., $(s' + s'', a) \simeq (s', a)$, for any additional set s'' of fresh keys. To summarize, for every $(s_1, a_1), (s_2, a_2) \in TAs$, $(s_1, a_1) \simeq (s_2, a_2)$ if and only if there is a finite set s_0 and two morphisms $i_1 : s_1 \to s_0$ and $i_2 : s_2 \to s_0$ such that $A(\mathbf{id}_s + i_1)a_1 = A(\mathbf{id}_s + i_2)a_2$.

3.3 Interprétation du métalangage cryptographique

Let Nat be the set of integers and $Bool$ be the set of two boolean values tt and ff. We define functors N, B and K from $\mathcal{I}$ to $\mathcal{S}et$:

$$\begin{aligned} Ns = Nat \quad & \& \quad Ni = \mathbf{id}_{Nat}, \\ Bs = Bool \quad & \& \quad Bi = \mathbf{id}_{Bool}, \\ Ks = s \quad & \& \quad Ki = i, \end{aligned}$$

where s is an object and i is a morphism in $\mathcal{I}$. These three functors are intended to be denotations of types nat, bool and key respectively.

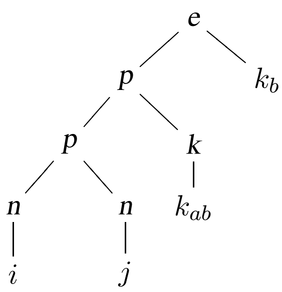
3.3.1 Dénotation de messages

For the message type msg, we define another functor $M : \mathcal{I} \to \mathcal{S}et$ by

- For every $s \in \mathcal{I}$, Ms is the smallest set which satisfies the following conditions:
  - If $a \in Ks$, then $\mathsf{k}(a) \in Ms$;
  - If $a \in Ns$, then $\mathsf{n}(a) \in Ms$;
  - If $a \in Ms$ and $k \in s$, then $\mathsf{e}(a, k) \in Ms$;
  - If $a_1, a_2 \in Ms$, then $\mathsf{p}(a_1, a_2) \in Ms$.
- For every $i : s \to s' \in \mathcal{I}$, Mi is a function from Ms to Ms' defined by:
  - If $x = \mathsf{n}(a)$ for some $a \in Ns$, then $Mi(x) = \mathsf{n}(Ni(a))$;

- If $x = k(a)$ for some $a \in Ks$, then $Mi(x) = k(Ki(a))$;
- If $x = e(a, k)$ for some $a \in Ms$ and $k \in s$, then $Mi(x) = e(Mi(a), Ki(k))$;
- If $x = p(a_1, a_2)$ for some $a_1, a_2 \in Ms$, then $Mi(x) = p(Mi(a_1), Mi(a_2))$.

Indeed, each Ms (for each $s \in \mathcal{I}$) can be regarded as a set of binary trees where an external node of such a tree is either a null value *nil*, an integer in the set Nat or a key in the set s, and each internal node is denoted by a symbol *n*, *k*, *e* or *p*, with some constraints (being consistent with the typing rules for messages in Figure 2.1), e.g., if a node is denoted by a symbol *e*, then its left child is another message tree and its right child is a key in the set s, which must be an external node. For instance, a message $e(p(p(n(i), n(j)), k(k_{ab})), k_b)$ is represented by:



.

A function Mi, for some $i : s \to s' \in \mathcal{I}$, simply maps a message tree in Ms to a tree in Ms', replacing each external node according to Ni and Ki, but without changing the tree structure. Since Ni is an identity function, Mi just changes external nodes of keys. For instance, the image of the above tree through Mi is the same tree except that k_{ab} and k_b are replaced by $i(k_{ab})$ and $i(k_b)$.

3.3.2 Interprétation du métalangage en $\mathcal{S}et^{\mathcal{I}}$

As in standard interpretations in cartesian closed categories, we translate types in the cryptographic metalanguage as objects of $\mathcal{S}et^{\mathcal{I}}$:

$$\begin{array}{rclrcl}
[\![\mathsf{nat}]\!] & = & N, & [\![\mathsf{bool}]\!] & = & B, \\
[\![\mathsf{key}]\!] & = & K, & [\![\mathsf{msg}]\!] & = & M, \\
[\![\tau \times \tau']\!] & = & [\![\tau]\!] \times [\![\tau']\!], & [\![\mathsf{opt}[\tau]]\!] & = & [\![\tau]\!] + 1_\perp, \\
[\![\tau \to \tau']\!] & = & [\![\tau']\!]^{[\![\tau]\!]}, & [\![\mathsf{T}\tau]\!] & = & \boldsymbol{T}[\![\tau]\!],
\end{array}$$

where we assume that $\perp$ is a terminal object in $\mathcal{S}et$ and $1_\perp$ is a terminal object in $\mathcal{S}et^{\mathcal{I}}$ defined by:

$$\forall s \in \mathcal{I},\ 1_\perp s = \{\perp\} \quad \text{and} \quad \forall i : s \to s' \in \mathcal{I},\ 1_\perp i = \mathbf{id}_{\{\perp\}}.$$

Each well-typed term $\Gamma \vdash t : \tau$ is then interpreted as a morphism from $[\![\Gamma]\!]$ to $[\![\tau]\!]$, where $[\![\Gamma]\!] = \prod_{x:\tau_i \in \Gamma} [\![\tau_i]\!]$. In other words, $[\![\Gamma \vdash t : \tau]\!]$ is a natural transformation such that for every

$s \in \mathcal{I}$, $[\![\Gamma \vdash t : \tau]\!]s$ is a function from $[\![\Gamma]\!]s$ to $[\![\tau]\!]s$. We then define a Γ-*environment* ρ, for every context Γ and every $s \in \mathcal{I}$, as a function which maps every variable x ($x : \tau \in \Gamma$) to an element of $[\![\tau]\!]s$.

Note that such an environment ρ can be seen as an element of $[\![\Gamma]\!]s$ and we shall write later on $\rho \in [\![\Gamma]\!]s$. When $[\![\Gamma]\!]s$ is an empty set, e.g., $\Gamma = \{x : \mathsf{key}\}$ and $s = \emptyset$, we simply mean that there is no such environment for these Γ and s. If $\rho \in [\![\Gamma]\!]s$, we write $\rho[x \mapsto a]$ as an environment mapping each variable $x' : \tau \in \Gamma$ to $\rho(x')$ and the variable x to the element a. We accordingly describe the meaning of a term $\Gamma \vdash t : \tau$ over a set s and in an environment $\rho \in [\![\Gamma]\!]s$, as a value $[\![\Gamma \vdash t : \tau]\!]s\rho$ in $[\![\tau]\!]s$, by induction on typing derivations, as shown in the Figure 3.1 and the Figure 3.2. We write $[\![t]\!]s$ instead of $[\![t]\!]s\rho$ when the environment ρ is irrelevant, e.g., t is a closed term.

We are using here some kind of free-algebra for interpreting those operations on the msg type. In particular, for the two cryptographic primitives `enc` and `dec`, a basic algebra property is required to hold:

$$\forall s \in \mathcal{I},\ a \in [\![\mathsf{msg}]\!]s,\ k \in s, \quad [\![\mathtt{dec}]\!]s([\![\mathtt{enc}]\!]s(a,k),k) = a.$$

As we have mentioned when introducing the msg type in our language, by defining such a general type for messages, we can easily extend our work to a richer language with more specific cryptographic primitives. All we need to do here is to redefine the denotation of the msg type so that it satisfies those specific algebraic properties for cryptography.

$$
\begin{array}{rcl}
[\![\Gamma, x:\tau \vdash x:\tau]\!]s\rho & = & \rho(x) \\
[\![\Gamma \vdash \lambda x.t : \tau \to \tau']\!]s\rho & = & \text{the unique } f \in [\![\tau \to \tau']\!]s \text{ such that} \\
 & & \text{for all } i : s \to s' \in \mathcal{I} \text{ and for all } a \in [\![\tau]\!]s', \\
 & & fs'(i,a) = [\![\Gamma, x:\tau \vdash t:\tau']\!]s'([\![\Gamma]\!]i(\rho) \cup \{x \mapsto a\}) \\
[\![\Gamma \vdash t_1 t_2 : \tau']\!]s\rho & = & [\![\Gamma \vdash t_1 : \tau \to \tau']\!]s\rho(\mathbf{id}_s, [\![\Gamma \vdash t_2 : \tau]\!]s\rho) \\
[\![\Gamma \vdash \langle t_1, t_2\rangle : \tau_1 \times \tau_2]\!]s\rho & = & ([\![\Gamma \vdash t_1 : \tau_1]\!]s\rho, [\![\Gamma \vdash t_2 : \tau_2]\!]s\rho) \\
[\![\Gamma \vdash \mathtt{proj}_i(t) : \tau_i]\!]s\rho & = & a_i, \text{ where } [\![\Gamma \vdash t : \tau_1 \times \tau_2]\!]s\rho = (a_1, a_2) \\
[\![\Gamma \vdash \mathtt{some}(t) : \mathsf{opt}[\tau]]\!]s\rho & = & [\![\Gamma \vdash t : \tau]\!]s\rho \\
[\![\Gamma \vdash \mathtt{case}\ t_1\ \mathtt{of}\ \mathtt{some}(x)\ \mathtt{in}\ t_2\ \mathtt{else}\ t_3 : \tau']\!]s\rho & = & \left\{ \begin{array}{l} [\![\Gamma, x:\tau \vdash t_2 : \tau']\!]s\rho \cup \{x \mapsto a\}, \text{ if } [\![\Gamma \vdash t : \mathsf{opt}[\tau]]\!]s\rho = a \neq \bot \\ [\![\Gamma \vdash t_3 : \tau']\!]s\rho \text{ if } [\![\Gamma \vdash t : \mathsf{opt}[\tau]]\!]s\rho = \bot \end{array} \right. \\
[\![\Gamma \vdash \mathtt{val}(t) : \mathsf{T}\tau]\!]s\rho & = & [\emptyset, [\![\Gamma \vdash t : \tau]\!]s\rho] \\
[\![\Gamma \vdash \mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2 : \mathsf{T}\tau']\!]s\rho & = & [s_1 + s_2,\ a_2], \text{where } [\![\Gamma \vdash t_1 : \mathsf{T}\tau]\!]s\rho = [s_1, a_1] \text{ and} \\
 & & [\![\Gamma, x:\tau \vdash t_2 : \mathsf{T}\tau']\!](s+s_1)\rho' = [s_2, a_2] \\
 & & \text{where } \rho'(x) = a_1,\ \rho'(y_i) = [\![\tau_i]\!](\mathbf{inl}_{s,s_1})(\rho(y_i)) \text{ for any } y_i : \tau_i \in \Gamma \\
[\![\Gamma \vdash \mathtt{new} : \mathsf{key}]\!]s\rho & = & [\{k\}, k],\ \text{where } k \notin s
\end{array}
$$

Figure 3.1: Interpretation of the metalanguage terms (I)

$$
\begin{array}{rcl}
[\![\Gamma \vdash \mathtt{enc}(t_1, t_2) : \mathsf{msg}]\!] s\rho & = & e([\![\Gamma \vdash t_1 : \mathsf{msg}]\!] s\rho, [\![\Gamma \vdash t_2 : \mathsf{key}]\!] s\rho) \\
[\![\Gamma \vdash \mathtt{dec}(t_1, t_2) : \mathsf{opt}[\mathsf{msg}]]\!] s\rho & = & \left\{ \begin{array}{ll} a, & \text{if } [\![\Gamma \vdash t_1 : \mathsf{msg}]\!] s\rho = e(a, k) \text{ and } [\![\Gamma \vdash t_2 : \mathsf{key}]\!] s\rho = k, \\ & \text{for some } a \in [\![\mathsf{msg}]\!] s \text{ and } k \in s \\ \bot, & \text{otherwise} \end{array} \right. \\
[\![\Gamma \vdash \mathtt{p}(t_1, t_2) : \mathsf{msg}]\!] s\rho & = & p([\![\Gamma \vdash t_1 : \mathsf{msg}]\!] s\rho, [\![\Gamma \vdash t_2 : \mathsf{msg}]\!] s\rho) \\
[\![\Gamma \vdash \mathtt{fst}(t) : \mathsf{msg}]\!] s\rho & = & \left\{ \begin{array}{ll} a_1, & \text{if } [\![\Gamma \vdash t : \mathsf{msg}]\!] s\rho = p(a_1, a_2) \text{for some } a_1, a_2 \in [\![\mathsf{msg}]\!] s \\ \bot, & \text{otherwise} \end{array} \right. \\
[\![\Gamma \vdash \mathtt{snd}(t) : \mathsf{msg}]\!] s\rho & = & \left\{ \begin{array}{ll} a_2, & \text{if } [\![\Gamma \vdash t : \mathsf{msg}]\!] s\rho = p(a_1, a_2) \text{for some } a_1, a_2 \in [\![\mathsf{msg}]\!] s \\ \bot, & \text{otherwise} \end{array} \right. \\
[\![\Gamma \vdash \mathtt{n}(t) : \mathsf{msg}]\!] s\rho & = & n([\![\Gamma \vdash t : \mathsf{nat}]\!] s\rho) \\
[\![\Gamma \vdash \mathtt{k}(t) : \mathsf{msg}]\!] s\rho & = & k([\![\Gamma \vdash t : \mathsf{key}]\!] s\rho) \\
[\![\Gamma \vdash \mathtt{getnum}(t) : \mathsf{opt}[\mathsf{nat}]]\!] s\rho & = & \left\{ \begin{array}{ll} a, & \text{if } [\![\Gamma \vdash t : \mathsf{msg}]\!] s\rho = n(a) \text{ for some } a \in Nat \\ \bot, & \text{otherwise} \end{array} \right. \\
[\![\Gamma \vdash \mathtt{getkey}(t) : \mathsf{opt}[\mathsf{key}]]\!] s\rho & = & \left\{ \begin{array}{ll} a, & \text{if } [\![\Gamma \vdash t : \mathsf{msg}]\!] s\rho = k(a) \text{ for some } a \in s \\ \bot, & \text{otherwise} \end{array} \right.
\end{array}
$$

Figure 3.2: Interpretation of the metalanguage terms (II)

3.4 Formes canoniques

In this section, we define the canonical form of terms in the cryptographic metalanguage and show that every term in the metalanguage has an equivalent canonical term (with the same semantics). Precisely, for every well-typed term $\Gamma \vdash t : \tau$ and for every $s \in \mathcal{I}$ and $\rho[\![\Gamma]\!]s$, if $[\![t]\!]s\rho \neq \bot$, then there exists a term $\Gamma \vdash t' : \tau$ in canonical form such that $[\![t]\!]s\rho = [\![t']\!]s\rho$.

A term in the cryptographic metalanguage is said to be in the *canonical form* if it is a term defined by the following grammar:

$$\begin{array}{rcl} u & ::= & x \mid n \mid \mathtt{true} \mid \mathtt{false} \mid \langle u, u\rangle \mid \mathtt{some}(u) \mid m \mid \\ & & \mathtt{let}\ s \Leftarrow \overline{\mathtt{new}}\ \mathtt{in}\ \mathtt{val}(u) \mid \lambda x_1 \cdots \lambda x_m.ut_1 \cdots t_{m'} \quad (m, m' \geq 0) \\ & & \mathtt{n}(x) \mid \mathtt{n}(n) \mid \mathtt{getnum}(u) \mid \mathtt{k}(x) \mid \mathtt{getkey}(u) \mid \mathtt{enc}(u, x) \mid \mathtt{dec}(u, x) \mid \\ & & \mathtt{p}(u, u) \mid \mathtt{fst}(u) \mid \mathtt{snd}(u), \end{array}$$

where n are integer constants $(0, 1, 2, \ldots)$ and $t_1, \ldots, t_m$ are terms in the metalanguage. The term $\mathtt{let}\ s \Leftarrow \overline{\mathtt{new}}\ \mathtt{in}\ \mathtt{val}(u)$, where $s = \{x_1, \ldots, x_n\}$ $(n \geq 0)$ is a set of bound variables of type key, abbreviates

$$\mathtt{let}\ x_1 \Leftarrow \mathtt{new}\ \mathtt{in} \cdots \mathtt{let}\ x_n \Leftarrow \mathtt{new}\ \mathtt{in}\ \mathtt{val}(u).$$

We shall also write $\overline{s : \mathsf{key}}$ for the typing context $x_1 : \mathsf{key}, \ldots, x_n : \mathsf{key}$, and $[\overline{s \mapsto s}]$ for the environment $[x_1 \mapsto x_1, \ldots, x_n \mapsto x_n]$, where we regard keys variables as identical as keys.

Because all canonical terms must be well typed, it is easy to check that canonical terms of type key must be variables (the constant $\mathtt{new}$ is of type Tkey, not key, and a term $\mathtt{getkey}(t)$ has type opt[key], not key either). $\mathtt{enc}(m, x)$ and $\mathtt{dec}(m, x)$ are canonical forms for encryption and decryption where x must be a variable. In particular, in a canonical decryption term $\mathtt{dec}(u, x)$, u must be of type msg, so it cannot be decryption $\mathtt{dec}(u', x')$, $\mathtt{fst}(u')$ or $\mathtt{snd}(u')$, because they are of type opt[msg]). If u is any other form besides an encryption, the value of $\mathtt{dec}(u, x)$ is always $\bot$, so we may further require that in canonical form $\mathtt{dec}(u, x)$, u must be of the form $\mathtt{enc}(u', x')$.

Lemma 3.1. *For every $s \in \mathcal{I}$ and every environment $\rho \in [\![\Gamma]\!]s$, if u_1, u_2 are two canonical terms such that $\Gamma, x : \tau \vdash u_1 : \tau'$ and $\Gamma \vdash u_2 : \tau$ hold, then either $[\![u_1[u_2/x]]\!]s\rho = \bot$, or there is another canonical term u such that $\Gamma \vdash u : \tau'$ hold and $[\![u]\!]s\rho = [\![u_1[u_2/x]]\!]s\rho$.*

Proof. We prove the statement by induction on the structure of u_1. Only some particular cases are listed here.

- $u_1 \equiv \lambda x_1. \cdots \lambda x_m.u'_1 t_1 \cdots t_{m'}$: By induction, there exists canonical term u''_1 such that $[\![u'_1[u_2/x]]\!]s\rho = [\![u''_1]\!]s\rho$, then $[\![u_1[u_2/x]]\!]s\rho = [\![\lambda x_1. \cdots \lambda x_m.u''_1 t'_1 \cdots t'_{m'}]\!]s\rho$, where $t'_1 = t_1[u_2/x]$, ... , $t'_{m'} = t_{m'}[u_2/x]$.

- $u_1 \equiv \mathtt{let}\ s_1 \Leftarrow \overline{\mathtt{new}}\ \mathtt{in}\ \mathtt{val}(u_1')$: Clearly, $\Gamma, \overline{s_1 : \mathsf{key}}, x : \tau \vdash u_1' : \tau'$ holds, then by induction, there is a canonical term u_1'' such that $\Gamma, \overline{s_1 : \mathsf{key}}, x : \tau \vdash u_1' : \tau'$ holds and $[\![u_1'']\!](s+s_1)\mathbf{inl}_{s,s_1}(\rho) = [\![u_1[u_2/x]]\!](s+s_1)\mathbf{inl}_{s,s_1}(\rho)$, if $[\![u_1[u_2/x]]\!](s+s_1)\mathbf{inl}_{s,s_1}(\rho) \neq \perp$. Therefore, $[\![u_1[u_2/x]]\!]s\rho = [\![\mathtt{let}\ s_1 \Leftarrow \overline{\mathtt{new}}\ \mathtt{in}\ \mathtt{val}(u_1'')]\!]s\rho$.

- $u_1 \equiv \mathtt{dec}(u_1', y)$: By induction, if $[\![u_1'[u_2/x]]\!]s\rho \neq \perp$, there is a canonical term u_1'' such that $[\![u_1'']\!]s\rho = [\![u_1'[u_2/x]]\!]s\rho$. Then $[\![u_1[u_2/x]]\!]s\rho = [\![\mathtt{enc}(u_1'', y)]\!]s\rho$, where $\mathtt{enc}(u_1'', y)$ is canonical.

- $u_1 \equiv \mathtt{dec}(u_1', y)$: If $[\![u_1'[u_2/x]]\!]s\rho \neq \perp$, by induction, there is a canonical term u_1'' such that $[\![u_1'']\!]s\rho = [\![u_1'[u_2/x]]\!]s\rho$, so $[\![u_1[u_2/x]]\!]s\rho = [\![\mathtt{dec}(u_1'', y)]\!]s\rho$. Clearly, the type of u_1'' must be msg. The only case where $[\![u_1[u_2/x]]\!]s\rho \neq \perp$ is that $u_1'' \equiv \mathtt{enc}(u_1''', z)$, for another canonical term u_1''', and $\rho(y) = \rho(z)$. In this case, $[\![u_1[u_2/x]]\!]s\rho = [\![u_1''']\!]s\rho$. In any other case, $[\![u_1[u_2/x]]\!]s\rho = \perp$. □

Proposition 3.2. *Let t be a term such that $\Gamma \vdash t : \tau$ is derivable. For every $s \in \mathcal{I}$ and $\rho \in [\![\Gamma]\!]s$, if $[\![t]\!]s\rho \neq \perp$, there exists a canonical term u such that $\Gamma \vdash u : \tau$ holds and $[\![t]\!]s\rho = [\![u]\!]s\rho$.*

Proof. We prove the statement by induction on the structure of term t. We show only the cases of functions, applications, computations and two message operations — encryption and decryption. Other cases are standard.

- $t \equiv \lambda x.t'$: Assume that t is of type $\tau \to \tau'$ and let $f = [\![t]\!]s\rho$. By the definition of $[\![\_]\!]s\rho$, for every $i : s \to s' \in \mathcal{I}$ and every value $a \in [\![\tau]\!]s'$, $fs'(i, a) = [\![t']\!]s'([\![\Gamma]\!]i(\rho)[x \mapsto a]) = [\![u']\!]s'([\![\Gamma]\!]i(\rho)[x \mapsto a])$, where by induction u is a term in canonical form. Then $\lambda x.u$ is also a canonical term and $[\![\lambda x.u]\!]s\rho = [\![\lambda x.t]\!]s\rho$.

- $t \equiv t_1t_2$: $[\![t]\!]s\rho = ([\![t_1]\!]s\rho)s(\mathbf{id}_s, [\![t_2]\!]s\rho)$. By induction, $[\![t_1]\!]s\rho = [\![u_1]\!]s\rho$ and $[\![t_2]\!]s\rho = [\![u_2]\!]s\rho$, where u_1 and u_2 are two canonical terms. u_1 must be of a function type, so it is
  - either a variable $y \in \Gamma$, then $[\![t]\!]s\rho = ([\![y]\!]s\rho)s(\mathbf{id}_s, [\![t_2]\!]s\rho) = [\![yt_2]\!]s\rho$, and yt_2 is in canonical form;
  - or an abstraction $\lambda x_1 \cdots x_n.u_1't_1' \cdots t_m'$, then

$$\begin{aligned}
[\![t]\!]s\rho &= \left[\!\left[\lambda x_2 \cdots x_n.u_1't_1' \cdots t_m'\right]\!\right]s\rho[x_1 \mapsto [\![u_2]\!]s\rho] \\
&= \left[\!\left[\lambda x_2 \cdots x_n.(u_1'[u_2/x_1])t_1'' \cdots t_m''\right]\!\right]s\rho \\
&= \left[\!\left[\lambda x_2 \cdots x_n.yt_1'' \cdots t_m''\right]\!\right]s\rho[y \mapsto \left[\!\left[u_1'[u_2/x_1]\right]\!\right]s\rho] \\
&= \left[\!\left[\lambda x_2 \cdots x_n.yt_1'' \cdots t_m''\right]\!\right]s\rho[y \mapsto \left[\!\left[u_1''\right]\!\right]s\rho] \\
&\quad \text{(by Lemma 3.1, } u_1'' \text{ is canonical and } [\![u_1'']\!]s\rho = [\![u_1'[u_2/x]]\!]s\rho\text{)} \\
&= \left[\!\left[\lambda x_2 \cdots x_n.u_1''t_1'' \cdots t_m''\right]\!\right]s\rho
\end{aligned}$$

where $t''_i = t'_i[u_2/x]$ $(i = 1, \ldots, m)$, and by Lemma 3.1, u''_1 is canonical, then $\lambda x_2 \cdots x_n.u''_1 t''_1 \cdots t''_m$ is canonical as well.

- $t \equiv \mathtt{new}$: $[\![\mathtt{new}]\!]s\rho = [\![\mathtt{let}\ y \Leftarrow \mathtt{new}\ \mathtt{in}\ \mathtt{val}(y)]\!]s\rho$, for some variable $y \notin \Gamma$.

- $t \equiv \mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2$: By induction, there are canonical terms $\mathtt{let}\ s_1 \Leftarrow \overline{\mathtt{new}}\ \mathtt{in}\ \mathtt{val}(u_1)$ and $\mathtt{let}\ s_2 \Leftarrow \overline{\mathtt{new}}\ \mathtt{in}\ \mathtt{val}(u_2)$ such that

$$\begin{aligned}[\![t_1]\!]s\rho &= [\![\mathtt{let}\ s_1 \Leftarrow \overline{\mathtt{new}}\ \mathtt{in}\ \mathtt{val}(u_1)]\!]s\rho \\ &= [s_1, [\![u_1]\!](s+s_1)\mathbf{inl}_{s,s_1}(\rho)[\overline{s_1 \mapsto s_1}]]\end{aligned}$$

and

$$\begin{aligned}[\![t_2]\!](s+s_1)\rho' &= [\![\mathtt{let}\ s_2 \Leftarrow \overline{\mathtt{new}}\ \mathtt{in}\ \mathtt{val}(u_2)]\!](s+s_1)\rho' \\ &= [s_2, [\![u_2]\!](s+s_1+s_2)\mathbf{inl}_{s+s_1,s_2}(\rho')[\overline{s_2 \mapsto s_2}]]\end{aligned}$$

where $\rho' = \mathbf{inl}_{s,s_1}(\rho)[x \mapsto a]$ for every $a \in [\![\tau']\!](s+s_1)$ (assuming that t_1 is of type $\mathsf{T}\tau'$). Let $a' = [\![u_1]\!](s+s_1)\mathbf{inl}_{s,s_1}(\rho)[\overline{s_1 \mapsto s_1}]$, then

$$[\![\tau']\!]\mathbf{inl}_{s+s_1,s_2}(a') = [\![u_1]\!](s+s_1+s_2)\mathbf{inl}s, s_1+s_2(\rho)[\overline{s_1 \mapsto s_1}],$$

hence,

$$\begin{aligned}&[\![u_2]\!](s+s_1+s_2)\mathbf{inl}_{s+s_1,s_2}(\mathbf{inl}_{s,s_1}(\rho[x \mapsto a'])[\overline{s_1 \mapsto s_1}, \overline{s_2 \mapsto s_2}] \\ =\ &[\![u_2]\!](s+s_1+s_2)\mathbf{inl}_{s,s_1+s_2}(\rho)[x \mapsto [\![\tau']\!]\mathbf{inl}_{s+s_1,s_2}(a'), \overline{s_1 \mapsto s_1}, \overline{s_2 \mapsto s_2}] \\ =\ &[\![u_2[u_1/x]]\!](s+s_1+s_2)\mathbf{inl}_{s,s_1+s_2}(\rho)[\overline{s_1 \mapsto s_1}, \overline{s_2 \mapsto s_2}] \\ =\ &[\![u']\!](s+s_1+s_2)\mathbf{inl}_{s,s_1+s_2}(\rho)[\overline{s_1 \mapsto s_1}, \overline{s_2 \mapsto s_2}],\end{aligned}$$

where by Lemma 3.1, u' is a canonical term such that $\Gamma, \overline{s_1, \mathsf{key}}, \overline{s_2, \mathsf{key}} \vdash u' : \tau$ holds. According to the interpretation in the model $\mathcal{S}et^{\mathcal{I}}$,

$$[\![t]\!]s\rho = [\![\mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2]\!]s\rho = [\![\mathtt{let}\ s_1 + s_2 \Leftarrow \overline{\mathtt{new}}\ \mathtt{in}\ \mathtt{val}(u')]\!]s\rho.$$

Clearly, $\mathtt{let}\ s_1 + s_2 \Leftarrow \overline{\mathtt{new}}\ \mathtt{in}\ \mathtt{val}(u')$ is canonical.

- $t \equiv \mathtt{enc}(t_1, t_2)$: by induction, there are canonical terms u_1 and u_2 such that $[\![t_1]\!]s\rho = [\![u_1]\!]s\rho$ and $[\![t_2]\!]s\rho = [\![u_2]\!]s\rho$, then

$$[\![t]\!]s\rho = e([\![t_1]\!]s\rho, [\![t_2]\!]s\rho) = e([\![u_1]\!]s\rho, [\![u_2]\!]s\rho) = [\![\mathtt{enc}(u_1, u_2)]\!]s\rho.$$

In particular, u_2 is of type key, so it must be a variable, so $\mathtt{enc}(u_1, u_2)$ is canonical.

- $t \equiv \mathtt{dec}(t_1, t_2)$: If $[\![t_1]\!]s\rho = e(a, [\![t_2]\!]s\rho)$ for some $a \in [\![\mathsf{msg}]\!]s$, by induction there is a canonical term $\mathtt{enc}(u_1, u_2)$ such that $[\![t_1]\!]s\rho = [\![\mathtt{enc}(u_1, u_2)]\!]s\rho$, then $[\![t]\!]s\rho = a = [\![u_1]\!]s\rho$ and u_1 is canonical. Otherwise, $[\![t]\!]s\rho = \bot$. □

Lemma 3.3. *For every term t such that $\overline{w : \mathsf{key}} \vdash t : \mathsf{msg}$ is derivable, $[\![t]\!]s\rho \in [\![\mathsf{msg}]\!]\rho(w)$, for every $s \in \mathcal{I}$ and $\rho \in [\![\overline{\mathsf{key}}]\!]s$.*

Proof. By Proposition 3.2, we simply consider the canonical form of t (denoted by u), and since there is no free variable of msg type and nat type, u could only be $\mathtt{k}(x)$, $\mathtt{k}(n)$, $\mathtt{enc}(u_1, u_2)$ or $\mathtt{p}(u_1, u_2)$. Then we can prove the desired property by induction on the structure of u. □

Given a typing context Γ and an environment $\rho \in [\![\Gamma]\!]s$, we say that a value $a \in [\![\tau]\!]s$ is *definable*, if there is a term t such that $\Gamma \vdash t : \tau$ is derivable, and $[\![t]\!]s\rho = a$. Note that $\rho(w) \subseteq s$, hence $[\![\mathsf{msg}]\!]\rho(w) \subseteq [\![\mathsf{msg}]\!]s$. The above lemma actually states the definability of elements in $[\![\mathsf{msg}]\!]s$ — a value in $[\![\mathsf{msg}]\!]s$ is definable if and only if it belongs to $[\![\mathsf{msg}]\!]\rho(w)$. Definability will be largely involved in the discussion of Chapter 6, where we shall concentrate on the completeness of logical relations.

3.5 Équivalence contextuelle

Contextual equivalence is a crucial notion in our approach, since we use it to describe security properties of protocols. Although there are standard definitions in lambda-calculus, it turns out that they do not fit well in the cryptographic metalanguage. Indeed, contextual equivalence for cryptographic protocols is very subtle and requires careful treatment.

We start by defining contextual equivalence in the simply-typed lambda-calculus. Contexts are simply programs, but we are not interested in arbitrary programs. We do not get concrete sense from values like functions or cipher-texts, so we consider only those programs that return observable values. Fix a set $\mathbf{Obs}$ of so-called *observation types*. Usually, $\mathbf{Obs}$ consists of any base type with decidable equality. For instance, bool and nat in the cryptographic metalanguage are observation types, but types key and msg are not. Usually, in a set-theoretic model of simply-typed lambda calculi, we say that two values $a_1, a_2 \in [\![\tau]\!]$, for the same type τ, are *contextually equivalent*, written as $a_1 \approx_\tau a_2$, if and only if, whatever the term $\mathbb{C}$ such that $x : \tau \vdash \mathbb{C} : o$ ($o \in \mathbf{Obs}$) is derivable,

$$[\![\mathbb{C}]\!][x \mapsto a_1] = [\![\mathbb{C}]\!][x \mapsto a_2].$$

Two closed terms t_1 and t_2 such that $\vdash t_1 : \tau$ and $\vdash t_2 : \tau$ are derivable, are contextually equivalent (written as $t_1 \approx_\tau t_2$), if and only if their denotations (in the set-theoretical model) are contextually equivalent.

Intuitively, this (essentially standard) notion captures the fact that we would like to consider t_1 and t_2 as equivalent provided, whatever the question we ask about them, the answer is the same for t_1 and t_2. Asking a question about t means executing t in a context (a.k.a., an operating system) $\mathbb{C}$ with an observation type, bool for example, i.e., running $\mathbb{C}$ when x takes the value of t and watching the output. If the answer differs for $t = t_1$ and for $t = t_2$ in the context $\mathbb{C}$, then there is an observable difference between t_1 and t_2.

Contextual equivalence for the key generation monad

Defining contextual equivalence in the cryptographic metalanguage (and hence in the categorical model $\mathcal{S}et^{\mathcal{I}}$) is a bit tricky. First, we have to consider contexts $\mathbb{C}$ of type $\mathsf{T}o$ ($o \in \mathbf{Obs}$), not of type o. Intuitively, contexts should be allowed to do some computations; were they of type o, they could only return values. In particular, note that contexts $\mathbb{C}$ such that $x : \mathsf{T}\tau \vdash \mathbb{C} : o$ is derivable, meant to observe computations at type τ, cannot observe anything. This is because the only way we can make use of values of computations in the metalanguage is to put computations in the `let` construction, while the (let) typing rule only allows us to use computations to build other computations, never values.

Another tricky aspect is that attackers in the metalanguage are actually modeled by contexts where a protocol can execute. Hence we cannot take contexts $\mathbb{C}$ that only depend on one variable $x : \tau$ as before. We must indeed assume that $\mathbb{C}$ can also depend on an arbitrary set of keys, keys disclosed to contexts (attackers). Given keys $k_1, \ldots, k_n$, the only way $\mathbb{C}$ can be made to depend on them is to assume that $\mathbb{C}$ has n free variables $z_1, \ldots, z_n$ of type key, which are mapped to $k_1, \ldots, k_n$. (It is more standard [PS93a, AG99] to consider expressions built on separate sets of variables and keys, thus introducing the semantic notion of keys in the syntax. It is more natural here to consider that there are variables z_m mapped, in a one-to-one way, to keys k_m.) Let s' be a set of keys containing $k_1, \ldots, k_n$, let w' be $\{z_1, \ldots, z_n\}$, and $i' : w' \to s'$ the injection mapping each z_m to k_m ($1 \leq m \leq n$). We shall use i' to denote the environment mapping every variable z_i to k_i:

$$i' \equiv [z_1 \mapsto k_1, \ldots, z_n \mapsto k_n].$$

We then consider contexts $\mathbb{C}$ such that $\overline{w' : \mathsf{key}}, x : \tau \vdash \mathbb{C} : \mathsf{T}o$ is derivable, evaluate $[\![\mathbb{C}]\!] s' i' [x \mapsto a_1]$ and compare it with $[\![\mathbb{C}]\!] s' i' [x \mapsto a_2]$ to decide whether a_1 and a_2 are contextually equivalent. This represents the fact that $\mathbb{C}$ is evaluated in a world where all keys in s' have been created, and where $\mathbb{C}$ has access to all (disclosed) keys in $i'(w')$.

This definition is not yet correct, as this would require a_1 and a_2 to be in $[\![\tau]\!] s'$, but they are in $[\![\tau]\!] s$ for some possibly different set s of keys created during the evaluation of a_1 and a_2. This is repaired by considering a coercion $[\![\tau]\!] l$, where $l : s \to s'$ is an injection in $\mathcal{I}$. We then arrive at the following definition of Contextual equivalence:

Definition 3.1. *$a_1, a_2 \in [\![\tau]\!]s$ are* contextually equivalent at *s, written as $a_1 \approx_\tau^s a_2$, if and only if, for every finite set of variables w', for any injections $i' : w' \to s'$ and $l : s \to s'$, for every term $\mathbb{C}$ such that*

$$\overline{w' : \mathsf{key}}, x : \tau \vdash \mathbb{C} : \mathsf{T}o, \quad (o \in \mathbf{Obs})$$

is derivable,

$$[\![\mathbb{C}]\!]s'i'[x \mapsto [\![\tau]\!]l(a_1)] = [\![\mathbb{C}]\!]s'i'[x \mapsto [\![\tau]\!]l(a_2)].$$

This notion is inspired by [PS93a, Definition 4]. Here we consider only contexts having access to *all* keys in s', i.e., $i'(w') = s'$, and we lose no generality by this simplification. Moreover, we can just equate w' with s', then $a_1 \approx_\tau^s a_2$ if and only if, for every injection $l : s \to s'$, for every term $\mathbb{C}$ such that

$$\overline{s' : \mathsf{key}}, x : \tau \vdash \mathbb{C} : \mathsf{T}o, \quad (o \in \mathbf{Obs})$$

is derivable,

$$[\![\mathbb{C}]\!]s'\mathbf{id}_{s'}[x \mapsto [\![\tau]\!]l(a_1)] = [\![\mathbb{C}]\!]s'\mathbf{id}_{s'}[x := [\![\tau]\!]l(a_2)],$$

where we see the *variables* in s' as denoting the *keys* in s' here, equating keys with variables.

Chapitre 4

Relations logiques

Il n'est pas facile de prouver l'équivalence contextuelle directement à cause de la quantification universelle sur les contextes. Heureusement, dans le lambda-calcul, nous pourrons déduire l'équivalence contextuelle en utilisant une technique appelé *relations logiques*. C'est un outil puissant des lambda-calculs typés qui permet de prouver de nombreux résultats importants dans ce domaine (voir [Mit96] pour une liste de résultats qui peuvent être prouvés à l'aide de relations logiques).

Essentiellement, une relation logique est un ensemble de relations, une pour chaque type, et les relations sont définies par récurrence sur les types. Le point crucial est que deux fonctions (forcément du même type) sont reliées si et seulement si les images de deux données reliées sont reliées. En général, les relations logiques sont définies d'une façon dénotationnelle [Plo80, Mit96] et la construction sur les catégories cartésiennes fermées permet de déduire les relations logiques des lambda-calculs simplement typés. Avec cette construction, nous pouvons définir les relations logiques sur presque tous les modèles concrets des lambda-calculs simplement typés, par exemple sur les modèles basés sur les ensembles ou sur la catégorie de foncteurs $\mathcal{S}et^{\mathcal{I}}$.

Pourtant, cette construction générale n'est pas suffisante pour déduire une relation logique d'un langage avec les types monadiques, puisqu'elle ignore la présence éventuelle de monades. Cette difficulté a été réglée par Goubault-Larrecq, Lasota et Nowak dans [GLLN02], où ils définissent une construction générale de relations logiques monadiques sur les catégories avec monades. Ainsi, ils montrent que certaines conditions doivent être satisfaites pour définir les relations logiques monadiques. Ils appliquent leur méthode à de nombreuses monades concrètes, y compris celle du modèle $\mathcal{S}et^{\mathcal{I}}$, et ils obtiennent des relations logiques concrètes pour des effets de bord divers.

Pourtant, nous observons que le modèle $\mathcal{S}et^{\mathcal{I}}$ n'est pas suffisant pour étudier les relations entre les programmes de notre métalangage, donc nous ne pouvons pas déduire sur cette catégorie des relations logiques assez puissantes pour le métalangage cryptographique. Afin de définir une relation logique pour la monade de génération de clés, nous avons besoin d'une catégorie plus riche, c'est-à-dire d'une catégorie avec plus d'informations.

Le chapitre 4, ainsi que le chapitre suivant, porte sur la construction des relations logiques du métalangage cryptographique. Cela comporte tout d'abord une discussion sur ce que doit être la catégorie correcte pour dériver les relations logiques.

Les deux premières parties sont des parties préliminaires sur les relations logiques. La partie 4.1 comporte une définition standard des relations logiques du lambda-calcul, notamment la construction catégorique basée sur les catégories cartésiennes fermées. Pour exemple, nous montrons comment dériver une relation logique sur la catégorie $\mathcal{S}et^{\mathcal{I}}$ (sans monade) selon la construction générale. Nous étendons ensuite la construction pour déduire les relations logiques monadiques sur les catégories avec monades. Plusieurs exemples sont donnés dans cette partie, y compris la monade de génération de clés (sur la catégorie $\mathcal{S}et^{\mathcal{I}}$). Pourtant, dans la partie 4.3, nous verrons que les relations logiques dérivées sur le modèle $\mathcal{S}et^{\mathcal{I}}$ est trop faible — nous ne pouvons pas relier des programmes qui sont équivalents contextuellement d'une manière évidente. Nous définissons alors une nouvelle catégorie $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ et la construction des relations logiques sur cette nouvelle catégorie est exposée dans la partie 4.4. Nous arrivons donc à une relation logique du métalangage cryptographique dans la partie 4.5, en définissant les relations des types bases, notamment du type msg. Nous vérifions aussi quelques propriétés de cette relation logique.

Contextual equivalence is not easy to prove directly, notably because of the universal quantification over an infinite number of contexts. In typed lambda-calculi, we are usually able to deduce contextual equivalence using a technique called *logical relations*. This is a powerful tool in typed lambda-calculi, which allows us to prove a number of important results in this domain (see [Mit96] for a list of results that can be proved via logical relations).

Essentially, a logical relation is a family of relations, one for each type, defined inductively. The crux is that relations for functions must be determined from the relations for arguments and results, in a way that guarantees closure under application and lambda abstraction. Logical relations are usually defined in a denotational way [Plo80, Mit96]. In particular, the construction based on cartesian closed categories [MS93, MR92] defines a general way for constructing logical relations in typed lambda-calculi. Following this construction, we can define logical relations over almost all concrete models of typed lambda-calculi (necessarily sound), e.g., a set-theoretical model or the functor category $\mathcal{S}et^{\mathcal{I}}$.

However, this general construction is not enough to construct logical relations for a language with monadic types, since a CCC is not equipped with a monad in general, hence there is no standard way to derive relations for monadic types. This was mended by the work of Goubault-Larrecq, Lasota and Nowak [GLLN02]. They define a general construction of logical relations over categories with monads, and show that for defining such logical relations, the model must satisfy certain properties. They also apply this method to a number of concrete monads, including the model $\mathcal{S}et^{\mathcal{I}}$, and obtain concrete logical relations for various forms of computation.

However, we observe that the model $\mathcal{S}et^{\mathcal{I}}$ is indeed insufficient for the study of relations between programs of the metalanguage, hence it is insufficient too for us to define logical relations for the cryptographic metalanguage. In order to define logical relations for dynamic key generation, we need a category with more information.

Chapter 4 and Chapter 5 are mainly about deriving logical relations for the cryptographic metalanguage. This includes in the first place a discussion on what should be the right category for deriving logical relations.

Section 4.1 and Section 4.2 are two introductory sections. Section 4.1 is about the standard definition of logical relations, in particular the categorical construction in cartesian closed categories. As an example, we also show how to follow the categorical construction to derive a logical relation over the category $\mathcal{S}et^{\mathcal{I}}$ (without monad). Section 4.2 then extends this construction to categories with monads. Some examples are also given in this section, notably the dynamic name creation monad (the model $\mathcal{S}et^{\mathcal{I}}$ with monad). Since logical relations derived over this category are indeed too weak to recognize some obviously contextually equivalent programs, we define a new category $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ in Section 4.3. Section 4.4 shows how to derive logical relations over this new category. By defining relations for base types, notably the msg type, we then arrive at a logical relation for the cryptographic metalanguage, in Section 4.5. Some properties are also checked about this logical relation.

However, logical relations derived over the category $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ are still too weak. We start Chapter 5 by a counter-example showing the weakness of the category $\mathcal{I}^{\rightarrow}$, then in Section 5.1 we revise $\mathcal{I}^{\rightarrow}$ by adding some constraints and call the revised category $\mathcal{PI}^{\rightarrow}$. In Section 5.2, we show that $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ is the right category for deriving logical relations for the cryptographic metalanguage. We arrive finally at a cryptographic logical relation for the metalanguage in Section 5.3 and check certain properties. We show that this logical relation can be used to verify protocols in Section 5.4, by checking the two protocols presented in Chapter 2. In Section 5.5, we do some comparison with logical relations for the nu-calculus. We in particular show that logical relations derived over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ are equivalent to Stark's (denotational) logical relations [Sta94].

4.1 Relations logiques

Consider a set-theoretical model of the simply-typed lambda calculus. A (binary) *logical relation* is a family $(\mathcal{R}_\tau)_{\tau \text{ type}}$ of binary relations $\mathcal{R}_\tau$ on $[\![\tau]\!]$, one for each type τ, which are defined by induction on the type structure. In particular, two functions are related if and only if they map related arguments to related results. Precisely, for every pair of functions $f_1, f_2 \in [\![\tau \rightarrow \tau']\!]$, the following condition must be always satisfied:

$$(\mathbf{Log}) \qquad f_1 \; \mathcal{R}_{\tau\rightarrow\tau'} \; f_2 \iff \forall a_1, a_2 \in [\![\tau]\!] \cdot a_1 \; \mathcal{R}_\tau \; a_2 \Rightarrow f_1(a_1) \; \mathcal{R}_{\tau'} \; f_2(a_2).$$

This is the standard definition of logical relations in the λ-calculus [Mit96]. We write $a_1 \; \mathcal{R} \; a_2$ to say that a_1 and a_2 are related by the binary relation $\mathcal{R}$.

Note that there is no constraint on relations for base types. In a simply-typed lambda calculus with only base types and function types, once the relations $\mathcal{R}_b$, for any base type b, are fixed, the condition above forces $(\mathcal{R}_\tau)_{\tau \text{ type}}$ to be uniquely determined, by induction on types. It is certainly possible to derive relations for other complex types by induction, for example, two pairs are related if and only if the components are related respectively.

The $(\mathbf{Log})$ condition entails notably the so-called *basic lemma*. To state it, first say that two Γ-environments ρ_1, ρ_2 are *related* by the logical relation, in notation $\rho_1 \; \mathcal{R}_\Gamma \; \rho_2$, if and only if $\rho_1(x) \; \mathcal{R}_\tau \; \rho_2(x)$ for every $x : \tau$ in Γ. The basic lemma states that if $\Gamma \vdash t : \tau$ is derivable, and ρ_1, ρ_2 are two related Γ-environments, then $[\![t]\!]\rho_1 \; \mathcal{R}_\tau \; [\![t]\!]\rho_2$. This is a simple induction on (the typing derivation of) t (see [Mit96] for details).

We are interested in the basic lemma because, as observed e.g. in [SP03], this implies that for all logical relations that coincide with equality on observation types, two terms with related values must be contextually equivalent. More precisely, assume that $\mathcal{R}_o$ is equality on $[\![o]\!]$ for every $o \in \mathbf{Obs}$. Take the simple notion of contextual equivalence in simply-typed lambda-calculi, i.e., for any pair of terms t_1, t_2 such that $\vdash t_1 : \tau$ and $\vdash t_2 : \tau$ are derivable, $t_1 \approx_\tau t_2$ if and only if, for any term $\mathbb{C}$ such that $x : \tau \vdash \mathbb{C} : o$ ($o \in \mathbf{Obs}$) is derivable, $[\![\mathbb{C}]\!][x \mapsto [\![t_1]\!]] = [\![\mathbb{C}]\!][x \mapsto [\![t_2]\!]]$. Then, if $[\![t_1]\!]\ \mathcal{R}_\tau\ [\![t_2]\!]$, we conclude that $t_1 \approx_\tau t_2$. Indeed, by the basic lemma, for every $\mathbb{C}$ such that $x : \tau \vdash \mathbb{C} : o$ is derivable ($o \in \mathbf{Obs}$), it holds that $[\![\mathbb{C}]\!][x \mapsto [\![t_1]\!]]\ \mathcal{R}_o\ [\![\mathbb{C}]\!][x \mapsto [\![t_2]\!]]$, i.e., $[\![\mathbb{C}]\!][x \mapsto [\![t_1]\!]] = [\![\mathbb{C}]\!][x \mapsto [\![t_2]\!]]$ since $\mathcal{R}_o$ is equality.

Categorical generalization

The standard definition of logical relations can be derived from a general construction over CCCs, using the notion of sconing [MS93]. Fix two categories $\mathcal{C}$ and $\boldsymbol{C}$ and a functor $|\_| : \mathcal{C} \to \boldsymbol{C}$. The *comma category* $(\boldsymbol{C} \downarrow |\_|)$ is the category whose objects are triples $\langle S, f, A\rangle$, with $f : S \to |A|$ in $\boldsymbol{C}$, and whose morphisms are pairs $\langle u, v\rangle : \langle S, f, A\rangle \to \langle S', f', A'\rangle$ with $u : S \to S' \in \boldsymbol{C}$ and $v : A \to A' \in \mathcal{C}$, such that the following square commutes in $\boldsymbol{C}$:

$$\begin{tikzcd} S \arrow[r, "f"] \arrow[d, "u"'] & {|A|} \arrow[d, "|v|"] \\ S' \arrow[r, "f'"] & {|A'|} \end{tikzcd} .$$

This category is also called the *scone of $\mathcal{C}$ over $\boldsymbol{C}$*. The second projection functor $U : (\boldsymbol{C} \downarrow |\_|) \to \mathcal{C}$ (also seen as a forgetful functor) maps $\langle S, f, A\rangle$ to A and a morphism $\langle u, v\rangle$ to v. The full subcategory of this scone consisting of all objects $\langle S, f, A\rangle$ with f a mono is called the *subscone of $\mathcal{C}$ over $\boldsymbol{C}$*, denoted by $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$.

A remarkable feature of sconing is that it preserves almost all additional categorical structures that $\mathcal{C}$ might have (see [FS90] for further discussion). In particular, if both $\mathcal{C}$ and $\boldsymbol{C}$ are CCCs and $\boldsymbol{C}$ has pull-backs and if the functor $|\_|$ preserves finite products, then $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$ is cartesian closed as well [Laf87, MR92]. Suppose $X = \langle S, f, A\rangle$ and $Y = \langle S', f', A'\rangle$ are two objects in $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$. The exponential Y^X is constructed as follows: $Y^X = \langle R, h, A'^A\rangle$, together with a morphism $g_0 : R \to S'^S$, such that the following diagram is a pull-back in $\boldsymbol{C}$:

$$\begin{tikzcd} R \arrow[r, hook, "h"] \arrow[d, "g_0"'] & {|A'^A|} \arrow[d, "g_2"] \\ S'^S \arrow[r, hook, "g_1"] & {|A'|^S} \end{tikzcd} ,$$

where g_1, g_2 are the unique morphisms making the following two diagrams commute:

$$\begin{tikzcd}
S'^S \times S \arrow[r, "\mathbf{eval}"] \arrow[d, "g_1 \times \mathbf{id}"'] & S' \arrow[d, hook, "f'"] \\
{|A'|^S \times S} \arrow[r, "\mathbf{eval}"] & {|A'|}
\end{tikzcd}, \qquad
\begin{tikzcd}
{|A'^A| \times S} \arrow[r, "\mathbf{id} \times f"] \arrow[d, "g_2 \times \mathbf{id}"'] & {|A'^A| \times |A|} \arrow[d, "|\mathbf{eval}|"] \\
{|A'|^S \times S} \arrow[r, "\mathbf{eval}"] & {|A'|}
\end{tikzcd},$$

and it can be verified that g_1 is a mono, hence h is a mono as well. The application morphism is defined by $\mathbf{eval}_{X,Y} = (\mathbf{eval}_{S,S'} \circ (g_0 \times \mathbf{id}_S), \mathbf{eval}_{A,A'})$. This is indeed a morphism from Y^X to Y in $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$ because the following diagram commutes:

$$\begin{tikzcd}
R \times S \arrow[rr, "h \times f"] \arrow[dr, "h \times \mathbf{id}"] \arrow[dd, "g_0 \times \mathbf{id}"'] & & {|A'^A| \times |A|} \arrow[d, equal] \\
& {|A'^A| \times S} \arrow[ur, "\mathbf{id} \times f"'] \arrow[d, "g_2 \times \mathbf{id}"] & {|A'^A \times A|} \arrow[dd, "|\mathbf{eval}|"] \\
S'^S \times S \arrow[r, "g_1 \times \mathbf{id}"] \arrow[d, "\mathbf{eval}"'] & {|A'|^S \times S} \arrow[dr, "\mathbf{eval}"] & \\
S' \arrow[rr, "f"] & & {|A'|}
\end{tikzcd}.$$

The uniqueness property of exponentiation also holds (see [MR92] for the detailed proof).

Now let Σ_b be the set of all base types in Σ, seen as a discrete category and let $\mathcal{C}$ be a Σ-CCC. As shown in Section 3.1.1, there is a unique representation $[\![\_]\!]_{\mathcal{C}}$ from the free Σ-CCC $\boldsymbol{\lambda}(\Sigma)$ to $\mathcal{C}$. Clearly, the following diagram commutes:

$$\begin{tikzcd}
\Sigma_b \arrow[r, "\subseteq"] \arrow[dr, "{[\![_]\!]_b}"'] & \boldsymbol{\lambda}(\Sigma) \arrow[d, "{[\![_]\!]_{\mathcal{C}}}"] \\
& \mathcal{C}
\end{tikzcd},$$

where $[\![\_]\!]_b$ is the functor representing the intended interpretation of base types in $\mathcal{C}$. Now assume $\boldsymbol{C}$ is another Σ-CCC, such that $\boldsymbol{C}$ has pull-backs. Let $|\_|$ be a functor from $\mathcal{C}$ to $\boldsymbol{C}$ that preserves terminal object, finite products and interpretations of Σ, i.e., $[\![b]\!]_{\boldsymbol{C}} = [\![b]\!]_{\mathcal{C}}$. Then $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$ is also a Σ-CCC, with $\langle [\![b]\!]_{\boldsymbol{C}}, \mathbf{id}, [\![b]\!]_{\mathcal{C}} \rangle$ as the denotation for base type b and $\langle [\![c]\!]_{\boldsymbol{C}}, [\![c]\!]_{\mathcal{C}} \rangle$ as the denotation for term constant c. Assume we are given a functor from Σ_b to $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$, i.e., a collection $\mathcal{R}_b$ of objects in $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$, one for each base type b. Then there is a unique representation $\mathcal{R}$ of Σ-CCCs from $\boldsymbol{\lambda}(\Sigma)$ to $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$ such that the following diagram commutes:

$$\begin{tikzcd} \Sigma_b \arrow[r, "\subseteq"] \arrow[d, "(\mathcal{R}_b)_{b\in\Sigma}"'] & \boldsymbol{\lambda}(\Sigma) \arrow[dl, "\mathcal{R}"] \\ \mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}} & \end{tikzcd} .$$

Now the crux of constructing logical relations in the model $\mathcal{C}$ is as follows. The forgetful functor $U : \mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}} \to \mathcal{C}$, which maps an object $\langle S, m, A\rangle$ to A and a morphism $\langle u, v\rangle$ to v, is also a representation of Σ-CCCs. It follows that $U \circ \mathcal{R}$ is a representation of Σ-CCCs again, from $\boldsymbol{\lambda}(\Sigma)$ to $\mathcal{C}$. If $U \circ (\mathcal{R}_b)_{b\in\Sigma} = [\![\_]\!]_b$, then by the uniqueness property of $[\![\_]\!]_{\mathcal{C}}$, we must have $U \circ \mathcal{R} = [\![\_]\!]_{\mathcal{C}}$, i.e., the following diagram commutes:

$$\begin{tikzcd} & \boldsymbol{\lambda}(\Sigma) \arrow[dl, "\mathcal{R}"'] \arrow[d, "[\![\_]\!]_{\mathcal{C}}"] \\ \mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}} \arrow[r, "U"'] & \mathcal{C} \end{tikzcd} . \tag{4.1}$$

Let $\boldsymbol{C} = \mathcal{S}et$, $\mathcal{C} = \mathcal{S}et \times \mathcal{S}et$ and let $|\_|$ be the functor $\mathbf{id} \times \mathbf{id}$, where $\mathbf{id}$ denotes the identity functor. Every binary relation $S \subseteq A_1 \times A_2$ has a representation $\langle \pi_1^S, \pi_2^S\rangle : S \hookrightarrow A_1 \times A_2$, where the arrow is the inclusion induced by two projections $\pi_1^S : S \to A_1$ and $\pi_2^S : S \to A_2$. $\mathcal{R}(\tau)$ is of the form $S \hookrightarrow [\![\tau]\!] \times [\![\tau]\!]$, where $[\![\_]\!]$ is the interpretation of lambda terms in set-theoretical models ($[\![\_]\!]_{\mathcal{C}} = \langle [\![\_]\!], [\![\_]\!]\rangle$), and S, up to isomorphism, is just a subset of $[\![\tau]\!] \times [\![\tau]\!]$. Then $(\mathcal{R}_\tau)_{(\tau \text{ type})}$ behaves like a logical relation: the object part of functor $\mathcal{R}$ yields logical relations (or extensions) in set-theoretical models. In particular, the fact that $\mathcal{R}$ preserve exponentials states the $(\mathbf{Log})$ condition:

$$(f_1, f_2) \in \mathcal{R}(\tau \to \tau') \iff \forall (a_1, a_2) \in \mathcal{R}(\tau) \cdot (f_1(a_1), f_2(a_2)) \in \mathcal{R}(\tau');$$

the morphism part of functor $\mathcal{R}$ maps each morphism $[y := t] : \Gamma \to \{y : \tau\}$ in $\boldsymbol{\lambda}(\Sigma)$, namely a typed term $\Gamma \vdash t : \tau$ modulo $\beta\eta$, to a morphism in the subscone, i.e., a pair $\langle u, v\rangle$, where $v = ([\![\Gamma \vdash t : \tau]\!], [\![\Gamma \vdash t : \tau]\!])$ according to the commuting diagram 4.1. The fact that $\langle u, v\rangle$ is a morphism, i.e., the following diagram commutes:

$$\begin{tikzcd} \mathcal{R}_\Gamma \arrow[r, hook] \arrow[d, "u"'] & {[\![\Gamma]\!] \times [\![\Gamma]\!]} \arrow[d, "v=([\![\Gamma\vdash t:\tau]\!],[\![\Gamma\vdash t:\tau]\!])"] \\ \mathcal{R}_\tau \arrow[r, hook] & {[\![\tau]\!] \times [\![\tau]\!]} \end{tikzcd} ,$$

states the basic lemma: for any pair of environments ρ_1 and ρ_2,

$$(\rho_1, \rho_2) \in \mathcal{R}(\Gamma) \Longrightarrow ([\![\Gamma \vdash t : \tau]\!]\rho_1, [\![\Gamma \vdash t : \tau]\!]\rho_2) \in \mathcal{R}(\tau).$$

Now consider the functor category $\mathcal{S}et^{\mathcal{I}}$. Let $\boldsymbol{C} = \mathcal{S}et^{\mathcal{I}}$ and $\mathcal{C} = \mathcal{S}et^{\mathcal{I}} \times \mathcal{S}et^{\mathcal{I}}$. Clearly, $\mathcal{S}et^{\mathcal{I}}$ has pull-backs, defined pointwise. Objects of the subscone over $\mathcal{S}et^{\mathcal{I}}$ give rise to $\mathcal{I}$-indexed *Kripke logical relations* [MM91]. Precisely, every relation in an $\mathcal{I}$-indexed Kripke logical relation is indexed not only by a type, but also by a set s in $\mathcal{I}$. This extra index is usually called a *world*, which is also seen as a representation of a computation stage. In the case of dynamic key generation, a world is just a set of keys that have been generated at that stage. If there is a morphism $i : s \to s' \in \mathcal{I}$, we say that s is a *smaller world* and s' is a *larger world*. Note that a smaller world does not mean a smaller set here, but a non-larger set, and similar for a larger world.

As an object in the subscone $\mathrm{Subscone}^{\mathcal{S}et}_{\mathcal{S}et \times \mathcal{S}et}$ is a representation of a binary relation, an object $\langle \pi_1^S, \pi_2^S \rangle : S \hookrightarrow A_1 \times A_2$ ($S, A_1, A_2 \in \mathcal{S}et^{\mathcal{I}}$) in the subscone $\mathrm{Subscone}^{\mathcal{S}et^{\mathcal{I}}}_{\mathcal{S}et^{\mathcal{I}} \times \mathcal{S}et^{\mathcal{I}}}$ is a representation of a series of binary relations, such that for any $s \in \mathcal{I}$, $\langle \pi_1^S, \pi_2^S \rangle s : Ss \hookrightarrow A_1 s \times A_2 s$ is an inclusion (up to isomorphism). In other words, S is a family of relations Ss between $A_1 s$ and $A_2 s$, functorial in s, and the functoriality requires that

$$\forall i : s \to s' \in \mathcal{I} \cdot (a_1, a_2) \in Ss \Longrightarrow (A_1 i(a_1), A_2 i(a_2)) \in Ss'.$$

This is the so-called *monotonicity* property of Kripke logical relations. Intuitively, it says that every related values at a smaller world must remain related when they are lifted to a larger world.

Exponentials in the subscone $\mathrm{Subscone}^{\mathcal{S}et^{\mathcal{I}}}_{\mathcal{S}et^{\mathcal{I}} \times \mathcal{S}et^{\mathcal{I}}}$ define relations for functions in Kripke logical relations, which are slightly different from those in standard logical relations. Consider the exponential $Y^X = S \hookrightarrow {B_1}^{A_1} \times {B_2}^{A_2}$, where $X = S_A \hookrightarrow A_1 \times A_2$ and $Y = S_B \hookrightarrow B_1 \times B_2$. For any $s \in \mathcal{I}$ and for any pair of functions $f_1 \in {B_1}^{A_1} s$, $f_2 \in {B_2}^{A_2} s$ (recall that in the category $\mathcal{S}et^{\mathcal{I}}$, f_i is a natural transformation such that for every $i : s \to s' \in \mathcal{I}$, $f_1 s'$ is a morphism $f_i s' : \mathcal{I}(s, s') \times A_i s' \to B_i s'$):

$$\begin{aligned}(f_1, f_2) \in Ss \Longleftrightarrow \\ &\forall i : s \to s' \in \mathcal{I} \cdot \forall a_1 \in A_1 s', a_2 \in A_2 s' \cdot \\ &\qquad (a_1, a_2) \in S_A s' \Rightarrow (f_1 s'(i, a_1), f_2 s'(i, a_2)) \in S_B s'.\end{aligned}$$

This is the so-called *comprehension* property of Kripke logical relations, which means that related functions at some world should map related arguments *at any larger world* to related results.

4.2 Relations logiques pour les types monadiques

Defining logical relations for Moggi's computational lambda-calculus follows the same pattern, but we need to consider the *free Σ-let-CCC* $\boldsymbol{Comp}(\Sigma)$ over Σ, instead of $\boldsymbol{\lambda}(\Sigma)$. Similarly, we get the following commuting diagram,

$$\begin{tikzcd}
\Sigma_b \arrow[r, "\subseteq"] \arrow[dr, "\llbracket _ \rrbracket_b"'] & \boldsymbol{Comp}(\Sigma) \arrow[d, "\llbracket _ \rrbracket_{\mathcal{C}}"] \\
& \mathcal{C}
\end{tikzcd} .$$

where $\mathcal{C}$ is a Σ-let-CCC, and $\llbracket \_ \rrbracket_{\mathcal{C}}$ is a representation of Σ-let-CCCs, i.e., a functor that preserves products, exponentials, interpretations of constants, and the strong monad (functor, unit, multiplication, strength).

We then need $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$ to be a Σ-let-CCC to establish the diagram

$$\begin{tikzcd}
\Sigma_b \arrow[r, "\subseteq"] \arrow[d, "(\mathcal{R}_b)_{b\in\Sigma}"'] & \boldsymbol{Comp}(\Sigma) \arrow[dl, "\mathcal{R}"] \\
\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}
\end{tikzcd} .$$

That is, we need to lift a strong monad $(T, \eta, \mu, \mathrm{t})$ on $\mathcal{C}$ to another monad $(\widetilde{T}, \widetilde{\eta}, \widetilde{\mu}, \widetilde{\mathrm{t}})$ on $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$ such that the following diagram

$$\begin{tikzcd}
\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}} \arrow[r, "\widetilde{T}"] \arrow[d, "U"'] & \mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}} \arrow[d, "U"] \\
\mathcal{C} \arrow[r, "T"] & \mathcal{C}
\end{tikzcd} \tag{4.2}$$

commutes, i.e., $T \circ U = U \circ \widetilde{T}$. Moreover, for any object $X \in \mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$, $U\widetilde{\eta}_X = \eta_{UX}$ and $U\widetilde{\mu}_X = \mu_{UX}$. This amounts to the requirement that the following two diagrams commute:

$$\begin{tikzcd}
& TUX \arrow[d, equal] \\
UX \arrow[ur, "\eta_{UX}"] \arrow[r, "U\widetilde{\eta}_X"] & U\widetilde{T}X
\end{tikzcd}
\qquad
\begin{tikzcd}
& T^2UX \arrow[dl, "\mu_{UX}"'] \arrow[d, equal] \\
TUX \arrow[d, equal] & TU\widetilde{T}X \arrow[d, equal] \\
U\widetilde{T}X & U\widetilde{T}^2X \arrow[l, "U\widetilde{\mu}_X"]
\end{tikzcd}$$

where by $=\!=\!=$ we mean the identity between two objects by (4.2).

According to [GLLN02], to lift a strong monad $(T, \eta, \mu, \mathrm{t})$ on $\mathcal{C}$ to $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$, we need:

- a category $\boldsymbol{C}$ with explicitly given finite products and natural isomorphisms $\boldsymbol{\gamma}$ and $\boldsymbol{\alpha}$:

  $$\boldsymbol{\gamma}_A : 1 \times A \to A \qquad \boldsymbol{\alpha}_{A,B,C} : (A \times B) \times C \to A \times (B \times C)$$

  for any $A, B, C \in \boldsymbol{C}$;

- a functor $|\_| : \mathcal{C} \to \boldsymbol{C}$, preserving finite products and natural isomorphisms (namely mapping γ and α in $\mathcal{C}$ to $\boldsymbol{\gamma}$ and $\boldsymbol{\alpha}$ respectively);

- a strong monad $(\boldsymbol{T}, \boldsymbol{\eta}, \boldsymbol{\mu}, \mathbf{t})$ on $\boldsymbol{C}$, related to $(T, \eta, \mu, \mathrm{t})$ by a natural transformation (called a *monad morphism*) [1] $\sigma : \boldsymbol{T}|\_| \to |T\_|$ making the following diagram commute:

$$
\begin{tikzcd}
|A| \times \boldsymbol{T}|B| \arrow[rr, "\mathbf{id}_{|A|} \times \sigma_B"] \arrow[d, "\mathbf{t}_{|A|,|B|}"'] & & {|A| \times |TB|} \arrow[r, equal] & {|A \times TB|} \arrow[d, "|\mathrm{t}_{A,B}|"] \\
\boldsymbol{T}(|A| \times |B|) \arrow[r, equal] & \boldsymbol{T}|A \times B| \arrow[rr, "\sigma_{A \times B}"] & & {|T(A \times B)|}
\end{tikzcd}
$$

- a *mono factorization* system on $\boldsymbol{C}$, which is essentially an epi-mono factorization [AHS90] without the requirement for epis. Formally, a mono factorization system is given by two distinguished subclasses of morphisms in $\boldsymbol{C}$, the so-called *pseudoepis* $\twoheadrightarrow$ and the so-called *relevant monos* $\multimap$. The latter must be monos, while the former are not necessarily epis. Both classes must contain all isomorphisms and be closed under composition with isomorphisms. Each morphism f in $\boldsymbol{C}$ must factor as $f = m \circ e$ for some pseudoepi e and some relevant mono m. For each commuting diagram as the one on the left, there is a unique diagonal making both triangles commute in the right diagram:

$$
\begin{tikzcd}
\bullet \arrow[r, two heads] \arrow[d] & \bullet \arrow[d] \\
\bullet \arrow[r, "" , hook] & \bullet
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\bullet \arrow[r, two heads] \arrow[d] & \bullet \arrow[d] \arrow[dl, dotted] \\
\bullet \arrow[r, hook] & \bullet
\end{tikzcd}
$$

- both $\boldsymbol{T}$ and finite products preserve pseudoepis.

All these requirements guarantee the correctness of the lifting of strong monad T of $\mathcal{C}$ to $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$. The detailed construction can be found in [GLLN02]. We only summarize the definition of the lifted monad $(\widetilde{T}, \widetilde{\eta}, \widetilde{\mu}, \widetilde{\mathrm{t}})$ on $\mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$:

- for any object $\langle S, f, A\rangle \in \mathrm{Subscone}_{\mathcal{C}}^{\boldsymbol{C}}$, $\widetilde{T}\langle S, f, A\rangle = \langle \widetilde{S}, m_{S,A}, TA\rangle$ such that

$$
\begin{tikzcd}
\boldsymbol{T}S \arrow[r, "\boldsymbol{T}f"] \arrow[d, two heads, "e_{S,A}"'] & \boldsymbol{T}|A| \arrow[d, "\sigma_A"] \\
\widetilde{S} \arrow[r, hook, "m_{S,A}"] & {|TA|}
\end{tikzcd}
$$

commutes, for some pseudoepi $e_{S,A}$;

[1] σ was named 'distributivity law' in [GLLN02]. We renamed it as one might confuse it with a distributive law by Beck [Bec69].

- for any morphism $\langle u, v\rangle : \langle S, f, A\rangle \to \langle S', f', A'\rangle \in \mathrm{Subscone}^{\boldsymbol{C}}_{\mathcal{C}}$, $\widetilde{T}\langle u, v\rangle = \langle \widetilde{u}, |Tv|\rangle$ where $\widetilde{u}$ is the unique morphism making the following diagram commute:

$$\begin{tikzcd}
\boldsymbol{T}S \arrow[r, two heads, "e_{S,A}"] \arrow[d, "\boldsymbol{T}u"'] & \widetilde{S} \arrow[d, "m_{S,A}"] \arrow[ddl, dotted, "\widetilde{u}"'] \\
\boldsymbol{T}S' \arrow[d, two heads, "e_{S',A'}"'] & |TA| \arrow[d, "|Tv|"] \\
\widetilde{S'} \arrow[r, "m_{S',A'}"] & |TA'|
\end{tikzcd}$$

- for every object $\langle S, f, A\rangle$, $\widetilde{\eta}_{\langle S,f,A\rangle} = \langle e_{S,A} \circ \boldsymbol{\eta}_S, |\eta_A|\rangle$ is a morphism from $\langle S, f, A\rangle$ to $\widetilde{T}\langle S, f, A\rangle$, i.e., $\langle \widetilde{S}, m_{S,A}, TA\rangle$;

- for every object $\langle S, f, A\rangle$, $\widetilde{\mu}_{\langle S,f,A\rangle} = \langle k, |\boldsymbol{T}_A|\rangle$ is a morphism from $\langle \widetilde{\widetilde{S}}, m_{\widetilde{S},TA}, \boldsymbol{T}^2A\rangle$ to $\langle \widetilde{S}, m_{S,A}, TA\rangle$, where $\langle \widetilde{\widetilde{S}}, m', \boldsymbol{T}^2A\rangle = \widetilde{T}\langle \widetilde{S}, m_{S,A}, A\rangle = \widetilde{T}^2\langle S, f, A\rangle$ and $k : \widetilde{\widetilde{S}} \to \widetilde{S}$ is the unique morphism making the following right diagram commute,

$$\begin{tikzcd}
\boldsymbol{T}^2S \arrow[r, two heads, "\boldsymbol{T}e_{S,A}"] \arrow[d, "\boldsymbol{\mu}_S"'] & \boldsymbol{T}\widetilde{S} \arrow[d, "\boldsymbol{T}m_{S,A}"] \arrow[dddl, dotted, "l"'] \\
\boldsymbol{T}S \arrow[dd, two heads, "e_{S,A}"'] & \boldsymbol{T}|TA| \arrow[d, "\sigma_{TA}"] \\
 & |\boldsymbol{T}^2A| \arrow[d, "|\mu_A|"] \\
\widetilde{S} \arrow[r, "m_{S,A}"] & |TA|
\end{tikzcd}
\qquad
\begin{tikzcd}
\boldsymbol{T}\widetilde{S} \arrow[r, two heads, "e_{\widetilde{S},TA}"] \arrow[dd, dashed, "l"'] & \widetilde{\widetilde{S}} \arrow[d, "m_{\widetilde{S},TA}"] \arrow[ddl, dotted, "k"'] \\
 & |\boldsymbol{T}^2A| \arrow[d, "|\mu_A|"] \\
\widetilde{S} \arrow[r, "m_{S,A}"] & |TA|
\end{tikzcd}$$

where l is the unique morphism making the above left diagram commute.

- for every pair of objects $\langle S, f, A\rangle$ and $\langle R, g, B\rangle$, $\widetilde{\mathrm{t}}_{\langle S,f,A\rangle,\langle R,g,B\rangle} = \langle h, \mathrm{t}_{A,B}\rangle$, is a morphism from $\langle S, f, A\rangle \times \widetilde{T}\langle R, g, B\rangle$, i.e., $\langle S \times \widetilde{R}, f \times m_{R,B}, A \times TB\rangle$, to $\langle \widetilde{S \times R}, m_{S\times R,A\times B}, T(A \times B)\rangle$ where $h : S \times \widetilde{R} \to \widetilde{S \times R}$ is the unique morphism making the following diagram commute:

$$
\begin{tikzcd}
S \times \boldsymbol{T}R \arrow[r, two heads, "\mathbf{id}_S \times e_{R,B}"] \arrow[d, "\mathbf{t}_{S,R}"'] & S \times \widetilde{R} \arrow[d, "f \times m_{R,B}"] \arrow[ddd, dotted, "h"'] \\
\boldsymbol{T}(S \times R) \arrow[dd, two heads, "e_{S\times R, A\times B}"'] & {|A| \times |TB|} \arrow[d, equal] \\
 & {|A \times TB|} \arrow[d, "|\mathrm{t}_{A,B}|"] \\
\widetilde{S \times R} \arrow[r, tail, "m_{S\times R, A\times B}"] & {|T(A \times B)|}
\end{tikzcd}
$$

Now assume that $\boldsymbol{C} = \mathcal{S}et$, $\mathcal{C} = \mathcal{S}et \times \mathcal{S}et$, $\boldsymbol{T}$ is a strong monad over $\mathcal{S}et$, and $|\_|$ is the functor $\mathbf{id} \times \mathbf{id}$. A strong monad T on $\mathcal{S}et \times \mathcal{S}et$ can be defined pairwise: $T(A_1, A_2) = (\boldsymbol{T}A_1, \boldsymbol{T}A_2)$. The monad morphism σ is then defined by the distributivity of the monad $\boldsymbol{T}$ on $\mathcal{S}et$ over binary products:

$$\sigma_{(A_1,A_2)} = (\boldsymbol{T}\pi_1, \boldsymbol{T}\pi_2) : \boldsymbol{T}(A_1 \times A_2) \to \boldsymbol{T}A_1 \times \boldsymbol{T}A_2$$

where π_1 and π_2 are the projections from $A_1 \times A_2$. Obviously $\mathcal{S}et$ has a mono factorization system with surjections as pseudoepis and injections as relevant monos. Then all the requirements are met for lifting the monad T on $\mathcal{S}et \times \mathcal{S}et$ to $\mathrm{Subscone}^{\mathcal{S}et}_{\mathcal{S}et\times\mathcal{S}et}$, with a strong monad $\widetilde{T}$. Every binary relation $S \subseteq A_1 \times A_2$ has a representation $(\pi_1^S, \pi_2^S) : S \hookrightarrow A_1 \times A_2$, and the lifted monad maps a relations S to another relation $\widetilde{S}$ between sets $\boldsymbol{T}A_1$ and $\boldsymbol{T}A_2$:

$$
\begin{tikzcd}
\boldsymbol{T}S \arrow[r, "\boldsymbol{T}\langle \pi_1^S, \pi_2^S \rangle"] \arrow[d, two heads] & \boldsymbol{T}(A_1 \times A_2) \arrow[d, "\sigma_{\langle A_1, A_2 \rangle}"] \\
\widetilde{S} \arrow[r, tail] & \boldsymbol{T}A_1 \times \boldsymbol{T}A_2
\end{tikzcd}
$$

where $\widetilde{S}$ is defined as the direct image of the function $\sigma_{\langle A_1,A_2\rangle} \circ \boldsymbol{T}\langle \pi_1^S, \pi_2^S \rangle$, which is proved to be equal to $\langle \boldsymbol{T}\pi_1^S, \boldsymbol{T}\pi_2^S \rangle$ [GLLN02].

Moreover, the following diagram commutes:

$$
\begin{tikzcd}
 & \boldsymbol{Comp}(\Sigma) \arrow[dl, "\mathcal{R}"'] \arrow[d, "\langle [\![\_]\!], [\![\_]\!] \rangle"] \\
\mathrm{Subscone}^{\mathcal{S}et}_{\mathcal{S}et\times\mathcal{S}et} \arrow[r, "U"] & \mathcal{S}et \times \mathcal{S}et
\end{tikzcd}
$$

where $\mathcal{R}$, U and $[\![\_]\!]$ are representations of Σ-let-CCCs, and $\mathcal{R}(\tau)$, up to isomorphism, is a subset of $[\![\tau]\!] \times [\![\tau]\!]$. The fact that $\mathcal{R}$ preserves (strong) monads gives rise to the logical relation for monadic types:

$$(a_1, a_2) \in \mathcal{R}(\mathsf{T}\tau) \iff (a_2, a_2) \in \widetilde{T}\mathcal{R}(\tau)$$

A list of logical relations defined over some concrete monads is given in [GLLN02]. For example, the relation $\widetilde{S}$ for the exception monad ($\boldsymbol{T}A = A + E$) is

$$\widetilde{S} = S \cup \{(e, e) \mid e \in E\},$$

where E is the set of exceptions, and $\widetilde{S}$ for the non-determinism monad ($\boldsymbol{T}A = \mathbb{P}_{fin}(A)$) is

$$\begin{aligned}(s_1, s_2) \in \widetilde{S} \iff & (\forall a_1 \in s_1.\exists a_2 \in s_2.(a_1, a_2) \in S) \;\; \& \\ & (\forall a_2 \in s_2.\exists a_1 \in s_1.(a_1, a_2) \in S)\end{aligned}$$

A logical relation for key generation monad

A logical relation for the name creation monad is also defined in [GLLN02]. To derive this logical relation, we shall consider the functor category $\mathcal{S}et^{\mathcal{I}}$. Precisely, let $\mathcal{C} = \mathcal{S}et^{\mathcal{I}} \times \mathcal{S}et^{\mathcal{I}}$, $\boldsymbol{C} = \mathcal{S}et^{\mathcal{I}}$ and $|\_|$ be the functor $\mathbf{id} \times \mathbf{id}$. $\mathcal{S}et^{\mathcal{I}}$ has a mono factorization consisting of pointwise surjections and pointwise injections. Take the strong monad T on $\mathcal{S}et^{\mathcal{I}}$ as defined in Section 3.2. For any $A_1, A_2 \in \mathcal{S}et^{\mathcal{I}}$, the functor morphism $\sigma_{\langle A_1, A_2\rangle} = \langle T\pi_1, T\pi_2\rangle$ is then a natural transformation from $T(A_1 \times A_2)$ to $TA_1 \times TA_2$ such that for any $s \in \mathcal{I}, a_1 \in A_1(s + s'), a_2 \in A_2(s + s')$:

$$\sigma_{\langle A_1, A_2\rangle} s[s', (a_1, a_2)] = ([s', a_1], [s', a_2]),$$

so we can lift the monad T to $\mathrm{Subscone}^{\mathcal{S}et^{\mathcal{I}}}_{\mathcal{S}et^{\mathcal{I}} \times \mathcal{S}et^{\mathcal{I}}}$.

Now consider $(\pi_1^S, \pi_2^S) : S \hookrightarrow A_1 \times A_2$ in the subscone $\mathrm{Subscone}^{\mathcal{S}et^{\mathcal{I}}}_{\mathcal{S}et^{\mathcal{I}} \times \mathcal{S}et^{\mathcal{I}}}$ as a representation of a series of binary relations (for every $s \in \mathcal{I}$, $Ss \subseteq A_1 s \times A_2 s$). For every $s \in \mathcal{I}$, $\widetilde{S}s$ is defined as the direct image of $\langle T\pi_1^S, T\pi_2^S\rangle s = \langle T\pi_1^S s, T\pi_2^S s\rangle$. When we consider the equivalence relation $\simeq$ between monadic values, this means that for any $[s_1, a_1] \in TA_1 s$ and $[s_2, a_2] \in TA_2 s$,

$$\begin{aligned}&[s_1, a_1]\; \widetilde{S}s\; [s_2, a_2] \iff \\ &\qquad \exists s' \in \mathcal{I}, a_1' \in A_1(s + s'), a_2' \in A_2(s + s') \text{ s.t.} \\ &\qquad\qquad (s_1, a_1) \simeq (s', a_1') \;\;\&\;\; (s_2, a_2) \simeq (s', a_2') \;\;\&\;\; a_1'\; S(s + s')\; a_2',\end{aligned}$$

which is proved to be equivalent to the following definition [GLLN02]:

$$\begin{aligned}&[s_1, a_1]\; \widetilde{S}s\; [s_2, a_2] \iff \\ &\qquad \exists s_0, i_1 : s_1 \to s_0, i_2 : s_2 \to s_0 \in \mathcal{I} \text{ s.t.} \\ &\qquad\qquad (A_1(\mathbf{id}_s + i_1)a_1)\; S(s + s_0)\; (A_2(\mathbf{id}_s + i_2)a_2).\end{aligned} \tag{4.3}$$

4.3 Le catégorie $\mathcal{I}^{\rightarrow}$

The category $\mathcal{S}et^{\mathcal{I}}$ is a perfectly adequate model for dynamic key generation, and we are able to derive logical relations through the general construction over this category. However, logical relations derived over this category are too weak in the sense that it is not sufficient for us to study relations between programs in a language involving dynamic key (or name) generation. In particular, logical relations for the cryptographic metalanguage depend on what we choose as relations between keys. This accordingly requires a proper definition of $\mathcal{R}^s_{\mathsf{key}}$. While $[\![\mathsf{key}]\!]s$ varies as s changes, $\mathcal{R}^s_{\mathsf{key}}$ should represent the variation of the relation between keys when s varies. But in the category $\mathcal{S}et^{\mathcal{I}}$, only some very naive relations for keys can be naturally defined, for instance, an empty relation or a full relation for every s in $\mathcal{I}$ (relations for keys are defined over s since $[\![\mathsf{key}]\!]s = s$).

A non-trivial relation on s is the identity relation, but it is noticed in [ZN03] that logical relations defined inductively from the identity relation on keys are too weak to recognize the contextual equivalence between certain obviously equivalent programs. Here is an example[2], where we have two programs

$$\begin{aligned} p_1 &= \texttt{let } k \Leftarrow \texttt{new in val}(\lambda x.\texttt{case dec}(x,k) \texttt{ of some}(\_) \texttt{ in true else false}), \\ p_2 &= \texttt{val}(\lambda x.\texttt{false}). \end{aligned}$$

In the first program, applying the function to any possible arguments, we always get the value `false`, since k is a fresh key and no context can build a encrypted message with k, hence the two programs are contextually equivalent, but they are not related by any logical relations that coincide with equality at the type key. Write the denotations of the two programs as $[\{k\}, f_1]$ and $[\emptyset, f_2]$. By (4.3), in order to relate these two computations, we should relate the two functions at $s + \{k\}$, but then we are allowed to apply these functions to messages built from k and we get non-related results, namely `true` and `false`.

Recall that our purpose is to use logical relations to prove contextual equivalence. Clearly, not every pair of keys, but only those disclosed keys can be compared by contexts. We should consider only the equality between these keys. One may argue that those keys that are not disclosed to contexts are also contextually equivalent since no context can tell the difference between them. But a subtle point about contextual equivalence is that contextually equivalent programs or values must be indeed accessible by contexts, while those non-disclosed keys are not, i.e., contexts are not able to get access to these keys, hence not able to test the equality between them.

[2]Note that the discussion in [ZN03] is based on the nu-calculus, a language without any constant for encryption, but a similar counterexample is given with a native constant of testing equality between names (seen equivalently as keys here) in that language.

In fact, there are already several known methods for defining relations between keys. A very popular way is to take as $\mathcal{R}^{s}_{\mathsf{key}}$ the identity on a certain set of "disclosed" keys as in the framed-bisimulation for Spi-calculus [AG99, AG98], where such a set is called a "frame". A more standard way is to consider two different sets of keys and to use a bijection between these two sets to represent the relation for keys, without forcing related keys to be equal. This is what Pitts and Stark do for defining an operational logical relation for the nu-calculus [PS93a].

However, in semantics, it is more natural to consider only one set of keys with a subset of keys seen as "disclosed keys". We can use the first method here to define $\mathcal{R}_{\mathsf{key}}$, letting our logical relation parameterized by a parameter *fr* (we follow the convention in the framed-bisimulation and call the parameter *fr*, denoting "frame"):

$$k_1 \; \mathcal{R}^{fr,s}_{\mathsf{key}} \; k_2 \Longleftrightarrow k_1 = k_2 \in fr$$

where $fr \subseteq s$, denoting those disclosed keys at the stage s. Obviously, the parameter *fr* varies as the set s changes, and one point that should be noticed in this variation is that, disclosed keys should always remain disclosed. In other words, when we pass from a smaller world s to a larger world s', keys in *fr* must be still in *fr*. Precisely, for any $i : s \to s' \in \mathcal{I}$ and for any $k \in fr_s$, $i(k) \in fr_{s'}$, where we write fr_s for the parameter *fr* at the world s.

The parameter *fr* can be formalized by using the comma category $\mathcal{I}^{\rightarrow}$:

Definition 4.1. *Category $\mathcal{I}^{\rightarrow}$ is the comma category $(\mathcal{I} \downarrow |\_|)$ where $|\_|$ is the identity functor from $\mathcal{I}$ to $\mathcal{I}$. Precisely, objects of $\mathcal{I}^{\rightarrow}$ are tuples $\langle w, i, s\rangle$ with $i : w \to s$ in $\mathcal{I}$, and whose morphisms are pairs $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ where $j : w \to w'$ in $\mathcal{I}$ and $l : s \to s'$ in $\mathcal{I}$ such that the following diagram commutes:*

$$\begin{tikzcd} w \arrow[r, "i"] \arrow[d, "j"'] & s \arrow[d, "l"] \\ w' \arrow[r, "i'"] & s'. \end{tikzcd} \tag{4.4}$$

The composition of two morphisms $(j, l) : i \to i'$ and $(j', l') : i' \to i''$ is $(j' \circ j, l \circ l)$. We write i for $\langle w, i, s\rangle$ when the domain w and the codomain s of i are clear from the context. Intuitively, every object $\langle w, i, s\rangle$ of $\mathcal{I}^{\rightarrow}$ represents a selection of disclosed keys from a set s of keys. Then naturally, w takes the place of the parameter *fr*.

In the next section, we shall build another functor category $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ and show how to derive logical relations over it. The following proposition is useful for refining logical relations derived over $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$.

Proposition 4.1. *Suppose that $l_1 : s \to s_1$ and $l_2 : s \to s_2$ are two morphisms in $\mathcal{I}$. There exists a set s' and two morphisms $l'_1 : s_1 \to s'$ and $l'_2 : s_2 \to s'$ in $\mathcal{I}$ such that $l'_1 \circ l_1 = l'_2 \circ l_2$.*

The proof of this proposition will become easier with the following property of $\mathcal{I}^{\rightarrow}$:

Proposition 4.2 (Cube property for $\mathcal{I}^{\rightarrow}$). *Suppose that $(j_1, l_1) : \langle w, i, s\rangle \to \langle w_1, i_1, s_1\rangle$ and $(j_2, l_2) : \langle w, i, s\rangle \to \langle w_2, i_2, s_2\rangle$ are two morphisms in $\mathcal{I}^{\rightarrow}$. There exists $\langle w', i', s'\rangle \in \mathcal{I}^{\rightarrow}$ and two morphisms $(j'_1, l'_1) : i_1 \to i'$ and $(j'_2, l'_2) : i_2 \to i'$ such that the following square commutes in $\mathcal{I}^{\rightarrow}$:*

$$\begin{tikzcd} & i \arrow[dl, "(j_1,l_1)"'] \arrow[dr, "(j_2,l_2)"] & \\ i_1 \arrow[dr, "(j'_1,l'_1)"'] & & i_2 \arrow[dl, "(j'_2,l'_2)"] \\ & i' & \end{tikzcd} \tag{4.5}$$

Proof. The fact that the diagram (4.5) commutes in $\mathcal{I}^{\rightarrow}$ is equivalent to the fact that the following cube commutes in $\mathcal{I}$:

$$\begin{tikzcd} & & w \arrow[dll, "j_1"'] \arrow[d, "i"] \arrow[drr, "j_2"] & & \\ w_1 \arrow[d, "i_1"'] \arrow[drr, "j'_1"] & & s \arrow[dll, "l_1"'] \arrow[drr, "l_2"] & & w_2 \arrow[d, "i_2"] \arrow[dll, "j'_2"'] \\ s_1 \arrow[drr, "l'_1"'] & & w' \arrow[d, "i'"] & & s_2 \arrow[dll, "l'_2"] \\ & & s' & & \end{tikzcd} \tag{4.6}$$

Let $w' = w_1 \oplus w_2 / \sim_w$ and $s' = s_1 \oplus s_2 / \sim_s$, where $\oplus$ denotes the disjoint union (without losing generality, we assume that $w_1 \cap w_2 = \emptyset$ and $s_1 \cap s_2 = \emptyset$) and the quotient relations $\sim_w, \sim_s$ are defined by

$$\begin{aligned} &\forall n_1 \in w_1, n_2 \in w_2, \quad n_1 \sim_w n_2 \Leftrightarrow \exists n \in s \text{ s.t. } k_1(n) = i_1(n_1) \ \ \& \ \ k_2(n) = i_2(n_2) \\ &\forall n_1 \in s_1, n_2 \in s_2, \quad n_1 \sim_s n_2 \Leftrightarrow \exists n \in s \text{ s.t. } k_1(n) = n_1 \ \ \& \ \ k_2(n) = n_2. \end{aligned}$$

We write $\widetilde{n}^w$ for the equivalence of $\sim_w$, and $\widetilde{n}^s$ for that of $\sim_s$ (we may omit w and s and simply write $\widetilde{n}$ when the index is clear from the context). If n is not $\sim$-related to other elements, we have $\widetilde{n} = n$.

Define $j'_l : w_l \to w'$ by $j'_l(n) = \widetilde{n}^w$ $(l = 1, 2)$, $k'_l : s_l \to s'$ by $k'_l(n) = \widetilde{n}^s$ $(l = 1, 2)$ and $i' : w' \to s'$ by

$$i'(\widetilde{n}^w) = \begin{cases} \widetilde{i_1(n)}^s & \text{if } n \in w_1, \\ \widetilde{i_2(n)}^s & \text{if } n \in w_2. \end{cases}$$

Obviously, j_1, j_2, k_1, k_2 are (well-defined) injections. We need to show that i' is also an injection. For any $\widetilde{n_1}^w, \widetilde{n_2}^w \in w'$ and $\widetilde{n_1}^w \neq \widetilde{n_2}^w$, hence $\widetilde{n_1}^w \not\sim_w \widetilde{n_2}^w$, either

- $n_1, n_2 \in w_1$, then $i_1(n_1), i_1(n_2) \in s_1$ and $i_1(n_1) \neq i_1(n_2)$ because of the injectivity of i_1. And by the definition of $\sim_s$, $i_1(n_1) \not\sim_s i_1(n_2)$, i.e., $\widetilde{i_1(n_1)}^s \neq \widetilde{i_1(n_2)}^s$, so $i'(\widetilde{n_1}^w) \neq i'(\widetilde{n_2}^w)$;

- or $n_1, n_2 \in w_2$. Similarly, we can get $i'(\widetilde{n_1}^w) \neq i'(\widetilde{n_2}^w)$;

- or n_1 and n_2 are from different sets. Without losing generality, we assume $n_1 \in w_1$ and $n_2 \in w_2$. Because $n_1 \not\sim_w n_2$, i.e., there is no $n \in s$ such that $k_1(n) = i_1(n_1)$ and $k_1(n) = i_2(n_2)$, by the definition of $\sim_s$, $i_1(n_1) \not\sim_s i_2(n_2)$, hence $\widetilde{i_1(n_1)}^s \neq \widetilde{i_2(n_2)}^s$, which immediately shows $i'(\widetilde{n_1}^w) \neq i'(\widetilde{n_2}^w)$.

Now, the commutativity of the cube (4.6) comes down to that of the following four diagrams:

$$
\begin{tikzcd}[column sep=small]
 & w \arrow[dl, "j_1"'] \arrow[dr, "j_2"] & \\
w_1 \arrow[dr, "j'_1"'] & a & w_2 \arrow[dl, "j'_2"] \\
 & w' &
\end{tikzcd}
\quad
\begin{tikzcd}[column sep=small]
 & s \arrow[dl, "k_1"'] \arrow[dr, "k_2"] & \\
s_1 \arrow[dr, "k'_1"'] & b & s_2 \arrow[dl, "k'_2"] \\
 & s' &
\end{tikzcd}
\quad
\begin{tikzcd}
w_1 \arrow[r, "i_1"] \arrow[d, "j'_1"'] \arrow[dr, phantom, "c"] & s_1 \arrow[d, "k'_1"] \\
w' \arrow[r, "i'"'] & s'
\end{tikzcd}
\quad
\begin{tikzcd}
w_2 \arrow[r, "i_2"] \arrow[d, "j'_2"'] \arrow[dr, phantom, "d"] & s_2 \arrow[d, "k'_2"] \\
w' \arrow[r, "i'"'] & s'
\end{tikzcd}.
$$

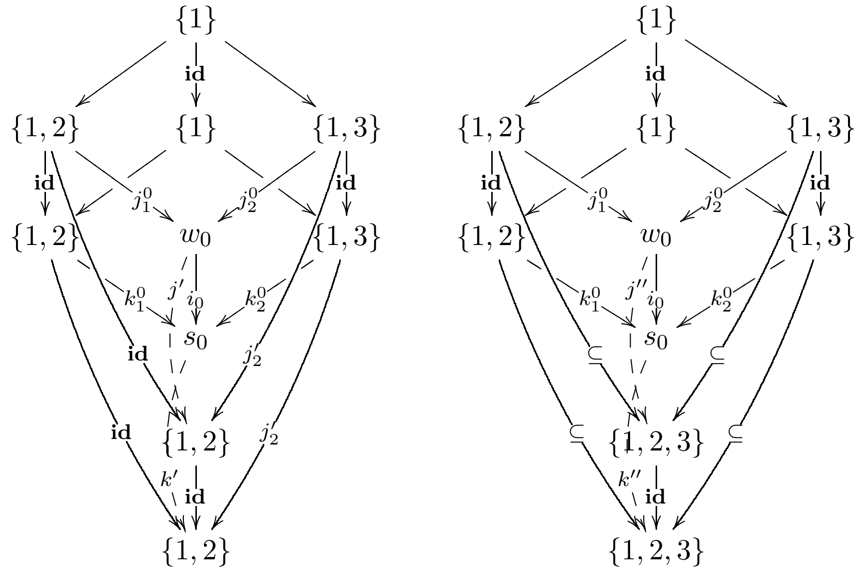
Commutativity of square a: For any $n \in w$, because squares $\begin{tikzcd} w \arrow[r, "i"] \arrow[d, "j_1"'] & s \arrow[d, "k_1"] \\ w_1 \arrow[r, "i_1"] & s_1 \end{tikzcd}$ and $\begin{tikzcd} w \arrow[r, "i"] \arrow[d, "j_2"'] & s \arrow[d, "k_2"] \\ w_2 \arrow[r, "i_2"] & s_2 \end{tikzcd}$ commute, we have $i_1(j_1(n)) = (k_1(i(n))$ and $i_2(j_2(n)) = k_2(i(n))$. By the definition of $\sim_w$, $j_1(n) \sim_w j_2(n)$, then $j'_1(j_1(n)) = \widetilde{j_1(n)}^w = \widetilde{j_2(n)}^w = j'_2(j_2(n))$.
Commutativity of square b: For any $n \in s$, $k_1(n) \sim_s k_2(n)$, so $k'_1(k_1(n)) = \widetilde{k_1(n)}^s = \widetilde{k_2(n)}^s = k'_2(k_2(n))$.
Commutativity of square c, d: For any $n \in w_l$ $(k = 1, 2)$, $i'(j'_l(n)) = i'(\widetilde{n}^w) = \widetilde{i_l(n)}^s = k'_l(i_l(n)),\ (k = 1, 2)$. □

Remark 4.1. Note that the construction in the above proof is not a push-out. Actually, the category $\mathcal{I}^{\to}$ has no push-outs. Consider the following diagram in $\mathcal{I}$:

$$
\begin{tikzcd}
 & \{1\} \arrow[dl, "\subseteq"'] \arrow[d, "\mathbf{id}"] \arrow[dr, "\subseteq"] & \\
\{1,2\} \arrow[d, "\mathbf{id}"'] & \{1\} \arrow[dl, "\subseteq"'] \arrow[dr, "\subseteq"] & \{1,3\} \arrow[d, "\mathbf{id}"] \\
\{1,2\} & & \{1,3\}
\end{tikzcd}
$$

where $\subseteq$ denotes the particular injection of inclusion. Assume this diagram has a push-out, composed by $w_0 \xrightarrow{i_0} s_0$ and two morphisms (j_1^0, k_1^0) (from $\{1,2\} \xrightarrow{\mathbf{id}} \{1,2\}$ to $w_0 \xrightarrow{i_0} s_0$) and (j_2^0, k_2^0)

(from $\{1,3\}\xrightarrow{\mathbf{id}}\{1.3\}$ to $w_0\xrightarrow{i_0}s_0$), then the following two diagrams should commute in $\mathcal{I}$:



where $j_2'(1) = 1, j_2'(3) = 2$. By the left diagram, $j'(j_1^0(2)) = 2$, $j'(j_2^0(3)) = j_2'(3) = 2$, then $j_1^0(2) = j_2^0(3)$ (since j' is an injection). While by the right diagram, $j''(j_1^0(2)) = 2$, $j''(j_2^0(3)) = 3$, and we get $j_1^0(2) \neq j_2^0(3)$ (j'' is an injection as well), which is a contradiction.

Proof of Proposition 4.1. Take objects $\langle\emptyset, i, s\rangle$, $\langle\emptyset, i_1, s_1\rangle$ and $\langle\emptyset, i_2, s_2\rangle$ in $\mathcal{I}^\to$, where i, i_1 and i_2 are all empty injections. Clearly, $(j_1, l_1) : i \to i_1$ and $(j_2, l_2) : i \to i_2$, where j_1 and j_2 are empty injections as well, are two morphisms in $\mathcal{I}^\to$. According to Proposition 4.2, there exist $\langle w', i', s'\rangle$ and $(j_1', l_1') : i_1 \to i'$, $(j_2', l_2') : i_2 \to i'$, such that $(j_1', l_1') \circ (j_1, l_1) = (j_2', l_2') \circ (j_2, l_2)$, which includes the equation $l_1' \circ l_1 = l_2' \circ l_2$. $\square$

4.4 Dérivation des relations logiques sur $\mathcal{S}et^{\mathcal{I}^\to}$

Let $\mathcal{S}et^{\mathcal{I}^\to}$ be the category of functors from $\mathcal{I}^\to$ to $\mathcal{S}et$ and natural transformations. Note that this is not a new model for the cryptographic metalanguage. We define the new category $\mathcal{S}et^{\mathcal{I}^\to}$ just for deriving a new logical relation and we still consider the interpretation of the metalanguage in the model $\mathcal{S}et^{\mathcal{I}}$.

For deriving logical relations over category $\mathcal{S}et^{\mathcal{I}^\to}$, we must first check the necessary requirements on categories.

$\mathcal{S}et^{\mathcal{I}^\to}$ is cartesian closed. Products and coproducts are defined pointwise. For any two functors $A, B : \mathcal{I}^\to \to \mathcal{S}et$, their exponent is defined by

$$B^A i = \mathcal{S}et^{\mathcal{I}^\to}(\mathcal{I}^\to(i, -) \times A, B)$$
$$(B^A(j,l)f)i''((j',l'),a) = fi''((j'\circ j, l'\circ l), a)$$

for any $\langle w,i,s\rangle \in \mathcal{I}^\rightarrow$, $(j,l) : i \to i'$, $(j',l') : i' \to i'' \in \mathcal{I}^\rightarrow$, $f \in B^A i$ and $a \in Ai$. $\mathcal{S}et^{\mathcal{I}^\rightarrow}$ has also pull-backs, defined pointwise. Now define a strong monad $(\boldsymbol{T}, \boldsymbol{\eta}, \boldsymbol{\mu}, \mathbf{t})$ on $\mathcal{S}et^{\mathcal{I}^\rightarrow}$ by:

- $\boldsymbol{T}A = \mathbf{colim}_{i'}\, A(\_ + i') : \mathcal{I}^\rightarrow \to \mathcal{S}et$. On objects, $\boldsymbol{T}Ai = \mathbf{colim}_{i'}\, A(i+i')$ is the set of equivalence classes of pairs (i',a), where $i' : w' \to s'$ in $\mathcal{I}$ and $a \in A(i+i')$, modulo the smallest equivalence relation $\sim$ such that $(i',a) \sim (i'', A(\mathbf{id}_i + (j,k))a)$ for each morphism $(j,k) : \langle w',i',s'\rangle \to \langle w'',i'',s''\rangle$ in $\mathcal{I}^\rightarrow$. We write $[i,a]$ for the equivalence class of (i,a). On morphisms, $\boldsymbol{T}A(j,k)$ maps the equivalence class of (i',a) to the equivalence class of $(i', A((j,k)+\mathbf{id}_{i'})a)$;
- for any $f : A \to B$ in $\mathcal{S}et^{\mathcal{I}^\rightarrow}$, $\boldsymbol{T}fi : \boldsymbol{T}Ai \to \boldsymbol{T}Bi$ is defined by $\boldsymbol{T}fi[i',a] = [i', f(i+i')a]$;
- $\boldsymbol{\eta}Ai : Ai \to \boldsymbol{T}Ai$ is defined by $\boldsymbol{\eta}Aia = [\emptyset, a]$, where $\emptyset$ denotes the empty function between empty sets;
- $\boldsymbol{\mu}Ai : \boldsymbol{T}^2Ai \to \boldsymbol{T}Ai$ is defined by $\boldsymbol{\mu}Ai[i',[i'',a]] = [i'+i'', a]$;
- $\mathbf{t}A, Bi : Ai \times \boldsymbol{T}Bi \to \boldsymbol{T}(A\times B)i$ is defined by $\mathbf{t}A,Bi(a,[i',b]) = [i', (Ai^{\rightarrow}_{i,i'}a, b)]$ where $i^{\rightarrow}_{i,i'} : i \to i+i'$ is the canonical injection.

Let $U : \mathcal{I}^\rightarrow \to \mathcal{I}$ be the forgetful functor which maps an object $\langle w,i,s\rangle$ to s and a morphism (j,l) to l. Let $|\_| : \mathcal{C} \to \boldsymbol{C}$ be the functor $\mathbf{id}_{\mathcal{S}et}{}^U$ (where $\mathcal{C} = \mathcal{S}et^{\mathcal{I}}$ and $\boldsymbol{C} = \mathcal{S}et^{\mathcal{I}^\rightarrow}$). On an object A, $|A|$ is equal to $A \circ U$, that is, for any object $\langle w,i,s\rangle \in \mathcal{I}^\rightarrow$, $|A|\langle w,i,s\rangle = As$, and for any morphism $(j,l) \in \mathcal{I}^\rightarrow$, $|A|(j,l) = Al$. On a morphism f (a natural transformation), for each $\langle w,i,s\rangle$ in $\mathcal{I}^\rightarrow$, the component $|f|_{\langle w,i,s\rangle}$ is equal to f_s. It is clear that the functor $|\_|$ preserves finite products.

Recall the strong monad T over the category $\mathcal{S}et^{\mathcal{I}}$, defined in Section 3.2. Let $\sigma : \boldsymbol{T}|\_| \to |T\_|$ be the monad morphism defined by $\sigma_A i[\langle w',i',s'\rangle, a] = [s',a]$, for any object A in $\mathcal{S}et^{\mathcal{I}}$ and $\langle w,i,s\rangle \in \mathcal{I}^\rightarrow$. This is well defined as $|A|(i+i') = A(s+s')$. We can then define $\sigma_{(A_1,A_2)} : \boldsymbol{T}|A_1 \times A_2| \to |TA_1| \times |TA_2|$ by

$$\sigma_{(A_1,A_2)} = (\sigma_{A_1} \circ \boldsymbol{T}|\pi_1|, \sigma_{A_2} \circ \boldsymbol{T}|\pi_2|),$$

i.e.,

$$\sigma_{\langle A_1,A_2\rangle} i[\langle w',i',s'\rangle, (a_1,a_2)] = ([s',a_1],[s',a_2]).$$

$\mathcal{S}et^{\mathcal{I}^\rightarrow}$ has a mono factorization system consisting of pointwise surjections and pointwise injections. And it is clear that $\boldsymbol{T}$ and finite products preserve pointwise surjections. Then all the requirements for building a logical relation on $\mathrm{Subscone}^{\mathcal{S}et^{\mathcal{I}^\rightarrow}}_{\mathcal{S}et^{\mathcal{I}} \times \mathcal{S}et^{\mathcal{I}}}$ are satisfied.

Now consider $f : S \hookrightarrow |A_1 \times A_2|$ ($A_1, A_2 \in \mathcal{S}et^{\mathcal{I}}$ and $S \in \mathcal{S}et^{\mathcal{I}^\rightarrow}$) as a representation of a series of binary relations such that for every $\langle w,i,s\rangle \in \mathcal{I}^\rightarrow$, $fi : Si \hookrightarrow |A_1 \times A_2|i =$

$(A_1 \times A_2)s = A_1 s \times A_2 s$ is an inclusion, representing a binary relation. In particular, the relation between monadic values is given by mono factorization of the composition of $\boldsymbol{T}f$ with $\sigma_{(A_1,A_2)}$:

$$
\begin{tikzcd}
\boldsymbol{T}S \arrow[r, "\boldsymbol{T}f"] \arrow[d, two heads] & \boldsymbol{T}|A_1 \times A_2| \arrow[d, "\sigma_{(A_1,A_2)}"] \\
\widetilde{S} \arrow[r, hook] & |TA_1| \times |TA_2|
\end{tikzcd}
$$

We take $\widetilde{S}$ as the direct image of $\sigma_{(A_1,A_2)} \circ \boldsymbol{T}f$, then for any $\langle w, i, s\rangle \in \mathcal{I}^{\to}$ and any pair of computations $[s_1, a_1] \in |TA_1|i$, $[s_2, a_2] \in |TA_2|i$,

$$
\begin{aligned}
&[s_1, a_1]\ \widetilde{S}i\ [s_2, a_2] \Longleftrightarrow \\
&\quad \exists \langle w', i', s'\rangle \in \mathcal{I}^{\to}, a'_1 \in A_1(s + s'), a'_2 \in A_2(s + s') \text{ s.t.} \\
&\qquad (s_1, a_1) \simeq (s', a'_1)\ \ \&\ \ (s_2, a_2) \simeq (s', a'_2)\ \ \&\ \ a'_1\ S(i + i')\ a'_2
\end{aligned}
\tag{4.7}
$$

According to the definition of $\simeq$, $(s_1, a_1) \simeq (s', a'_1)$ means that there is a finite set s'_1 and two morphisms $l_1 : s_1 \to s'_1$, $l'_1 : s' \to s'_1$ in $\mathcal{I}$ such that $A_1(\mathbf{id}_s + l_1)a_1 = A_1(\mathbf{id}_s + l'_1)a'_1$. Similarly, $(s_2, a_2) \simeq (s', a'_2)$ means that there is a finite set s'_2 and two morphisms $l_2 : s_2 \to s'_2$, $l'_2 : s' \to s'_2$ in $\mathcal{I}$ such that $A_2(\mathbf{id}_s + l_2)a_2 = A_2(\mathbf{id}_s + l'_2)a'_2$. By Proposition 4.1, there exist s_0 with two morphisms $l''_1 : s'_1 \to s_0$, $l''_2 : s'_2 \to s_0$ in $\mathcal{I}$, such that $l''_1 \circ l'_1 = l''_2 \circ l'_2$. hence

$$\mathbf{id}_s + (l''_1 \circ l'_1) = \mathbf{id}_s + (l''_2 \circ l'_2).$$

Take an arbitrary object $\langle w_0, i_0, s_0\rangle$ and a morphism $(j, l) : i' \to i_0$ in $\mathcal{I}^{\to}$ where $l = l''_1 \circ l'_1 = l''_2 \circ l'_2$ (such objects and morphisms necessarily exist). Because S is a functor from $\mathcal{I}^{\to}$ to $\mathcal{S}et$,

$$
\begin{aligned}
& S(\mathbf{id}_i + (j, l))(a'_1, a'_2) \\
= \ & |A_1 \times A_2|(\mathbf{id}_i + (j, l))(a'_1, a'_2) \\
= \ & (A_1(\mathbf{id}_s + l)a'_1, A_2(\mathbf{id}_s + l)a'_2) \\
= \ & (A_1(\mathbf{id}_s + (l''_1 \circ l'_1))a'_1, A_2(\mathbf{id}_s + (l''_2 \circ l'_2))a'_2) \\
= \ & (A_1(\mathbf{id}_s + (l''_1 \circ l_1))a_1, A_2(\mathbf{id}_s + (l''_2 \circ l_2))a_2)
\end{aligned}
$$

i.e., $(A_1(\mathbf{id}_s + (l''_1 \circ l_1))a_1, A_2(\mathbf{id}_s + (l''_2 \circ l_2))a_2) \in S(i + i_0)$. So if $[s_1, a_1]\ \widetilde{S}i\ [s_2, a_2]$, then there are some object $\langle w_0, i_0, s_0\rangle$ in $\mathcal{I}^{\to}$ and some injections $l^0_1 : s_1 \to s_0$ and $l^0_2 : s_2 \to a_0$, namely $l^0_1 = l''_1 \circ l'_1$ and $l^0_2 = l''_2 \circ l'_2$, such that

$$A_1(\mathbf{id}_s + l^0_1)a_1\ S(i + i_0)\ A_2(\mathbf{id}_s + l^0_2)a_2. \tag{4.8}$$

Conversely, if (4.8) holds, then the right-hand side of (4.7) holds as well, for $i' = i_0$, $a'_1 = A_1(\mathbf{id}_s + l_1^0)a_1$ and $a'_2 A_2(\mathbf{id}_s + l_2^0)a_2$. (4.7) is thus equivalent to

$$\begin{aligned}[s_1, a_1]\ \widetilde{S}i\ [s_2, a_2] \iff \\ \exists\langle w_0, i_0, s_0\rangle \in \mathcal{I}^{\rightarrow}, l_1 : s_1 \rightarrow s_0 \in \mathcal{I}, l_2 : s_2 \rightarrow s_0 \in \mathcal{I}\cdot \\ A_1(\mathbf{id}_s + l_1)(a_1)\ S(i + i_0)\ A_2(\mathbf{id}_s + l_2)(a_2).\end{aligned} \tag{4.9}$$

So we arrive at the definition of logical relations for monadic types, derived over $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$.

4.5 Une relation logique pour le métalangage

The derivation of logical relations in the last section, on the category $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$, allows us to derive a logical relation for the cryptographic metalanguage, which requires in the first place the definitions of relations for base types.

Recall that the very essential point about logical relations is that the Basic Lemma must hold. The categorical derivation of logical relations already guarantees this, if every constant is related to itself. Furthermore, logical relations defined over a functor category are Kripke logical relations and they must satisfy the monotonicity property so that the Basic Lemma will hold for these logical relations. So basically, when defining a logical relation based on a category like $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$, we must check the following two conditions:

- every constant is related to itself;
- the logical relation is monotonic.

As discussed at the beginning of last section, a proper way to define the relation between keys is to fix a set of "disclosed" keys (denoted by a parameter *fr*). This parameter can be defined in a natural way, using the category $\mathcal{I}^{\rightarrow}$. The relation for keys is then defined as follows: for any object $\langle w, i, s\rangle \in \mathcal{I}^{\rightarrow}$ and any pair of keys $k_1, k_2 \in [\![\mathsf{key}]\!]s = s$,

$$k_1\ \mathcal{R}^i_{\mathsf{key}}\ k_2 \iff k_1 = k_2 \in i(w).$$

We say that a key $k \in s$ is *auto-related* if $k \in i(w)$. The original parameter *fr* becomes $i(w)$. Indeed, an injection $i : w \rightarrow s$ in $\mathcal{I}$ can be seen as a selection of disclosed keys in s. In a Kripke logical relation derived from the category $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$, a world — an object in $\mathcal{I}^{\rightarrow}$ — is then a set of keys together with a selection of disclosed keys.

As for relations for types bool and nat, to identify contextual equivalence, the identity relation is the only choice.

4.5.1 La relation entre les messages

To define the relation for messages, we should pay attention to those "message-related" constants, namely

$$\Sigma_m = \{\mathtt{k}, \mathtt{getkey}, \mathtt{n}, \mathtt{getnum}, \mathtt{p}, \mathtt{fst}, \mathtt{snd}, \mathtt{enc}, \mathtt{dec}\}.$$

Every constant in Σ_m is used to either construct a message from other messages or concrete values like integers or keys, or destruct a message into smaller messages or concrete values. In a word, they change the structure of messages, so a natural way is to define the relation for messages by induction on the message structure, from those relations for integers and keys.

First, for every injection $i : w \to s \in \mathcal{I}$, define a relation $\mathcal{MR}^i_\perp \subseteq [\![\mathsf{msg}]\!]s \times [\![\mathsf{msg}]\!]s$ by

- $(n(n_1), n(n_2)) \in \mathcal{MR}^i_\perp$, for all $(n_1, n_2) \in \mathcal{R}^i_{\mathsf{nat}}$;
- $(k(k_1), k(k_2)) \in \mathcal{MR}^i_\perp$, for all $(k_1, k_2) \in \mathcal{R}^i_{\mathsf{key}}$;
- if $(m_1, m_2) \in \mathcal{MR}^i_\perp$ and $(m'_1, m'_2) \in \mathcal{MR}^i_\perp$, then $(p(m_1, m'_1), p(m_2, m'_2)) \in \mathcal{MR}^i_\perp$;
- if $(m_1, m_2) \in \mathcal{MR}^i_\perp$ and $k \in i(w)$, then $(e(m_1, k), e(m_2, k)) \in \mathcal{MR}^i_\perp$.

The relation $\mathcal{MR}_\perp$ is indeed built by induction on the message structure, from relations $\mathcal{R}_{\mathsf{nat}}$ and $\mathcal{R}_{\mathsf{key}}$. However, since keys are divided into "disclosed" keys and "secret" keys and none of "secret" keys is related by $\mathcal{R}_{\mathsf{key}}$, messages, in particular cipher-texts, are accordingly divided into two parts — "non-secret" cipher-texts that are encrypted using disclosed keys and "secret" cipher-texts that are encrypted using secret keys. The so inductively defined relation $\mathcal{MR}_\perp$ touches only those non-secret cipher-texts and does not say anything about secret cipher-texts. In other words, a cipher-text encrypted with a secret key (a non-auto-related key) is not related to any other message by $\mathcal{MR}_\perp$. Of course, this is too strict. In our model, if we cannot decrypt two cipher-texts, we shall then consider them as equivalent, since there is no way for us to see whether the two corresponding plain-texts are equivalent. Basically, we can simply let all secret cipher-texts be related with each other and we define another relation $\mathcal{MR}^i_\top \subseteq [\![\mathsf{msg}]\!]s \times [\![\mathsf{msg}]\!]s$, for every injection $i : w' \to s \in \mathcal{I}$, by induction on the structure of messages:

- $(n(n_1), n(n_2)) \in \mathcal{MR}^i_\top$, for all $(n_1, n_2) \in \mathcal{R}^i_{\mathsf{nat}}$;
- $(k(k_1), k(k_2)) \in \mathcal{MR}^i_\top$, for all $(k_1, k_2) \in \mathcal{R}^i_{\mathsf{key}}$;
- if $(m_1, m_2) \in \mathcal{MR}^i_\top$ and $(m'_1, m'_2) \in \mathcal{MR}^i_\top$, then $(p(m_1, m'_1), p(m_2, m'_2)) \in \mathcal{MR}^i_\top$;
- if $(m_1, m_2) \in \mathcal{MR}^i_\top$ and $k \in i(w)$, then $(e(m_1, k), e(m_2, k)) \in \mathcal{MR}^i_\top$.
- if $(m_1, m_2) \in \mathcal{MR}^i_\top$ and $k_1, k_2 \notin i(w)$, then $(e(m_1, k_1), e(m_2, k_2)) \in \mathcal{MR}^i_\top$.

Clearly, for any $\mathbf{id}_s : s \to s$, $\mathcal{MR}_\perp^{\mathbf{id}_s} = \mathcal{MR}_\top^{\mathbf{id}_s}$ and we write it as $\mathcal{MR}^s$.

To make the *Basic Lemma* hold, the relation for messages must be sandwiched between the relation $\mathcal{MR}_\perp$ and the relation $\mathcal{MR}_\top$ [SP03, GLLNZ04]. In particular, if we do not consider dynamic key generation and simply take a set-theoretical model, the *Basic Lemma* holds for all relations sandwiched between $\mathcal{MR}_\perp$ and $\mathcal{MR}_\top$ [GLLNZ04].

The largest relation $\mathcal{MR}_\top$ is somehow too much when contextual equivalence is our concern. Consider the following program:

$$p(n) = (\nu k).\langle \{n * 2\}_k, \lambda x.\mathtt{dec}(x,k) \mathbin{\mathtt{mod}} 2\rangle.$$

It is clear that if k is secret, then two instances of this program, with concrete numbers for the argument n, should be contextually equivalent, but to relate these instances, we must not relate encrypted odd numbers, by key k, with encrypted even numbers, otherwise, instances of the function (the second component) are not related. This indeed shows that we should not simply consider all secret cipher-texts as equivalent.

Furthermore, it should be also noticed that, to some extent, logical relations for deducing contextual equivalence also indicate that contexts can get access to related values. For example, a proper relation between secret cipher-texts for the above program should not relate encrypted odd numbers (by key k) with whatever, since they are never produced. This point does interfere with our choice of relations for the msg type.

In order to define uniquely a logical relation for the cryptographic metalanguage, a natural way is to make the relation between secret cipher-texts as a parameter. This is exactly what Sumii and Pierce did in the cryptographic lambda-calculus [SP01].

For every injection $i : w \to s \in \mathcal{I}$, define a function φ^i which maps a pair of secret keys to a set of message pairs:

$$\varphi^i : (s - i(w)) \times (s - i(w)) \to \mathbb{P}([\![\mathsf{msg}]\!]s \times [\![\mathsf{msg}]\!]s) \tag{4.10}$$

where $\mathbb{P}$ denotes the power set. φ is called a *cipher function*. It is indeed a group of functions indexed by injections in $\mathcal{I}$, i.e., objects of the category $\mathcal{I}^\to$.

Given a cipher function φ, we can then define a unique relation for messages.

Definition 4.2. *For every injection $i : w \to s \in \mathcal{I}$, a* cryptographic message relation $\mathcal{MR}^{i,\varphi} \subseteq [\![\mathsf{msg}]\!]s \times [\![\mathsf{msg}]\!]s$*, is the smallest relation such that*

- $(n(n_1), n(n_2)) \in \mathcal{MR}^{i,\varphi}$*, for all* $(n_1, n_2) \in \mathcal{R}^i_{\mathsf{nat}}$*;*
- $(k(k_1), k(k_2)) \in \mathcal{MR}^{i,\varphi}$*, for all* $(k_1, k_2) \in \mathcal{R}^i_{\mathsf{key}}$*;*
- *if* $(m_1, m_2) \in \mathcal{MR}^{i,\varphi}$ *and* $(m_1', m_2') \in \mathcal{MR}^{i,\varphi}$*, then* $(p(m_1, m_1'), p(m_2, m_2')) \in \mathcal{MR}^{i,\varphi}$*;*

- *if* $(m_1, m_2) \in \mathcal{MR}^{i,\varphi}$ *and* $(k_1, k_2) \in \mathcal{R}^i_{\mathsf{key}}$, *then* $(\mathsf{e}(m_1, k), \mathsf{e}(m_2, k)) \in \mathcal{MR}^{i,\varphi}$.
- *if* $k_1, k_2 \in s - i(w)$ *and* $(m_1, m_2) \in \varphi^i(k_1, k_2)$, *then* $(\mathsf{e}(m_1, k_1), \mathsf{e}(m_2, k_2)) \in \mathcal{MR}^{i,\varphi}$.

We say that a cipher function φ is *logical* if, for every $\langle w, i, s\rangle \in \mathcal{I}^{\rightarrow}$ and every pair of keys $k_1, k_2 \in s - i(w)$,

$$(m_1, m_2) \in \varphi^i(k_1, k_2) \Longrightarrow (\mathsf{e}(m_1, k_1), \mathsf{e}(m_2, k_2)) \in \mathcal{MR}^s.$$

In other words, a logical cipher function must be consistent with the inductively defined message relation where we assume that all keys are disclosed. In particular, since both $\mathcal{R}^{\mathbf{id}_s}_{\mathsf{nat}}$ and $\mathcal{R}^{\mathbf{id}_s}_{\mathsf{key}}$ are identity relations, $\mathcal{MR}^s$ is just the identity relation over $[\![\mathsf{msg}]\!]s$. Clearly, in a logical cipher function φ, if $k_1 \neq k_2$, then $\varphi^i(k_1, k_2) = \emptyset$, and for any $k \in s - i(w)$ and any pair of messages $(m_1, m_2) \in \varphi^i(k, k)$, we have $m_1 = m_2$.

Lemma 4.3. *If the cipher function φ is logical, then any pair of messages related by $\mathcal{MR}^{i,\varphi}$ are identical, i.e.,*

$$(m_1, m_2) \in \mathcal{MR}^{i,\varphi} \Longrightarrow m_1 = m_2.$$

Proof. By induction on the message structure. □

We say that a cipher function φ is *monotonically logical* if it is logical and those related cipher-texts according to φ should remain related when they are lifted to a larger world. Precisely, for every morphism $(j, l) : \langle w, i, s\rangle \rightarrow \langle w', i', s'\rangle$ in $\mathcal{I}^{\rightarrow}$ and every key $k \in s - i(w)$ such that $l(k) \in s' - i'(w')$, if a pair of messages $(m_1, m_2) \in \varphi^i(k, k)$, then $([\![\mathsf{msg}]\!]l(m_1), [\![\mathsf{msg}]\!]l(m_2)) \in \varphi^{i'}(l(k), l(k))$. Indeed, a logical cipher function is a partial identity relation over secret cipher messages.

Given a cipher function φ, if for every morphism $(j, l) : \langle w, i, s\rangle \rightarrow \langle w', i', s'\rangle \in \mathcal{I}^{\rightarrow}$ and for every pair of values $m_1, m_2 \in [\![\mathsf{msg}]\!]s$,

$$(m_1, m_2) \in \mathcal{MR}^{i,\varphi} \Longrightarrow ([\![\mathsf{msg}]\!]l(m_1), [\![\mathsf{msg}]\!]l(m_2)) \in \mathcal{MR}^{i',\varphi},$$

then we say that the cryptographic message relation $\mathcal{MR}$ is *monotonic on* $\mathcal{I}^{\rightarrow}$.

Lemma 4.4. *If the cipher function φ is monotonically logical, then the cryptographic message relation $\mathcal{MR}$ is monotonic on $\mathcal{I}^{\rightarrow}$.*

Proof. We prove the statement by induction on the message structure. It is easy to check that this property holds when the message is of the form *n*(_), *k*(_) or *p*(_, _).

If $m_1 = e(m_1', k_1)$ and $m_2 = e(m_2', k_2)$, $[\![\mathsf{msg}]\!]l(m_1) = e([\![\mathsf{msg}]\!]l(m_1'), l(k_1))$ and $[\![\mathsf{msg}]\!]l(m_2) = e([\![\mathsf{msg}]\!]l(m_2'), l(k_2))$. According to Lemma 4.3, $m_1 = m_2$, so $k_1 = k_2$. Because l is injective and so is $[\![\mathsf{msg}]\!]l$, $l(k_1) = l(k_2)$ and $[\![\mathsf{msg}]\!]l(m_1) = [\![\mathsf{msg}]\!]l(m_2)$.

- If $k_1 = k_2 \in i(w)$, then $l(k_1) = l(k_2) \in i'(w')$ and $(m'_1, m'_2) \in \mathcal{MR}^{i,\varphi}$. By induction, $([\![\mathsf{msg}]\!]l(m'_1), [\![\mathsf{msg}]\!]l(m'_2)) \in \mathcal{MR}^{i',\varphi}$, hence

$$(e([\![\mathsf{msg}]\!]l(m'_1), l(k_1), e([\![\mathsf{msg}]\!]l(m'_2), l(k_2)) \in \mathcal{MR}^{i',\varphi}.$$

- If $k_1 = k_2 \notin i(w)$ but $l(k_1) = l(k_2) \in i'(w')$, consider the contents of m'_1 and m'_2. If m'_1 and m'_2 contain no secret cipher-texts, i.e., $m_1, m_2 \in [\![\mathsf{msg}]\!](i(w))$, because φ is logical, $(m'_1, m'_2) \in \mathcal{MR}^{i,\varphi}$. If m'_1, m'_2 contain secret cipher-texts, these secret cipher-texts must be related by $\mathcal{MR}^{i,\varphi}$ according to φ, hence $(m'_1, m'_2) \in \mathcal{MR}^{i,\varphi}$ as well. Then by induction,

$$([\![\mathsf{msg}]\!]l(m'_1), [\![\mathsf{msg}]\!]l(m'_2)) \in \mathcal{MR}^{i',\varphi},$$

  and consequently $([\![\mathsf{msg}]\!]l(m_1), [\![\mathsf{msg}]\!]l(m_2)) \in \mathcal{MR}^{i',\varphi}$.

- If $l(k_1) = l(k_2) \notin i'(w')$, $([\![\mathsf{msg}]\!]l(m'_1), [\![\mathsf{msg}]\!]l(m'_2)) \in \varphi^{i'}(l(k_1), l(k_2))$ because φ is monotonically logical, hence

$$(e([\![\mathsf{msg}]\!]l(m'_1), l(k_1), e([\![\mathsf{msg}]\!]l(m'_2), l(k_2)) \in \mathcal{MR}^{i',\varphi}.$$

□

The relation $\mathcal{MR}^{i,\varphi}$ is not monotonic on $\mathcal{I}^{\rightarrow}$ in general. For example, consider a pair of messages $(m_1, m_2) \in \varphi^{\langle w,\subseteq,s\rangle}(k_1, k_2)$, for some pair of keys $k_1, k_2 \in s - w$. In particular, we assume that $k_1 \neq k_2$. Take a morphism $\langle j, \mathbf{id}_s\rangle : \langle w, \subseteq, s\rangle \rightarrow \langle w + \{k_1\}, \subseteq, s\rangle$ where j is inclusion, then two encrypted messages $(e(m_1, k_1), e(m_2, k_2)) \in \mathcal{MR}^{\langle w,\subseteq,s\rangle,\varphi}$, are not related by $\mathcal{MR}^{\langle w+\{k_1\},\subseteq,s\rangle,\varphi}$, whatever φ is.

Since we allow contexts to be defined at larger worlds, the monotonicity property will interfere a lot with the choice of relations for base types, as we wish to use $\mathcal{MR}$ to identify contextual equivalence between messages. Thus, only relating messages at a certain world is not enough to show that they are equivalent. We must also be able to relate them at every larger world, against attackers defined at larger worlds.

4.5.2 Une relation logique cryptographique faible

Putting together relations for different types, we then arrive at a logical relation for the cryptographic metalanguage, defined over the category $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ according to the general construction.

Definition 4.3. *For every object $\langle w, i, s\rangle \in \mathcal{I}^{\rightarrow}$, the relations $\mathcal{R}^{i,\varphi}_{\tau} \subseteq [\![\tau]\!]s \times [\![\tau]\!]s$, where φ is a cipher function as specified in (4.10), are defined by induction over the structure of type τ,* *according to:*

$$
\begin{aligned}
& b_1\ \mathcal{R}^{i,\varphi}_{\mathsf{bool}}\ b_2 \iff b_1 = b_2 \\
& n_1\ \mathcal{R}^{i,\varphi}_{\mathsf{nat}}\ n_2 \iff n_1 = n_2 \\
& k_1\ \mathcal{R}^{i,\varphi}_{\mathsf{key}}\ k_2 \iff k_1 = k_2 \in i(w) \\
& m_1\ \mathcal{R}^{i,\varphi}_{\mathsf{msg}}\ m_2 \iff (m_1, m_2) \in \mathcal{MR}^{i,\varphi} \\
& (a_1, a_1')\ \mathcal{R}^{i,\varphi}_{\tau\times\tau'}\ (a_2, a_2') \iff a_1\ \mathcal{R}^{i,\varphi}_{\tau}\ a_2\ \ \&\ \ a_1'\ \mathcal{R}^{i,\varphi}_{\tau'}\ a_2' \\
& a_1\ \mathcal{R}^{i,\varphi}_{\mathsf{opt}[\tau]}\ a_2 \iff (a_1\ \mathcal{R}^{i,\varphi}_{\tau}\ a_2)\ \mathit{or}\ (a_1 = a_2 = \bot) \\
& f_1\ \mathcal{R}^{i,\varphi}_{\tau\to\tau'}\ f_2 \iff \\
& \qquad\qquad \forall (j,l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle \in \mathcal{I}^{\to} \cdot\ \forall a_1, a_2 \in [\![\tau]\!]s' \cdot \\
& \qquad\qquad\qquad (a_1\ \mathcal{R}^{i',\varphi}_{\tau}\ a_2 \Rightarrow f_1 s'(l, a_1)\ \mathcal{R}^{i',\varphi}_{\tau'}\ f_2 s'(l, a_2)) \\
& [s_1, a_1]\ \mathcal{R}^{i,\varphi}_{\mathsf{T}\tau}\ [s_2, a_2] \iff \\
& \qquad\qquad \exists \langle w_0, i_0, s_0\rangle \in \mathcal{I}^{\to}, l_1 : s_1 \to s_0 \in \mathcal{I}, l_2 : s_2 \to s_0 \in \mathcal{I} \cdot \\
& \qquad\qquad\qquad [\![\tau]\!](\mathbf{id}_s + l_1)(a_1)\ \mathcal{R}^{i+i_0,\varphi}_{\tau}\ [\![\tau]\!](\mathbf{id}_s + l_2)(a_2)
\end{aligned}
$$

where $\mathcal{MR}^{i,\varphi}$ *is a cryptographic message relation.*

Keep in mind that we are using logical relations to deduce contextual equivalence and the Basic Lemma is very crucial for this purpose. Does the Basic Lemma hold for this logical relation? Recall that the categorical derivation of logical relations guarantees the Basic Lemma, but we must check two conditions — every constant is related to itself, and logical relations must be monotonic. Let us check the monotonicity property first.

Lemma 4.5. *The logical relation* $\mathcal{R}^{i,\varphi}_{\tau}$ *is monotonic when* φ *is monotonically logical, i.e., for every morphism* $(j,l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle \in \mathcal{I}^{\to}$ *and every pair of values* $a_1, a_2 \in [\![\tau]\!]s$*,*

$$a_1\ \mathcal{R}^{i,\varphi}_{\tau}\ a_2 \Longrightarrow [\![\tau]\!]l(a_1)\ \mathcal{R}^{i',\varphi}_{\tau}\ [\![\tau]\!]l(a_2).$$

Proof. It is obvious that relations for base types are monotonic, in particular the monotonicity of $\mathcal{R}^{\varphi}_{\mathsf{msg}}$ is shown by Lemma 4.4. For complex types, the monotonicity is proved by induction on the type structure. Here are the cases for function types and computation types. Others are standard.

- Function types $\tau \to \tau'$: Consider two related functions $f_1, f_2 \in [\![\tau \to \tau']\!]$ such that $f_1\ \mathcal{R}^{i,\varphi}_{\tau\to\tau'}\ f_2$. For any morphism $\langle j', l'\rangle : \langle w', i', s'\rangle \to \langle w'', i'', s''\rangle \in \mathcal{I}^{\to}$ ($\langle j' \circ j, l' \circ l\rangle$ is then a morphism from $\langle w, i, s\rangle$ to $\langle w'', i'', s''\rangle$), and for any $a_1, a_2 \in [\![\tau]\!]s''$ such that $a_1\ \mathcal{R}^{i'',\varphi}_{\tau}\ a_2$, because $f_1\ \mathcal{R}^{i,\varphi}_{\tau\to\tau'}\ f_2$,

$$f_1 s''(l' \circ l, a_1)\ \mathcal{R}^{i'',\varphi}_{\tau'}\ f_2 s''(l' \circ l, a_2).$$

By the definition of exponential in the category $\mathcal{S}et^{\mathcal{I}}$, for every function $f \in [\![\tau \to \tau']\!]s$ and every value $a \in [\![\tau]\!]s''$,

$$([\![\tau']\!]^{[\![\tau]\!]}l(f))s''(l', a) = fs''(l' \circ l, a)$$

hence

$$([\![\tau']\!]^{[\![\tau]\!]}l(f_1))s''(l', a_1)\ \mathcal{R}^{i'',\varphi}_{\tau'}\ ([\![\tau']\!]^{[\![\tau]\!]}l(f_2))s''(l', a_2),$$

and consequently $[\![\tau']\!]^{[\![\tau]\!]}l(f_1)\ \mathcal{R}^{i',\varphi}_{\tau\to\tau'}\ [\![\tau']\!]^{[\![\tau]\!]}l(f_2)$.

- Computation types $\mathsf{T}\tau$:

$$[\![\mathsf{T}\tau]\!]l([s, a]) = T[\![\tau]\!]l([s, a]) = [s, [\![\tau]\!](l + \mathbf{id}_s)(a)].$$

If $[s_1, a_1]\ \mathcal{R}^{i,\varphi}_{\mathsf{T}\tau}\ [s_2, a_2]$, we must show that

$$[s_1, [\![\tau]\!](l + \mathbf{id}_{s_1})(a_1)]\ \mathcal{R}^{i',\varphi}_{\mathsf{T}\tau}\ [s_2, [\![\tau]\!](l + \mathbf{id}_{s_2})(a_2)].$$

According to the definition of $\mathcal{R}_{\mathsf{T}\tau}$, we should find a pair $\langle w_0', i_0', s_0'\rangle \in \mathcal{I}^{\to}$ and two injections $l_1' : s_1 \to s_0'$ and $l_2' : s_2 \to s_0'$ in $\mathcal{I}$ such that

$$[\![\tau]\!](\mathbf{id}_{s'} + l_1')([\![\tau]\!](l + \mathbf{id}_{s_1})(a_1))\ \mathcal{R}^{i'+i_0',\varphi}_{\tau}\ [\![\tau]\!](\mathbf{id}_{s'} + l_2')([\![\tau]\!](l + \mathbf{id}_{s_2})(a_2))$$

which, by the functorality of $[\![\tau]\!]$, is equivalent to

$$[\![\tau]\!]k_1(a_1)\ \mathcal{R}^{i'+i_0',\varphi}_{\tau}\ [\![\tau]\!]k_2(a_2)$$

where

$$k_m = (\mathbf{id}_{s'} + l_m') \circ (l + \mathbf{id}_{s_m}) :\ s + s_m \xrightarrow{l+\mathbf{id}_{s_m}} s' + s_m \xrightarrow{\mathbf{id}_{s'}+l_m'} s' + s_0'\ , \quad (m = 1, 2).$$

Since $[s_1, a_1]\ \mathcal{R}^{i,\varphi}_{\mathsf{T}\tau}\ [s_2, a_2]$, there exist an object $\langle w_0, i_0, s_0\rangle$ in $\mathcal{I}^{\to}$ and two injections $l_1 : s_1 \to s_0$, $l_2 : s_2 \to s_0$ such that $[\![\tau]\!](\mathbf{id}_s + l_1)(a_1)\ \mathcal{R}^{i+i_0}_{\tau}\ [\![\tau]\!](\mathbf{id}_s + l_2)(a_2)$.

  - If $s' \cap s_0 = \emptyset$, simply let $s_0' = s_0$, $w_0' = w_0$, $i_1' = i_1$ and $i_2' = i_2$, then

$$(j + \mathbf{id}_{w_0}, l + \mathbf{id}_{s_0}) : \langle w + w_0, i + i_0, s + s_0\rangle \to \langle w' + w_0, i' + i_0, s' + s_0\rangle$$

is a morphism in $\mathcal{I}^{\to}$, and by induction (the relation at type τ is monotonic),

$$[\![\tau]\!](l + \mathbf{id}_{s_0})([\![\tau]\!](\mathbf{id}_s + l_1)(a_1))\ \mathcal{R}^{i'+i_0,\varphi}_{\tau}\ [\![\tau]\!](l + \mathbf{id}_{s_0})([\![\tau]\!](\mathbf{id}_s + l_2)(a_2)),$$

where the two elements equal $[\![\tau]\!]k_1(a_1)$ and $[\![\tau]\!]k_2(a_2)$ respectively, because the following square commutes in $\mathcal{I}$

$$\begin{tikzcd} s + s_m \arrow[r, "l+\mathbf{id}_{s_m}"] \arrow[d, "\mathbf{id}_s+l_m"'] & s' + s_m \arrow[d, "\mathbf{id}_{s'}+l_m"] \\ s + s_0 \arrow[r, "l+\mathbf{id}_{s_0}"] & s' + s_0 \end{tikzcd} \qquad (m = 1, 2).$$

  - If $s' \cap s_0 \neq \emptyset$, it is always possible to find a set s_0'' which is isomorphic to s_0 such that $s' \cap s_0'' = \emptyset$, then let $s_0' = s_0''$ and the proof goes identically as in the previous case.

So $[\![\mathsf{T}\tau]\!]l([s_1, a_1])\ \mathcal{R}^{i',\varphi}_{\mathsf{T}\tau}\ [\![\mathsf{T}\tau]\!]l([s_2, a_2])$. □

Given a cipher function φ, say that two environments $\rho_1, \rho_2 \in [\![\Gamma]\!]s$ are related at $\langle w, i, s\rangle$, written as $\rho_1\ \mathcal{R}^{i,\varphi}_{\Gamma}\ \rho_2$, if and only if, for every variable $x : \tau \in \Gamma$, $\rho_1(x)\ \mathcal{R}^{i,\varphi}_{\tau}\ \rho_2(x)$.

Lemma 4.6. *For every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$, if $\rho_1\ \mathcal{R}^{i,\varphi}_{\Gamma}\ \rho_2$ and the cipher function φ is monotonically logical, then $[\![\Gamma]\!]i(\rho_1)\ \mathcal{R}^{i',\varphi}_{\Gamma}\ [\![\Gamma]\!]i(\rho_2)$.*

Proof. This is a corollary of Lemma 4.5 since $[\![\Gamma]\!]$ is interpreted as $[\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$. □

The logical relation $\mathcal{R}^{\langle w,i,s\rangle,\varphi}_{\tau}$ is sound, i.e., the Basic Lemma holds, when the cipher function φ is monotonically logical.

Proposition 4.7 (Basic Lemma). *Let $\Gamma \vdash t : \tau$ be a well typed term in the cryptographic metalanguage. For every $\langle w, i, s\rangle \in \mathcal{I}^{\to}$ and every monotonically logical cipher function φ, if two environments $\rho_1, \rho_2 \in [\![\Gamma]\!]s$ are related, i.e., $\rho_1\ \mathcal{R}^{i,\varphi}_{\Gamma}\ \rho_2$, then $[\![t]\!]s\rho_1\ \mathcal{R}^{i,\varphi}_{\tau}\ [\![t]\!]s\rho_2$.*

Proof. This is proved by induction on the structure of the term t. We show several induction steps here, notably functions, applications, computation constants and cryptographic primitives.

- Functions $\lambda x.t$ of type $\tau \to \tau'$: according to Figure 3.1, $[\![\lambda x.t]\!]s\rho$ is some function $f \in [\![\tau \to \tau']\!]s$ such that for any $l : s \to s' \in \mathcal{I}$ and for any $a \in [\![\tau]\!]s'$, $fs'(l, a) = [\![t]\!]s'([\![\Gamma]\!]l(\rho) \cup \{x \mapsto a\})$. Let $f_1 = [\![\lambda x.t]\!]s\rho_1$ and $f_2 = [\![\lambda x.t]\!]s\rho_2$. For every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$, let $a_1, a_2 \in [\![\tau]\!]s'$ be related values at $\langle w', i', s'\rangle$, i.e., $a_1\ \mathcal{R}^{i',\varphi}_{\tau}\ a_2$. According to Lemma 4.5, $\mathcal{R}^{i,\varphi}$ is monotonic, so

$$[\![\Gamma]\!]l(\rho_1) \cup \{x \mapsto a_1\}\ \mathcal{R}^{i',\varphi}_{\Gamma}\ [\![\Gamma]\!]l(\rho_2) \cup \{x \mapsto a_2\}$$

  and by induction,

$$[\![t]\!]s'([\![\Gamma]\!]l(\rho_1) \cup \{x \mapsto a_1\})\ \mathcal{R}^{i',\varphi}_{\tau'}\ [\![t]\!]s'([\![\Gamma]\!]l(\rho_2) \cup \{x \mapsto a_2\}),$$

  hence $f_1\ \mathcal{R}^{i,\varphi}_{\tau\to\tau'}\ f_2$.

- Applications $t_1 t_2$ of type τ', where t_1 is of type $\tau \to \tau'$ and t_2 is of type τ: First, $[\![t_1]\!]s\rho = f \in [\![\tau \to \tau']\!]s$ such that for any $l : s \to s' \in \mathcal{I}$, $fs'(l, a') = [\![t_1 t_2]\!]s'([\![\Gamma]\!]l(\rho))$, where $a' = [\![t_2]\!]s'([\![\Gamma]\!]l(\rho))$. Then $[\![t_1 t_2]\!]s\rho = fs(\mathbf{id}_s, a)$ where $a = [\![t_2]\!]s\rho$. Let $f_1 = [\![t_1]\!]s\rho_1$, $f_2 = [\![t_1]\!]s\rho_2$, $a_1 = [\![t_2]\!]s\rho_1$ and $a_2 = [\![t_2]\!]s\rho_2$. By induction, $f_1\ \mathcal{R}^{i,\varphi}_{\tau\to\tau'}\ f_2$, $a_1\ \mathcal{R}^{i,\varphi}_{\tau}\ a_2$, so $f_1 s(\mathbf{id}_s, a_1)\ \mathcal{R}^{i,\varphi}_{\tau'}\ f_2 s(\mathbf{id}_s, a_2)$.

- Fresh key generation constant new, of type Tkey: $[\![\mathtt{new}]\!]s\rho = [\{k\}, k]$, where $k \notin s$, no matter what ρ is. Take $\langle \{k\}, \mathbf{id}_{\{k\}}, \{k\} \rangle \in \mathcal{I}^{\to}$. It is clear that $k \; \mathcal{R}_{\mathsf{key}}^{i+\mathbf{id}_{\{k\}},\varphi} \; k$, hence $[\{k\}, k] \; \mathcal{R}_{\mathsf{Tkey}}^{i,\varphi} \; [\{k\}, k]$.

- Trivial computations $\mathtt{val}(t)$ of type $\mathsf{T}\tau$, where t is of type τ: Because $[\![\mathtt{val}(t)]\!]s\rho = [\emptyset, [\![t]\!]s\rho]$ and by induction, $[\![t]\!]s\rho_1 \; \mathcal{R}_{\tau}^{i,\varphi} \; [\![t]\!]s\rho_2$, according to the definition of $\mathcal{R}_{\mathsf{T}\tau}$, $[\![\mathtt{val}(t)]\!]s\rho_1 \; \mathcal{R}_{\mathsf{T}\tau}^{i,\varphi} \; [\![\mathtt{val}(t)]\!]s\rho_2$.

- Sequential computations $\mathtt{let}\; x \Leftarrow t_1 \;\mathtt{in}\; t_2$ of type $\mathsf{T}\tau'$, where x is of type τ, t_1 is of type $\mathsf{T}\tau$ and t_2 is of type $\mathsf{T}\tau'$: Let $[\![t_1]\!]s\rho_1 = [s_1, a_1]$ and $[\![t_1]\!]s\rho_2 = [s_2, a_2]$. By induction, $[s_1, a_1] \; \mathcal{R}_{\mathsf{T}\tau}^{i,\varphi} \; [s_2, a_2]$, so there exist some $\langle w_0, i_0, s_0 \rangle$ in $\mathcal{I}^{\to}$ and two injections $l_1 : s_1 \to s_0$, $l_2 : s_2 \to s_0$ such that

$$[\![\tau]\!](l_1 + \mathbf{id}_s)a_1 \; \mathcal{R}_{\tau}^{i+i_0,\varphi} \; [\![\tau]\!](l_2 + \mathbf{id}_s)a_2.$$

  Let $[\![t_2]\!](s + s_0)\rho'_1 = [s'_1, b_1]$ and $[\![t_2]\!](s + s_0)\rho'_2 = [s'_2, b_2]$, where

$$\rho'_m = [\![\Gamma]\!]\mathbf{inl}_{s,s_0}(\rho_m) \cup \{x \mapsto [\![\tau]\!](l_m + \mathbf{id}_s)a_m\}, \quad (m = 1, 2).$$

  Again by induction, there exist some $\langle w'_0, i'_0, s'_0 \rangle$ in $\mathcal{I}^{\to}$ and two injections $l'_1 : s'_1 \to s'_0$, $l'_2 : s'_2 \to s'_0$ such that

$$[\![\tau]\!](l'_1 + \mathbf{id}_{s+s_0})b_1 \; \mathcal{R}_{\tau}^{i+i_0+i'_0,\varphi} \; [\![\tau]\!](l'_2 + \mathbf{id}_{s+s_0})b_2.$$

  According to the definition of monad T on $\mathcal{S}et^{\mathcal{I}}$, $[s_m, a_m] = [s_0, [\![\tau]\!](l_m + \mathbf{id}_s)a_m]$, so

$$[\![\mathtt{let}\; x \Leftarrow t_1 \;\mathtt{in}\; t_2]\!]s\rho_m = [s_0 + s'_m, b_m].$$

  Considering $\langle w_0 + w'_0, i_0 + i'_0, s_0 + s'_0 \rangle \in \mathcal{I}^{\to}$ and injections

$$\mathbf{id}_{s_0} + l'_1 : s_0 + s_1 \to s_0 + s'_0, \quad \mathbf{id}_{s_0} + l'_2 : s_0 + s_2 \to s_0 + s'_0,$$

  we get $[s_0 + s'_1, b_1] \; \mathcal{R}_{\mathsf{T}\tau'}^{i,\varphi} \; [s_0 + s'_2, b_2]$.

- Encryption $\mathtt{enc}(t_1, t_2)$ of type msg, where t is of type msg and t_2 is of type key: First, let $a_1 = [\![t_1]\!]s\rho_1$, $a_2 = [\![t_1]\!]s\rho_2$, $k_1 = [\![t_2]\!]s\rho_1$ and $k_2 = [\![t_1]\!]s\rho_2$, then $[\![\mathtt{enc}(t_1, t_2)]\!]s\rho_1 = \mathsf{e}(a_1, k_1)$ and $[\![\mathtt{enc}(t_1, t_2)]\!]s\rho_2 = \mathsf{e}(a_2, k_2)$. By induction, $a_1 \; \mathcal{R}_{\mathsf{msg}}^{i,\varphi} \; a_2$ and $k_1 \; \mathcal{R}_{\mathsf{key}}^{i,\varphi} \; k_2$, i.e., $k_1 = k_2 \in i(w)$, then according to the definition of the cryptographic message relation, $\mathsf{e}(a_1, k_1)\mathcal{MR}^{i,\varphi}\mathsf{e}(a_2, k_2)$, i.e., $[\![\mathtt{enc}(t_1, t_2)]\!]s\rho_1 \; \mathcal{R}_{\mathsf{msg}}^{i,\varphi} \; [\![\mathtt{enc}(t_1, t_2)]\!]s\rho_2$.

- Decryption $\mathtt{dec}(t_1, t_2)$ of type opt[msg], where t is of type msg and t_2 is of type key: First, let $a_1 = [\![t_1]\!]s\rho_1$, $a_2 = [\![t_1]\!]s\rho_2$, $k_1 = [\![t_2]\!]s\rho_1$ and $k_2 = [\![t_1]\!]s\rho_2$. By induction,

$k_1\ \mathcal{R}^{i,\varphi}_{\mathsf{key}}\ k_2$, i.e., $k_1 = k_2 \in i(w)$. If $a_1 = \mathsf{e}(a'_1, k)$ and $a_2 = \mathsf{e}(a'_2, k)$, for some $a'_1, a'_2 \in [\![\mathsf{msg}]\!]s$ and $k = k_1 = k_2$, then $[\![\mathtt{dec}(t_1, t_2)]\!]s\rho_1 = a'_1$ and $[\![\mathtt{dec}(t_1, t_2)]\!]s\rho_2 = a'_2$, and by the definition of the cryptographic message relation, $a'_1 \mathcal{MR}^{i,\varphi} a'_2$; if $a_1 \neq \mathsf{e}(a'_1, k)$ and $a_2 \neq \mathsf{e}(a'_2, k)$, then $[\![\mathtt{dec}(t_1, t_2)]\!]s\rho_1 = [\![\mathtt{dec}(t_1, t_2)]\!]s\rho_2 = \bot$. It is not possible that one of a_1, a_2 is a cipher-text encrypted with key k while the other is not, because it implies that a_1 and a_2 must not be related. Therefore, $[\![\mathtt{dec}(t_1, t_2)]\!]s\rho_1\ \mathcal{R}^{i,\varphi}_{\mathsf{opt[msg]}}\ [\![\mathtt{dec}(t_1, t_2)]\!]s\rho_2$. □

The Basic Lemma for the logical relation $\mathcal{R}^{i,\varphi}_{\tau}$ holds when the cipher function φ is monotonically logical. This is indeed a very strict restriction, since such a cipher function relates only identical cipher-texts. When we take a more general cipher function, it is very likely that the Basic Lemma does not hold any more in general. For example, for the term $y : \mathsf{msg} \vdash \lambda x.y : \mathsf{bool} \to \mathsf{msg}$, take a cipher function φ where $(m_1, m_2) \in \varphi^{\langle w, \subseteq, s\rangle}(k_1, k_2)$ $(k_1 \neq k_2)$ and two environments $\rho_1, \rho_2 \in [\![\Gamma]\!]s$ $(\Gamma = \{y : \mathsf{msg}\})$ such that $\rho_1(y) = \mathsf{e}(m_1, k_1)$ and $\rho_2(y) = \mathsf{e}(m_2, k_2)$. Obviously, $\rho_1\ \mathcal{R}^{\langle w, \subseteq, s\rangle, \varphi}_{\Gamma}\ \rho_2$. But $[\![\lambda x.y]\!]s\rho_1$ and $[\![\lambda x.y]\!]s\rho_2$ are not related by $\mathcal{R}^{\langle w, \subseteq, s\rangle, \varphi}_{\mathsf{bool}\to\mathsf{msg}}$, because related functions should map related arguments to related results, *at every larger world*, while at the world $\langle w + \{k_1\}, s\rangle$, the two encrypted messages $\mathsf{e}(m_1, k_1), \mathsf{e}(m_2, k_2)$ are not related any more and we get non-related results.

Logical relations derived over the category $\mathcal{S}et^{\mathcal{I}^{\to}}$ are too weak in the sense that some non-trivial contextually equivalent programs are not related by these relations. Recall the two programs given at the beginning of Section 4.3. Actually, with logical relations derived over the category $\mathcal{S}et^{\mathcal{I}^{\to}}$, we are still not able to relate those two programs. We shall see a more realistic example in the next chapter. It turns out that we can derive stronger logical relations with a subtler category, where we are in particular allowed to choose more general cipher functions without breaking the soundness of logical relations. We shall later on refer to the logical relation defined in Definition 4.3 as the *weak cryptographic logical relation*.

Chapitre 5

Relations logiques cryptographiques

Dans ce chapitre, nous continuons notre discussion sur la construction des relations logiques du métalangage cryptographique. La relation logique définie dans le chapitre 4 n'est pas assez puissante puisque le lemme fondamental n'est correct que pour un ensemble très limité de fonctions de chiffrement φ. Le contre-exemple à la fin du chapitre montre en particulier que le lemme fondamental n'est plus correct si les fonctions de chiffrement permettent de relier des messages chiffrés par des clés différentes. En effet, même si les fonctions de chiffrement ne relient que les messages chiffrés par la même clé, il existe encore des programmes équivalents contextuellement qui ne peuvent pas être reliés par la relation logique.

Nous commençons ce chapitre par un autre exemple qui montre la faiblesse des relations logiques construites sur la catégorie $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$. Nous montrons aussi que la catégorie $\mathcal{I}^{\rightarrow}$ doit satisfaire certaines propriétés algébriques pour qu'elle représente une bonne relation entre les clés. Puis, dans la partie 5.1, nous corrigeons la catégorie $\mathcal{I}^{\rightarrow}$ en ajoutant certaines contraintes et nous appelons la nouvelle catégorie $\mathcal{PI}^{\rightarrow}$. Nous montrons alors que la catégorie $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ est la bonne catégorie où construire les relations logiques du métalangage cryptographique, et la dérivation est donnée dans la partie 5.2. Dans la partie suivante, nous arrivons finalement à une relation logique cryptographique de notre métalangage et nous vérifions certaines propriétés de cette relation logique. De plus, cette relation logique peut servir à vérifier des protocoles concrets. Nous montrons ceci dans la partie 5.4, en vérifiant les deux protocoles du chapitre 2. Dans la dernière partie, nous comparons nos relations logiques dérivées sur la catégorie $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ avec les relations logiques dénotationnelles du nu-calcul de Stark [Sta94] et nous prouvons que les deux sont en effet équivalentes.

The logical relation defined in Section 4.5.2 is too weak because the Basic Lemma holds only for a rather restricted collection of cipher functions. The counterexample at the end of Section 4.5.2 in particular shows that if cipher functions are allowed to relate cipher-texts encrypted by different keys, the Basic Lemma does not hold any more.

However, even if we allow cipher functions to relate only messages encrypted by the same key, there are still equivalent programs that cannot be related by the logical relation. Consider the following two programs (using abbreviations in Figure 2.2):

$$\begin{aligned} p_1 &= \nu k.\langle \{0\}_k,\ \{1\}_k,\ \lambda x.\mathtt{getnum}(\mathtt{dec}(x,k))\rangle, \\ p_2 &= \nu k.\langle \{1\}_k,\ \{0\}_k,\ \lambda x.\mathtt{letopt}\ y \Leftarrow \mathtt{getnum}(\mathtt{dec}(x,k))\ \mathtt{in}\ \mathtt{some}(1-y)\rangle, \end{aligned}$$

both of type

$$\mathsf{T}(\mathsf{msg} \times \mathsf{msg} \times (\mathsf{msg} \to \mathsf{opt}[\mathsf{nat}])),$$

whose denotations in $\mathcal{S}et^{\mathcal{I}}$ at some set s are

$$\begin{aligned} [\![p_1]\!]s &= [\{k\}, (\{0\}_k,\ \{1\}_k,\ f_1)] \\ [\![p_2]\!]s &= [\{k\}, (\{1\}_k,\ \{0\}_k,\ f_2)], \end{aligned}$$

where we write $\{0\}_k$ for $\mathit{e}(\mathit{n}(0), k)$ and $\{1\}_k$ for $\mathit{e}(\mathit{n}(1), k)$, for the sake of clarity, and $f_1 = [\![\lambda x.\mathtt{dec}(x,k)]\!](s + \{k\})$, $f_2 = [\![\lambda x.\mathtt{letopt}\ y \Leftarrow \mathtt{dec}(x,k)\ \mathtt{in}\ (1-y)]\!](s+\{k\})$.

The two programs are contextually equivalent because the first and the second components of the two tuples are messages encrypted by some secret key and they do not give (directly) useful information to contexts. What contexts can do with these encrypted messages (and eventually get some meaningful values) is to apply the third component — a decryption function — to these messages and see whether some meaningful values will be returned. However, when the functions are applied to either the first two messages, or the second two messages, the results are always the same (0 for the first two components and 1 for the second two).

In order to relate these two programs at a certain world $\langle w, s\rangle$, we need to relate the two tuples at some world $\langle w', s'\rangle$, where $s' = s + \{k\}$. Obviously, we must have that $k \notin w'$ and $(\mathit{n}(0), \mathit{n}(1)), (\mathit{n}(1), \mathit{n}(0)) \in \varphi^{w', s+\{k\}}(k,k)$. However, even just for the world $\langle w', s'\rangle$, the two functions are not related, because they have to map related values at any larger world to related results, while it is possible that at some larger world $\langle w'', s''\rangle$, we have $k \in w''$, and then the two functions may get non-related results. For example, applying the two functions to the value $\mathit{e}(\mathit{n}(0), k)$, which is related to itself at $\langle w'', s''\rangle$ since $k \in w''$, we get 0 and 1 respectively.

What we learn from this example, as well as the one at the end of Chapter 4, is that the weakness of logical relations built over the category $\mathcal{S}et^{\mathcal{I}^{\to}}$ is indeed caused by the relation between different worlds. Intuitively, the commuting diagram (4.4) says that, when we pass from a smaller world $\langle w, i, s\rangle$ to a larger world $\langle w', i', s'\rangle$, we can actually get all keys in s, not just those in $i(w)$. In other words, this means that all non-disclosed keys at a certain world, may become known to contexts or attackers at some larger world. This is too much, because it is then impossible for us to hide information from all possible attackers — there are always attackers who know every key and can reveal every secret.

We need more restriction on the category $\mathcal{I}^{\rightarrow}$. Intuitively, we should not allow attackers to get access to keys that are not disclosed, at any larger world. This could seem too strict, as in practice, there are often cases where we only need to hold a secret for a limited duration, so it is certainly possible that we generate a fresh key and we do not disclose it immediately, but at some later stage. However, in our approach, we have to consider this key as a disclosed key, for any world containing it. The reason is that in our model, computation stages are represented by sets of keys, not by time. There is no state in the cryptographic metalanguage, hence no way to disclose a key at later *time*. Once we generate a key in a program and we find that at a certain point this key can be accessed by contexts, then we just take it as a disclosed key when building logical relations, otherwise, it is seen as a non-disclosed key.

More precisely, every morphism $(j, l) : \langle w, i, s\rangle \rightarrow \langle w', i', s'\rangle$ in $\mathcal{I}^{\rightarrow}$ should make the following two conditions hold:

- for any $k \in w$, $i'(j(k)) = l(i(k))$, i.e., every key that is disclosed at the world i, must remain disclosed at the world i';
- for any $k \in s$ but $k \notin i(w)$, $l(k) \notin i'(w')$, i.e., every key that is not disclosed at the world i, must remain secret at the world i'.

For short, these two conditions are just equivalent to the following equation:

$$i'(j(w)) = l(i(w)) = l(s) \cap i'(w'), \tag{5.1}$$

where $i' \circ j = l \circ i$.

5.1 La catégorie $\mathcal{PI}^{\rightarrow}$

To achieve the equation (5.1), we should add some restriction on the category $\mathcal{I}^{\rightarrow}$. In fact, it turns out that if the diagram $\begin{array}{ccc} w & \xrightarrow{i} & s \\ {\scriptstyle j}\downarrow & & \downarrow{\scriptstyle l} \\ w' & \xrightarrow{i'} & s' \end{array}$ is a pull-back in $\mathcal{I}$, then the equation (5.1) necessarily holds (and conversely).

Lemma 5.1. *For any morphism $(j, l) : \langle w, i, s\rangle \rightarrow \langle w', i', s'\rangle$ in $\mathcal{I}^{\rightarrow}$, $i'(j(w)) = l(i(w)) = l(s) \cap i'(w')$ if and only if the commuting square $\begin{array}{ccc} w & \xrightarrow{i} & s \\ {\scriptstyle j}\downarrow & & \downarrow{\scriptstyle l} \\ w' & \xrightarrow{i'} & s' \end{array}$ is a pull-back in $\mathcal{I}$.*

Proof. First, assume that $i'(j(w)) = l(i(w)) = l(s) \cap i'(w')$, then we need to show that the following diagram commutes for any $i_0 : w_0 \to s$ and any $(j_0, l) : i_0 \to i' \in \mathcal{I}^{\to}$:

$$\begin{tikzcd}
w_0 \arrow[drr, bend left, "i_0"] \arrow[dr, dashed, "f"] \arrow[ddr, bend right, "j_0"'] & & \\
& w \arrow[r, "i"] \arrow[d, "j"'] & s \arrow[d, "l"] \\
& w' \arrow[r, "i'"] & s'
\end{tikzcd} . \tag{5.2}$$

We must prove that the two triangles commute and the injection $f : w_0 \to w$ is unique. Note that because (j_0, l) is a morphism in $\mathcal{I}^{\to}$, $l(i_0(w_0)) = i'(j_0(w_0)) \subseteq l(s) \cap i'(w')$. Define f by $f(k_0) = j^{-1}(j_0(k_0))$, for every $k_0 \in w_0$, then

- f is a well-defined injection: because $i'(j_0(w_0)) \subseteq l(s) \cap i'(w') = i'(j(w))$ and i' is injection, $j_0(w_0) \subseteq j(w)$, hence $j^{-1}(j_0(k_0))$ is defined for any $k_0 \in w_0$. Obviously, f is injective since both j (as well as j^{-1} restricted over $j_0(w_0)$) and j_0 are injective;

- $j \circ f = j_0$ and $i \circ f = i_0$: the first is obvious. For any $k_0 \in w_0$,

$$\begin{aligned}
l(i(f(k_0))) &= i'(j(f(k_0))) \qquad \text{(because } i' \circ j = l \circ i, \text{ in } \mathcal{I}^{\to}) \\
&= i'(j(j^{-1}(j_0(k_0)))) \\
&= i'(j_0(k_0)) = l(i_0(k_0)) \qquad \text{(because } i' \circ j_0 = l \circ i_0, \text{ in } \mathcal{I}^{\to}),
\end{aligned}$$

  then $i(f(k_0)) = i_0(k_0)$ since l is injective;

- f is unique: suppose that there is another injection $f' : w_0 \to w$ such that the diagram (5.2) commutes. Take any $k_0 \in w_0$, $j(f(k_0)) = j_0(k_0) = j(f'(k_0))$ and because j is injective, $f(k_0) = f'(k_0)$, hence $f = f'$.

Now suppose that diagram $\begin{tikzcd} w \arrow[r, "i"] \arrow[d, "j"'] & s \arrow[d, "l"] \\ w' \arrow[r, "i'"] & s' \end{tikzcd}$ is a pull-back of i' and l. First note that because this diagram commutes, $l(i(w)) = i'(j(w)) \subseteq l(s) \cap i'(w')$. Assume that there exists some $k \in l(s) \cap i'(w')$ but $k \notin l(i(w))$, then $i'^{-1}(k) \notin j(w)$, and $l^{-1}(k) \notin i(w)$. Let $w_0 = w + \{k\}$ and define $j_0 : w_0 \to w'$ and $i_0 : w_0 \to s$ by: $j_0(k') = j(k'), i_0(k') = i(k')$ for any $k' \in w$, and $j_0(k) = i'^{-1}(k), i_0(k) = l^{-1}(k)$. Clearly, $i' \circ j_0 = l \circ i$, i.e., (j_0, l) is a morphism from i_0 to i' in $\mathcal{I}^{\to}$, but there is no injection from w_0 to w, which is a contradiction to the fact that diagram 5.2 is a pull-back, hence $l(s) \cap i'(w') \subseteq l(i(w)) = i'(j(w))$. $\square$

We can then define a new category $\mathcal{PI}^{\to}$ for worlds as follows:

Definition 5.1. *$\mathcal{PI}^{\rightarrow}$ is a category where objects are tuples $\langle w, i, s\rangle$ with $i : w \to s \in \mathcal{I}$, and morphisms are pairs $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$, where $j : w \to w' \in \mathcal{I}$ and $l : s \to s' \in \mathcal{I}$, such that the following commuting diagram is a pull-back in $\mathcal{I}$:*

$$\begin{tikzcd} w \arrow[r, "i"] \arrow[d, "j"'] & s \arrow[d, "l"] \\ w' \arrow[r, "i'"] & s' \end{tikzcd}$$

We write i for $\langle w, i, s\rangle$ when the domain w and the codomain s of i are clear from the context. The composite of morphisms in $\mathcal{PI}^{\rightarrow}$ is well defined since the composite of two pull-backs is still a pull-back.

Lemma 5.2. *If* $\begin{tikzcd} w \arrow[r, "i"] \arrow[d, "j"'] & s \arrow[d, "l"] \\ w' \arrow[r, "i'"] & s' \end{tikzcd}$ *and* $\begin{tikzcd} w' \arrow[r, "i'"] \arrow[d, "j'"'] & s' \arrow[d, "l'"] \\ w'' \arrow[r, "i''"] & s'' \end{tikzcd}$ *are two pull-backs in $\mathcal{I}$, then the square* $\begin{tikzcd} w \arrow[r, "i"] \arrow[d, "j' \circ j"'] & s \arrow[d, "l \circ l'"] \\ w'' \arrow[r, "i''"] & s'' \end{tikzcd}$ *is a pull-back as well.*

Proof. This is standard in category theory, but here we also make a set-theoretical proof.

By Lemma 5.1, we just need to show that

$$l'(l(i(w))) = i''(j'(j(w))) = l'(l(s)) \cap i''(w'').$$

Without causing confusion, we shall use (j, l) refer to the diagram defining the morphism (j, l), and similarly for (j', l') and $(j' \circ j, l' \circ l)$.

First, $l'(l(i(w))) = l'(i'(j(w))) = i''(j'(j(w)))$, according to the commuting squares (j, l) and (j', l'). Second, because the diagram (j, l) and (j', l') are pull-backs, by Lemma 5.1, $l(i(w)) = i'(j(w)) = l(s) \cap i'(w')$ and $l'(i'(w')) = i''(j'(w')) = l'(s) \cap i''(w'')$. Then $l'(l(i(w))) = l'(l(s) \cap i'(w')) = l'(l(s)) \cap l'(i'(w'))$, since l' is injective. Moreover, because $l(s) \subseteq s'$ ($l : s \to s'$ is an injection) and l' is injective, $l'(l(s)) \subseteq l'(s')$, hence $l'(l(s)) \cap l'(i'(w')) = l'(l(s)) \cap (l'(s) \cap i''(w'')) = l'(l(s)) \cap i''(w'')$. $\square$

Remark 5.1. There is another way to define a category such that the equation (5.1) holds. We take the same objects of $\mathcal{I}^{\rightarrow}$, i.e., morphisms in $\mathcal{I}$, and we take pairs $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ as morphisms such that the following diagram commutes in the category $\mathcal{P}fun$ (the category of sets and partial functions):

$$\begin{tikzcd} w \arrow[r, "i"] & s \\ w' \arrow[r, "i'"] \arrow[u, "j^{-1}"] & s' \arrow[u, "l^{-1}"'] \end{tikzcd} \tag{5.3}$$

where $j : w \to w', l : s \to s' \in \mathcal{I}$ and j^{-1}, l^{-1} stand for the reverse (partial) injection of j, l respectively. The identity of an object $\langle w, i, s\rangle$ is just $(\mathbf{id}_w, \mathbf{id}_s)$, and compositions of morphisms are also defined by compositions of partial functions.

The commuting square also implies the equation (5.1), i.e., $i'(j(w)) = l(i(w)) = l(s) \cap i'(w'')$. First, for any $k \in w$, $j(n)$ is in the domain of definition of j^{-1}, hence $i'(j(n))$ must also be in the domain of definition of l^{-1} and $l^{-1}(i'(j(k))) = i'(j^{-1}(j(k))) = i(k)$. Because l is injective, $l(i(k)) = l(l^{-1}(i'(j(k)))) = i'(j(k))$. Clearly, $l(i(k)) \in l(i(w)) \subseteq l(s)$ and $i'(j(k)) \in i'(j(w)) \subseteq i'(w')$. Second, if $k \in l(s) \cap i'(w')$, then k is in the domain of definition of l^{-1} and there exists some $k' \in w'$ such that $i'(k') = k$. Clearly, k' is in the domain of definition of $l^{-1} \circ i'$, so by the commuting square (5.3), it must be also in the domain of $i \circ j^{-1}$, consequently in the domain of j^{-1}. Thus there exists $k'' \in w$ such that $j(k'') = k'$, therefore, $k = i'(k') = i'(j(k'')) \in i'(j(w))$.

The category defined her is basically the same category as defined in Definition 5.1: the condition that the diagram (5.3) commutes is equivalent to the pull-back condition in Definition 5.1. We prefer the definition using pull-backs to the one using partial functions because pull-backs have certain nice properties. For instance, to check the composition of morphisms, it is standard that the composition of two pull-backs is still a pull-back, hence Lemma 5.2 is straightforward (although we also made a set-theoretical proof), but this is not the case if we the definition using partial functions. Furthermore, taking partial functions into consideration might also make our discussion more complicated, notably on the derivation of logical relations.

Clearly, the category $\mathcal{PI}^{\to}$ is a subcategory of $\mathcal{I}^{\to}$. More specifically, it is a subcategory of $\mathcal{I}^{\to}$, where we have the same collection of objects ($\mathbf{Obj}(\mathcal{PI}^{\to}) = \mathbf{Obj}(\mathcal{I}^{\to})$), but fewer morphisms.

Lemma 5.3. *For every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$, there exists some object $\langle w_0, i_0, s_0\rangle \in \mathcal{PI}^{\to}$ such that $\langle w', i', s'\rangle$ and $\langle w + w_0, i + i_0, s + s_0\rangle$ are isomorphic.*

Proof. Let $w_0 = w' - j(w)$, $s_0 = s' - l(s)$ and i_0 be i' limited to the domain w_0. Define $j' : w + w_0 \to w'$ by $j'(k) = j(k)$ for any $k \in w$ and $j'(k) = k$ for any $k \in w_0$ and define $l' : s + s_0 \to s'$ by $l'(k) = l(k)$ for any $k \in s$ and $l'(k) = k$ for any $k \in s_0$. Clearly, both j' and l' are bijective, and it is easy to check that the diagram

$$\begin{tikzcd} w + w_0 \arrow[r, "i+i_0"] \arrow[d, "j'"'] & s + s_0 \arrow[d, "l'"] \\ w' \arrow[r, "i'"] & s' \end{tikzcd}$$

commutes and is a pull-back in $\mathcal{I}$, hence (j', l') is an isomorphism in $\mathcal{PI}^{\to}$. $\square$

The category $\mathcal{PI}^{\rightarrow}$ also satisfies the "cube property".

Proposition 5.4 (Cube property for $\mathcal{PI}^{\rightarrow}$). *Suppose that $(j_1, l_1) : \langle w, i, s\rangle \to \langle w_1, i_1, s_1\rangle$ and $(j_2, l_2) : \langle w, i, s\rangle \to \langle w_2, i_2, s_2\rangle$ are two morphisms in $\mathcal{PI}^{\rightarrow}$. There exists $\langle w', i', s'\rangle \in \mathcal{PI}^{\rightarrow}$ and two morphisms $(j_1', l_1') : i_1 \to i'$ and $(j_2', l_2') : i_2 \to i'$ such that the following square commutes in $\mathcal{PI}^{\rightarrow}$:*

$$\begin{tikzcd}
 & i \arrow[dl, "(j_1,l_1)"'] \arrow[dr, "(j_2,l_2)"] & \\
i_1 \arrow[dr, "(j_1',l_1')"'] & & i_2 \arrow[dl, "(j_2',l_2')"] \\
 & i' &
\end{tikzcd} \tag{5.4}$$

Proof. According to Lemma 5.3, there exist objects $\langle w_0^1, i_0^1, s_0^1\rangle$ and $\langle w_0^2, i_0^2, s_0^2\rangle$ in $\mathcal{PI}^{\rightarrow}$ such that $i + i_0^1$ is isomorphic to i_1 and $i + i_0^2$ is isomorphic to i_2. Let $\langle w', i', s'\rangle$ be the object $\langle w + w_0^1 + w_0^2, i + i_0^1 + i_0^2, s + s_0^1 + s_0^2\rangle$. Consider the diagram

$$\begin{tikzcd}
 & i \arrow[dl, "(j_1,l_1)"'] \arrow[dr, "(j_2,l_2)"] & \\
i_1 \arrow[d, "(j_1^0,l_1^0)"'] & & i_2 \arrow[d, "(j_2^0,l_2^0)"] \\
i + i_0^1 \arrow[dr, "(\mathbf{inl},\mathbf{inl})"'] & & i + i_0^2 \arrow[dl, "(\mathbf{inl},\mathbf{inl})"] \\
 & i + i_0^1 + i_0^2 &
\end{tikzcd}$$

where (j_1^0, l_1^0) and (j_2^0, l_2^0) are isomorphisms such that $j_1^0 \circ j_1$, $l_1^0 \circ l_1$, $j_2^0 \circ j_2$, $l_2^0 \circ l_2$ are all inclusions. This diagram commutes since both paths are pairs of inclusions. $\square$

5.2 Dérivation des relations logiques sur $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$

To define logical relations for the cryptographic metalanguage, we switch from $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ to $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, the category of functors from $\mathcal{PI}^{\rightarrow}$ to $\mathcal{S}et$ and natural transformations. Necessary properties for the derivation must be checked.

First of all, $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ is Cartesian closed. Products and coproducts are still defined pointwise. For any two functors $A, B : \mathcal{PI}^{\rightarrow} \to \mathcal{S}et$, their exponent is defined by

$$B^A i = \mathcal{S}et^{\mathcal{PI}^{\rightarrow}}(\mathcal{PI}^{\rightarrow}(i, -) \times A, B)$$
$$(B^A(j, l)f)i''((j', l'), a) = fi''((j' \circ j, l' \circ l), a)$$

for any $\langle w, i, s\rangle \in \mathcal{PI}^{\rightarrow}$, $(j,l) : i \rightarrow i'$, $(j',l') : i' \rightarrow i'' \in \mathcal{PI}^{\rightarrow}$, $f \in B^A i$ and $a \in Ai$. $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ has pull-backs, taken pointwise.

Define a strong monad $(\boldsymbol{T}, \boldsymbol{\eta}, \boldsymbol{\mu}, \mathbf{t})$ on $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ by:

- $\boldsymbol{T}A = \mathbf{colim}_{i'}\, A(\_ + i') : \mathcal{PI}^{\rightarrow} \rightarrow \mathcal{S}et$. On objects, $\boldsymbol{T}Ai = \mathbf{colim}_{i'}\, A(i + i')$ is the set of equivalence classes of pairs (i', a), where $i' : w' \rightarrow s'$ in $\mathcal{I}$ and $a \in A(i + i')$, modulo the smallest equivalence relation $\sim$ such that $(i', a) \sim (i'', A(\mathbf{id}_i + (j,l))a)$ for each morphism $(j,l) : \langle w', i', s'\rangle \rightarrow \langle w'', i'', s''\rangle$ in $\mathcal{PI}^{\rightarrow}$. We write $[i, a]$ for the equivalence class of (i, a). On morphisms, $\boldsymbol{T}A(j,l)$ maps the equivalence class of (i', a) to the equivalence class of $(i', A((j,l) + \mathbf{id}_{i'})a)$;
- for any $f : A \rightarrow B$ in $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, $\boldsymbol{T}fi : \boldsymbol{T}Ai \rightarrow \boldsymbol{T}Bi$ is defined by $\boldsymbol{T}fi[i', a] = [i', f(i+i')a]$;
- $\boldsymbol{\eta}Ai : Ai \rightarrow \boldsymbol{T}Ai$ is defined by $\boldsymbol{\eta}Aia = [\emptyset, a]$, where $\emptyset$ denotes the empty function between empty sets;
- $\boldsymbol{\mu}Ai : \boldsymbol{T}^2Ai \rightarrow \boldsymbol{T}Ai$ is defined by $\boldsymbol{\mu}Ai[i', [i'', a]] = [i' + i'', a]$;
- $\mathbf{t}A, Bi : Ai \times \boldsymbol{T}Bi \rightarrow \boldsymbol{T}(A \times B)i$ is defined by $\mathbf{t}A, Bi(a, [i', b]) = [i', (Ai^{\rightarrow}_{i,i'}a, b)]$ where $i^{\rightarrow}_{i,i'} : i \rightarrow i + i'$ is the canonical injection.

Recall the forgetful functor $U : \mathcal{I}^{\rightarrow} \rightarrow \mathcal{I}$ mapping an object $\langle w, i, s\rangle$ to s and a morphism (j, l) to l. Clearly, this is also a functor from $\mathcal{PI}^{\rightarrow}$ to $\mathcal{I}$. Let $|\_| : \mathcal{S}et^{\mathcal{I}} \rightarrow \mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ be the functor $\mathbf{id}_{\mathcal{S}et}{}^U$. $|\_|$ preserves finite products. Define the monad morphism $\sigma : \boldsymbol{T}|\_| \rightarrow |T\_|$, where T is the strong monad over the category $\mathcal{S}et^{\mathcal{I}}$, by $\sigma_A i[\langle w', i', s'\rangle, a] = [s', a]$, for any object A in $\mathcal{S}et^{\mathcal{I}}$ and $\langle w, i, s\rangle \in \mathcal{PI}^{\rightarrow}$. Accordingly, define $\sigma_{(A_1,A_2)} : \boldsymbol{T}|A_1 \times A_2| \rightarrow |TA_1| \times |TA_2|$ by

$$\sigma_{(A_1,A_2)} = (\sigma_{A_1} \circ \boldsymbol{T}|\pi_1|, \sigma_{A_2} \circ \boldsymbol{T}|\pi_2|)$$

$\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ has a mono factorization system consisting of pointwise surjections and pointwise injections and it is clear that $\boldsymbol{T}$ and finite products preserve pointwise surjections. All these allow us to define a logical relation on $\mathrm{Subscone}^{\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}}_{\mathcal{S}et^{\mathcal{I}} \times \mathcal{S}et^{\mathcal{I}}}$.

The derivation of logical relations over $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ can be adapted here without much change. Consider $f : S \hookrightarrow |A_1 \times A_2|$ ($A_1, A_2 \in \mathcal{S}et^{\mathcal{I}}$ and $S \in \mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$) as a representation of a series of binary relations such that for every $\langle w, i, s\rangle \in \mathcal{PI}^{\rightarrow}$, $fi : Si \hookrightarrow |A_1 \times A_2|i = (A_1 \times A_2)s = A_1s \times A_2s$ is an inclusion, representing a binary relation. In particular, the relation between monadic values is the same as that derived from $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ since the two categories $\mathcal{I}^{\rightarrow}$ and $\mathcal{PI}^{\rightarrow}$ have the same objects:

$$\begin{aligned}
&[s_1, a_1]\ \widetilde{S}i\ [s_2, a_2] \iff \\
&\qquad \exists \langle w_0, i_0, s_0\rangle \in \mathcal{PI}^{\rightarrow}, l_1 : s_1 \rightarrow s_0 \in \mathcal{I}, l_2 : s_2 \rightarrow s_0 \in \mathcal{I}\cdot \\
&\qquad\qquad A_1(\mathbf{id}_s + l_1)(a_1)\ S(i + i_0)\ A_2(\mathbf{id}_s + l_2)(a_2).
\end{aligned} \tag{5.5}$$

The only difference is relations for function types:

$$\begin{aligned} &f_1 \ \mathcal{R}_{B^A}^{\langle w,i,s\rangle} \ f_2 \Longleftrightarrow \\ &\qquad \forall\langle j,l\rangle : \langle w,i,s\rangle \to \langle w',i',s'\rangle \in \mathcal{PI}^{\rightarrow} \cdot \ \forall a_1, a_2 \in As' \cdot \\ &\qquad\qquad (a_1 \ \mathcal{R}_A^{i'} \ a_2 \Rightarrow f_1 s'\langle l, a_1\rangle \ \mathcal{R}_B^{i'} \ f_2 s'\langle l, a_2\rangle). \end{aligned} \tag{5.6}$$

A function relation derived over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ quantifies over a relatively smaller collection of worlds as there are fewer morphisms in $\mathcal{PI}^{\rightarrow}$ than in $\mathcal{I}^{\rightarrow}$, hence we can relate more functions with this logical relation.

Lemma 5.5. *The full subcategory $\mathcal{PI}^{\subseteq}$ of $\mathcal{PI}^{\rightarrow}$ consisting only of inclusions, i.e., objects are inclusions ($\langle w, \subseteq, s\rangle$) of $\mathcal{I}$, is equivalent to the whole category $\mathcal{PI}^{\rightarrow}$.*

Proof. Let $F : \mathcal{PI}^{\subseteq} \to \mathcal{PI}^{\rightarrow}$ be the inclusion functor, and $G : \mathcal{PI}^{\rightarrow} \to \mathcal{PI}^{\subseteq}$ be the functor which maps $\langle w,i,s\rangle$ to $\langle i(w), \subseteq, s\rangle$, and $(j,l) : \langle w,i,s\rangle \to \langle w',i',s'\rangle$ to $(i' \circ j \circ i^{-1}, l) : \langle i(w), \subseteq, s\rangle \to \langle i'(w'), \subseteq, s'\rangle$. Then clearly, $G \circ F$ is the identity, and $F \circ G$ maps $\langle w,i,s\rangle$ to $\langle i(w), \subseteq, s\rangle$ which are isomorphic through $(i, \mathbf{id}_s)$ and $(i^{-1} \restriction_{i(w)}, \mathbf{id}_s)$. So (F, G) is an equivalence of categories. ☐

We shall write $\langle w, s\rangle$ for $\langle w, \subseteq, s\rangle$ and (l,l) for the morphism from $\langle w,s\rangle$ to $\langle w',s'\rangle$ in $\mathcal{PI}^{\subseteq}$, where l is an injection from s to s', and the first l of (l,l) actually denotes $l \restriction_w$ with the codomain w'.

Lemma 5.6. *The category $\mathcal{S}et^{\mathcal{PI}^{\subseteq}}$ is equivalent to the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$.*

Proof. Similarly as in the proof of Lemma 5.5, we let $F' : \mathcal{S}et^{\mathcal{PI}^{\subseteq}} \to \mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ be the functor such that

$$\forall A \in \mathcal{S}et^{\mathcal{PI}^{\subseteq}}, \forall\langle w,i,s\rangle \in \mathcal{PI}^{\rightarrow}, \ F'(A)(\langle w,i,s\rangle) = A(\langle i(w), s\rangle),$$

and $G' : \mathcal{S}et^{\mathcal{PI}^{\rightarrow}} \to \mathcal{S}et^{\mathcal{PI}^{\subseteq}}$ be the functor such that

$$\forall A \in \mathcal{S}et^{\mathcal{PI}^{\rightarrow}}, \forall\langle w,s\rangle \in \mathcal{PI}^{\subseteq}, \ G'(A)(\langle w,s\rangle) = A(\langle w, \subseteq, s\rangle).$$

Clearly, $G' \circ F'$ is the identity, and $F' \circ G'$ maps every functor $A \in \mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ to another functor $A' \in \mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ such that

$$\forall\langle w,i,s\rangle \in \mathcal{PI}^{\rightarrow}, A'(< w,i,s >) = A(i(w), \subseteq, s),$$

and these two functors are isomorphic through $A(i, \mathbf{id}_s)$ and $A(i^{-1} \restriction_{i(w)}, \mathbf{id}_s)$. So (F', G') is an equivalence of categories. ☐

If we switch from the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ to the equivalent category $\mathcal{S}et^{\mathcal{PI}^{\subseteq}}$, the definition of $\widetilde{S}i$ (5.5) is then equivalent to

$$\begin{aligned}
&[s_1, a_1]\ \widetilde{S}\langle w, s\rangle\ [s_2, a_2] \Longleftrightarrow \\
&\qquad \exists \langle w_0, s_0\rangle \in \mathcal{PI}^{\subseteq}, l_1 : s_1 \to s_0 \in \mathcal{I}, l_2 : s_2 \to s_0 \in \mathcal{I}\cdot \\
&\qquad\qquad A_1(\mathbf{id}_s + l_1)(a_1)\ S\langle w + w_0, s + s_0\rangle\ A_2(\mathbf{id}_s + l_2)(a_2).
\end{aligned}$$

The function relation (5.6) is equivalent to

$$\begin{aligned}
&f_1\ \mathcal{R}_{BA}^{\langle w,s\rangle}\ f_2 \Longleftrightarrow \\
&\qquad \forall \langle w_0, s_0\rangle \in \mathcal{PI}^{\subseteq}\cdot\ \forall a_1, a_2 \in A(s + s_0)\cdot \\
&\qquad\qquad (a_1\ \mathcal{R}_A^{\langle w+w_0, s+s_0\rangle}\ a_2 \Rightarrow \\
&\qquad\qquad\qquad f_1 s'\langle \mathbf{inl}_{s,s_0}, a_1\rangle\ \mathcal{R}_B^{\langle w+w_0, s+s_0\rangle}\ f_2 s'\langle \mathbf{inl}_{s,s_0}, a_2\rangle).
\end{aligned}$$

Since logical relations derived over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ are Kripke logical relations, they must satisfy the monotonicity property so that the Basic Lemma will hold. The following propositions shows that logical relations derived over $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ are monotonic if relations for base types are monotonic.

Proposition 5.7 (Monotonicity). *Suppose that $(\mathcal{R}_\tau)_{\tau \text{ type}}$ is a logical relation derived from the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. If $\mathcal{R}_b$ is monotonic for every base type b, then $\mathcal{R}_\tau$ is monotonic for every type τ, in the sense that for every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle \in \mathcal{PI}^{\rightarrow}$ and every pair of values $a_1, a_2 \in [\![\tau]\!]s$,*

$$a_1\ \mathcal{R}_\tau^{\langle w,i,s\rangle}\ a_2 \Longrightarrow [\![\tau]\!]l(a_1)\ \mathcal{R}_\tau^{\langle w',i',s'\rangle}\ [\![\tau]\!]l(a_2).$$

Proof. Similar as the proof of Lemma 4.5, we prove the monotonicity by induction on types. We do not detail the induction steps, which are almost the same as for Lemma 4.5. ☐

5.3 Relations logiques cryptographiques

We have been very careful in defining the cipher function and the cryptographic message relation according to injections in $\mathcal{I}$, not objects in $\mathcal{I}^{\rightarrow}$, although $\mathcal{I}^{\rightarrow}$ and $\mathcal{PI}^{\rightarrow}$ have exactly the same collection of objects. Definitions based on injections of $\mathcal{I}$ allow us to reuse them directly to define a logical relation over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$.

Definition 5.2 (Cryptographic logical relation). *Suppose that $\langle w, i, s\rangle$ is an object in $\mathcal{PI}^{\rightarrow}$ and φ is a cipher function. The relations $\boldsymbol{\mathcal{R}}_\tau^{\langle w,i,s\rangle,\varphi} \subseteq [\![\tau]\!]s \times [\![\tau]\!]s$ ($\boldsymbol{\mathcal{R}}_\tau^{i,\varphi}$ for short) are defined by* *induction over the structure of type τ, as follows:*

$$
\begin{aligned}
& b_1 \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\mathsf{bool}}\, b_2 \iff b_1 = b_2, \\
& n_1 \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\mathsf{nat}}\, n_2 \iff n_1 = n_2, \\
& k_1 \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\mathsf{key}}\, k_2 \iff k_1 = k_2 \in i(w), \\
& m_1 \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\mathsf{msg}}\, m_2 \iff (m_1, m_2) \in \mathcal{MR}^{i,\varphi}, \\
& (a_1, a_1') \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\tau\times\tau'}\, (a_2, a_2') \iff a_1 \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\tau}\, a_2 \;\; \& \;\; a_1' \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\tau'}\, a_2', \\
& a_1 \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\mathsf{opt}[\tau]}\, a_2 \iff a_1 \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\tau}\, a_2 \textit{ or } a_1 = a_2 = \bot, \\
& f_1 \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\tau\to\tau'}\, f_2 \iff \\
& \qquad\qquad \forall (j,l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle \in \mathcal{PI}^{\to} \cdot \; \forall a_1, a_2 \in [\![\tau]\!] s' \cdot \\
& \qquad\qquad\qquad (a_1 \,\boldsymbol{\mathcal{R}}^{i',\varphi}_{\tau}\, a_2 \Rightarrow f_1 s'(l, a_1) \,\boldsymbol{\mathcal{R}}^{i',\varphi}_{\tau'}\, f_2 s'(l, a_2)), \\
& [s_1, a_1] \,\boldsymbol{\mathcal{R}}^{i,\varphi}_{\mathsf{T}\tau}\, [s_2, a_2] \iff \\
& \qquad\qquad \exists \langle w_0, i_0, s_0\rangle \in \mathcal{PI}^{\to}, l_1 : s_1 \to s_0, l_2 : s_2 \to s_0 \in \mathcal{I} \cdot \\
& \qquad\qquad\qquad [\![\tau]\!](\mathbf{id}_s + l_1)(a_1) \,\boldsymbol{\mathcal{R}}^{i+i_0,\varphi}_{\tau}\, [\![\tau]\!](\mathbf{id}_s + l_2)(a_2),
\end{aligned}
$$

where $\mathcal{MR}^{i,\varphi}$ is the cryptographic message relation of Definition 4.2.

A cipher function φ is *monotonic* if for any morphism $(j,l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$, and for every $m_1, m_2 \in [\![\mathsf{msg}]\!] s$,

$$(m_1, m_2) \in \varphi^i(k_1, k_2) \Longrightarrow ([\![\mathsf{msg}]\!] l(m_1), [\![\mathsf{msg}]\!] l(m_2)) \in \varphi^{i'}(l(k_1), l(k_2)).$$

This is well-defined because the category $\mathcal{PI}^{\to}$ guarantees that if $k_1, k_2 \notin w$, then $l(k_1), l(k_2) \notin w'$ either. Furthermore, a cipher function φ is said to be *consistent* if

$$(m_1, m_2) \in \varphi^i(k_1, k_2) \iff ([\![\mathsf{msg}]\!] l(m_1), [\![\mathsf{msg}]\!] l(m_2)) \in \varphi^{i'}(l(k_1), l(k_2)).$$

Again we must check the Basic Lemma. Recall that this is meant to check the condition of monotonicity and the one that every constant is related to itself. Once we get these two conditions satisfied, the categorical construction automatically guarantees that the Basic Lemma necessarily holds. According to Proposition 5.7, logical relations derived over the category $\mathcal{S}et^{\mathcal{PI}^{\to}}$ are monotonic if all relations for base types are monotonic. It is easy to check that relations for types nat, bool and key are monotonic. The following lemma shows that the relation for messages are monotonic if the cipher function φ is monotonic.

Lemma 5.8. *If the cipher function φ is monotonic, then for any morphism $(j,l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle \in \mathcal{PI}^{\to}$ and for any pair of values $m_1, m_2 \in [\![\mathsf{msg}]\!] s$,*

$$(m_1, m_2) \in \mathcal{MR}^{i,\varphi} \Longrightarrow ([\![\mathsf{msg}]\!] l(m_1), [\![\mathsf{msg}]\!] l(m_2)) \in \mathcal{MR}^{i',\varphi}.$$

Proof. We prove by induction on the message structure. In particular, when $m_1 = \mathsf{e}(m_1', k_1)$ and $m_2 = \mathsf{e}(m_2', k_2)$,

- either $k_1 = k_2 \in w$ and $(m_1', m_2') \in \mathcal{MR}^{i,\varphi}$, then $l(k_1) = l(k_2) \in i'(w')$. By induction, $([\![\mathsf{msg}]\!]l(m_1'), [\![\mathsf{msg}]\!]l(m_2')) \in \mathcal{MR}^{i',\varphi}$, hence
$$(\mathsf{e}([\![\mathsf{msg}]\!]l(m_1'), l(k_1)), \mathsf{e}([\![\mathsf{msg}]\!]l(m_2'), l(k_2))) \in \mathcal{MR}^{i',\varphi},$$
- or $k_1, k_2 \notin w$ and $(m_1', m_2') \in \varphi^i(k_1, k_2)$, then according to the definition of $\mathcal{PI}^{\rightarrow}$, $l(k_1), l(k_2) \notin i'(w')$. Because φ is monotonic,
$$([\![\mathsf{msg}]\!]l(m_1'), [\![\mathsf{msg}]\!]l(m_2')) \in \varphi^{i'}(l(k_1), l(k_2)),$$
hence $(\mathsf{e}([\![\mathsf{msg}]\!]l(m_1'), l(k_1)), \mathsf{e}([\![\mathsf{msg}]\!]l(m_2'), l(k_2))) \in \mathcal{MR}^{i',\varphi}$. □

Proposition 5.9 (Monotonicity). *The cryptographic logical relation $\boldsymbol{\mathcal{R}}_\tau^{\langle w,i,s\rangle,\varphi}$ is monotonic for any monotonic cipher functions φ, in the sense that for every morphism $(j, l) : \langle w, i, s\rangle \rightarrow \langle w', i', s'\rangle \in \mathcal{PI}^{\rightarrow}$ and every pair of values $a_1, a_2 \in [\![\tau]\!]s$,*
$$a_1 \, \boldsymbol{\mathcal{R}}_\tau^{\langle w,i,s\rangle,\varphi} \, a_2 \Longrightarrow [\![\tau]\!]l(a_1) \, \boldsymbol{\mathcal{R}}_\tau^{\langle w',i',s'\rangle,\varphi} \, [\![\tau]\!]l(a_2).$$

Proof. Clearly, in the cryptographic logical relation, relations for every base type are monotonic. In particular, Lemma 5.8 shows that $\boldsymbol{\mathcal{R}}_{\mathsf{msg}}^{i,\varphi}$ is monotonic when the cipher function φ is monotonic. Since the cryptographic logical relation is derived over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, according to Proposition 5.7, it is monotonic for every type τ. □

Lemma 5.10. *For any monotonic cipher function φ and for any morphism $(j, l) : \langle w, i, s\rangle \rightarrow \langle w', i', s'\rangle \in \mathcal{PI}^{\rightarrow}$, if $\rho_1 \, \boldsymbol{\mathcal{R}}_\Gamma^{i,\varphi} \, \rho_2$, then $[\![\Gamma]\!]l(\rho_1) \, \boldsymbol{\mathcal{R}}_\Gamma^{i',\varphi} \, [\![\Gamma]\!]l(\rho_2)$.*

Proof. This is a corollary of Proposition 5.9. □

The Basic Lemma of the cryptographic logical relation holds for a non-trivial collection of cipher functions, namely for all monotonic cipher functions.

Proposition 5.11 (Basic Lemma for the cryptographic logical relations). *Suppose that $\Gamma \vdash t : \tau$ is a well-typed term and φ is a monotonic cipher function. For every $\langle w, i, s\rangle \in \mathcal{PI}^{\rightarrow}$ and every pair of environments $\rho_1, \rho_2 \in [\![\Gamma]\!]s$ such that $\rho_1 \, \mathcal{R}_\Gamma^{i,\varphi} \, \rho_2$, $[\![t]\!]s\rho_1 \, \boldsymbol{\mathcal{R}}_\tau^{i,\varphi} \, [\![t]\!]s\rho_2$.*

Proof. It is proved by induction on the structure of term t, which includes proving that every constant is related to itself.. We do not detail the induction steps, which are quite similar as in the proof of Proposition 4.7. □

If we consider the equivalent subcategory $\mathcal{S}et^{\mathcal{PI}^{\subseteq}}$ of $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, Definition 5.2 is actually equivalent to the following one:

Definition 5.3. *Let w and s be two sets in $\mathcal{I}$ and $w \subseteq s$. φ is a cipher function. The relations $\boldsymbol{\mathcal{R}}_{\tau}^{\langle w,s\rangle,\varphi} \subseteq [\![\tau]\!]s \times [\![\tau]\!]s$ are defined by induction over the structure of type τ, as follows:*

$$
\begin{aligned}
& b_1 \, \boldsymbol{\mathcal{R}}_{\mathsf{bool}}^{\langle w,s\rangle,\varphi} \, b_2 \iff b_1 = b_2, \\
& n_1 \, \boldsymbol{\mathcal{R}}_{\mathsf{nat}}^{\langle w,s\rangle,\varphi} \, n_2 \iff n_1 = n_2, \\
& k_1 \, \boldsymbol{\mathcal{R}}_{\mathsf{key}}^{\langle w,s\rangle,\varphi} \, k_2 \iff k_1 = k_2 \in w, \\
& m_1 \, \boldsymbol{\mathcal{R}}_{\mathsf{msg}}^{\langle w,s\rangle,\varphi} \, m_2 \iff (m_1, m_2) \in \mathcal{MR}^{\langle w,s\rangle,\varphi}, \\
& (a_1, a_1') \, \boldsymbol{\mathcal{R}}_{\tau\times\tau'}^{\langle w,s\rangle,\varphi} \, (a_2, a_2') \iff a_1 \, \boldsymbol{\mathcal{R}}_{\tau}^{\langle w,s\rangle,\varphi} \, a_2 \;\; \& \;\; a_1' \, \boldsymbol{\mathcal{R}}_{\tau'}^{\langle w,s\rangle,\varphi} \, a_2', \\
& a_1 \, \boldsymbol{\mathcal{R}}_{\mathsf{opt}[\tau]}^{\langle w,s\rangle,\varphi} \, a_2 \iff a_1 \, \boldsymbol{\mathcal{R}}_{\tau}^{\langle w,s\rangle,\varphi} \, a_2 \textit{ or } a_1' = a_2' = \bot, \\
& f_1 \, \boldsymbol{\mathcal{R}}_{\tau\rightarrow\tau'}^{\langle w,s\rangle,\varphi} \, f_2 \iff \\
& \qquad\qquad \forall w', s' \in \mathcal{I} \textit{ s.t. } w' \subseteq s' \cdot \; \forall a_1, a_2 \in [\![\tau]\!](s+s') \cdot \\
& \qquad\qquad\quad (a_1 \, \boldsymbol{\mathcal{R}}_{\tau}^{\langle w+w',s+s'\rangle,\varphi} \, a_2 \Rightarrow \\
& \qquad\qquad\qquad f_1(s+s')(\mathbf{inl}_{s,s'}, a_1) \, \boldsymbol{\mathcal{R}}_{\tau'}^{\langle w+w',s+s'\rangle,\varphi} \, f_2(s+s')(\mathbf{inl}_{s,s'}, a_2)), \\
& [s_1, a_1] \, \boldsymbol{\mathcal{R}}_{\mathsf{T}\tau}^{\langle w,s\rangle,\varphi} \, [s_2, a_2] \iff \\
& \qquad\qquad \exists w_0, s_0 \in \mathcal{I} \textit{ s.t. } w_0 \subseteq s_0 \cdot \exists l_1 : s_1 \rightarrow s_0 \in \mathcal{I}, l_2 : s_2 \rightarrow s_0 \in \mathcal{I} \cdot \\
& \qquad\qquad\quad [\![\tau]\!](\mathbf{id}_s + l_1)(a_1) \, \boldsymbol{\mathcal{R}}_{\tau}^{\langle w+w_0,s+s_0\rangle,\varphi} \, [\![\tau]\!](\mathbf{id}_s + l_2)(a_2),
\end{aligned}
$$

where $\mathcal{MR}^{\langle w,s\rangle,\varphi}$ is a cryptographic message relation.

This definition takes as parameters a pair of sets and simply requires w to be a subset of s, i.e., the set of disclosed keys is a subset of those keys that have been created at that "world", so it is much closer to the intuition and we shall use this logical relations to check the relation between concrete programs.

5.4 Vérification des protocoles à l'aide de relations logiques

The point of checking whether two concrete programs are related is to distinguish between disclosed keys and secret keys and to find a proper cipher function. Consider the counterexample at the beginning of this chapter:

$$
\begin{aligned}
p_1 &= \nu k.\langle \{0\}_k, \; \{1\}_k, \; \lambda x.\mathtt{getnum}(\mathtt{dec}(x,k))\rangle, \\
p_2 &= \nu k.\langle \{1\}_k, \; \{0\}_k, \; \lambda x.\mathtt{letopt}\; y \Leftarrow \mathtt{getnum}(\mathtt{dec}(x,k)) \;\mathtt{in}\; \mathtt{some}(1-y)\rangle,
\end{aligned}
$$

with the denotations

$$\begin{array}{rcl}[\![p_1]\!]s & = & [\{k\}, (\{0\}_k,\ \{1\}_k,\ f_1)] \\ [\![p_2]\!]s & = & [\{k\}, (\{1\}_k,\ \{0\}_k,\ f_2)].\end{array}$$

We are now able to relate these two programs with the cryptographic logical relation. Without loss of generality, we can just start with a world $\langle\emptyset, \emptyset\rangle$. Clearly, the fresh key k is secret, i.e., $k \notin w$, then we need to relate the two tuples at the world $\langle\emptyset, \{k\}\rangle$. For this, we define the cipher function at this world as

$$\varphi^{\langle\emptyset,\{k\}\rangle}(k, k) = \{(0, 1), (1, 0)\}.$$

Note that in order to keep the soundness of the logical relation, cipher-functions must be monotonic, i.e., for every larger world $\langle w, s\rangle$ such that $k \notin w$ and $k \in s$, $(0, 1)$ and $(1, 0)$ must be included in $\varphi^{\langle w,s\rangle}(k, k)$. Then the cipher-texts in the two programs are related because of this cipher function. The two functions f_1 and f_2 are related as well because they expect cipher-texts as arguments, but we are only allowed to apply them to related cipher-texts. In this case, we shall always get the decryption error ($\perp$) unless we apply these two functions to secret cipher-texts — $(\{1\}_k, \{0\}_k), (\{0\}_k, \{1\}_k)$ — given by the cipher-function, where we get related results.

This section shows that the cryptographic logical relation can be used to relate the different instances of the protocols in Chapter 2, by carefully choosing the set of secret keys and the cipher-function φ.

5.4.1 Le protocole de l'échange de clés symétriques

Recall the encoding of the (fixed) symmetric key establishment protocol:

$$P(m) \equiv \nu(k_{as}, k_{bs}, k_{es}, k_{ab}).\langle k_{es}, f_a, f_s\rangle,$$

where

$$\begin{array}{rcl} f_a & \equiv & \langle [A, B, \{B, k_{ab}\}_{k_{as}}], \{m\}_{k_{ab}}\rangle \\ f_s & \equiv & \lambda x.\mathtt{letopt}\ x_a \Leftarrow \mathtt{getnum}(\pi^3_1(x))\ \mathtt{in} \\ & & \quad \mathtt{letopt}\ x_b \Leftarrow \mathtt{getnum}(\pi^3_2(x))\ \mathtt{in} \\ & & \quad \mathtt{letopt}\ y \Leftarrow \mathtt{dec}(\pi^3_3(x), \mathtt{K}(x_a, s))\ \mathtt{in} \\ & & \quad \mathtt{letopt}\ x'_b \Leftarrow \mathtt{getnum}(\mathtt{fst}(y))\ \mathtt{in} \\ & & \quad \mathtt{if}\ x_b = x'_b\ \mathtt{then}\ \mathtt{some}([\mathtt{n}(x_a), \mathtt{n}(x_b), \{\mathtt{snd}(y)\}_{\mathtt{K}(x_b,s)}]) \\ & & \qquad\qquad \mathtt{else}\ \mathtt{error}\end{array}$$

Basically, f_s expects messages of the form $[X, Y, \{Y, Z\}_{k_x s}]$ and will output messages $[X, Y, \{Y, Z\}_{k_y s}]$. If the argument message is of a wrong format, f_s always returns an error ($\perp$).

Proposition 5.12. *For any two different messages $m_1 \neq m_2$, there exists a monotonic cipher-function φ such that $[\![P(m_1)]\!] \, \boldsymbol{\mathcal{R}}_\tau^{\langle\emptyset,\emptyset\rangle,\varphi} \, [\![P(m_2)]\!]$.*

Proof. Consider the denotation of the protocol program

$$[\{k_{as}, k_{bs}, k_{es}, k_{ab}\}, \langle k_{es}, [A, B, \{B, k_{ab}\}_{k_{as}}], \{m\}_{k_{ab}}, f\rangle],$$

where f is the denotation of f_s defined over $\{k_{as}, k_{bs}, k_{es}, k_{ab}\}$. In order to relate the two instances where $m = m_1$ and $m = m_2$, we must select carefully the set w of disclosed keys such that the tuples are related.

Relating tuples is simply relating components. Relating the first components $k_e s$ forces us to put k_{es} in w. Relating the third components, we must not put k_{ab} into w, since m_1 and m_2 are different. This accordingly requires that the key k_{as} should be secret, because in the second component, it is used to encrypt the secret key k_{ab}. We also let k_{bs} be a secret key since it is not disclosed in the program. Let $w = \{k_{es}\}$ and $s = \{k_{as}, k_{bs}, k_{es}, k_{ab}\}$. Then at the world $\langle w, s\rangle$, the first three components of the tuple are related, with the cipher function defined as (first try):

$$\begin{aligned}
\varphi^{\langle w,s\rangle}(k_{ab}, k_{ab}) &= \{(m_1, m_2)\},\\
\varphi^{\langle w,s\rangle}(k_{as}, k_{as}) &= \{([B, k_{ab}], [B, k_{ab}])\},\\
\varphi^{\langle w,s\rangle}(k_{bs}, k_{bs}) &= \emptyset.
\end{aligned}$$

We still need to check whether the function f is related to itself. Function f returns meaningful results (non-error) only when it is applied to messages of expected format. In this program, the possible messages that contexts can build and the corresponding responses from f are:

$$\begin{aligned}
[A, B, \{B, k_{ab}\}_{k_{as}}] &\mapsto [A, B, \{B, k_{ab}\}_{k_{bs}}],\\
[E, A, \{A, k\}_{k_{es}}] &\mapsto [E, A, \{A, k\}_{k_{as}}],\\
[E, B, \{B, k\}_{k_{es}}] &\mapsto [E, B, \{B, k\}_{k_{bs}}].
\end{aligned}$$

where k is either k_{es} or some fresh key not in s. We should then revise the cipher function so that this function is related to itself:

$$\begin{aligned}
\varphi^{\langle w',s'\rangle}(k_{ab}, k_{ab}) &= \{(m_1, m_2)\}\\
\varphi^{\langle w',s'\rangle}(k_{as}, k_{as}) &= \{([B, k_{ab}], [B, k_{ab}]), ([A, k], [A, k])\}\\
\varphi^{\langle w',s'\rangle}(k_{bs}, k_{bs}) &= \{([B, k], [B, k])\},
\end{aligned}$$

where $w' = w + w_0$ and $s' = s + s_0$ for any $\langle w_0, s_0 \rangle \in \mathcal{PI}^{\subseteq}$, and $k \in w'$. It is clear that this cipher function is monotonic. Moreover, with this cipher function, other components of the protocol program are still related. $\square$

5.4.2 Le protocole de Needham-Schroeder-Lowe

In the cryptographic metalanguage, G. Lowe's fixed version of the Needham-Schroeder public key protocol is encoded as

$$NS(m) \equiv \nu(k_a, k_b, k_e).\langle \lambda x.\{x\}_{k_a}, \lambda x.\{x\}_{k_b}, k_e, \langle \mathtt{n}(A), f_a \rangle, f_b \rangle,$$

where

$$\begin{array}{rcl} f_a & \equiv & \lambda\{x\}_{k_a}.\mathtt{letopt}\ x' \Leftarrow \mathtt{getnum}(\mathtt{snd}(x))\ \mathtt{in} \\ & & \qquad\qquad \mathtt{some}(\nu(N_a).\langle \{[\mathtt{k}(N_a), \mathtt{fst}(x), A]\}_{\mathtt{K}_a(x')},\ f'_a \rangle) \\ f'_a & \equiv & \lambda\{x''\}_{k_a}.\mathtt{letopt}\ x''' \Leftarrow \mathtt{getkey}(x'')\ \mathtt{in} \\ & & \qquad\qquad \mathtt{if}\ x''' = N_a\ \mathtt{then}\ \mathtt{some}(\{i\}_{N_a})\ \mathtt{else}\ \mathtt{error} \\ f_b & \equiv & \lambda y.\mathtt{letopt}\ y' \Leftarrow \mathtt{getnum}(y)\ \mathtt{in} \\ & & \qquad\qquad \mathtt{some}(\nu(N_b).\langle \{[\mathtt{k}(N_b), \mathtt{n}(B)]\}_{\mathtt{K}_b(y')},\ f'_b \rangle) \\ f'_b & \equiv & \lambda\{y''\}_{k_b}.\mathtt{letopt}\ y''_2 \Leftarrow \mathtt{getkey}(\pi^3_2(y''))\ \mathtt{in} \\ & & \qquad\qquad \mathtt{letopt}\ y''_3 \Leftarrow \mathtt{getnum}(\pi^3_3(y''))\ \mathtt{in} \\ & & \qquad\qquad \mathtt{if}\ (y''_2 = N_b\ \mathtt{and}\ y' = y''_3) \\ & & \qquad\qquad\quad \mathtt{then}\ \mathtt{some}(\{\pi^3_1(y'')\}_{\mathtt{K}_b(y')})\ \mathtt{else}\ \mathtt{error}. \end{array}$$

Basically, f_a expects messages $\{N_b, B\}_{k_a}$ and returns messages $\{N_a, N_x, A\}_{k_x}$, where N_a is a fresh nonce generated by f_a; f'_a then expects exactly the message $\{N_a\}_{k_a}$ and returns the secret messages encrypted with N_a; f_b expects a principle identity X and returns $\{N_b, B\}_{k_x}$ where N_b is a fresh nonce generated by f_b; then f'_b expects messages $\{N_x, N_b, X\}_{k_b}$ and returns messages $\{N_x\}_{k_x}$. In particular, f'_b checks whether the principle identity X it receives is the same one as f_b receives.

Proposition 5.13. *For any two different messages $m_1 \neq m_2$, there exists a monotonic cipher-function φ such that $[\![NS(m_1)]\!]\ \boldsymbol{\mathcal{R}}^{\langle\emptyset,\emptyset\rangle,\varphi}_{\tau}\ [\![NS(m_2)]\!]$.*

Proof. Consider the denotation of the protocol program

$$[\{k_a, k_b, k_e\}, \langle pk_a, pk_b, k_e, \langle A, f_1 \rangle, f_2 \rangle],$$

where f_1 and f_2 are denotations of f_a and f_b defined over the set $\{k_a, k_b, k_e\}$. Clearly, keys k_a and k_b should be considered as secret keys, and k_e must be a disclosed key so that it is related to itself. Let $w = \{k_e\}$ and $s = \{k_a, k_b, k_e\}$. Then we must relate the two public key functions pk_a and pk_b, and the two principle functions f_1 and f_2, at the world $\langle w, s\rangle$.

To relate functions pk_a and pk_b with themselves, the cipher function should relate every related messages encrypted by k_a or by k_b:

$$\begin{array}{rcl}\varphi^{\langle w',s'\rangle}(k_a, k_a) & = & \{(m_1, m_2) \mid (m_1, m_2) \in \mathcal{MR}^{\langle w',s'\rangle,\varphi}\} \\ \varphi^{\langle w',s'\rangle}(k_b, k_b) & = & \{(m_1, m_2) \mid (m_1, m_2) \in \mathcal{MR}^{\langle w',s'\rangle,\varphi}\}\end{array}$$

where $w' = w + w_0$ and $s' = s + s_0$ for any $\langle w_0, s_0\rangle \in \mathcal{PI}^{\subseteq}$. While $\mathcal{MR}$ is defined according to φ, this cipher function is indeed recursively defined.

The function f_1 is a mapping

$$\{N, B\}_{k_a} \mapsto [\{N_a\}, \langle\{N_a, N, A\}_{k_b}, f_1'\rangle],$$

where f_1' is a mapping

$$\{N_a\}_{k_a} \mapsto \{m\}_{N_a}.$$

Clearly, N_a should be secret, otherwise f_1' is not related to itself when m is replaced by two different messages m_1 and m_2. Furthermore, $\varphi(N_a, N_a)$ should contains (m_1, m_2). To relate the function f_1, $\varphi(k_b, k_b)$ should contain $([N_a, N, A], [N_a, N, A])$, for any key N.

The function f_2 maps a principle identity to an encrypted message with the secret key of the received identity. Consider two possible arguments:

$$\begin{array}{rcl}A & \mapsto & [\{N_b\}, \langle\{N_b, B\}_{k_a}, f_2'\rangle] \\ E & \mapsto & [\{N_b\}, \langle\{N_b, B\}_{k_e}, f_2'\rangle],\end{array}$$

where f_2' is a mapping

$$\begin{array}{rcll}\{N', N_b, A\}_{k_b} & \mapsto & \{N'\}_{k_a} & \text{when } f_1 \text{ is applied to } A \\ \{N', N_b, E\}_{k_b} & \mapsto & \{N'\}_{k_e} & \text{when } f_1 \text{ is applied to } E.\end{array}$$

N_b is not secret, otherwise the function f_2 will return unrelated results when it is applied to E. Because N_a is secret, the only message of the form $\{N', N_b, A\}_{k_b}$ that could be applied to f_2' is $\{N_a, N_b, A\}_{k_b}$, i.e., $N' = N_a$, so $\varphi(k_a, k_a)$ must contain (N_a, N_a) in order to relate f_2'. If f_2 is applied to E, then N' must be a disclosed key according to $\varphi(k_b, k_b)$, so f_2' is always related to itself.

To summarize, the different instances of the protocol are related in the cryptographic logical relation, with the cipher function

$$\begin{array}{rcl}
\varphi^{\langle w',s'\rangle}(k_a,k_a) & = & \{(m_1,m_2) \mid (m_1,m_2) \in \mathcal{MR}^{\langle w',s'\rangle,\varphi}\} \\
\varphi^{\langle w',s'\rangle}(k_b,k_b) & = & \{(m_1,m_2) \mid (m_1,m_2) \in \mathcal{MR}^{\langle w',s'\rangle,\varphi}\} \\
\varphi^{\langle w'',s''\rangle}(k_a,k_a) & = & \{(m_1,m_2) \mid (m_1,m_2) \in \mathcal{MR}^{\langle w'',s''\rangle,\varphi}\} \\
& & \cup \{(N_a,N_a)\} \\
\varphi^{\langle w'',s''\rangle}(k_b,k_b) & = & \{(m_1,m_2) \mid (m_1,m_2) \in \mathcal{MR}^{\langle w'',s''\rangle,\varphi}\} \\
& & \cup \{([N_a,N_b,A],[N_a,N_b,A])\} \\
\varphi^{\langle w'',s''\rangle}(N_a,N_a) & = & \{(m_1,m_2)\}
\end{array}$$

where $w' = \{k_e\} + w_0$, $s' = \{k_a,k_b,k_e\} + s_0$, $w'' = \{k_e,N_b\} + w_0$ and $s'' = \{k_a,k_b,k_e,N_a,N_b\} + s_0$ for any $\langle w_0,s_0\rangle \in \mathcal{PI}^{\subseteq}$. □

5.5 Comparaisons avec les relations logiques du nu-calcul

In the nu-calculus, Pitts and Stark proposed an *operational logical relation* [1] for reasoning about the contextual equivalence [PS93a]. The operational logical relation is defined over the syntax of the nu-calculus and relies largely on the operational semantics. Stark later rebuilt the categorical model $\mathcal{S}et^{\mathcal{I}}$ using the machinery of *categories with relations* and defined the category $\mathcal{P}$ [Sta94]. This category gives a denotational semantics for the nu-calculus which directly validates most contextually equivalent programs. In particular, it allows us to derive logical relations for the nu-calculus, which are proved to be equivalent to the operational logical relation for types up to second order. We shall show in this section that logical relations defined over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ are indeed equivalent to those derived from Stark's category $\mathcal{P}$.

A category with relations is a category with a collection of binary *relations* between pairs of objects, represented $R : A \leftrightarrow B$, and *parametric squares* of the form

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, leftrightarrow, "R"'] & A' \arrow[d, leftrightarrow, "R'"] \\ B \arrow[r, "g"] & B' \end{tikzcd},$$

where R, R' are relations and f, g are morphisms. Relations, like morphisms, are simply abstract data. We rebuild the model $\mathcal{S}et^{\mathcal{I}}$ using the machinery of categories with relations, by equipping both the index category $\mathcal{I}$ and the base category $\mathcal{S}et$ with relations. For the index category $\mathcal{I}$, a

[1] In [ZN03], we asserted wrongly that logical relations derived over $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ can identify Pitts and Stark's operational logical relation.

relation $R : s_1 \leftrightarrow s_2$ on $\mathcal{I}$ consists of a finite set R and a pair of injections $s_1 \leftarrowtail R \rightarrowtail s_2$. Such a relation is also called a *span* in the operational logical relation. The operation '+' on $\mathcal{I}$ extends to relations: if $R : s_1 \leftrightarrow s_2$ and $R' : s_1' \leftrightarrow s_2'$ are two spans in $\mathcal{I}$, then $R+R' : s_1+s_1' \leftrightarrow s_2+s_2'$ is a span as well. A square in $\mathcal{I}$ is parametric if and only if both squares in
$$\begin{tikzcd} s_1 \arrow[r] & s_1' \\ R \arrow[r] \arrow[u] \arrow[d] & R' \arrow[u] \arrow[d] \\ s_2 \arrow[r] & s_2' \end{tikzcd}$$
are pull-backs.

Up to isomorphism, all parametric squares in $\mathcal{I}$ are of the form
$$\begin{tikzcd} s_1 \arrow[r, "\mathbf{inl}"] \arrow[d, leftrightarrow, "R"'] & s_1 + s_1' \arrow[d, leftrightarrow, "R+R'"] \\ s_2 \arrow[r, "\mathbf{inl}"] & s_2 + s_2' \end{tikzcd}.$$

The base category $\mathcal{S}et$ is extended with ordinary binary relations and a square
$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, leftrightarrow, "R"'] & A' \arrow[d, leftrightarrow, "R'"] \\ B \arrow[r, "f"] & B' \end{tikzcd}$$
are parametric if and only if

$$\forall a \in A, b \in B \text{ s.t. } (a,b) \in R \Rightarrow (fa, gb) \in R'.$$

We then take the ordinary category $\mathcal{P}$ of parametric functors and parametric natural transformations from $\mathcal{I}$ to $\mathcal{S}et$.

Every relation in $\mathcal{I}$ identifies an object in $\mathcal{P}\mathcal{I}^{\rightarrow}$ and every parametric square identifies a morphism. Precisely, for every relation $R : s_1 \leftrightarrow s_2$, define an object $\langle R, i_R, s_1 +_R s_2 \rangle$, where $s_1 +_R s_2$ is $s_1 + s_2$ modulo the relation R, and i_R is just the injection mapping every element in R to the equivalent classes of either of its components. Clearly, if both squares
$$\begin{tikzcd} R \arrow[r] \arrow[d] & s_1 \arrow[d] \\ R' \arrow[r] & s_1' \end{tikzcd}$$
and
$$\begin{tikzcd} R \arrow[r] \arrow[d] & s_2 \arrow[d] \\ R' \arrow[r] & s_2' \end{tikzcd}$$
are pull-backs, then
$$\begin{tikzcd} R \arrow[r] \arrow[d] & s_1 +_R s_1' \arrow[d] \\ R' \arrow[r] & s_2 +_{R'} s_2' \end{tikzcd}$$
is a pull-back as well. Conversely, for any object $\langle w, i, s \rangle$, we can simply build a relation $R : s \leftrightarrow s$ with $R_w = \mathbf{id}_w$.

The category $\mathcal{P}$ is Cartesian closed. In particular, exponentials are defined by

$$\begin{aligned} B^A s &= \mathcal{P}(\mathcal{I}(s, -) \times A, B), \\ B^A i f s''(j, a) &= f s''(j \circ i, a), \end{aligned}$$

where $s, s', s'' \in \mathcal{I}$, $i : s \to s'$, $j : s' \to s'' \in \mathcal{I}$, $f \in B^A s$ and $a \in As''$. Furthermore,

$$(f_1, f_2) \in B^A R \iff$$

for all parametric squares $\begin{array}{ccc} s_1 & \xrightarrow{l_1} & s_1' \\ {\scriptstyle R}\updownarrow & & \updownarrow{\scriptstyle R'} \\ s_2 & \xrightarrow{l_2} & s_2' \end{array}$ and elements $a_1 \in As_1', a_2 \in As_2'$,

$$(a_1, a_2) \in AR' \Rightarrow (f_1 s_1'(l_1, a_1), f_2 s_2'(l_2, a_2)) \in BR',$$

where $s_1, s_2 \in \mathcal{I}$, $f_1 \in B^A s_1$ and $f_2 \in B^A s_2$. The monad is specified exactly as for the model $\mathcal{S}et^{\mathcal{I}}$ except that on objects, it is the quotient

$$TAs = \{(s', a) \mid s' \in \mathcal{I}, a \in A(s + s')\}/\simeq$$

where $(s_1', a_1) \simeq (s_2', a_2)$ if and only if there is some $R' : s_1' \leftrightarrow s_2'$ such that $(a_1, a_2) \in A(\mathbf{id}_s + R')$. The relation $TAR : TAs_1 \leftrightarrow TAs_2$, for any $R : s_1 \leftrightarrow s_2$ in $\mathcal{I}$, is given by

$$\begin{aligned} &(e_1, e_2) \in TAR \iff \\ &\quad \exists R' : s_1' \leftrightarrow s_2', a_1 \in A(s_1 + s_1'), a_2 \in A(s_2 + s_2') \text{ s.t.} \\ &\quad\quad e_1 = [s_1', a_1] \ \& \ e_2 = [s_2', a_2] \ \& \ (a_1, a_2) \in A(R + R'). \end{aligned}$$

Logical relations derived over $\mathcal{S}et^{\mathcal{PI}^{\to}}$ and $\mathcal{P}$ are equivalent in the sense that for any type τ and any relation $R : s_1 \leftrightarrow s_2$, there is some object $\langle w, i, s\rangle$ in $\mathcal{PI}^{\to}$ such that values related by $[\![\tau]\!]R$ are also related by $\mathcal{R}^i_\tau$ after being lifted to the proper world, and conversely, for any object $\langle w, i, s\rangle$ of $\mathcal{PI}^{\to}$, there is some relation R such that values related by $\mathcal{R}^i_\tau$ are related by $[\![\tau]\!]R$. The equivalence can be proved by induction on types when the relations for base types are carefully defined. The induction steps of function types and monadic types are shown by the following two propositions, while others are standard. Let $\mathcal{R}$ be a logical relation defined over $\mathcal{S}et^{\mathcal{PI}^{\to}}$. In particular, $\mathcal{R}_{TA}$ and $\mathcal{R}_{B^A}$ are defined by (5.5) and (5.6) respectively.

Proposition 5.14. *A is an object in $\mathcal{S}et^{\mathcal{I}}$. For every relation $R : s_1 \leftrightarrow s_2$ and every object $\langle w, i, s\rangle \in \mathcal{PI}^{\to}$, if*

$$\begin{aligned} (a_1, a_2) \in AR \quad &\Rightarrow \quad (Ai_1(a_1), Ai_2(a_2)) \in \mathcal{R}^{i_R}_A, \\ (a_1, a_2) \in \mathcal{R}^i_A \quad &\Rightarrow \quad (a_1, a_2) \in AR_w, \end{aligned}$$

where $i_R : R \to s_1 +_R s_2$ is the injection identifying the relation R, $i_1 : s_1 \to s_1 +_R s_2, i_2 : s_2 \to s_1 +_R s_2$ are canonical injections in $\mathcal{I}$, and $R_w : s \leftrightarrow s$ is the identity relation on w, then

$$\begin{aligned} (e_1, e_2) \in TAR \quad &\Rightarrow \quad (TAi_1(e_1), TAi_2(e_2)) \in \mathcal{R}^{i_R}_{TA}, \\ (e_1, e_2) \in \mathcal{R}^i_{TA} \quad &\Rightarrow \quad (e_1, e_2) \in TAR_w. \end{aligned}$$

Proof. If $(e_1, e_2) \in TAR$, according to the definition of TAR, there exist some relation $R' : s'_1 \leftrightarrow s'_2$ and two elements $a_1 \in A(s_1+s'_1)$, $a_2 \in A(s_2+s'_2)$ such that $e_1 = [s'_1, a_1]$, $e_2 = [s'_2, a_2]$ and $(a_1, a_2) \in A(R + R')$. Then $TAi_1(e_1) = [s'_1, A(i_1 + \mathbf{id}_{s'_1})a_1]$ and $TAi_2(e_2) = [s'_2, A(i_2 + \mathbf{id}_{s'_2})a_2]$. Let $i_{R'} : R' \to s'_1 +_{R'} s'_2$ be the injection identified by R' and $i'_1 : s'_1 \to s'_1 +_{R'} s'_2$ and $i'_2 : s'_2 \to s'_1 +_{R'} s'_2$ be the canonical injections, so

$$i_R + i_{R'} : R + R' \to (s_1 +_R s_2) + (s'_1 +_{R'} s'_2)$$

is the injection identified by $R + R'$. Because $(a_1, a_2) \in A(R + R')$, by the hypothesis, $(A(i_1 + i'_1)a_1, A(i_2 + i'_2)a_2) \in \mathcal{R}_A^{i_R+i_{R'}}$, hence $(TAi_1(e_1), TAi_2(e_2)) \in \mathcal{R}_{TA}^{i_R}$ according to (5.5).

Suppose that $e_1 = [s'_1, a_1]$ and $e_2 = [s'_2, a_2]$. If $(e_1, e_2) \in \mathcal{R}_{TA}^{i}$, according to (5.5), there exist some $\langle w_0, i_0, s_0\rangle$ in $\mathcal{PI}^{\to}$ and two injections $l_1 : s'_1 \to s_0$, $l_2 : s'_2 \to s_0$ such that $(A(\mathbf{id}_s + l_1)a_1, A(\mathbf{id}_s + l_2)a_2) \in \mathcal{R}_A^{i+i_0}$. Then by the hypothesis,

$$(A(\mathbf{id}_s + l_1)a_1, A(\mathbf{id}_s + l_2)a_2) \in AR_{w+w_0} = A(R_w + R_{w_0}).$$

Clearly, $(s_1, a_1) \simeq (s_0, A(\mathbf{id}_s + l_1)a_1)$ by taking l_1 as a relation between s_1 and s_0. Also, $(s_1, a_1) \simeq (s_0, A(\mathbf{id}_s + l_1)a_1)$, hence $(e_1, e_2) \in TAR_w$. ☐

Proposition 5.15. *Suppose that A, B are two objects in $\mathcal{S}et^{\mathcal{I}}$. For every relation $R : s_1 \leftrightarrow s_2$ and every object $\langle w, i, s\rangle \in \mathcal{PI}^{\to}$, if*

$$\begin{aligned}
(a_1, a_2) \in AR &\;\Rightarrow\; (Ai_1(a_1), Ai_2(a_2)) \in \mathcal{R}_A^{i_R},\\
(b_1, b_2) \in BR &\;\Rightarrow\; (Bi_1(b_1), Bi_2(b_2)) \in \mathcal{R}_B^{i_R},\\
(a_1, a_2) \in \mathcal{R}_A^{i} &\;\Rightarrow\; (a_1, a_2) \in AR_w,\\
(b_1, b_2) \in \mathcal{R}_B^{i} &\;\Rightarrow\; (b_1, b_2) \in BR_w,
\end{aligned}$$

where $i_R : R \to s_1 +_R s_2$ is the injection identifying the relation R, $i_1 : s_1 \to s_1 +_R s_2, i_2 : s_2 \to s_1 +_R s_2$ are canonical injections in $\mathcal{I}$, and $R_w : s \leftrightarrow s$ is the identity relation on w, then

$$\begin{aligned}
(f_1, f_2) \in B^A R &\;\Rightarrow\; (B^A i_1(f_1), B^A i_2(f_2)) \in \mathcal{R}_{B^A}^{i_R},\\
(f_1, f_2) \in \mathcal{R}_{B^A}^{i} &\;\Rightarrow\; (f_1, f_2) \in B^A R_w.
\end{aligned}$$

Proof. If $(f_1, f_2) \in B^A R$, let $(j, l) : i_R \to i'$ be an arbitrary morphism in $\mathcal{PI}^{\to}$, where $i' : w' \to s' \in \mathcal{I}$, then by Lemma 5.3, there exists some object $\langle w_0, i_0, s_0\rangle$ such that (j, l) is equivalent to the morphism $(\mathbf{inl}, \mathbf{inl})$ from i_R to $i_R + i_0$. Take two arbitrary elements $a_1, a_2 \in A((s_1 +_R s_2) + s_0)$ such that $(a_1, a_2) \in \mathcal{R}_A^{i_R+i_0}$. By hypothesis, $(a_1, a_2) \in A(R' + R_{w_0})$, where $R' : s_1 +_R s_2 \leftrightarrow s_1 +_R s_2$ is the relation equivalent to R, so the square
$$\begin{tikzcd} s_1 \arrow[r, "i_1"] \arrow[d, leftrightarrow, "R"'] & s_1 +_R s_2 \arrow[d, leftrightarrow, "R'"] \\ s_2 \arrow[r, "i_2"] & s_1 +_R s_2 \end{tikzcd}$$
is

parametric in $\mathcal{I}$, hence $(B^A i_1(f_1), B^A i_2(f_2)) \in B^A R'$ by the functoriality. Because the square
$$\begin{tikzcd} s_1 +_R s_2 \arrow[r, "\mathbf{inl}"] \arrow[d, leftrightarrow, "R'"] & (s_1 +_R s_2) + s_0 \arrow[d, "R'+R_{w_0}"] \\ s_1 +_R s_2 \arrow[r, "\mathbf{inl}"] & (s_1 +_R s_2) + s_0 \end{tikzcd}$$
is also parametric,

$$(B^A i_1(f_1)((s_1 +_R s_2) + s_0)(\mathbf{inl}, a_1), B^A i_2(f_2)((s_1 +_R s_2) + s_0)(\mathbf{inl}, a_2)) \in B^A(R' + R_{w_0}),$$

and by hypothesis, they are related by $\mathcal{R}_B^{i_R + i_0}$, hence $(B^A i_1(f_1), B^A i_2(f_2)) \in \mathcal{R}_{B^A}^{i_R}$.

If $(f_1, f_2) \in \mathcal{R}_{B^A}^{i}$, take an arbitrary parametric square
$$\begin{tikzcd} s \arrow[r, "l_1"] \arrow[d, leftrightarrow, "R_w"'] & s'_1 \arrow[d, leftrightarrow, "R'"] \\ s \arrow[r, "l_2"] & s'_2 \end{tikzcd}$$
, which, up to isomorphism, is equivalent to
$$\begin{tikzcd} s \arrow[r, "\mathbf{inl}"] \arrow[d, leftrightarrow, "R_w"'] & s + s_1^0 \arrow[d, leftrightarrow, "R_w+R_0"] \\ s \arrow[r, "\mathbf{inl}"] & s + s_2^0 \end{tikzcd}$$
for some relation $R_0 : s_1^0 \leftrightarrow s_2^0$. Let $i_{R_0} : R_0 \to s_1^0 +_{R_0} s_2^0$ be the injection identified by R_0, $i_1^0 : s_1^0 \to s_1^0 +_{R_0} s_2^0$ and $i_2^0 : s_2^0 \to s_1^0 +_{R_0} s_2^0$ be the canonical injections, and $R'_0 : s_1^0 +_{R_0} s_2^0 \leftrightarrow s_1^0 +_{R_0} s_2^0$ be the equivalent relation to R_0. Then for every elements $a_1 \in A(s + s_1^0), a_2 \in A(S + s_2^0)$ such that $(a_1, a_2) \in A(R_w + R_0)$, by the hypothesis,
$$(A(\mathbf{id}_s + i_1^0)a_1, A(\mathbf{id}_s + i_2^0)a_2) \in \mathcal{R}_A^{i + i_{R_0}}.$$
Because $(\mathbf{inl}, \mathbf{inl}) : i \to i + i_{R_0}$ is a morphism in $\mathcal{PI}^{\to}$,

$$(f_1(s + (s_1^0 +_{R_0} s_2^0))(\subseteq, A(\mathbf{id}_s + i_1^0)a_1), f_2(s + (s_1^0 +_{R_0} s_2^0))(\subseteq, A(\mathbf{id}_s + i_2^0)a_2) \in \mathcal{R}_B^{i + i_{R_0}}$$

and by hypothesis, they are also related by $B(R_w + R'_0)$, so $f_1(s + s_1)(\mathbf{inl}, a_1)$ and $(f_2(s + s_2)(\mathbf{inl}, a_2)$ must be related by $B(R_w + R_0)$, otherwise the functoriality would imply that the above two lifted elements are not related by $B(R_w + R'_0)$. Hence, $(f_1, f_2) \in B^A R_w$. $\square$

Chapitre 6

Complétude des relations logiques

Lorsque nous utilisons les relations logiques pour déduire l'équivalence contextuelle, la complétude des relations logiques est un sujet que nous ne devons pas ignorer. La complétude des relations logiques peut avoir deux sens :

- dans le sens *fort*, les relations logiques sont complètes, par rapport à l'équivalence contextuelle, si et seulement s'il existe une relation logique spécifique telle que tous les programmes équivalents peuvent être reliés par cette relation ;
- dans le sens *faible*, les relations logiques sont complètes si et seulement si pour chaque paire de programmes contextuellement équivalent, il existe une relation logique qui relie les deux programmes.

Dans cette thèse, nous nous concentrons sur la complétude des relations logiques dans le sens fort, ce qui est plus utile et plus pratique pour étudier l'équivalence contextuelle.

En général, dans le lambda-calcul simplement typé, les relations logiques ne sont complètes que pour les types du premier ordre. Nous commençons le chapitre par une revue brève de la preuve de complétude (pour les types du premier ordre) dans les lambda-calculs typés standards. La complétude de relations logiques pour les types monadiques est plus subtile. D'abord, la définition standard d'équivalence contextuelle du lambda-calcul typé ne s'applique pas dans le métalangage cryptographique. Il faut trouver une définition plus adaptée à notre langage. Ensuite, il s'avère très difficile d'obtenir un résultat général de complétude pour toutes les monades à cause de la grande différence entre les propriétés spécifiques des monades, ainsi qu'entre les définitions des relations logiques. Nous exposerons la difficulté en essayant de faire une preuve générale pour les types du premier ordre. Le lecteur intéressé par ce sujet peuvent se référer à l'annexe B pour une discussion détaillée sur la complétude des relations logiques monadiques.

Dans ce chapitre, nous nous concentrons sur les relations logiques pour la monade de la génération de clés. Tout d'abord, nous continuons notre discussion de la notion d'équivalence contextuelle commencée à la fin du chapitre 3. Nous arrivons donc, dans la partie 6.1, à une définition finale de l'équivalence contextuelle pour les protocoles cryptographiques, et nous montrons que la relation logique cryptographique (définie au chapitre 5) permet de déduire cette notion d'équivalence contextuelle. Dans la partie 6.2 et la partie 6.3, nous examinons la question de la complétude des relations logiques dérivées sur la catégorie $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. En particulier, nous trouvons que ces relations logiques ne sont pas complètes pour tous les types du premier ordre. Nous dégageons un sous-ensemble des types du premier ordre pour lesquels nous pouvons obtenir la complétude. Afin d'obtenir la complétude pour tous les types, nous utilisons la notion de relations logiques lâches (lax). Dans la partie 6.4, nous définissons, sur la même catégorie $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, une relation logique complète qui est lâche pour les types de fonctions et les types monadiques, mais stricte (non lâche) pour les autres.

Completeness is an important concern about logical relations that we should not bypass when we are inclined to use logical relations to deduce contextual equivalence. There are two senses of completeness for logical relations:

- In the *strong* sense, we say that logical relations are complete, w.r.t. contextual equivalence, if and only if there exist a logical relation such that all contextually equivalent programs can be related by this specific logical relation;

- In the *weak* sense, we say that logical relations are complete, if and only if for every pair of contextually related programs, there exists a logical relation such that the equivalent programs can be related.

We shall focus in this thesis on completeness of the strong sense, which is more practical than the weak one for studying equivalence between programs.

However, in simply-typed lambda-calculus, logical relations are only complete for types up to first order in general. Recall the standard definition of contextual equivalence in simply-typed lambda-calculus, defined in a set-theoretical model. Two closed terms t_1, t_2 of the same type τ, are contextually equivalent ($t_1 \approx_\tau t_2$), if and only if, whatever the term $\mathbb{C}$ such that $x : \tau \vdash \mathbb{C} : o$ ($o \in \mathbf{Obs}$) is derivable, it holds that

$$[\![\mathbb{C}]\!][x \mapsto [\![t_1]\!]] = [\![\mathbb{C}]\!][x \mapsto [\![t_2]\!]].$$

Logical relations for simply typed lambda-calculus are complete up to first-order types, in the strong sense that there exists a logical relation $(\mathcal{R}_\tau)_{\tau \text{ type}}$ which is partial equality on observation types, such that if $\vdash t_1 : \tau$ and $\vdash t_2 : \tau$ are derivable, for any type τ up to first order, it holds that

$$t_1 \approx_\tau t_2 \implies [\![t_1]\!] \; \mathcal{R}_\tau \; [\![t_2]\!].$$

Say that a value $a \in [\![\tau]\!]$ is *definable* if and only if there exists a closed term t such that $\vdash t : \tau$ is derivable and $a = [\![t]\!]$. We define the relation $\sim_\tau$ by $a_1 \sim_\tau a_2$ (for $a_1, a_2 \in [\![\tau]\!]$) if and only if a_1, a_2 are definable and $a_1 \approx_\tau a_2$. Let $(\mathcal{R}_\tau)_{\tau \text{ type}}$ be the logical relation induced by $\mathcal{R}_b = \sim_b$ at every base type b.

The proof of completeness is by induction on τ. Case $\tau = b$ is obvious. Let $\tau = b \to \tau'$. Take two terms t_1, t_2 of type $b \to \tau'$ such that t_1 and t_2 are related by $\approx_{b \to \tau'}$. Let $f_1 = [\![t_1]\!]$ and $f_2 = [\![t_2]\!]$. Assume that $a_1, a_2 \in [\![b]\!]$ are related by $\mathcal{R}_b$, therefore $a_1 \sim_b a_2$ since $\mathcal{R}_b = \sim_b$. Clearly, a_1 and a_2 are definable, say by terms u_1 and u_2, respectively. Then, for any context $\mathbb{C}$

such that $x : \tau' \vdash \mathbb{C} : o$ ($o \in \mathbf{Obs}$) is derivable,

$$\begin{aligned}
& [\![\mathbb{C}]\!][x \mapsto f_1(a_1)] \\
= \; & [\![\mathbb{C}[xu_1/x]]\!][x \mapsto f_1] \quad \text{(since } a_1 = [\![u_1]\!]\text{)} \\
= \; & [\![\mathbb{C}[xu_1/x]]\!][z \mapsto f_2] \quad \text{(since } f_1 \approx_{b \to \tau'} f_2\text{)} \\
= \; & [\![\mathbb{C}]\!][x \mapsto f_2(a_1)] \\
= \; & [\![\mathbb{C}[t_2 x/x]]\!][x \mapsto a_1] \quad \text{(since } f_2 = [\![t_2]\!]\text{)} \\
= \; & [\![\mathbb{C}[t_2 x/x]]\!][x \mapsto a_2] \quad \text{(since } a_1 \approx_b a_2\text{)} \\
= \; & [\![\mathbb{C}]\!][x \mapsto f_2(a_2)].
\end{aligned}$$

Hence $f_1(a_1) \approx_{\tau'} f_2(a_2)$. By induction hypothesis, $f_1(a_1)\ \mathcal{R}_{\tau'}\ f_2(a_2)$. Because a_1 and a_2 are arbitrary, it holds that $f_1\ \mathcal{R}_{b \to \tau'}\ f_2$.

Completeness of monadic logical relations is much subtler. First, as we have seen in Section 3.5, the standard definition of contextual equivalence for simply-typed lambda-calculus is not suitable for the computational lambda-calculus. We revise the definition and consider contexts of type $\mathsf{T}o$ ($o \in \mathbf{Obs}$): two closed terms t_1, t_2 of the same type τ, are contextually equivalent ($t_1 \approx_\tau t_2$), if and only if, whatever the term $\mathbb{C}$ such that $x : \tau \vdash \mathbb{C} : \mathsf{T}o$ ($o \in \mathbf{Obs}$) is derivable, it holds that

$$[\![\mathbb{C}]\!][x \mapsto [\![a_1]\!]] = [\![\mathbb{C}]\!][x \mapsto [\![a_2]\!]].$$

Second, it is very difficult to get a general result on completeness for all monads, since specific properties of particular monads (and corresponding logical relations) are quite different. Furthermore, since contexts are involved, language constants play an important role in discussions of completeness and they vary widely for different forms of computations.

We investigate completeness up to first-order types in the strong sense, in a similar way as in simply-typed lambda-calculus. We aim at finding a logical relation $(\mathcal{R}_\tau)_{\tau \text{ type}}$ such that if $\vdash t_1 : \tau$ and $\vdash t_2 : \tau$ are derivable, for any type τ up to first order, it holds that

$$t_1 \approx_\tau t_2 \implies [\![t_1]\!]\ \mathcal{R}_\tau\ [\![t_2]\!].$$

Or, shortly: $\sim_\tau \subseteq \mathcal{R}_\tau$. We again induce a logical relation $(\mathcal{R}_\tau)_{\tau \text{ type}}$ by $\mathcal{R}_b = \sim_b$, for every base type b. Then the proof would go by induction over τ, to show $\sim_\tau \subseteq \mathcal{R}_\tau$ for an arbitrary monad *T* and every first-order type τ. Cases $\tau = b$ and $\tau = b \to \tau'$ go identically as in the above proof for simply-typed lambda-calculus. The difficult case is $\tau = \mathsf{T}\tau'$, i.e., the induction step

$$\sim_\tau \subseteq \mathcal{R}_\tau \implies \sim_{\mathsf{T}\tau} \subseteq \mathcal{R}_{\mathsf{T}\tau}. \tag{6.1}$$

We did not find any general way to prove this for an arbitrary monad. In fact, this does not hold for all first-order types. For certain concrete monads, in order to show (6.1), we must have further restrictions on τ. Interested readers are referred to Appendix B for further discussion on completeness of monadic logical relations, where we show (6.1) for a list of concrete monads (sometimes with further restrictions on τ).

In this chapter, we shall concentrate ourselves on logical relations for the special monad of dynamic key generation. First of all, we continue the discussion at the end of Chapter 3, on the notion of contextual equivalence. We give, in Section 6.1, the final definition of the contextual equivalence for cryptographic protocols and show that the cryptographic logical relation defined in Section 5.3 can be used to deduce the contextual equivalence. Then we start to investigate the completeness of logical relations derived over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. In particular, it turns out that completeness of these logical relations does not hold for every type up to first order, but it does hold for a certain subset of first-order types. In Section 6.2, we investigate completeness for non-monadic types, and Section 6.3 is about monadic types. To get completeness for all types, we switch to the notion of *lax logical relations*. In Section 6.4, we define a complete lax logical relation over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, which is lax at function types and monadic types, but can be strict at various other types.

6.1 Équivalence contextuelle des protocoles cryptographiques

Recall the definition of contextual equivalence in the end of Chapter 3: two values $a_1, a_2 \in [\![\tau]\!]s$ are contextually equivalent *at* s, ($a_1 \approx_\tau^s a_2$), if and only if, for every finite set of variables w', every injections $i' : w' \to s'$ and $l : s \to s'$ and every term $\mathbb{C}$ such that

$$\overline{w' : \mathsf{key}}, x : \tau \vdash \mathbb{C} : \mathsf{T}o, \quad (o \in \mathbf{Obs})$$

is derivable,

$$[\![\mathbb{C}]\!]s'i'[x \mapsto [\![\tau]\!]l(a_1)] = [\![\mathbb{C}]\!]s'i'[x \mapsto [\![\tau]\!]l(a_2)].$$

This definition potentially allows contexts to have access to *all* keys, which is too powerful. The key of defining contextual equivalence for cryptographic protocols is that contexts must represent honestly the power of attackers. Obviously, attackers do not necessarily know every key.

The category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ has been proved very useful for defining logical relations for the cryptographic metalanguage. It can also be used here to define a more reasonable notion of contextual equivalence. Note that we shall consider here the category equivalent to $\mathcal{PI}^{\rightarrow}$ where w is restricted to be a finite set of *variables* and continue to call this category $\mathcal{PI}^{\rightarrow}$. Objects $\langle w, i, s\rangle$ are then sets w of variables denoting those disclosed keys in s, together with an injection i. We also use i to denote the environment $[\overline{w \mapsto i(w)}]$. Using the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, we then arrive at the following definition of contextual equivalence for dynamic key generation:

Definition 6.1 (Contextual equivalence for key generation). *Suppose that $i : w \to s$ is an injection in $\mathcal{I}$. Two values $a_1, a_2 \in [\![\tau]\!]s$ are said to be* contextually equivalent at $\langle w, i, s\rangle$, *written $a_1 \approx_\tau^{\langle w,i,s\rangle} a_2$, if and only if, for every morphisms $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$, for every term $\mathbb{C}$ such that*

$$\overline{w' : \mathsf{key}}, x : \tau \vdash \mathbb{C} : \mathsf{T}o, \quad (o \in \mathbf{Obs})$$

is derivable,

$$[\![\mathbb{C}]\!]s'i'[x \mapsto [\![\tau]\!]l(a_1)] = [\![\mathbb{C}]\!]s'i'[x \mapsto [\![\tau]\!]l(a_2)].$$

We often write $\approx_\tau^i$ for short, when the domain and codomain of i is clear from the context.

This definition is more general than the one we introduced in the end of Chapter 3. In particular, $a_1 \approx_\tau^s a_2$ if and only if $a_1 \approx_\tau^{\mathbf{id}_s} a_2$.

However, contexts in this definition do not represent the full power of real attackers. This is because morphisms in category $\mathcal{PI}^{\to}$ do not allow contexts defined at $\langle w, i, s\rangle$ to get access to keys in $s - i(w)$, hence contexts cannot build any messages including cipher-texts encrypted by secret keys. This is too strict, because in reality, attackers are certainly able to make use of those encrypted messages passing through the network, even though they are not able to decrypt them. In other words, a context for cryptographic protocols depends not only on a set of disclosed keys, but also on a set of cipher-texts encrypted by secret keys, which we call the *knowledge* of the context.

Formally, a knowledge κ is a family of sets of cipher-texts such that for every $\langle w, i, s\rangle \in \mathcal{PI}^{\to}$, $\kappa^{\langle w,i,s\rangle}$ is a set of cipher-texts encrypted by a key in $s - i(w)$, i.e.,

$$\kappa^{\langle w,i,s\rangle} \subseteq \{\mathsf{e}(m, k) \mid m \in [\![\mathsf{msg}]\!]s \ \ \& \ \ k \in s - i(w)\}.$$

We also write κ^i for $\kappa^{\langle w,i,s\rangle}$. Note that in some formal models, the term "knowledge" represents all messages that an attacker is able to access, not just secret cipher-texts, which is different from our notion here.

We say that a knowledge κ is *monotonic* if and only if for every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$, and every message $a \in [\![\mathsf{msg}]\!]s$,

$$a \in \kappa^i \Longrightarrow [\![\mathsf{msg}]\!]l(a) \in \kappa^{i'},$$

A knowledge κ is *consistent* if and only if for every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$, and every message $a \in [\![\mathsf{msg}]\!]s$,

$$a \in \kappa^i \Longleftrightarrow [\![\mathsf{msg}]\!]l(a) \in \kappa^{i'}.$$

Moreover, a knowledge κ is *finite* if and only if for every $\langle w, i, s\rangle \in \mathcal{PI}^{\to}$, κ^i is finite.

Our notion of contextual equivalence is defined over denotational models, so the way that a context accesses those secret cipher-texts is by context variables. Given a typing context Γ and a knowledge κ, for every injection $i : w \to s$, we say that an environment $\rho \in [\![\Gamma]\!]s$ is a *κ-honest environment* if and only if for every variable $x : \mathsf{msg} \in \Gamma$, $\rho(x) \in \kappa^i$. If κ is finite, an environment ρ is said to *have full access to κ* if for every $a \in \kappa^i$, there is a variable $x : \mathsf{msg} \in \Gamma$ such that $\rho(x) = a$. This implicitly requires that there must be enough variables of type msg in Γ.

Definition 6.2 (Contextual equivalence for cryptographic protocols). *Suppose that κ is a knowledge and $i : w \to s$ is an injection in $\mathcal{I}$. Two values $a_1, a_2 \in [\![\tau]\!]s$ are* contextually equivalent at $\langle w, i, s\rangle$ and κ, *written as $a_1 \approx_\tau^{\langle w,i,s\rangle,\kappa} a_2$, if and only if, for every morphisms $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$ and every term $\mathbb{C}$ such that*

$$x : \tau, \overline{w' : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash \mathbb{C} : \mathsf{T}o, \quad (o \in \mathbf{Obs})$$

is derivable, and for any κ-based environment ρ,

$$[\![\mathbb{C}]\!]s'\rho[x \mapsto [\![\tau]\!]l(a_1), \overline{w' \mapsto i'(w')}] = [\![\mathbb{C}]\!]s'\rho[x \mapsto [\![\tau]\!]l(a_2), \overline{w' \mapsto i'(w')}],$$

where ρ is any κ-honest environment.

Two terms t_1 and t_2 such that $\overline{w : \mathsf{key}} \vdash t_1 : \tau$ and $\overline{w : \mathsf{key}} \vdash t_2 : \tau$ are derivable, are contextually equivalent at $\langle w, i, s\rangle$ and κ *if and only if $[\![t_1]\!]si \approx_\tau^{\langle w,i,s\rangle,\kappa} [\![t_2]\!]si$, where i also denotes the environment $[\overline{w \mapsto i(w)}]$.*

Given a cipher function φ, for every $\langle w, i, s\rangle \in \mathcal{PI}^{\to}$, consider

$$\begin{aligned} |\varphi|^i \quad = \quad \{\mathsf{e}(a,k) \mid \exists k' \in s - i(w), a' \in [\![\mathsf{msg}]\!]s, \\ \text{s.t. } (a,a') \in \varphi^i(k,k') \text{ or } (a',a) \in \varphi^i(k',k)\}. \end{aligned}$$

we call $|\varphi|$ the *knowledge* of φ, which defines uniquely a knowledge. Using logical relations to derive contextual equivalence then requires at least two conditions: first, the knowledge of the cipher function must contain all cipher-texts in the knowledge of contexts, i.e., $\kappa \subseteq |\varphi|$; second, the cipher function φ must respect the identity of the knowledge κ of contexts, i.e., $(a,a) \in \varphi^i(k,k)$ for every $\langle w, i, s\rangle \in \mathcal{PI}^{\to}$ and every $\mathsf{e}(a,k) \in \kappa^i$.

With the cryptographic logical relation $(\boldsymbol{\mathcal{R}}_\tau)_{\tau \text{ type}}$, as defined in Definition 5.2, we can deduce the contextual equivalence in the cryptographic metalanguage.

Theorem 6.1 (Soundness of the cryptographic logical relation). *Suppose that φ is a monotonic cipher function, κ is a monotonic knowledge and φ respects the identity of κ. For every injection $i : w \to s$ in $\mathcal{I}$ and every pair of values $a_1, a_2 \in [\![\tau]\!]s$, if $a_1 \; \boldsymbol{\mathcal{R}}_\tau^{i,\varphi} \; a_2$, then $a_1 \approx_\tau^{i,\kappa} a_2$, where $(\boldsymbol{\mathcal{R}}_\tau)_{\tau \text{ type}}$ is the cryptographic logical relation as defined in Definition 5.2.*

Proof. Take any morphism $(j,l) : \langle w,i,s\rangle \to \langle w',i',s'\rangle$ in $\mathcal{PI}^{\to}$. Because $\boldsymbol{\mathcal{R}}^{i,\varphi}_{\tau}$ is monotonic (Proposition 5.9), we have $[\![\tau]\!]l(a_1)\ \boldsymbol{\mathcal{R}}^{i',\varphi}_{\tau}\ [\![\tau]\!]l(a_2)$. Take any context $\mathbb{C}$ such that

$$x : \tau, \overline{w' : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash \mathbb{C} : \mathsf{Tnat}$$

is derivable, and let ρ_1, ρ_2 be two environments

$$\begin{aligned}\rho_1 &= \rho[x \mapsto [\![\tau]\!]l(a_1), w' \mapsto i'(w')],\\ \rho_2 &= \rho[x \mapsto [\![\tau]\!]l(a_2), w' \mapsto i'(w')],\end{aligned}$$

where ρ is any κ-honest environment. Obviously, $\rho_1\ \boldsymbol{\mathcal{R}}^{i',\varphi}_{\Gamma}\ \rho_2$ since φ respects the identity of κ, where $\Gamma = \{x : \tau, \overline{w' : \mathsf{key}}, \overline{v : \mathsf{msg}}\}$. Then by the Basic Lemma (Proposition 5.11), $[\![\mathbb{C}]\!]s'\rho_1\ \boldsymbol{\mathcal{R}}^{i',\varphi}_{\mathsf{Tnat}}\ [\![\mathbb{C}]\!]s'\rho_2$, i.e., $[\![\mathbb{C}]\!]s'\rho_1 = [\![\mathbb{C}]\!]s'\rho_2$, hence $a_1 \approx^{i,\kappa}_{\tau} a_2$. $\square$

Note that we state this soundness theorem for all values, not necessarily definable values, while in our later discussion on completeness, we shall focus on definable values.

6.2 Complétude pour les types non-monadiques

While we can use logical relations to deduce contextual equivalence, it is natural to wonder whether all contextually equivalent programs can be related by logical relations. This is what we call the completeness of logical relations. However, in the cryptographic metalanguage, if we do not have restrictions on computation types, logical relations for the key generation monad (derived over the category $\mathcal{S}et^{\mathcal{PI}^{\to}}$) is not even complete for zero order types. Consider the following two programs of type TTkey:

$$\begin{aligned}&\mathtt{let}\ k \Leftarrow \mathtt{new}\ \mathtt{in}\ \mathtt{val}(\mathtt{let}\ k' \Leftarrow \mathtt{new}\ \mathtt{in}\ \mathtt{val}(k)),\\ &\mathtt{let}\ k \Leftarrow \mathtt{new}\ \mathtt{in}\ \mathtt{val}(\mathtt{let}\ k' \Leftarrow \mathtt{new}\ \mathtt{in}\ \mathtt{val}(k')).\end{aligned}$$

It is easy to compute their denotations in $\mathcal{S}et^{\mathcal{I}}$, $[\{k_1\}, [\{k_1'\}, k_1]]$ and $[\{k_2\}, [\{k_2'\}, k_2']]$ respectively. They are not logically related because there are two levels of computations, and logical relations have to be constructed at each level. Since in the first term the value (k_1) is a key generated during the outer computation, while the value of the other term (k_2') is from the inner computation, which is again fresh for all keys generated during the outer one, there is no way to define these two values as the same fresh key in semantics. But the only way to distinguish these two terms is retrieving their values (two fresh keys) and do some comparison. Since both are fresh, no context can distinguish them, so these two terms are indeed contextually equivalent.

The point is that logical relations are defined by induction on types, but contextual equivalence is not. For those terms including several levels of computations, contexts are usually not able to know the exact level where a fresh key is generated, but in semantics, these levels are explicitly identified, by types indeed. This problem exists not only for types of the form $\mathsf{TT}\tau$, but for all computation types $\mathsf{T}\tau$ where τ contains again some computations, e.g., $\mathsf{T}(\mathsf{key} \times \mathsf{Tkey})$, except when they are inside a function, e,g., $\mathsf{T}(\mathsf{key} \to \mathsf{Tkey})$. Encoding of protocols may need this kind of types for typing the program, but this can be avoided by a careful encoding, e.g., the encoding of the symmetric key establishment protocol in Chapter 2.

Now that we are not able to show the completeness for types that are simply classified according to the order, it is better to do more refined classification on types. Here is a classification on a subset of first-order types.

$$\begin{array}{rcl} \tau^0 & ::= & b \mid \tau^0 \times \tau^0 \mid \mathsf{opt}[\tau^0] \\ \tau^1_p & ::= & b \mid \tau^1_p \times \tau^1_p \mid \mathsf{opt}[\tau^1_p] \mid b \to \tau^1_p \\ \tau^1 & ::= & \tau^0 \mid \mathsf{T}\tau^0 \mid b \to \tau^1 \mid \mathsf{T}(b \to \tau^1) \end{array}$$

where $b \in \Sigma$ is a base type. Class τ^0 consists of types of zero order but not containing any computation, and we call them *plain zero-order types*. Class τ^1_p is simply the class τ^0 plus first-order functions, but it still contains no computations. We call types in τ^1_p *plain first-order types*. We then have computations in the class τ^1, but these computations can only return values of plain zero-order types or functions, so types like TTkey are not allowed. We call types in class τ^1 *one-level first-order types*. In particular, this class of types is sufficient for typing most protocols.

We shall temporarily forget monadic types and investigate the completeness of logical relations derived over the category $\mathcal{S}et^{\mathcal{PI}^{\to}}$, for plain first-order types. In particular, the discussion shows how complete the cryptographic logical relation $(\boldsymbol{\mathcal{R}}_\tau)_{\tau \text{ type}}$ is.

First, say that a value $a \in [\![\tau]\!]s$ is *definable at* $\langle w, i, s\rangle$, where $i : w \to s$ is an injection in $\mathcal{PI}^{\to}$, if and only if there is a term t such that $\overline{w : \mathsf{key}} \vdash t : \tau$ is derivable and $[\![t]\!]si = a$ (i denoting the environment $[\overline{w \mapsto i(w)}]$).

Lemma 6.2. *For any object $\langle w, i, s\rangle \in \mathcal{PI}^{\to}$, a value $a \in [\![\mathsf{msg}]\!]s$ is* definable at $\langle w, i, s\rangle$ *if and only if $a \in [\![\mathsf{msg}]\!](i(w))$.*

Proof. First, we show that for any value $a \in [\![\mathsf{msg}]\!](i(w))$, there is a term t such that $\overline{w : \mathsf{key}} \vdash t : \mathsf{msg}$ holds and $[\![t]\!]si = a$. This can be proved by induction on the structure of the value a.

Now consider a value $a \in [\![\mathsf{msg}]\!](i(w))$ which is definable at $\langle w, i, s\rangle$ (by a term t). According to Lemma 3.3, $a \in [\![\mathsf{msg}]\!](i(w))$. $\square$

Lemma 6.3. *For any morphism $\langle j, l\rangle : \langle w, i, s\rangle \to \langle w', i', s'\rangle \in \mathcal{PI}^{\to}$, if a value $a \in [\![\tau]\!]s$ is definable at i, then $[\![\tau]\!]l(a)$ is also definable at i'.*

Proof. Suppose that a is definable by a term t such that $\overline{w : \mathsf{key}} \vdash t : \tau$ is derivable, i.e., $a = [\![t]\!]si$, then

$$\begin{aligned}
[\![\tau]\!]l(a) &= [\![\tau]\!]l([\![t]\!]s[\overline{w \mapsto i(w)}]) \\
&= [\![t]\!]s'[\overline{w \mapsto [\![\mathsf{key}]\!]l(i(w))}] \\
&\qquad \text{(by naturality of } [\![t]\!]) \\
&= [\![t]\!]s'[\overline{w \mapsto l(i(w))}] \\
&\qquad \text{(because } [\![\mathsf{key}]\!]l = l \text{ by definition)} \\
&= [\![t]\!]s'[\overline{w \mapsto i'(j(w))}] \\
&\qquad \text{(because } l \circ i = i' \circ j\text{, by the definition of } \mathcal{PI}^{\rightarrow}) \\
&= [\![t'j]\!]s'[\overline{w \mapsto i'(j(w))}] \\
&\qquad \text{(where } t' = tj^{-1}\text{, seeing } j \text{ and } j^{-1} \text{ as substitutions)} \\
&= [\![t']\!]s'([\![j]\!]s'[\overline{w \mapsto i'(j(w))}]) \\
&\qquad \text{(because } [\![_]\!] \text{ is functorial)} \\
&= [\![t']\!]s'[\overline{j(w) \mapsto i'(j(w))}] \\
&= [\![t']\!]s'[\overline{w' \mapsto i'(w')}],
\end{aligned}$$

hence $[\![\tau]\!]l(a)$ is definable. □

For every type τ, every injection $i : w \to s$ in $\mathcal{I}$, define a relation $\sim_\tau^{\langle w,i,s\rangle,\kappa}$, where κ is a context knowledge, by: for any values $a_1, a_2 \in [\![\tau]\!]s$, $a_1 \sim_\tau^{i,\kappa} a_2$ if and only if a_1, a_2 are definable at $\langle w, i, s\rangle$ and $a_1 \approx_\tau^{i,\kappa} a_2$. We then get the following completeness for logical relations derived over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$.

Proposition 6.4 (Completeness for plain first-order types). *Let κ be a finite context knowledge. Logical relation for the cryptographic metalanguage is complete for all plain first-order types in the strong sense: there exists a logical relation $(\mathcal{R}_\tau)_{\tau \text{ type}}$ derived over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ such that for every injection $i : w \to s$ in $\mathcal{I}$ and every terms t_1, t_2 such that $\overline{w : \mathsf{key}} \vdash t_1 : \tau$ and $\overline{w : \mathsf{key}} \vdash t_2 : \tau$ are derivable (τ a plain first-order type), if $t_1 \approx_\tau^{i,\kappa} t_2$, then $[\![t_1]\!]si\ \mathcal{R}_\tau^i\ [\![t_2]\!]si$.*

Proof. Let $\mathcal{R}_b^i = \sim_b^{i,\kappa}$ for every injection $i : w \to s$ in $\mathcal{I}$ and every base type b. Define $(\mathcal{R}_\tau)_{\tau \text{ type}}$ as the logical relation induced by $\mathcal{R}_b^i$ according to the derivation of logical relations over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, i.e., using the non-base type clauses of Definition 5.2. In particular,

$$\begin{aligned}
&f_1\ \mathcal{R}_{\tau\to\tau'}^i\ f_2 \iff \\
&\qquad \forall (j,l) : \langle w,i,s\rangle \to \langle w',i',s'\rangle \in \mathcal{PI}^{\rightarrow} \cdot \\
&\qquad\qquad \forall a_1, a_2 \in [\![\tau]\!]s' \cdot\ (a_1\ \mathcal{R}_\tau^{i'}\ a_2 \Rightarrow f_1 s'(l,a_1)\ \mathcal{R}_{\tau'}^{i'}\ f_2 s'(l,a_2))
\end{aligned}$$

We then prove by induction on τ.

If τ is a product type $\tau' \times \tau''$, then $a_1 = (a_1', a_1'')$ and $a_2 = (a_2', a_2'')$, where $a_1', a_2' \in [\![\tau']\!]s$, $a_1'', a_2'' \in [\![\tau'']\!]s$. Assume that a_1 and a_2 are definable at $\langle w, i, s\rangle$, so are a_1', a_1'', a_2', a_2'', e.g., a_1' is defined by $\mathtt{proj}_1(t)$ where $[\![t]\!]si = a_1$. Again assume that $(a_1, a_2) \notin \mathcal{R}^i_\tau$, then either $(a_1', a_2') \notin \mathcal{R}^i_{\tau'}$ or $(a_1'', a_2'') \notin \mathcal{R}^i_{\tau''}$. Without loss of generality, suppose that $(a_1', a_2') \notin \mathcal{R}^i_{\tau'}$. We then show that this implies that $a_1 \not\approx^{i,\kappa}_{\tau'\times\tau''} a_2$. By induction, there is a context $\mathbb{C}$ such that $y : \tau', \overline{w : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash \mathbb{C} : \mathsf{Tnat}$ is derivable and $[\![\mathbb{C}]\!]si[y \mapsto a_1'] \neq [\![\mathbb{C}]\!]si[y \mapsto a_2']$. Then the program

$$x : \tau' \times \tau'', \overline{w : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash (\lambda y.\mathbb{C})(\mathtt{proj}_1(x)) : \mathsf{Tnat}$$

can be used to distinguish a_1 and a_2.

If τ is an option type $\mathsf{opt}[\tau']$, for two values $a_1, a_2 \in [\![\mathsf{opt}[\tau]]\!]s$, assume that $(a_1, a_2) \notin \mathcal{R}^i_{\mathsf{opt}[\tau']}$, then either one of a_1 and a_2 equals $\perp$ while the other does not, or both do not equal $\perp$ but $(a_1, a_2) \notin \mathcal{R}^i_\tau$. However, in both cases, we can find programs that can distinguish a_1 and a_2, which is a contradiction to $a_1 \approx^{i,\kappa}_{\mathsf{opt}[\tau']} a_2$. In the former case, the program

$$x : \mathsf{opt}[\tau'], \overline{w : \mathsf{key}} \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{some}(\_)\ \mathtt{in}\ \mathtt{val}(1)\ \mathtt{else}\ \mathtt{val}(0) : \mathsf{Tnat}$$

can be used to distinguish a_1 and a_2. In the latter case, by induction, $a_1 \not\approx^{i,\kappa}_{\tau'} a_2$, so there is a context $\mathbb{C}$ such that $y : \tau', \overline{w : \mathsf{key}} \vdash \mathbb{C} : \mathsf{Tnat}$ is derivable and $[\![\mathbb{C}]\!]si[y \mapsto a_1'] \neq [\![\mathbb{C}]\!]si[y \mapsto a_2']$. Then the program

$$x : \mathsf{opt}[\tau'], \overline{w : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash \mathtt{case}\ x\ \mathtt{of}\ \mathtt{some}(y)\ \mathtt{in}\ \mathbb{C}\ \mathtt{else}\ \mathtt{val}(0) : \mathsf{Tnat}$$

can be used to distinguish a_1 and a_2.

If τ is a function type $b \to \tau'$, where $b \in \Sigma$ is a base type. Suppose that $f_1, f_2 \in [\![b \to \tau']\!]s$ and f_1, f_2 are definable at $\langle w, i, s\rangle$. Take any morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle \in \mathcal{PI}^{\to}$. According to Lemma 6.3, $[\![\tau]\!]l(f_1)$ and $[\![\tau]\!]l(f_2)$ are definable at $\langle w', i', s'\rangle$. Take any pair of values $a_1, a_2 \in [\![b]\!]s'$ such that $a_1\ \mathcal{R}^{i'}_b\ a_2$, then a_1 and a_2 are definable at $\langle w', i', s'\rangle$ and $a_1 \approx^{i',\kappa}_b a_2$. Let $\mathbb{C}$ be an arbitrary term such that $x : \tau', \overline{w : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash \mathbb{C} : \mathsf{Tnat}$ is derivable, and ρ be a κ-honest environment. Take any morphism $(j', l') : \langle w', i', s' \rangle \to \langle w'', i'', s'' \rangle \in \mathcal{PI}^{\to}$,

$$\begin{array}{rl}
& [\![\mathbb{C}]\!] s'' \rho[x \mapsto [\![\tau']\!] l'(f_1 s'(l, a_1)), \overline{w'' \mapsto i''(w'')}] \\
= & [\![\mathbb{C}]\!] s'' \rho[x \mapsto ([\![\tau]\!](l' \circ l) f_1) s''(\mathbf{id}_{s''}, [\![\tau']\!] l'(a_1)), \overline{w'' \mapsto i''(w'')}] \\
& \quad \text{(by the naturality of } f_1\text{)} \\
= & [\![\mathbb{C}[t_1 y/x]]\!] s'' \rho[y \mapsto [\![\tau']\!] l'(a_1), \overline{w'' \mapsto i''(w'')}] \\
& \quad \text{(because } [\![\tau]\!](l' \circ l) f_1 \text{ is definable at } \langle w'', i'', s'' \rangle\text{)} \\
= & [\![\mathbb{C}[t_1 y/x]]\!] s'' \rho[y \mapsto [\![\tau']\!] l'(a_2), \overline{w'' \mapsto i''(w'')}] \\
& \quad \text{(because } a_1 \approx_b^{i'} a_2\text{)} \\
= & [\![\mathbb{C}]\!] s'' \rho[x \mapsto ([\![\tau]\!](l' \circ l) f_1) s''(\mathbf{id}_{s''}, [\![\tau]\!] l'(a_2)), \overline{w'' \mapsto i''(w'')}] \\
= & [\![\mathbb{C}[zt/x]]\!] s'' \rho[z \mapsto [\![\tau]\!](l' \circ l) f_1, \overline{w'' \mapsto i''(w'')}] \\
& \quad \text{(because } [\![\tau']\!] l'(a_2) \text{ is definable at } \langle w'', i'', s'' \rangle\text{)} \\
= & [\![\mathbb{C}[zt/x]]\!] s'' \rho[z \mapsto [\![\tau]\!](l' \circ l) f_2, \overline{w'' \mapsto i''(w'')}] \\
& \quad \text{(because } f_1 \approx_{b \to \tau'}^{i} f_2\text{)} \\
= & [\![\mathbb{C}]\!] s'' \rho[x \mapsto [\![\tau']\!] l'(f_2 s'(l, a_2)), \overline{w'' \mapsto i''(w'')}]
\end{array}$$

so $f_1 s'(l, a_1) \approx_{\tau'}^{i';\kappa} f_2 s'(l, a_2)$. Again, because $[\![\tau]\!] l(f_1)$ and a_1 is definable at $\langle w', i', s' \rangle$, $f_1 s'(l, a_1) = [\![\tau]\!] l(f_1) s'(\mathbf{id}_{s'}, a_1)$ is definable at $\langle w', i', s' \rangle$ as well. Similarly, $f_2 s'(l, a_2)$ is definable. Then by induction, $f_1 s'(l, a_1)\ \mathcal{R}_{\tau'}^{i'}\ f_2 s'(l, a_2)$, hence $f_1\ \mathcal{R}_{b \to \tau'}^{i}\ f_2$ since a_1, a_2 and (j, l) are taken arbitrarily. ☐

In the previous proposition, we have obtained completeness for a logical relation defined at base types by $\mathcal{R}_b^i = \sim_b^{i,\kappa}$. Let us now come back to the cryptographic logical relation defined in Chapter 5. Now the question is: for every base type b, does it hold that

$$\sim_b^{i,\kappa} = \boldsymbol{\mathcal{R}}_b^{i,\varphi}, \tag{6.2}$$

for every injection $i : w \to s$ in $\mathcal{I}$?

Lemma 6.5. *Assume that observation types have no junk, in the sense that every value of* $[\![o]\!] s$ *(*$o \in \mathbf{Obs}$*) is definable at every* $\langle w, i, s \rangle$*. Then* $\sim_o^{i,\kappa}$ *is equality on* $[\![o]\!] s$*, and* $\sim_{\mathsf{T}o}^{i,\kappa}$ *is equality on* $[\![\mathsf{T}o]\!] s$ *for any observation type* o*.*

Proof. Clearly $\sim_o^{i,\kappa}$ contains equality. Conversely, let $a_1, a_2 \in [\![o]\!] s$ such that $a_1 \sim_o^{i,\kappa} a_2$. Take (j, l) to be the identity morphism from $\langle w, i, s \rangle$ to itself and $\mathbb{C}$ to be the context $\mathtt{val}(c = x)$ (so that $\overline{w : \mathsf{key}}, x : o \vdash \mathbb{C} : \mathsf{Tbool}$ is derivable), where c is any term such that $\overline{w : \mathsf{key}} \vdash c : o$ is derivable, and expand the definition of $\approx_o^{i,\kappa}$: $a_1 = [\![c]\!] s[\overline{w \mapsto i(w)}]$ if and only if $a_2 = [\![c]\!] s[\overline{w \mapsto i(w)}]$. Since o contains no junk, and c is arbitrary, $a_1 = a_2$.

The argument is similar for $\sim^{i,\kappa}_{\mathsf{T}o}$, taking $\texttt{let}\ z \Leftarrow c\ \texttt{in}\ \texttt{val}\ (z = x)$ for $\mathbb{C}$ instead. We just have to prove that $\mathsf{T}o$ has no junk. For every observation type o, $[\![o]\!]$ is a constant functor, so the elements of $[\![\mathsf{T}o]\!]s$ are of the form $[s', b] = [\emptyset, b]$, where $b \in [\![o]\!](s + s') = [\![o]\!]s$. Given any element $[\emptyset, b]$ of $[\![\mathsf{T}o]\!]s$, since o has no junk, we may write b as the value of some term c, hence $[\emptyset, b]$ is the value of $\texttt{val}(c)$. □

In the cryptographic metalanguage, nat and bool are two observation types, then $\sim^{i,\kappa}_{\mathsf{nat}}$ and $\sim^{i,\kappa}_{\mathsf{bool}}$ are the identity, which are exactly what we define in the cryptographic logical relation. The following lemma shows that $\sim_{\mathsf{key}} = \boldsymbol{\mathcal{R}}_{\mathsf{key}}$ holds for type key as well.

Lemma 6.6. *Let $i : w \to s$ be an injection in $\mathcal{I}$ and κ be a context knowledge. Then for every pair of keys $k_1, k_2 \in [\![\mathsf{key}]\!]s$, $k_1 \sim^{i,\kappa}_{\mathsf{key}} k_2$ if and only if $k_1 = k_2 \in i(w)$.*

Proof. We first claim that $(\star_{\mathsf{key}})$: the only values k in $[\![\mathsf{key}]\!]s = s$ that are definable at $\langle w, i, s\rangle$ are the keys in $i(w)$. One first observes, by applying any morphism l from s to s that is the identity on $i(w)$, that the only possible exceptions k to $(\star_{\mathsf{key}})$ must be fix-points of l: letting $k = [\![t]\!]s[\overline{w \mapsto i(w)}]$,

$$\begin{aligned}
l(k) &= [\![\mathsf{key}]\!]l(k) = [\![\mathsf{key}]\!]l([\![t]\!]s[\overline{w \mapsto i(w)}]) \\
&= [\![t]\!]s[\overline{w \mapsto l(i(w))}] \qquad \text{(since } [\![t]\!] \text{ is natural)} \\
&= [\![t]\!]s[\overline{w \mapsto i(w)}] \qquad \text{(since } l \text{ is the identity on } i(w)) \\
&= k.
\end{aligned}$$

Since l is arbitrary such that it restricts to the identity on $i(w)$, $(\star_{\mathsf{key}})$ can only fail when s consists of $i(w)$ plus just the one extra key k. Let then s' be s plus another key k'. There is an obvious morphism (j, l) from $\langle w, i, s\rangle$ to $\langle w, i, s'\rangle$ and we have seen that in this case $[\![\mathsf{key}]\!]l(k) = k$ is again definable at $\langle w, i, s'\rangle$. But this is impossible, since s' contains *two* keys outside of $i(w)$.

If $k_1 \sim^{i,\kappa}_{\mathsf{key}} k_2$, then both k_1 and k_2 are definable at $\langle w, i, s\rangle$, so by $(\star_{\mathsf{key}})$, $k_1 = i(z_1)$ for some $z_1 \in w$, and $k_2 = i(z_2)$ for some $z_2 \in w$. Since $k_1 \approx^{i,\kappa}_{\mathsf{key}} k_2$, if $k_1 \neq k_2$, then the context

$$x : \mathsf{key}, \overline{w : \mathsf{key}} \vdash \texttt{case dec}(\{1\}_{z_1}, x)\ \texttt{of some(\_) in val}(1)\ \texttt{else val}(0) : \mathsf{T}\mathsf{nat}$$

can distinguish the two keys, hence $k_1 = k_2$.

Conversely, if $k_1 = k_2 \in i(w)$, i.e., there is a variable $z \in w$ such that $n_1 = n_2 = i(z)$, then clearly $k_1 \sim^{i,\kappa}_{\mathsf{key}} k_2$. □

For the msg type, the equation (6.2) depends on the context knowledge κ and the cipher function φ. Indeed, $\approx^{i,\kappa}_{\mathsf{msg}}$ involves the relation between secret messages in the knowledge κ, precisely, the identity relation on $\kappa^{\langle w,i,s\rangle}$ for every $i : w \to s$, but any secret message in κ^i is definitely not in $[\![\mathsf{msg}]\!](i(w))$, hence according to Lemma 6.2, it is not definable at $\langle w, i, s\rangle$. So interestingly, it holds indeed that

$$\sim^{i,\kappa}_{\mathsf{msg}} \subseteq \mathcal{MR}^{i,\varphi},$$

for any cipher function φ.

Lemma 6.7. *Let κ be a context knowledge and $i : w \to s$ be an injection in $\mathcal{I}$. For every pair of values $a_1, a_2 \in [\![\mathsf{msg}]\!]s$, if a_1, a_2 are definable at $\langle w, i, s\rangle$ and $a_1 \approx^{i,\kappa}_{\mathsf{msg}} a_2$, then $(a_1, a_2) \in \mathcal{MR}^{i,\varphi}$, for any cipher function φ.*

Proof. Because $a_1 \approx^{i}_{\tau} a_2$, a_1 and a_2 must have the same head message token (*n*, *k*, *p* or *e*), i.e., the tokens at the roots of the two message trees are identical. Otherwise, it is easy to find a context to distinguish these two messages. For instance, if $a_1 = n(\_)$ and $a_2 = e(\_,\_)$, then the context

$$x : \mathsf{msg}, \overline{w : \mathsf{key}} \vdash \mathtt{case\ getnum}(x)\ \mathtt{of\ some}(\_)\ \mathtt{in\ val}(1)\ \mathtt{else\ val}(0) : \mathsf{Tnat}$$

can distinguish them.

Assume that a_1 and a_2 are definable at $\langle w, i, s\rangle$ but $(a_1, a_2) \notin \mathcal{MR}^{i,\varphi}$. We then prove that $a_1 \not\approx^{i,\kappa}_{\mathsf{msg}} a_2$, by induction on the message structure:

- If $a_1 = n(n_1)$ and $a_2 = n(n_2)$ for some $n_1, n_2 \in Nat$, because $(a_1, a_2) \notin \mathcal{MR}^{i,\varphi}$, it holds that $n_1 \neq n_2$. Then the context

  $$x : \mathsf{msg}, \overline{w : \mathsf{key}} \vdash \mathtt{case\ getnum}(x)\ \mathtt{of\ some}(y)\ \mathtt{in\ val}(y+1)\ \mathtt{else\ val}(0) : \mathsf{Tnat}$$

  can distinguish a_1 and a_2.

- If $a_1 = k(k_1)$ and $a_2 = k(k_2)$, because they are definable at $\langle w, i, s\rangle$, $k_1, k_2 \in i(w)$, so there exist two variables $z_1, z_2 \in w$ such that $i(z_1) = k_1$ and $i(z_2) = k_2$. Since $(a_1, a_2) \notin \mathcal{MR}^{i,\varphi}$, $k_2 \neq k_2$, the context

  $$\begin{array}{rcl} x : \mathsf{msg}, \overline{w : \mathsf{key}} & \vdash & \mathtt{case\ getkey}(x)\ \mathtt{of\ some}(y) \\ & & \mathtt{in\ case\ dec}(\{1\}_{z_1}, y)\ \mathtt{of\ some}(\_) \\ & & \quad \mathtt{in\ val}(1)\ \mathtt{else\ val}(0) \\ & & \mathtt{else\ val}(0) : \mathsf{Tnat} \end{array}$$

  can be used to distinguish these two messages.

- If $a_1 = p(a_1', a_1'')$ and $a_2 = p(a_2', a_2'')$, then either $(a_1', a_2') \notin \mathcal{MR}^{i,\varphi}$ or $(a_1'', a_2'') \notin \mathcal{MR}^{i,\varphi}$. Without loss of generality, suppose that $(a_1', a_2') \notin \mathcal{MR}^{i,\varphi}$. By induction, there is a context

$\mathbb{C}$ such that $y : \mathsf{msg}, \overline{w : \mathsf{key}} \vdash \mathbb{C} : \mathsf{Tnat}$ is derivable and $[\![\mathbb{C}]\!]si[y \mapsto a'_1] \neq [\![\mathbb{C}]\!]si[y \mapsto a'_2]$. Then the program

$$x : \mathsf{msg}, \overline{w : \mathsf{key}} \vdash \mathtt{case}\ \mathtt{fst}(x)\ \mathtt{of}\ \mathtt{some}(y)\ \mathtt{in}\ \mathbb{C}\ \mathtt{else}\ \mathtt{val}(0) : \mathsf{Tnat}$$

can be used to distinguish a_1 and a_2.

- If $a_1 = \mathsf{e}(a'_1, k_1)$ and $a_2 = \mathsf{e}(a'_2, k_2)$, because a_1, a_2 are definable at $\langle w, i, s\rangle$, k_1 and k_2 must be in $i(w)$, then either $k_1 \neq k_1$ or $(a'_1, a'_2) \notin \mathcal{MR}^{i,\varphi}$. If $k_1 \neq k_2$, the program

$$\begin{array}{rcl} x : \mathsf{msg}, \overline{w : \mathsf{key}} & \vdash & \mathtt{case}\ \mathtt{dec}(x, z_1)\ \mathtt{of}\ \mathtt{some}(\_) \\ & & \mathtt{in}\ \mathtt{val}(1)\ \mathtt{else}\ \mathtt{val}(0) : \mathsf{Tnat}, \end{array}$$

where z_1 is a variable in w and $i(z_1) = k_1$, can be used to distinguish a_1 and a_2; if $k_1 = k_2$ but $(a_1, a_2) \notin \mathcal{MR}^{i,\varphi}$, by induction, there is a context $\mathbb{C}$ such that $y : \mathsf{msg}, \overline{w : \mathsf{key}} \vdash \mathbb{C} : \mathsf{Tnat}$ is derivable and $[\![\mathbb{C}]\!]si[y \mapsto a'_1] \neq [\![\mathbb{C}]\!]si[y \mapsto a'_2]$. Then the program

$$x : \mathsf{msg}, \overline{w : \mathsf{key}} \vdash \mathtt{case}\ \mathtt{fst}(x)\ \mathtt{of}\ \mathtt{some}(y)\ \mathtt{in}\ \mathbb{C}\ \mathtt{else}\ \mathtt{val}(0) : \mathsf{Tnat}$$

can be used to distinguish a_1 and a_2. □

Whereas $\sim^{i,\kappa}_{\mathsf{msg}} \subseteq \mathcal{MR}^{i,\varphi}$ holds for any cipher function φ, the Lemma 6.2 shows that any definable messages (at $\langle w, i, s\rangle$) must be in $[\![\mathsf{msg}]\!](i(w))$, but a non-empty cipher function φ necessarily involves secret messages defined over keys that are not in $i(w)$, hence $\sim^{i,\kappa}_{\mathsf{msg}} = \mathcal{MR}^{i,\varphi}$ holds only if the cipher function φ is *empty*. Otherwise, assume that there are two keys $k_1, k_2 \in s - i(w)$ such that $\varphi^i(k_1, k_2) \neq \emptyset$ and take two messages a_1, a_2 such that $(a_1, a_2) \in \varphi^i(k_1, k_2)$, then clearly, $(\mathsf{e}(a_1, k_1), \mathsf{e}(a_2, k_2)) \in \mathcal{MR}^{i,\varphi}$, but they are not definable at $\langle w, i, s\rangle$, hence $\mathsf{e}(a_1, k_1) \not\sim^{i,\kappa}_{\mathsf{msg}} \mathsf{e}(a_2, k_2)$.

Proposition 6.8. *Let κ be a context knowledge and $i : w \to s$ be an injection in $\mathcal{I}$. For every pair of values $a_1, a_2 \in [\![\mathsf{msg}]\!]s$, $\sim^{i,\kappa}_{\mathsf{msg}} = \mathcal{MR}^{i,\varphi}$ holds if and only if the cipher function φ is empty.*

Proof. Lemma 6.7 shows that $\sim^{i,\kappa}_{\mathsf{msg}} \subseteq \mathcal{MR}^{i,\varphi}$ for any cipher function φ. We shall show that $\mathcal{MR}^{i,\varphi} \subseteq \sim^{i,\kappa}_{\mathsf{msg}}$ holds if and only if φ is empty.

The "only if" direction is obvious. For the "if" direction, take any two messages $a_1, a_2 \in \mathcal{MR}^{i,\emptyset}$ (we write $\mathcal{MR}^i$ for short). It is clear that a_1 and a_2 must have the same head message token (*n*, *k*, *p* or *e*). We can then prove it by induction on the structure of a_1 and a_2. For instance, if $a_1 = \mathsf{e}(a'_1, k_1)$ and $a_2 = \mathsf{e}(a'_2, k_2)$, then clearly, $k_1 = k_2 \in i(w)$ and $(a'_1, a'_2) \in \mathcal{MR}^i$. By induction, $a'_1 \sim^{i,\kappa}_{\mathsf{msg}} a'_2$, hence a_1, a_2 are definable at $\langle w, i, s\rangle$. Furthermore, take any context $\mathbb{C}$

such that $x : \mathsf{T}\tau', \overline{w : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash \mathbb{C} : \mathsf{Tnat}$ is derivable, for any morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$ and any κ-honest environment ρ,

$$\begin{aligned}
& [\![\mathbb{C}]\!] s'\rho[x \mapsto [\![\mathsf{msg}]\!] l(e(a'_1, k_1)), \overline{w' \mapsto i'(w')}] \\
= \; & [\![\mathbb{C}[\mathtt{enc}(y, z)/x]]\!] s'\rho[y \mapsto [\![\mathsf{msg}]\!] l(a'_1), \overline{w' \mapsto i'(w')}] \\
& \quad \text{(where } z \in w' \text{ and } i'(z) = l(k_1) = l(k_2)\text{)} \\
= \; & [\![\mathbb{C}[\mathtt{enc}(y, z)/x]]\!] s'\rho[y \mapsto [\![\mathsf{msg}]\!] l(a'_2), \overline{w' \mapsto i'(w')}] \\
& \quad \text{(because } a'_1 \approx^{i,\kappa}_{\mathsf{msg}} a'_2\text{)} \\
= \; & [\![\mathbb{C}]\!] s'\rho[x \mapsto [\![\mathsf{msg}]\!] l(e(a'_2, k_2)), \overline{w' \mapsto i'(w')}],
\end{aligned}$$

so $a_1 \approx^{i,\kappa}_{\mathsf{msg}} a_2$, and consequently $a_1 \sim^{i,\kappa}_{\mathsf{msg}} a_2$. □

6.3 Complétude pour les types monadiques

Completeness for monadic relations are much subtler — we have seen already a counter-example where two contextually equivalent computations of a zero-order type cannot be related by the logical relations derived over $\mathcal{S}et^{\mathcal{PI}^{\to}}$.

However, even though we consider only computations that return directly concrete values, i.e., those "one-level" computations, the completeness of monadic logical relations is still hard to prove. The general induction step, i.e.,

$$\approx^{i,\kappa}_{\tau} \subseteq \mathcal{R}_\tau \implies \approx^{i,\kappa}_{\mathsf{T}\tau} \subseteq \mathcal{R}_{\mathsf{T}\tau}$$

is difficult to achieve. For instance, let $[s_1, a_1], [s_2, a_2] \in [\![\mathsf{T}\tau]\!] s$ be two computations such that $[s_1, a_1] \approx^{i,\kappa}_{\mathsf{T}\tau} [s_2, a_2]$, and assume that they are not related. According to the logical relation for computations, a natural thought is to show that this implies that their values are not related (if these computations finally return some values), from which, by induction, we obtain a context $\mathbb{C}$ that distinguishes the values. Then our next step is to construct another context $\mathbb{C}'$ from $\mathbb{C}$ such that $\mathbb{C}'$ can distinguish the two computations. This is a general technique to prove completeness for monadic types and it does work for most monads (see Appendix B for examples).

However, this technique does not work for the key generation monad. The obstacle is the construction of a context for computations from contexts for values. For this form of computations, the definability of contexts interferes with such constructions. Actually, we are not able to do this kind of constructions. For example, in the above two computations, the two values a_1 and a_2 are defined at $s + s_0$ (for simplicity, we assume that $s_1 = s_2 = s_0$). If there is a context that can distinguish these two values, it must be defined at a larger world. Without loss of generality, let us just consider a context $\mathbb{C}$ defined over $s + s_0$, but then it is very difficult to construct another context $\mathbb{C}'$ (for computations) from $\mathbb{C}$, because this requires that $\mathbb{C}'$ is also defined at the world $s+s_0$, while in the case of key generation, values will be defined over $s+s_0+s_0$, not containing any keys in the first s_0, and we might not be able to make use of the original context $\mathbb{C}$, which never involves any key in the second s_0.

We shall next investigate the completeness of monadic logical relations for some specific types. Consider computations of type $\mathsf{T}\tau$ where τ is from a subset of plain zero-order types, defined by

$$\tau ::= \mathsf{nat} \mid \mathsf{bool} \mid \mathsf{key} \mid \tau \times \tau \mid \mathsf{opt}[\tau]. \tag{6.3}$$

In particular, we do not consider the msg type, which we shall discuss later. Indeed, without msg type, we can simply take the contextual equivalence for key generation (Definition 6.1).

Proposition 6.9. *Let $i : w \to s$ be an injection in $\mathcal{I}$. There exists a logical relation $(\mathcal{R}_\tau)_{\tau \text{ type}}$, derived over $\mathcal{S}et^{\mathcal{PI}^{\to}}$, such that for every pair of computations $[s_1, a_1]$ and $[s_2, a_2]$ (both in $[\![\mathsf{T}\tau]\!]s$, where τ are types defined by (6.3)),*

$$[s_1, a_1] \approx^i_{\mathsf{T}\tau} [s_2, a_2] \;\Rightarrow\; [s_1, a_1]\ \mathcal{R}^i_{\mathsf{T}\tau}\ [s_2, a_2].$$

Proof. Let $(\mathcal{R}_\tau)_{\tau \text{ type}}$ be a logical relation induced by $\mathcal{R}^i_b = \sim^{i,\kappa}_b$ for every injection $i : w \to s$ in $\mathcal{I}$, derived over the category $\mathcal{S}et^{\mathcal{PI}^{\to}}$. Then $\mathcal{R}^i_{\mathsf{nat}}$ and $\mathcal{R}^i_{\mathsf{bool}}$ are just the identity, and $\mathcal{R}^i_{\mathsf{key}}$ is the partial identity over s, i.e., the identity over $i(w)$.

In a logical relation, $[s_1, a_1]\ \mathcal{R}^i_{\mathsf{T}\tau}\ [s_2, a_2]$ if and only if there are injections $i_0 : w_0 \to s_0$ and $l_1 : s_1 \to s_0$, $l_2 : s_2 \to s_0$ in $\mathcal{I}$ such that

$$[\![\tau]\!](\mathbf{id}_s + l_1)(a_1)\ \mathcal{R}^{i+i_0}_\tau\ [\![\tau]\!](\mathbf{id}_s + l_2)(a_2).$$

We then prove by induction on type τ. Without loss of generality, we assume that $|s_1| \geq |s_2|$. We simply take $s_0 = s_1$, $l_1 = \mathbf{id}_{s_1}$, $i_0 = \mathbf{id}_{s_0}$ and an injection from s_2 to s_1 as l_2.

- If τ is bool or nat, then $[s_1, a_1] = [s_2, a_2]$. Clearly, a_1 and a_2 must be identical.
- For type key, either both a_1 and a_2 are fresh or both are not fresh, otherwise (assume that a_1 is fresh and a_2 not, then there must be some variables $z \in w$ such that $i(z) = a_2$), the context

$$\overline{w : \mathsf{key}}, x : \mathsf{Tkey} \;\vdash\; \begin{array}{l} \mathtt{let}\ y \Leftarrow x\ \mathtt{in} \\ \mathtt{case}\ \mathtt{dec}(\mathtt{enc}(0, z), y)\ \mathtt{of}\ \mathtt{some}(\_) \\ \mathtt{in}\ \mathtt{val}(1)\ \mathtt{else}\ \mathtt{val}(0) : \mathsf{Tnat} \end{array}$$

  can distinguish the two computations. If a_1, a_2 are not fresh, then $a_1 = a_2 \in i(w)$ (otherwise the above program can also distinguish the two computations), hence

$$[\![\mathsf{key}]\!](\mathbf{id}_s + \mathbf{id}_{s_1})(a_1)\ \mathcal{R}^{i+\mathbf{id}_{s_1}}_{\mathsf{key}}\ [\![\mathsf{key}]\!](\mathbf{id}_s + l_2)(a_2)$$

holds for every injection $l_2 : s_2 \to s_1$. If a_1, a_2 are fresh, i.e., $a_1 \in s_1$ and $a_2 \in s_2$, for every injections $l_2 : s_2 \to s_1$ such that $l_2(a_2) = a_1$, it holds that

$$[\![\mathsf{key}]\!](\mathbf{id}_s + \mathbf{id}_{s_1})(a_1)\ \mathcal{R}^{i+\mathbf{id}_{s_1}}_{\mathsf{key}}\ [\![\mathsf{key}]\!](\mathbf{id}_s + l_2)(a_2).$$

- If $\tau \equiv \mathsf{opt}[\tau']$, either both a_1 and a_2 are $\bot$, or both are not $\bot$ (otherwise it is easy to find a program to distinguish the two computations). Clearly, in the former case, the two values are always related. In the latter case, $[s_1, a_1] \in [\![\mathsf{T}\tau']\!]s$ and $[s_2, a_2] \in [\![\mathsf{T}\tau']\!]s$. Take any context $\mathbb{C}$ such that $x : \mathsf{T}\tau', \overline{w : \mathsf{key}} \vdash \mathbb{C} : \mathsf{T}\mathtt{nat}$ is derivable, for every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$,

$$\begin{aligned}
& [\![\mathbb{C}]\!]s'i'[x \mapsto [\![\mathsf{T}\tau']\!]l[s_1, a_1]] \\
= \quad & [\![\mathbb{C}']\!]s'i'[y \mapsto [\![\mathsf{T}\tau]\!]l[s_1, a_1]] \qquad \text{(because } a_1 \neq \bot) \\
= \quad & [\![\mathbb{C}']\!]s'i'[y \mapsto [\![\mathsf{T}\tau]\!]l[s_2, a_2]] \qquad \text{(because } [s_1, a_1] \approx^i_{\mathsf{T}\tau} [s_2, a_2]) \\
= \quad & [\![\mathbb{C}]\!]s'i'[x \mapsto [\![\mathsf{T}\tau']\!]l[s_2, a_2]],
\end{aligned}$$

where

$$\mathbb{C}' \equiv \mathtt{let}\ z \Leftarrow y\ \mathtt{in}\ \mathtt{case}\ z\ \mathtt{of}\ \mathtt{some}(x)\ \mathtt{in}\ \mathbb{C}\ \mathtt{else}\ \mathtt{val}(0),$$

so $[s_1, a_1] \approx^i_{\mathsf{T}\tau'} [s_2, a_2]$ as well. Then by induction, there is an injection l_2 such that $[\![\tau']\!](\mathbf{id}_s + \mathbf{id}_{s_1})(a_1)\ \mathcal{R}^{i+\mathbf{id}_{s_1}}_{\tau'}\ [\![\tau']\!](\mathbf{id}_s + l_2)(a_2)$, hence it holds that

$$[\![\tau]\!](\mathbf{id}_s + \mathbf{id}_{s_1})(a_1)\ \mathcal{R}^{i+\mathbf{id}_{s_1}}_{\mathsf{opt}[\tau']}\ [\![\tau]\!](\mathbf{id}_s + l_2)(a_2).$$

- If $\tau \equiv \tau' \times \tau''$, then $a_1 = (a'_1, a''_1)$ and $a_2 = (a'_2, a''_2)$. Take any context $\mathbb{C}$ such that $x : \mathsf{T}\tau', \overline{w : \mathsf{key}} \vdash \mathbb{C} : \mathsf{T}\mathtt{nat}$ is derivable, for every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$,

$$\begin{aligned}
& [\![\mathbb{C}]\!]s'i'[x \mapsto [\![\mathsf{T}\tau']\!]l[s_1, a'_1]] \\
= \quad & [\![\mathbb{C}[\mathtt{let}\ z \Leftarrow y\ \mathtt{in}\ \mathtt{val}(\mathtt{proj}_1(z))/x]]\!]s'i'[y \mapsto [\![\mathsf{T}\tau]\!]l[s_1, (a'_1, a''_1)]] \\
= \quad & [\![\mathbb{C}[\mathtt{let}\ z \Leftarrow y\ \mathtt{in}\ \mathtt{val}(\mathtt{proj}_1(z))/x]]\!]s'i'[y \mapsto [\![\mathsf{T}\tau]\!]l[s_2, (a'_2, a''_2)]] \\
& \qquad \text{(because } [s_1, (a'_1, a''_1)] \approx^i_{\mathsf{T}(\tau' \times \tau'')} [s_2, (a'_2, a''_2)]) \\
= \quad & [\![\mathbb{C}]\!]s'i'[x \mapsto [\![\mathsf{T}\tau']\!]l[s_2, a'_2]],
\end{aligned}$$

so $[s_1, a'_1] \approx^i_{\mathsf{T}\tau'} [s_2, a'_2]$. Similarly, $[s_1, a''_1] \approx^i_{\mathsf{T}\tau''} [s_2, a''_2]$. Then by induction, there are injections l'_2 and l''_2 such that

$$[\![\tau']\!](\mathbf{id}_s + \mathbf{id}_{s_1})(a'_1)\ \mathcal{R}^{i+\mathbf{id}_{s_1}}_{\tau'}\ [\![\tau']\!](\mathbf{id}_s + l'_2)(a'_2)$$

and

$$\llbracket \tau'' \rrbracket (\mathbf{id}_s + \mathbf{id}_{s_1})(a_1'') \; \mathcal{R}_{\tau''}^{i+\mathbf{id}_{s_1}} \; \llbracket \tau'' \rrbracket (\mathbf{id}_s + l_2'')(a_2'').$$

In particular, we can always define l_1' and l_2'' as the same injection, then

$$\llbracket \tau \rrbracket (\mathbf{id}_s + \mathbf{id}_{s_1})(a_1) \; \mathcal{R}_{\tau' \times \tau''}^{i+\mathbf{id}_{s_1}} \; \llbracket \tau \rrbracket (\mathbf{id}_s + l_2')(a_2).$$

□

Putting together all the results on completeness that have been presented in this section and in the previous section, we get the following theorem on the completeness of cryptographic logical relations:

Theorem 6.10 (Completeness of cryptographic logical relations). *The cryptographic logical relation as defined in Definition 5.2 is complete for types*

$$\tau_c^1 ::= b \mid \mathsf{T}\tau_c^0 \mid \tau_c^1 \times \tau_c^1 \mid \mathsf{opt}[\tau_c^1] \mid b \to \tau_c^1,$$

where $b \in \{\mathsf{bool}, \mathsf{nat}, \mathsf{key}, \mathsf{msg}\}$ *and* τ_c^0 *is defined by*

$$\tau_c^0 ::= \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{key} \mid \tau_c^0 \times \tau_c^0 \mid \mathsf{opt}[\tau_c^0],$$

in the sense that for every injection $i : w \to s$ *in* $\mathcal{I}$ *and every terms* t_1, t_2 *such that* $\overline{w : \mathsf{key}} \vdash t_1 : \tau_c^1$ *and* $\overline{w : \mathsf{key}} \vdash t_2 : \tau_c^1$ *are derivable, if* $t_1 \approx_{\tau_c^1}^{i,\emptyset} t_2$*, then* $\llbracket t_1 \rrbracket si \; \mathcal{R}_\tau^{i,\emptyset} \; \llbracket t_2 \rrbracket si$*.*

Proof. According to Proposition 6.4, Lemma 6.6, Proposition 6.7 and Proposition 6.9. □

A special type which is not included in τ_c^1 is Tmsg. Given two contextually equivalent computations of this type, to check whether they are related, we must construct the cipher function for those freshly generated keys that are not disclosed. We did not manage to get a formal proof of completeness for this type, but we conjecture that the cryptographic logical relation is complete on this type and we provide here an algorithm for constructing the cipher function.

Let $i : w \to s$ be an injection in $\mathcal{I}$ and $[s_1, a_1]$, $[s_2, a_2]$ be two computations in $\llbracket \mathsf{Tmsg} \rrbracket s$ such that $[s_1, a_1] \approx_{\mathsf{Tmsg}}^{i,\kappa} [s_2, a_2]$. Since we aim at constructing the cipher functions for those fresh keys, without loss of generality, we can simply let κ be an empty knowledge. Clearly, a_1 and a_2 must have the same head message token, just as shown in the proof of Lemma 6.7. We then show the existence of $i_0 : w_0 \to s_0$ and φ by executing the following "message-checking" algorithm with the pair of messages (a_1, a_2). In particular, we simply take i_0 as an inclusion. A state of this algorithm is a 4-tuple $\langle w_0, l_1, l_2, \varphi \rangle$, where l_1, l_2 will finally be defined as injections from s_1 and s_2 to s_0 respectively. We set the initial state of the algorithm with w_0 being an empty set, l_1 and l_2 being empty injections and φ being an empty cipher functions.

- If a_1, a_2 are not of the same form, i.e., with different head message token, then stop with error.

- If $a_1 = n(n_1)$ and $a_2 = n(n_2)$, then if $n_1 = n_2$, do nothing and stop normally; otherwise, stop with error.

- If $a_1 = k(k_1)$ and $a_2 = k(k_2)$, then if $k_1 = k_2 \in i(w)$, stop normally; if $k_1, k_2 \notin s$ and $l_1(k_1) = l_2(k_2)$ are already defined, stop normally; if $k_1, k_2 \notin s$ and $l_1(k_1), l_2(k_2)$ are not defined, let $l_1(k_1) = l_2(k_2) = k$ for some $k \notin s + w_0$ and $w_0 := w_0 + \{k\}$. If $\varphi(k_1, k_2) \neq \emptyset$, then for every $(a_1', a_2') \in \varphi(k_1, k_2)$, execute the "message-checking" algorithm with (a_1', a_2') at current state. If every execution stops normally, then let $\varphi(k_1, k_2) := \emptyset$ and stop normally; in any other case, stop with error.

- If $a_1 = e(a_1', k_1)$ and $a_2 = e(a_2', k_2)$, then if $k_1, k_2 \notin i(w) + w_0$ and k_1, k_2 are not defined by l_1 and l_2, then let $\varphi(k_1, k_2) := \varphi(k_1, k_2) \cup \{(a_1', a_2')\}$ and stop normally; if $k_1 = k_2 \in i(w) + w_0$ or $l_1(k_1) = l_2(k_2) \in w_0$, then execute the "message-checking" algorithm with messages a_1' and a_2' at the current state. if it stops normally, then stop normally; otherwise, stop with error.

- If $a_1 = p(a_1', a_1'')$ and $a_2 = p(a_2', a_2'')$, then execute the "message-checking" algorithm with message pairs (a_1', a_2') and (a_1'', a_2''), at current state. If both stop normally, with states $\langle w_0', l_1', l_2', \varphi' \rangle$ and $\langle w_0'', l_1'', l_2'', \varphi'' \rangle$ respectively, then if these two states are exactly the same as the initial states of the two executions, then stop normally; if the states are different from initial states and they are consistent, then merge these two states into a new state and execute the "message-checking" algorithm with message pairs (a_1', a_2') and (a_1'', a_2''), at this new state; in any other case, stop with error. Two states s_1 and s_2 are called consistent if, for any keys $k_1 \in s_1$ and $k_2 \in s_2$ that are defined by l_1', l_1'' and l_2', l_2'' respectively, $l_1'(k_1) = l_2'(k_2)$ if and only if $l_1''(k_1) = l_2''(k_2)$. To merge two states in to a new state $\langle w_0, l_1, l_2, \varphi \rangle$ means that for every $k \in s_1$, if it is defined by l_1' then $l_1(k) = l_1'(k)$, otherwise, if it is defined by l_1'', then $l_1(k) = l_1''(k)$. Similar for defining l_2. Then w_0 is the set $\{l_1(k) \mid k \in s_1 \ \& \ k \text{ is defined by } l_1\}$. φ is the union of φ' and φ'' point-wisely, but with $\varphi(k_1, k_2) = \emptyset$ for every k_1 that is defined by l_1 and every k_2 that is defined by l_2.

This algorithm necessarily stops, since there are only finitely many fresh keys and messages are finite trees. It is clear that this algorithm just destructs the two messages with the same operation, so if it stops normally, then the messages must be related at the terminal state, defining l_1 and l_2 as the identity for every key that is not defined during the algorithm and s_0 as $l_1(s_1) \cup l_2(s_2)$. If it stops with error, then the execution can be encoded in the cryptographic metalanguage and the program can be used to distinguish these two messages, which is a contradiction of the hypothesis that $[s_1, a_1] \approx^{i,\kappa}_{\mathsf{T}\tau} [s_2, a_2]$.

6.4 Relations logiques lax complètes

To get the completeness w.r.t. contextual equivalence at any type, we shall appeal to the notion of lax logical relations [PPST00]. Recall the categorical construction of logical relations using subscones, via the following diagram:

$$\begin{tikzcd} & \boldsymbol{\lambda}(\Sigma) \arrow[dl, "\mathcal{R}"'] \arrow[d, "[\![\_]\!]_{\mathcal{C}}"] \\ \mathrm{Subscone}^{\mathcal{C}}_{\mathcal{C}} \arrow[r, "U"'] & \mathcal{C} \end{tikzcd} \tag{6.4}$$

For logical relations, $\mathcal{R}$ is a representation of CCCs, in which case, as we have seen in Chapter 4, this diagram necessarily commutes. *Lax* logical relations are just product preserving functors $\mathcal{R}$ such that Diagram (6.4) commutes [PPST00, Section 6]. The equality in Diagram 6.4 is the key to make $\mathcal{R}$ satisfy the basic lemma.

The main difference is that, with lax logical relations, we do not require $\mathcal{R}$ to be representations of CCCs, just product preserving functors. Furthermore, when we consider the monadic lambda-calculus, we say that $\mathcal{R}$ is *strict at monadic types* if the functor $\mathcal{R}$ also preserves the (strong) monad.

We then consider, for any injection $i : w \to s$ in $\mathcal{I}$ and any context knowledge κ, the relation $\sim^{i,\kappa}_{\tau}$: for every pair of values $a_1, a_2 \in [\![\tau]\!]s$, $a_1 \sim^{i,\kappa}_{\tau} a_2$ if and only if a_1 and a_2 are definable at $\langle w, i, s\rangle$ and $a_1 \approx^{i,\kappa}_{\tau} a_2$. We shall show that if the context knowledge κ is monotonic, then $\sim^{i,\kappa}_{\tau}$ is indeed a lax logical relation, defined over the category $\mathcal{S}et^{\mathcal{PI}^{\to}}$.

Lemma 6.11. *Let $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ be a morphism in $\mathcal{PI}^{\to}$ and κ be a monotonic context knowledge. For every pair of values $a_1, a_2 \in [\![\tau]\!]s$, $a_1 \approx^{i,\kappa}_{\tau} a_2$ if and only if $[\![\tau]\!]l(a_1) \approx^{i',\kappa}_{\tau} [\![\tau]\!]l(a_2)$.*

Proof. We first show the monotonicity property:

$$a_1 \approx^{i,\kappa}_{\tau} a_2 \;\Rightarrow\; [\![\tau]\!]l(a_1) \approx^{i',\kappa}_{\tau} [\![\tau]\!]l(a_2)$$

Take any term $\mathbb{C}$ such that $x : \tau, \overline{w'' : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash \mathbb{C} : \mathsf{Tbool}$ is derivable, for every morphism $(j', l') : \langle w', i', s'\rangle \to \langle w'', i'', s''\rangle$ in $\mathcal{PI}^{\to}$ and every κ-honest environment ρ:

$$\begin{array}{ll} & [\![\mathbb{C}]\!]s''\rho[x \mapsto [\![\tau]\!]l'([\![\tau]\!]l(a_1)), \overline{w'' \mapsto i''(w'')}] \\ = & [\![\mathbb{C}]\!]s''\rho[x \mapsto [\![\tau]\!](l' \circ l)(a_1), \overline{w'' \mapsto i''(w'')}] \\ & \qquad \text{(because } [\![\tau]\!] \text{ is a functor)} \\ = & [\![\mathbb{C}]\!]s''\rho[x \mapsto [\![\tau]\!](l' \circ l)(a_2), \overline{w'' \mapsto i''(w'')}] \\ & \qquad \text{(because } a_1 \approx^{i,\kappa}_{\tau} a_2 \text{ and } (j' \circ j, l' \circ l) \text{ is a morphism in } \mathcal{PI}^{\to}) \\ = & [\![\mathbb{C}]\!]s''\rho[x \mapsto [\![\tau]\!]l'([\![\tau]\!]l(a_2)), \overline{w'' \mapsto i''(w'')}], \end{array}$$

hence $[\![\tau]\!]l(a_1) \approx_\tau^{i',\kappa} [\![\tau]\!]l(a_2)$.

The anti-monotonicity is equivalent to:

$$a_1 \not\approx_\tau^{i,\kappa} a_2 \ \Rightarrow\ [\![\tau]\!]l(a_1) \not\approx_\tau^{i',\kappa} [\![\tau]\!]l(a_2).$$

Assume that $a_1 \not\approx_\tau^{i,\kappa} a_2$, so there is some term $\mathbb{C}$ such that

$$x : \tau, \overline{w_0 : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash \mathbb{C} : \mathsf{Tbool}$$

is derivable, with a morphism $(j_0, l_0) : \langle w, i, s\rangle \to \langle w_0, i_0, s_0\rangle$ in $\mathcal{PI}^\to$ and a κ-honest environment ρ, such that

$$[\![\mathbb{C}]\!]s_0\rho[x \mapsto [\![\tau]\!]l_0(a_1), w_0 \mapsto i_0(w_0)] \neq [\![\mathbb{C}]\!]s_0\rho[x \mapsto [\![\tau]\!]l_0(a_2), w_0 \mapsto i_0(w_0)]$$

By the *Cube property* (Proposition 5.4), we have a commuting square:

$$\begin{tikzcd}
 & & w \arrow[dll, "j"'] \arrow[d, "i"] \arrow[drr, "j_0"] & & \\
w' \arrow[d, "i'"'] \arrow[drr, "j'"] & & s' \arrow[dll, "l"'] \arrow[drr, "l_0"] & & w_0 \arrow[dll, "j'_0"'] \arrow[d, "i_0"] \\
s \arrow[drr, "l'"'] & & w'' \arrow[d, "i''"] & & s_0 \arrow[dll, "l'_0"] \\
 & & s'' & &
\end{tikzcd} \tag{6.5}$$

To prove $[\![\tau]\!]l(a_1) \not\approx_\tau^{i',\kappa} [\![\tau]\!]l(a_2)$, it is sufficient to check that, for some context $\mathbb{C}'$ such that

$$x : \tau, \overline{w'' : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash \mathbb{C}' : \mathsf{Tbool}$$

is derivable, the following holds for some κ-honest environment ρ',

$$\begin{aligned} & [\![\mathbb{C}']\!]s''\rho'[x \mapsto [\![\tau]\!]l'([\![\tau]\!]l(a_1)), w'' \mapsto i''(w'')] \\ \neq\ & [\![\mathbb{C}']\!]s''\rho'[x \mapsto [\![\tau]\!]l'([\![\tau]\!]l(a_2)), w'' \mapsto i''(w'')]. \end{aligned}$$

Take $\mathbb{C}' := \mathbb{C}[j'_0(w_0)/w_0]$ and $\rho' = [\![\Gamma]\!]l'_0(\rho)$. Because κ is monotonic, ρ' is still a κ-honest environment.

$$
\begin{aligned}
& \left[\!\left[\mathbb{C}'\right]\!\right] s''\rho'[x \mapsto [\![\tau]\!] l'([\![\tau]\!] l(a_1)), w'' \mapsto i''(w'')] \\
= \ & \left[\!\left[\mathbb{C}[j_0'(w_0)/w_0]\right]\!\right] s''\rho'[x \mapsto [\![\tau]\!](l' \circ l)(a_1), w'' \mapsto i''(w'')] \\
= \ & \left[\!\left[\mathbb{C}[j_0'(w_0)/w_0]\right]\!\right] s''\rho'[x \mapsto [\![\tau]\!](l_0' \circ l_0)(a_1), w'' \mapsto i''(w'')] \\
& \qquad \text{(by the commuting square (6.5))} \\
= \ & \left[\!\left[\mathbb{C}[j_0'(w_0)/w_0]\right]\!\right] s''\rho'[x \mapsto [\![\tau]\!] l_0'([\![\tau]\!] l_0(a_1)), w'' \mapsto i''(w'')] \\
= \ & [\![\mathbb{C}]\!] s''\rho'[x \mapsto [\![\tau]\!] l_0'([\![\tau]\!] l_0(a_1)), w_0 \mapsto [\![\mathsf{key}]\!] i''(j_0'(w_0))] \\
& \qquad \text{(renaming of free variables does not change the interpretation)} \\
= \ & [\![\mathbb{C}]\!] s''\rho'[x \mapsto [\![\tau]\!] l_0'([\![\tau]\!] l_0(a_1)), w_0 \mapsto i''(j_0'(w_0))] \\
& \qquad \text{(}[\![\mathsf{key}]\!] \text{ is an identity functor)} \\
= \ & [\![\mathbb{C}]\!] s''\rho'[x \mapsto [\![\tau]\!] l_0'([\![\tau]\!] l_0(a_1)), w_0 \mapsto l_0'(i_0(w_0))] \\
& \qquad \text{(again by the commuting square (6.5))} \\
= \ & [\![\mathsf{Tbool}]\!] l_0'([\![\mathbb{C}]\!] s_0\rho[x \mapsto [\![\tau]\!] l_0(a_1), w_0 \mapsto i_0(w_0)] \\
& \qquad \text{(by the naturality of } [\![\mathbb{C}]\!]\text{).}
\end{aligned}
$$

Similarly,

$$
\begin{aligned}
& \left[\!\left[\mathbb{C}'\right]\!\right] s''\rho'[x \mapsto [\![\tau]\!] l'([\![\tau]\!] l(a_2)), w'' \mapsto i''(w'')] \\
= \ & [\![\mathsf{Tbool}]\!] l_0'([\![\mathbb{C}]\!] s_0\rho[x \mapsto [\![\tau]\!] l_0(a_2), w_0 \mapsto i_0(w_0)].
\end{aligned}
$$

Since $[\![\mathsf{Tbool}]\!]$ is an identity functor,

$$
\begin{aligned}
& \left[\!\left[\mathbb{C}'\right]\!\right] s''\rho'[x \mapsto [\![\tau]\!] l'([\![\tau]\!] l(a_1)), w'' \mapsto i''(w'')] \\
\neq \ & \left[\!\left[\mathbb{C}'\right]\!\right] s''\rho'[x \mapsto [\![\tau]\!] l'([\![\tau]\!] l(a_2)), w'' \mapsto i''(w'')],
\end{aligned}
$$

hence $[\![\tau]\!] l(a_1) \not\approx_\tau^{i',\kappa} [\![\tau]\!] l(a_2)$. □

Theorem 6.12. *Lax logical relations are complete for contextual equivalence in the cryptographic metalanguage, in the strong sense that there is a lax logical relation $(\mathcal{R}_\tau)_{\tau \text{ type}}$ such that, for every injection $i : w \to s$ in $\mathcal{I}$ and every pair of terms t_1, t_2 such that $\overline{w : \mathsf{key}} \vdash t_1 : \tau$ and $\overline{w : \mathsf{key}} \vdash t_2 : \tau$ are derivable, $t_1 \approx_\tau^{i,\kappa} t_2$ if and only if $[\![t_1]\!] si\ \mathcal{R}_\tau^i\ [\![t_2]\!] si$, where κ is a monotonic context knowledge.*

Proof. Define $\mathcal{R}_\tau^{\langle w,i,s\rangle}$ as the relation $\sim_\tau^{\langle w,i,s\rangle,\kappa}$.

We first need to show that $\mathcal{R}_\tau$, mapping $\langle w, i, s\rangle$ to $\mathcal{R}_\tau^{\langle w,i,s\rangle}$, defines an object of $\mathcal{S}et^{\mathcal{PI}^{\to}}$, i.e., a functor from $\mathcal{PI}^{\to}$ to $\mathcal{S}et$. The action on morphisms (j, l) is given by our requirement that

$U \circ \mathcal{R} = [\![\_]\!]_1$, where $\mathcal{R}$ maps τ to $\mathcal{R}_\tau$, and $[\![\tau]\!]_1\langle w, i, s\rangle = [\![\tau]\!]s \times [\![\tau]\!]s$ and $[\![t]\!]_1\langle w, i, s\rangle = [\![t]\!]s \times [\![t]\!]s$. Expand the equation $U \circ \mathcal{R} = [\![\_]\!]_1$: $\mathcal{R}_\tau(j, l)$ must map $(a_1, a_2) \in [\![\tau]\!]s \times [\![\tau]\!]s$ to $([\![\tau]\!]l(a_1), [\![\tau]\!]l(a_2)) \in [\![\tau]\!]s' \times [\![\tau]\!]s'$. To check that $\mathcal{R}_\tau$ is a functor, we must check that if $a_1 \mathcal{R}^i_\tau a_2$, then $[\![\tau]\!]l(a_1) \mathcal{R}^{i'}_\tau [\![\tau]\!]l(a_2)$, for every morphism (j, l) in $\mathcal{PI}^\rightarrow$:

- First, a_1 and a_2 are definable at $\langle w, i, s\rangle$; by Lemma 6.3, $[\![\tau]\!]l(a_1)$ and $[\![\tau]\!]l(a_2)$ are definable at $\langle w', i', s'\rangle$.

- Second, $a_1 \approx^i_\tau a_2$ implies $[\![\tau]\!]k(a_1) \approx^{i'}_\tau [\![\tau]\!]k(a_2)$, according to Lemma 6.11.

Next, we need to show that $\mathcal{R}_\tau$ is the object part of a product-preserving functor $\mathcal{R}$ from $\boldsymbol{Comp}(\Sigma)$ to $\mathrm{Subscone}^{C}_{\mathcal{C}}$ such that $U \circ \mathcal{R} = [\![\_]\!]_1$. This means showing that, for every typing context $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, for every type τ such that $\Gamma \vdash t : \tau$ is derivable, for every object $\langle w, i, s\rangle \in \mathcal{PI}^\rightarrow$, if $a_m \mathcal{R}^i_{\tau_m} a'_m$ for every m ($1 \leq m \leq n$), then

$$[\![t]\!]s[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n] \mathcal{R}^i_\tau [\![t]\!]s[x_1 \mapsto a'_1, \ldots, x_n \mapsto a'_n].$$

Since both a_m and a'_m are definable at i, write $a_m = [\![t_m]\!]si$ for some t_m such that $\overline{w : \mathsf{key}} \vdash t_m : \tau_m$ is derivable, and similarly $a'_m = [\![t'_m]\!]si$. Then, it is clear that $[\![t]\!]s[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$ is definable at i, by the term $t[t_1/x_1, \ldots, t_n/x_n]$, and similarly for $[\![t]\!]s[x_1 \mapsto a'_1, \ldots, x_n \mapsto a'_n]$. Second, take any $\mathbb{C}$ such that

$$x : \tau, \overline{w' : \mathsf{key}}, \overline{v : \mathsf{msg}} \vdash \mathbb{C} : \mathsf{Tbool}$$

is derivable, for every morphism $(j, l) : \langle w, i, s\rangle \rightarrow \langle w', i', s'\rangle$ in $\mathcal{PI}^\rightarrow$, for every κ-honest environment ρ

$$
\begin{aligned}
& [\![\mathbb{C}]\!]s'\rho[x \mapsto [\![\tau]\!]l([\![t]\!]s[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]), \overline{w' \mapsto i'(w')}] \\
= \; & [\![\mathbb{C}]\!]s'[x \mapsto [\![t]\!]s'\rho[x_1 \mapsto [\![\tau_1]\!]l(a_1), \ldots, x_n \mapsto [\![\tau_n]\!]l(a_n)], \overline{w' \mapsto i'(w')}] \\
& \qquad \text{(since } [\![t]\!] \text{ is a natural transformation)} \\
= \; & [\![\mathbb{C}[t/x]]\!]s'\rho[x_1 \mapsto [\![\tau_1]\!]l(a_1), \ldots, x_n \mapsto [\![\tau_n]\!]l(a_n), \overline{w' \mapsto i'(w')}] \\
= \; & \left[\!\!\left[\mathbb{C}[t/x][t_1 j^{-1}/x_1, \ldots, t_{n-1} j^{-1}/x_{n-1}]\right]\!\!\right] s'[x_n \mapsto [\![\tau_n]\!]l(a_n), \overline{w' \mapsto i'(w')}] \\
= \; & \left[\!\!\left[\mathbb{C}[t/x][t_1 j^{-1}/x_1, \ldots, t_{n-1} j^{-1}/x_{n-1}]\right]\!\!\right] s'[x_n \mapsto [\![\tau_n]\!]l(a'_n), \overline{w' \mapsto i'(w')}] \\
& \text{(because } a_n \approx_{\tau_n}^{w \xrightarrow{i} s} a'_n) \\
= \; & \ldots \\
= \; & \left[\!\!\left[\mathbb{C}[t/x][t_1 j^{-1}/x_1, \ldots, t_{m-1} j^{-1}/x_{m-1}, t'_{m+1} j^{-1}/x_{m+1}, \ldots, t'_n j^{-1}/x_n]\right]\!\!\right] s' \\
& \qquad [x_m \mapsto [\![\tau_m]\!]l(a_m), \overline{w' \mapsto i'(w')}] \\
= \; & \left[\!\!\left[\mathbb{C}[t/x][t_1 j^{-1}/x_1, \ldots, t_{m-1} j^{-1}/x_{m-1}, t'_{m+1} j^{-1}/x_{m+1}, \ldots, t'_n j^{-1}/x_n]\right]\!\!\right] s' \\
& \qquad [x_m \mapsto [\![\tau_m]\!]l(a'_m), \overline{w' \mapsto i'(w')}] \qquad \text{(because } a_m \approx_{\tau_m}^{w \xrightarrow{i} s} a'_m) \\
= \; & \ldots \\
= \; & \left[\!\!\left[\mathbb{C}[t/x][t'_2 j^{-1}/x_2, \ldots, t'_n j^{-1}/x_n]\right]\!\!\right] s'[x_1 \mapsto [\![\tau_1]\!]l(a'_1), \overline{w' \mapsto i'(w')}] \\
= \; & [\![\mathbb{C}[t/x]]\!]s'\rho[x_1 \mapsto [\![\tau_1]\!]l(a'_1), \ldots, x_n \mapsto [\![\tau_n]\!]l(a'_n), \overline{w' \mapsto i'(w')}] \\
= \; & [\![\mathbb{C}]\!]s'\rho[x \mapsto [\![t]\!]s'[x_1 \mapsto [\![\tau_1]\!]l(a'_1), \ldots, x_n \mapsto [\![\tau_n]\!]l(a'_n)]), \overline{w' \mapsto i'(w')}] \\
= \; & [\![\mathbb{C}]\!]s'\rho[x \mapsto [\![\tau]\!]l([\![t]\!]s[x_1 \mapsto a'_1, \ldots, x_n \mapsto a'_n]), \overline{w' \mapsto i'(w')}]
\end{aligned}
$$

Here we notice that, since a_m and a'_m are definable at i by t_m and t'_m, respectively, then $[\![\tau_m]\!]l(a_m)$ and $[\![\tau_m]\!]l(a'_m)$ are definable at i' by $t_m j^{-1}$, $t'_m j^{-1}$, respectively. So

$$[\![t]\!]s[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n] \; \mathcal{R}^i_\tau \; [\![t]\!]s[x_1 \mapsto a'_1, \ldots, x_n \mapsto a'_n].$$

$\mathcal{R}$ is a lax logical relation since $U \circ \mathcal{R} = [\![\_]\!]_1$ by construction. ☐

The (non-lax) logical relations (the cryptographic logical relations) are defined on key by: $k_1 \; \mathcal{R}^{\langle w,i,s\rangle}_{\mathsf{key}} \; k_2$ if and only if $k_1 = k_2 \in i(w)$. Hence, by Lemma 6.6 the lax logical relation $\sim^{i,\kappa}_\tau$ and the logical relation coincide at type key type.

For soundness, although $\sim^{i,\kappa}_{\mathsf{T}o}$ is the identity on $[\![\mathsf{T}o]\!]s$, because a non-empty knowledge κ introduces messages that are not definable $\langle w, i, s\rangle$, we are not able to apply the Basic Lemma. But if κ is empty, it is then sound for contextual equivalence since we do not need to consider contexts with free message variables. Indeed, by the basic lemma $U \circ \mathcal{R} = [\![\_]\!]_1$, whenever $a_1 \sim^{i,\emptyset}_\tau a_2$ ($\emptyset$ denoting the empty knowledge), then for any $\mathbb{C}$ such that $x : \tau, \overline{w' : \mathsf{key}} \vdash \mathbb{C} : \mathsf{T}o$

($o \in \mathbf{Obs}$) is derivable, for any morphism $(j,l) : \langle w, i, w\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$,

$$[\![\mathbb{C}]\!]s'[\overline{w_1 := i_1(w_1)}, x := [\![\tau]\!]l_1(a_1)] \sim^{i',\emptyset}_{\mathsf{T}o} [\![\mathbb{C}]\!]s'[\overline{w_1 := i_1(w_1)}, x := [\![\tau]\!]l_1(a_2)];$$

so $a_1 \approx^{i,\emptyset}_{\tau} a_2$.

The lax logical relation $\sim^{i,\kappa}_{\tau}$ is also monotonic:

Proposition 6.13. *Let κ be a monotonic context knowledge. The relation $\sim^{i,\kappa}_{\tau}$ is monotonic: for every morphism $(j,l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$ and every pair of values $a_1, a_2 \in [\![\tau]\!]s$,*

$$a_1 \sim^{i}_{\tau} a_2 \Longrightarrow [\![\tau]\!]l(a_1) \sim^{i',\kappa}_{\tau} [\![\tau]\!]l(a_2).$$

Proof. According to Lemma 6.3, if $a_1, a_2 \in [\![\tau]\!]s$ are definable at $\langle w, i, s\rangle$, $[\![\tau]\!]l(a_1)$ and $[\![\tau]\!]l(a_2)$ are also definable at $\langle w', i', s'\rangle$. Again by Lemma 6.11, $a_1 \approx^{i,\kappa}_{\tau} a_2$ implies $[\![\tau]\!]l(a_1) \approx^{i',\kappa}_{\tau} [\![\tau]\!]l(a_2)$, so $a_1 \sim^{i','know}_{\tau} a_2$. $\square$

Chapitre 7

Décidabilité de l'équivalence contextuelle

Étant donnés deux programmes, le problème consistant à savoir s'ils sont contextuellement équivalent est-il décidable ? Nous répondrons à cette question dans certains cas. Les relations logiques caractérisent l'équivalence contextuelle pour un ensemble de types, comme nous avons vu dans le chapitre précédant, donc il est naturel d'étudier d'abord le problème de savoir si deux valeurs arbitraires données sont reliées. Plus précisément, étant donnés une relation logique $(\mathcal{R}_\tau)_{\tau \text{ type}}$ et deux valeurs d'un type τ, est-ce que nous pouvons décider si elles sont reliées par la relation $\mathcal{R}_\tau$? Même si ceci est décidable pour toutes les types de base, il est difficile de décider si des valeurs d'un type complexe sont reliées, notamment les fonctions. Ce n'est pas étonnant puisque la définition des relations entre fonctions comporte une quantification universelle sur les paramètres reliés de ces fonctions — notamment lorsque l'espace de ces paramètres est infini.

Les types monadiques sont des types complexes particuliers du lambda-calcul computationnel. Les définitions concrètes de relations logiques pour les types monadiques varient beaucoup selons les différentes formes d'effets de bord et il est souvent très difficile d'étudier leurs propriétés d'une façon générale, ainsi que la décidabilité. Nous nous concentrons sur les relations logiques de la monade de la génération de clés, dérivées sur la catégorie $\mathcal{S}et^{\mathcal{PI}^{\to}}$. C'est aussi un cas difficile parce que la définition des relations pour les types monadiques comporte aussi une recherche d'un “monde” convenable dans un espace infini.

Nous étudierons la décidabilité dans plusieurs cas, pour les relations logiques dérivées sur la catégorie $\mathcal{S}et^{\mathcal{PI}^{\to}}$. Comme ce sont des relations logiques de Kripke, le problème de la décidabilité se divise en deux cas :

– étant donné un monde et un type, est-il décidable de vérifier si deux valeurs sont reliées à ce monde spécifique ?

- sans fixer le monde, l'existence d'un monde auquel les deux valeurs données sont reliées est-elle décidable ?

Nous nous concentrons sur le premier cas où nous supposons toujours que le monde est donné. Une technique générale pour cette étude est de se ramener à un cas où la quantification porte sur des éléments pris dans un espace fini.

Nous classifions notre étude dans ce chapitre selon les types. En particulier, nous explorerons la décidabilité du problème qui consiste à relier deux fonctions (dans la partie 7.1) et à relier deux éléments d'un type monadique (dans la partie 7.2). La dernière partie résume et présente un résultat de décidabilité de l'équivalence contextuelle du métalangage cryptographique, en considérant la complétude des relations logiques dérivées sur $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. Nous montrons aussi qu'il est en général indécidable de vérifier si deux programmes sont équivalents, en codant une machine à deux compteurs dans le métalangage.

Given two programs, is it decidable whether they are contextually equivalent or not? In this chapter, we shall try to answer this question for certain cases. Since logical relations can identify the contextual equivalence for certain types, as discussed in last chapter, it is natural to start by exploring the decidability of the problem of determining whether two values are related. In other words, given a logical relation $(\mathcal{R}_\tau)_{\tau \text{ type}}$ and two values of a type τ, is it decidable whether they are related by the relation $\mathcal{R}_\tau$? This problem depends indeed on the decidability of relating values of base types, since logical relations are defined inductively from relations for base types, but even if all relations for base types are decidable, it is still very hard to decide relations for complex types. The main obstacle is relations for functions, because they include universal quantifications over all related arguments, which are usually infinitely many, especially when arguments themselves are again functions.

Monadic types are special complex types in the computational lambda-calculus. Concrete relations for monadic types vary a lot when the monad is specialized in different forms of computation and it is usually very difficult to study their properties — including decidability — in a general way. We focus on logical relations for the dynamic key generation monad, derived over $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. This is indeed a difficult case, because the definition of monadic relations involves also an existential quantification over an infinite space, namely searching some object $\langle w_0, i_0, s_0 \rangle$ in $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ to render uniform the two sets of fresh keys.

We shall study several cases of decidability for logical relations derived over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. However, logical relations derived over $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ are necessarily Kripke logical relations, and the decidability problem is divided into two cases for this kind of logical relations:

- given a world as well as a type, is it decidable whether two values are related at this specific world?

- or without fixing the world, is it decidable whether there exist a world such that the given two values can be related?

we shall focus on the first question where we always assume that the world is given. A common technique used in our proofs is to restrict quantifications over an infinite space to ones over a finite one.

The cases that we shall discuss in this chapter are classified by types. In particular, we shall investigate whether it is decidable of relating two values of a function type (Section 7.1) or a computation type (Section 7.2). In the last section, we summarize these cases, and we also show that the contextual equivalence is in general undecidable by encoding the 2-counter machine in the metalanguage.

7.1 Décidabilité dans le cas des fonctions

Deciding relations for functions is difficult, especially for higher-order functions, because we shall probably be forced to do the quantification over infinitely many related arguments. Furthermore, a denotational model such as $\mathcal{S}et^{\mathcal{I}}$ may contain junks, especially non-computable functions, which make it even more difficult to decide relations for higher-order functions. The situation becomes even worse for Kripke logical relations. In a Kripke logical relation, to check whether two functions are related at a certain world, we are required to check it with related arguments at *every larger world*.

We shall next study several cases of the decidability of relations for first-order functions, i.e., functions which accept only values of base types as arguments. A notable case is relations for types $\mathsf{key} \to \tau$.

First, say that b is a *regular base type* if and only if in the model $\mathcal{S}et^{\mathcal{I}}$, b can be interpreted as a constant functor from $\mathcal{I}$ to $\mathcal{S}et$, i.e., $[\![b]\!]s = [\![b]\!]s'$ for any $s, s' \in \mathcal{I}$, and for every value $a \in [\![b]\!]s$, $[\![b]\!]l(a) = a$ for any injection $l : s \to s' \in \mathcal{I}$. Two base types in the cryptographic metalanguage — the boolean type bool and the integer type nat — are typical regular base types.

Lemma 7.1. *Suppose that $i : w \to s$ is an injection in $\mathcal{I}$ and $(\mathcal{R}_\tau)_{\tau \text{ type}}$ is a monotonic logical relation derived over the category $\mathcal{S}et^{\mathcal{PI}^{\to}}$. For every type $b \to \tau$, where b is a regular base type, and every pair of functions $f_1, f_2 \in [\![b \to \tau]\!]s$, $f_1 \; \mathcal{R}^i_{b\to\tau} \; f_2$ if and only if, for every pair of values $a_1, a_2 \in [\![b]\!]s$,*

$$a_1 \; \mathcal{R}^i_b \; a_2 \Longrightarrow f_1 s(\mathbf{id}_s, a_1) \; \mathcal{R}^i_\tau \; f_2 s(\mathbf{id}_s, a_2).$$

Proof. The "only if" direction is obvious. We prove the "if" direction.

For any morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle \in \mathcal{PI}^{\to}$, and for every pair of values $a_1, a_2 \in [\![b]\!]s'$,

$$\begin{array}{ll}
 & a_1 \; \mathcal{R}^{i'}_b \; a_2 \\
\Longleftrightarrow & a_1 \; \mathcal{R}^i_b \; a_2 \\
 & \quad (a_1, a_2 \in [\![b]\!]s \text{ and } b \text{ is a regular base type}) \\
\Longrightarrow & f_1 s(\mathbf{id}_s, a_1) \; \mathcal{R}^i_\tau \; f_2 s(\mathbf{id}_s, a_2) \\
\Longrightarrow & [\![\tau]\!]l(f_1 s(\mathbf{id}_s, a_1)) \; \mathcal{R}^{i'}_\tau \; [\![\tau]\!]l(f_2 s(\mathbf{id}_s, a_2)) \\
 & \quad (\mathcal{R}_\tau \text{ is monotonic}) \\
\Longleftrightarrow & f_1 s'(l \circ \mathbf{id}_s, [\![b]\!]l(a_1)) \; \mathcal{R}^{i'}_\tau \; f_2 s'(l \circ \mathbf{id}_s, [\![b]\!]l(a_2)) \\
 & \quad (\text{by the naturality of } f_1 \text{ and } f_2) \\
\Longleftrightarrow & f_1 s'(l, a_1) \; \mathcal{R}^{i'}_\tau \; f_2 s'(l, a_2)
\end{array}$$

hence $f_1 \; \mathcal{R}^i_{b\to\tau} \; f_2$. $\square$

Clearly, if b is also a finite base type, i.e., $[\![b]\!]s$ is finite for any $s \in \mathcal{I}$, then the decidability of $\mathcal{R}_{b\to\tau}$ will depend on the decidability of $\mathcal{R}_\tau$.

Proposition 7.2. *Suppose that $i : w \to s$ is an injection in $\mathcal{I}$ and $(\mathcal{R}_\tau)_{\tau \text{ type}}$ is a monotonic logical relation derived over the category $\mathcal{S}et^{\mathcal{PI}^\to}$. For every type $b \to \tau$, where b is a regular finite base type, the relation $\mathcal{R}_{b\to\tau}$ is decidable, if and only if $\mathcal{R}_\tau$ is decidable.*

Proof. According to Lemma 7.1, to check whether two functions are related by $\mathcal{R}^i_{b\to\tau}$, we just consider related arguments at the world $\langle w, i, s\rangle$. Because b is a finite base types, $\mathcal{R}^i_b$ is finite as well, so we just apply the two functions to every pair of related values and see whether the results are related by $\mathcal{R}^i_\tau$, which is also decidable. □

As for the cryptographic logical relation as defined in Definition 5.2, we must moreover require the cipher function to be monotonic so that the above proposition holds.

The type key is a special base type. In the model $\mathcal{S}et^{\mathcal{I}}$, $[\![\mathsf{key}]\!]s$ varies as s varies. Although for a certain s, $[\![\mathsf{key}]\!]s$ is finite and the relation between keys are decidable, to relate two functions of a type $\mathsf{key} \to \tau$, we have to consider those related keys at every larger world, which are obviously infinite. Fortunately, this infinite quantification can be reduced to a finite one.

Lemma 7.3. *Suppose that $i : w \to s$ is an injection in $\mathcal{I}$ and $(\mathcal{R}_\tau)_{\tau \text{ type}}$ is a monotonic logical relation derived over the category $\mathcal{S}et^{\mathcal{PI}^\to}$. For every type $\mathsf{key} \to \tau$ and every pair of functions $f_1, f_2 \in [\![\mathsf{key} \to \tau]\!]s$, $f_1 \; \mathcal{R}^i_{\mathsf{key}\to\tau} \; f_2$ if and only if,*

$$\forall k \in i(w).\ f_1 s(\mathbf{id}_s, k)\ \mathcal{R}^i_\tau\ f_2 s(\mathbf{id}_s, k)$$

and

$$\exists k_0 \notin s.\ f_1 s_0(\mathbf{inl}_{s,\{k_0\}}, k_0)\ \mathcal{R}^{i_0}_\tau\ f_2 s_0(\mathbf{inl}_{s,\{k_0\}}, k_0)$$

where $w_0 = w + \{k_0\}$, $s_0 = s + \{k_0\}$ and i_0 is the injection $i + \mathbf{id}_{\{k_0\}} : w_0 \to s_0$.

Proof. The “only if” direction is obvious. We prove the “if” direction.

To relate the two functions f_1, f_2, we must check that for any morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^\to$ and any $k' \in i'(w')$, $f_1 s'(l, k')\ \mathcal{R}^{i'}_\tau\ f_2 s'(l, k')$.

- If $k' \in l(i(w)) = i'(j(w))$, there exists some $k \in i(w)$ such that $k' = l(k)$, so by hypothesis, $f_1 s(\mathbf{id}_s, k)\ \mathcal{R}^i_\tau\ f_2 s(\mathbf{id}_s, k)$, then

  $$[\![\tau]\!]l(f_1 s(\mathbf{id}_s, k))\ \mathcal{R}^{i'}_\tau\ [\![\tau]\!]l(f_2 s(\mathbf{id}_s, k)),$$

  because $\mathcal{R}_\tau$ is monotonic (Proposition 5.9). By the naturality of f_1 and f_2,

  $$[\![\tau]\!]l(f_m s(\mathbf{id}_s, k)) = f_m s'(l, l(k)), \quad (m = 1, 2),$$

  we have $f_1 s'(l, k')\ \mathcal{R}^{i'}_\tau\ f_2 s'(l, k')$;

- If $k' \notin l(i(w))$, define injections $j' : w + \{w_0\} \to w'$ and $l' : s + \{k_0\} \to s'$ as

$$\begin{array}{ll} j'(k_0) = k', & j'(x) = j(x) \text{ for every } x \in w, \\ l'(k_0) = k', & l'(x) = l(x) \text{ for every } x \in s, \end{array}$$

then $(j', l') : \langle w_0, i_0, s_0 \rangle \to \langle w', i', s' \rangle$ is a morphism in $\mathcal{PI}^{\to}$, and $l' \circ \mathbf{inl}_{s,\{k_0\}} = l$. Because $f_1 s_0(\mathbf{inl}_{s,\{k_0\}}, k_0) \ \mathcal{R}_\tau^{i_0} \ f_2 s_0(\mathbf{inl}_{s,\{k_0\}}, k_0)$, and $\mathcal{R}_\tau$ is monotonic,

$$[\![\tau]\!] l'(f_1 s_0(\mathbf{inl}_{s,\{k_0\}}, k_0)) \ \mathcal{R}_\tau^{i'} \ [\![\tau]\!] l'(f_2 s_0(\mathbf{inl}_{s,\{k_0\}}, k_0)),$$

where, by the naturality of f_1 and f_2,

$$\begin{array}{rcl} [\![\tau]\!] l'(f_m s_0(\mathbf{inl}_{s,\{k_0\}}, k_0)) & = & f_m s'(l' \circ \mathbf{inl}_{s,\{k_0\}}, l'(k_0)) \\ & = & f_m s'(l, k'), \qquad (m = 1, 2), \end{array}$$

hence $f_1 s'(l, k') \ \mathcal{R}_\tau^{i'} \ f_2 s'(l, k')$.

$\square$

Lemma 7.3 shows the fact that, in order to relate functions of a type $\mathsf{key} \to \tau$, at a certain world $\langle w, i, s \rangle$, we do not need to quantify over all larger worlds, as prescribed by the *comprehension* property of Kripke logical relations. Instead, by simply considering those related keys at the original world $\langle w, i, s \rangle$ plus a related "fresh" key not in s, whatever it is, we can check whether two functions are related.

This lemma recognizes the "Some/Any" property for logical relations. This property is well specified in Pitts' nominal logic and is proved very important for reasoning about fresh resources (names, keys, nonces, etc.) [Pit03]. Intuitively, if a fact satisfies the "Some/Any" property, then it holds for *all* fresh keys if and only if it holds for *some* fresh key.

According to Lemma 7.3, if a logical relation is monotonic, deciding the relation for functions of a type $\mathsf{key} \to \tau$, depends indeed on the decidability of relating values of type τ.

Proposition 7.4. *Suppose that $i : w \to s$ is an injection in $\mathcal{I}$ and $\mathcal{R}_\tau^i$ is a monotonic logical relation derived over the category $\mathcal{S}et^{\mathcal{PI}^{\to}}$. For every type $\mathsf{key} \to \tau$, the relation $\mathcal{R}_{\mathsf{key}\to\tau}$ is decidable if $\mathcal{R}_\tau$ is decidable.*

Proof. According to Lemma 7.3, to check whether two functions are related by $\mathcal{R}^i_{b\to\tau}$, we just apply the two functions to every key in $i(w)$ and an arbitrary key that is not in s, and see whether the results are related by $\mathcal{R}_\tau^i$, which is decidable. $\square$

7.2 Décidabilité dans le cas monadique

According to the definition of monadic logical relations over category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, the point of relating two computations is to find a proper substitution for fresh keys, together with a proper selection of disclosed keys from the fresh keys, such that the corresponding values are related. Formally, suppose that s_1 and s_2 are two sets of fresh keys generated during two computations respectively, then to relate the two computations at a certain world $\langle w, i, s\rangle$, we should find a set s_0 together with an injection $i_0 : w_0 \to s_0$, and two injections $l_1 : s_1 \to s_0$, $l_2 : s_2 \to s_0$, so that the two corresponding values are related at the world $\langle w + w_0, i + i_0, s + s_0\rangle$. While this is also an exploration over a infinite space of worlds, a natural thought is to restrict this space to a finite space of worlds where we do not consider fresh keys that are not generated during the two computations, that is, the largest s_0 that we need to consider is $s_1 + s_2$. However, reducing a relation at a larger world (larger than $s_1 + s_2$) to a smaller world (smaller than $s_1 + s_2$) requires then the *op-monotonicity* property of logical relations.

We say that a logical relation $(\mathcal{R}_\tau)_{\tau \text{ type}}$ defined over $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ is *op-monotonic* if and only if, for every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\rightarrow}$ and every pair of values $a_1, a_2 \in [\![\tau]\!]s$,

$$[\![\tau]\!]l(a_1)\ \mathcal{R}_\tau^{i'}\ [\![\tau]\!]l(a_2) \Longrightarrow a_1\ \mathcal{R}_\tau^{i}\ a_2$$

Lemma 7.5. *Suppose that $i : w \to s$ is an injection in $\mathcal{I}$ and $(\mathcal{R}_\tau)_{\tau \text{ type}}$ is an op-monotonic logical relation derived over the model $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$. For every pair of computations $[s_1, a_1]$, $[s_2, a_2]$ in $[\![\mathsf{T}\tau]\!]s$, if $[s_1, a_1]\ \mathcal{R}_{\mathsf{T}\tau}^{i}\ [s_2, a_2]$, then there exists a pair $\langle w_0, i_0, s_0\rangle$ with $\max(|s_1|, |s_2|) \le |s_0| \le |s_1| + |s_2|$ and two injections $l_1 : s_1 \to s_0$, $l_2 : s_2 \to s_0$, such that $[\![\tau]\!](\mathbf{id}_s + l_1)(a_1)\ \mathcal{R}_\tau^{i+i_0}\ [\![\tau]\!](\mathbf{id}_s + l_2)(a_2)$, where $|s|$ denotes the cardinality of the set s.*

Proof. Because $[s_1, a_1]\ \mathcal{R}_{\mathsf{T}\tau}^{i}\ [s_2, a_2]$, by the derivation of logical relations over $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ (5.5), there exist injections $i'_0 : w'_0 \to s'_0$, $l'_1 : s_1 \to s'_0$ and $l'_2 : s_2 \to s'_0$ in $\mathcal{I}$ such that

$$[\![\tau]\!](\mathbf{id}_s + l'_1)(a_1)\ \mathcal{R}_\tau^{i+i'_0}\ [\![\tau]\!](\mathbf{id}_s + l'_2)(a_2).$$

Now let $s_0 = l'_1(s_1) \cup l'_2(s_2)$, $w_0 = w'_0 \cap s_0$ and $i_0 : w_0 \to s_0$ be the injection i'_0 restricted on the domain w_0. Clearly, $\max(|s_1|, |s_2|) \le |s_0| \le |s_1| + |s_2|$ holds. Let $j_0 : w_0 \to w'_0$ and $l_0 : s_0 \to s'_0$ be inclusions in $\mathcal{I}$, then $(\mathbf{id}_w + j_0, \mathbf{id}_s + l_0)$ is a morphism from $\langle w + w_0, i + i_0, s + s_0\rangle$ to $\langle w + w'_0, i + i'_0, s + s'_0\rangle$ in $\mathcal{PI}^{\rightarrow}$.

Define $l_m : s_m \to s_0$ by: for any $x \in s_m$, $l_m(x) = l'_m(x)$ $(m = 1, 2)$, and we have $l_0 \circ l_m = l'_m$. Because

$$\begin{array}{rl} & [\![\tau]\!](\mathbf{id}_s + l'_m)(a_m) \\ = & [\![\tau]\!]((\mathbf{id}_s + l_0) \circ (\mathbf{id}_s + l_m))(a_m) \\ = & [\![\tau]\!](\mathbf{id}_s + l_0)([\![\tau]\!](\mathbf{id}_s + l_m)(a_m)), \end{array}$$

and $\mathcal{R}^i_\tau$ is op-monotonic, $[\![\tau]\!](\mathbf{id}_s + l_1)(a_1)\ \mathcal{R}^{i+i_0}_\tau\ [\![\tau]\!](\mathbf{id}_s + l_2)(a_2)$. □

According to this lemma, if a logical relation is op-monotonic, then to decide whether two computations are related, we only need to consider a finite number of substitutions for fresh keys (up to isomorphism) and check whether the corresponding values under these substitutions are related. In other words, the decidability of relations for type $\mathsf{T}\tau$ depends on the decidability of relations for type τ.

Proposition 7.6. *Suppose that $i : w \to s$ is an injection in $\mathcal{I}$ and $(\mathcal{R}_\tau)_{\tau \text{ type}}$ is an op-monotonic logical relation derived over the category $\mathcal{S}et^{\mathcal{PI}^{\to}}$. For every type $\mathsf{T}\tau$, the relation $\mathcal{R}_{\mathsf{T}\tau}$ is decidable if $\mathcal{R}_\tau$ is decidable.*

Proof. According to Lemma 7.5, for every two computations of type $\mathsf{T}\tau$, if there exist injections $i_0 : w_0 \to s_0$, $l_1 : s_1 \to s_0$ and $l_2 : s_2 \to s_0$, with $\max(|s_1|, |s_2|) \leq |s_0| \leq |s_1| + |s_2|$, such that the two values (being lifted properly) are related, then the two computations are related. Since both s_1 and s_2 are finite, those possible injections (up to isomorphism) are also finite, and can be enumerated. □

The hypothesis of this proposition is that logical relations under consideration must be op-monotonic, so if we want to use this result to study the decidability of the cryptographic logical relation, we must check its op-monotonicity property. It is clear that relations for type bool and nat are op-monotonic. The relation for keys is also op-monotonic. Consider a morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in $\mathcal{PI}^{\to}$ and a pair of keys $k_1, k_2 \in s$. If $l(k_1)\ \boldsymbol{\mathcal{R}}^{i',\varphi}_{\mathsf{key}}\ l(k_2)$, then $l(k_1) = l(k_2) \in i'(w')$, so $l(k_1), l(k_2) \in i'(w') \cap l(s)$. Moreover, in the category $\mathcal{PI}^{\to}$, $i'(w') \cap l(s) = l(i(w))$, so $k_1 = k_2 \in i(w)$ since l is injective, i.e., $k_1\ \boldsymbol{\mathcal{R}}^{i,\varphi}_{\mathsf{key}}\ k_2$.

Lemma 7.7. *Let φ be a consistent cipher function, then the cryptographic message relation $\mathcal{MR}^{i,\varphi}$ is op-monotonic in the sense that for every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in the category $\mathcal{PI}^{\to}$, and every pair of messages $a_1, a_2 \in [\![\mathsf{msg}]\!]s$, if $([\![\mathsf{msg}]\!]l(a_1), [\![\mathsf{msg}]\!]l(a_2)) \in \mathcal{MR}^{i',\varphi}$, then $(a_1, a_2) \in \mathcal{MR}^{i,\varphi}$.*

Proof. This is proved by induction on the message structure. We show here the case where the two messages are cipher-texts.

Let $a_1 = \mathsf{e}(a'_1, k_1)$ and $a_2 = \mathsf{e}(a'_2, k_2)$, then $[\![\mathsf{msg}]\!]l(a_1) = \mathsf{e}([\![\mathsf{msg}]\!]l(a_1), l(k_1))$ and $[\![\mathsf{msg}]\!]l(a_1) = \mathsf{e}([\![\mathsf{msg}]\!]l(a_2), l(k_2))$.

- If $l(k_1) = l(k_2) \in i'(w')$, by the definition of $\mathcal{MR}^{i,\varphi}$,

$$([\![\mathsf{msg}]\!]l(a'_1), [\![\mathsf{msg}]\!]l(a'_2)) \in \mathcal{MR}^{i',\varphi},$$

and by induction, $(a'_1, a'_2) \in \mathcal{MR}^{i,\varphi}$. And

$$l(k_1) = l(k_2) \in i'(w') \cap l(s) = l(i(w)),$$

so $k_1 = k_2 \in i(w)$, hence $(\mathsf{e}(a'_1, k_1), \mathsf{e}(a'_2, k_2)) \in \mathcal{MR}^{i,\varphi}$.

- If $l(k_1), l(k_2) \notin i'(w')$, then

$$([\![\mathsf{msg}]\!]l(a'_1), [\![\mathsf{msg}]\!]l(a'_2)) \in \varphi^{i'}(l(k_1), l(k_2)),$$

and $l(k_1), l(k_2) \notin i'(w') \cap l(s) = l(i(w))$, so $k_1, k_2 \notin i(w)$. Because φ is consistent, we have $(a'_1, a'_2) \in \varphi^i(k_1, k_2)$, hence $(\mathsf{e}(a'_1, k_1), \mathsf{e}(a_2, k_2)) \in \mathcal{MR}^{i,\varphi}$.

□

We then prove that the cryptographic logical relation $(\boldsymbol{\mathcal{R}}_\tau)_{\tau \text{ type}}$ (Definition 5.2) is op-monotonic for certain types.

Proposition 7.8. *Let φ be a consistent cipher function. For every morphism $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ in the category $\mathcal{PI}^{\to}$, and every pair of values $a_1, a_2 \in [\![\tau]\!]s$, if $[\![\tau]\!]l(a_1)\ \boldsymbol{\mathcal{R}}_\tau^{i',\varphi}\ [\![\tau]\!]l(a_2)$, then $a_1\ \boldsymbol{\mathcal{R}}_\tau^{i,\varphi}\ a_2$, where $\boldsymbol{\mathcal{R}}_\tau^{i,\varphi}$ is the cryptographic logical relation defined in Definition 5.2 and τ is any type defined by the grammar*

$$\tau ::= b \mid b' \to \tau \mid \mathsf{T}\tau,$$

where $b \in \{\mathsf{nat}, \mathsf{bool}, \mathsf{key}, \mathsf{msg}\}$ and $b' \in \{\mathsf{nat}, \mathsf{bool}, \mathsf{key}\}$.

Proof. We prove the statement by induction on types. The relations for base types are monotonic.

For types $b \to \tau$ where $b \in \{\mathsf{nat}, \mathsf{bool}\}$, consider any pair of morphisms $(j, l) : \langle w, i, s\rangle \to \langle w', i', s'\rangle$ and $(j', l') : \langle w', i', s'\rangle \to \langle w'', i'', s''\rangle$ in $\mathcal{PI}^{\to}$, then $(j' \circ j, l' \circ l)$ is a morphism from $\langle w, i, s\rangle$ to $\langle w'', i'', s''\rangle$. Suppose that f_1 and f_2 are two functions in $[\![b \to \tau]\!]s$ and

$$[\![\tau]\!]^{[\![b]\!]}l(f_1)\ \boldsymbol{\mathcal{R}}_{b\to\tau}^{i',\varphi}\ [\![\tau]\!]^{[\![b]\!]}l(f_2),$$

then for any $a_1, a_2 \in [\![b]\!]s$,

$$\begin{array}{ll} & a_1 \,\boldsymbol{\mathcal{R}}_b^{i,\varphi}\, a_2 \\ \Leftrightarrow & a_1 \,\boldsymbol{\mathcal{R}}_b^{i'',\varphi}\, a_2 \\ & \qquad \text{(because } \boldsymbol{\mathcal{R}}_\tau \text{ is a regular logical relation)} \\ \Rightarrow & ([\![\tau]\!]^{[\![b]\!]} l(f_1))s''(l', a_1)) \,\boldsymbol{\mathcal{R}}_\tau^{i'',\varphi}\, ([\![\tau]\!]^{[\![b]\!]} i(f_2))s''(l', a_2) \\ & \qquad \text{(because } [\![\tau]\!]^{[\![b]\!]} l(f_1) \,\boldsymbol{\mathcal{R}}_{b\to\tau}^{i',\varphi}\, [\![\tau]\!]^{[\![b]\!]} l(f_2)) \\ \Leftrightarrow & f_1 s''(l' \circ l, a_1) \,\boldsymbol{\mathcal{R}}_\tau^{i'',\varphi}\, f_2 s''(l' \circ l, a_2) \\ & \qquad \text{(by the naturality of } f_1 \text{ and } f_2) \\ \Leftrightarrow & [\![\tau]\!](l' \circ l)(f_1 s(\mathbf{id}_s, a_1)) \,\boldsymbol{\mathcal{R}}_\tau^{i'',\varphi}\, [\![\tau]\!](l' \circ l)(f_2 s(\mathbf{id}_s, a_2)) \\ & \qquad \text{(again by the naturality of } f_1 \text{ and } f_2) \\ \Rightarrow & f_1 s(\mathbf{id}_s, a_1) \,\boldsymbol{\mathcal{R}}_\tau^{i,\varphi}\, f_2 s(\mathbf{id}_s, a_2) \\ & \qquad \text{(because } \boldsymbol{\mathcal{R}}_\tau \text{ is op-monotonic)} \end{array}$$

and by Lemma 7.1, $f_1 \,\boldsymbol{\mathcal{R}}_{b\to\tau}^{i,\varphi}\, f_2$.

For types $\mathsf{key} \to \tau$, let $f_1, f_2 \in [\![\mathsf{key} \to \tau]\!]s$ and $[\![\tau]\!]^{[\![\mathsf{key}]\!]} l(f_1) \,\boldsymbol{\mathcal{R}}_{\mathsf{key}\to\tau}^{i',\varphi}\, [\![\tau]\!]^{[\![\mathsf{key}]\!]} l(f_2)$. To prove $f_1 \,\boldsymbol{\mathcal{R}}_{\mathsf{key}\to\tau}^{i,\varphi}\, f_2$, by Lemma 7.3, it is sufficient to check the following two facts:

$$\begin{array}{ll} \text{Fact 1:} & \forall k \in i(w).\ f_1 s(\mathbf{id}_s, k) \,\boldsymbol{\mathcal{R}}_\tau^{i,\varphi}\, f_2 s(\mathbf{id}_s, k) \\ \text{Fact 2:} & \exists k_0 \notin s.\ f_1 s_0(\mathbf{inl}_{s,\{k_0\}}, k_0) \,\boldsymbol{\mathcal{R}}_\tau^{i_0,\varphi}\, f_2 s_0(\mathbf{inl}_{s,\{k_0\}}, k_0) \end{array}$$

where $w_0 = w + \{k_0\}$, $s_0 = s + \{k_0\}$ and i_0 is the injection $i + \mathbf{id}_{\{k_0\}} : w_0 \to s_0$.

- Fact 1: For any $k \in i(w)$, $l(k) \in l(i(w)) = i'(w') \cap l(s)$, so $l(k) \in i'(w')$.

$$([\![\tau]\!]^{[\![\mathsf{key}]\!]} l(f_1))s'(\mathbf{id}_{s'}, l(k)) \,\boldsymbol{\mathcal{R}}_\tau^{i',\varphi}\, ([\![\tau]\!]^{[\![\mathsf{key}]\!]} l(f_2))s'(\mathbf{id}_{s'}, l(k)).$$

Because f_1 and f_2 are natural transformations,

$$\begin{array}{ll} & ([\![\tau]\!]^{[\![\mathsf{key}]\!]} l(f_m))s'(\mathbf{id}_{s'}, l(k)) \\ = & f_m s'(\mathbf{id}_{s'} \circ l, l(k)) \\ = & [\![\tau]\!] l(f_m s(\mathbf{id}_s, k)), \quad (m = 1, 2) \end{array}$$

by induction, $\boldsymbol{\mathcal{R}}_\tau$ is op-monotonic, so $f_1 s(\mathbf{id}_s, k) \,\boldsymbol{\mathcal{R}}_\tau^{i,\varphi}\, f_2 s(\mathbf{id}_s, k)$;

- Fact 2: Because $[\![\tau]\!]^{[\![\mathsf{key}]\!]} l(f_1) \,\boldsymbol{\mathcal{R}}_{\mathsf{key}\to\tau}^{i',\varphi}\, [\![\tau]\!]^{[\![\mathsf{key}]\!]} l(f_2)$, for some key $k_0 \notin s'$,

$$([\![\tau]\!]^{[\![\mathsf{key}]\!]} l(f_1))s'_0(\mathbf{inl}_{s',\{k_0\}}, k_0) \,\boldsymbol{\mathcal{R}}_\tau^{i'_0,\varphi}\, ([\![\tau]\!]^{[\![\mathsf{key}]\!]} l(f_2))s'_0(\mathbf{inl}_{s',\{k_0\}}, k_0).$$

where $s'_0 = s' + \{k_0\}$, $w'_0 = w' + \{k_0\}$ and i'_0 is the injection $i' + \mathbf{id}_{\{k_0\}} : w'_0 \to s'_0$. Let $w_0 = w + \{k_0\}$ and $s_0 = s + \{k_0\}$. Because the following square commutes:

$$\begin{tikzcd} s \arrow[r, "\mathbf{inl}_{s,\{k_0\}}"] \arrow[d, "i"'] & s_0 \arrow[d, "i+\mathbf{id}_{\{k_0\}}"] \\ s' \arrow[r, "\mathbf{inl}s',\{k'_0\}"] & s'_0 \end{tikzcd}$$

we have

$$\begin{array}{rl} & ([\![\tau]\!]^{[\![\mathsf{key}]\!]} l(f_m)) s'_0(\mathbf{inl}_{s',\{k_0\}}, k_0) \\ = & f_m s'_0(\mathbf{inl}_{s',\{k_0\}} \circ i, k_0) \\ = & f_m s'_0((i + \mathbf{id}_{\{k_0\}}) \circ \mathbf{inl}_{s,\{k_0\}}, k_0) \\ = & [\![\tau]\!](i + \mathbf{id}_{\{k_0\}})(f_m s_0(\mathbf{inl}_{s,\{k_0\}}, k_0)), \end{array}$$

and because $(j + \mathbf{id}_{\{k_0\}}, l + \mathbf{id}_{\{k_0\}})$ is indeed a morphism from $\langle w_0, i_0, s_0\rangle$ to $\langle w'_0, i'_0, s'_0\rangle$ in $\mathcal{PI}^{\to}$, by induction, $f_1 s_0(\mathbf{inl}_{s,\{k_0\}}, k_0)\ \mathcal{R}^{i_0,\varphi}_{\tau}\ f_2 s_0(\mathbf{inl}_{s,\{k_0\}}, k_0)$.

From the above two facts, we can deduce that $f_1\ \mathcal{R}^{i,\varphi}_{\mathsf{key}\to\tau}\ f_2$.

For computation types $\mathsf{T}\tau$, consider any pair of computations $[s_1, a_1], [s_2, a_2] \in [\![\mathsf{T}\tau]\!]s$,

$$[\![\mathsf{T}\tau]\!]l([s_m, a_m]) = \boldsymbol{T}[\![\tau]\!]l([s_m, a_m]) = [s_m, [\![\tau]\!](l + \mathbf{id}_{s_m})(a_m)] \quad (m = 1, 2)$$

if $[s_1, [\![\tau]\!](l + \mathbf{id}_{s_1})(a_1)]\ \mathcal{R}^{i',\varphi}_{\mathsf{T}\tau}\ [s_2, [\![\tau]\!](l + \mathbf{id}_{s_2})(a_2)]$, there exist injections $i_0 : w_0 \to s_0$, $l_1 : s_1 \to s_0$ and $l_2 : s_2 \to s_0$ in $\mathcal{I}$ such that

$$[\![\tau]\!](\mathbf{id}_{s'} + l_1)([\![\tau]\!](l + \mathbf{id}_{s_1})(a_1))\ \mathcal{R}^{i'+i_0,\varphi}_{\tau}\ [\![\tau]\!](\mathbf{id}_{s'} + l_2)([\![\tau]\!](l + \mathbf{id}_{s_2})(a_2))$$

Because the square

$$\begin{tikzcd} s + s_m \arrow[r, "\mathbf{id}_s + l_m"] \arrow[d, "l+\mathbf{id}_{s_m}"'] & s + s_0 \arrow[d, "l+\mathbf{id}_{s_0}"] \\ s' + s_m \arrow[r, "\mathbf{id}_{s'} + l_m"] & s' + s_0 \end{tikzcd}$$

commutes,

$$\begin{array}{rl} & [\![\tau]\!](\mathbf{id}_{s'} + l_m)([\![\tau]\!](l + \mathbf{id}_{s_m})(a_m)) \\ = & [\![\tau]\!]((\mathbf{id}_{s'} + l_m) \circ (l + \mathbf{id}_{s_m}))(a_m)) \\ & \quad \text{(by the functoriality of } [\![\tau]\!]) \\ = & [\![\tau]\!]((l + \mathbf{id}_{s_0}) \circ (\mathbf{id}_s + l_m))(a_m) \\ & \quad \text{(by the above commuting square)} \\ = & [\![\tau]\!](l + \mathbf{id}_{s_0})([\![\tau]\!](\mathbf{id}_s + l_m)(a_m)) \\ & \quad \text{(by the functoriality of } [\![\tau]\!]) \end{array}$$

and $(j + \mathbf{id}_{s_0}, l + \mathbf{id}_{s_0}) : \langle w + w_0, i + i_0, s + s_0\rangle \to \langle w' + w_0, i + i'_0, s' + s_0\rangle$ is a morphism in $\mathcal{PI}^{\to}$, by induction $[\![\tau]\!](\mathbf{id}_s + l_1)(a_1)\ \boldsymbol{\mathcal{R}}_\tau^{i+i_0,\varphi}\ [\![\tau]\!](\mathbf{id}_s + l_2)(a_2)$, hence $[s_1, a_1]\ \boldsymbol{\mathcal{R}}_{\mathsf{T}\tau}^{i,\varphi}\ [s_2, a_2]$. $\square$

7.3 Décidabilité de l'équivalence contextuelle

There are four base types in the cryptographic metalanguage — nat, bool, key and msg. It is clear that relating two values of type nat or bool is decidable, as well as two keys since for every given world $\langle w, i, s\rangle \in \mathcal{PI}^{\to}$, the set s is always finite. Deciding whether two messages are related depends on the cipher function. We say that a cipher function φ is decidable if and only if for every injection $i : w \to s \in \mathcal{I}$, every pair of keys $k_1, k_2 \in s - i(w)$, it is decidable whether two messages $m_1, m_2 \in [\![\mathsf{msg}]\!]s$ are in $\varphi^i(k_1, k_2)$.

Lemma 7.9. *Suppose $i : w \to s$ is an injection in $\mathcal{I}$ and φ is a decidable cipher function. For every pair of messages $m_1, m_2 \in [\![\mathsf{msg}]\!]s$, it is decidable whether $(m_1, m_2) \in \mathcal{MR}^i$, where $\mathcal{MR}$ is the cryptographic message relation as defined in Definition 4.2.*

Proof. The statement is easy to check by decomposing the two messages (both are of finite size). $\square$

To sum up, relating values with the cryptographic logical relation $(\boldsymbol{\mathcal{R}}_\tau)_{\tau\ \mathrm{type}}$ (Definition 5.2) is decidable for a certain set of types, namely

$$\tau ::= b \mid \mathsf{bool} \to \tau \mid \mathsf{key} \to \tau \mid \mathsf{T}\tau \mid \tau \times \tau \mid \mathsf{opt}[\tau] \tag{7.1}$$

where $b \in \{\mathsf{nat}, \mathsf{bool}, \mathsf{key}, \mathsf{msg}\}$.

Theorem 7.10. *Suppose $i : w \to s$ is an injection in $\mathcal{I}$ and φ is a decidable and consistent cipher function. For every $a_1, a_2 \in [\![\tau]\!]s$, it is decidable whether $a_1\ \mathcal{R}_\tau^{i,\varphi,\varphi}\ a_2$, where τ is a type defined by the grammar (7.1).*

Proof. We prove the statement by induction on type τ. Induction steps are straightforward from Lemma 7.9, Proposition 7.2, Proposition 7.4 and Proposition 7.6. $\square$

Consider the completeness of the cryptographic logical relation, we get then the following result on the decidability of contextual equivalence.

Corollary 7.11. *Suppose $i : w \to s$ is an injection in $\mathcal{I}$ and κ is an empty context knowledge. For every pair of terms t_1, t_2 such that $\overline{w : \mathsf{key}} \vdash t_1 : \tau$ and $\overline{w : \mathsf{key}} \vdash t_2 : \tau$ are derivable, it is decidable whether $t_1 \approx_\tau^{i,\kappa} t_2$, where τ is defined by*

$$\tau ::= b \mid \mathsf{bool} \to \tau \mid \mathsf{key} \to \tau \mid \mathsf{T}\tau^0 \mid \tau \times \tau \mid \mathsf{opt}[\tau]$$

where $b \in \{\mathsf{nat}, \mathsf{bool}, \mathsf{key}, \mathsf{msg}\}$ *and* τ^0 *is defined by*

$$\tau^0 ::= \mathsf{nat} \mid \mathsf{bool} \mid \mathsf{key} \mid \tau^0 \times \tau^0 \mid \mathsf{opt}[\tau^0]$$

Proof. According to Theorem 6.1, Proposition 6.4, Lemma 6.6, Proposition 6.7 and Proposition 6.9, $\sim_\tau^{i,\emptyset} = \boldsymbol{\mathcal{R}}_\tau^{i,\emptyset}$ holds for these types. Then by Theorem 7.10, the contextual equivalence for these types is decidable. □

However, the contextual equivalence for the cryptographic metalanguage is in general undecidable. We shall show this by encoding the 2-counter machine [Min61] in our cryptographic metalanguage. Such a technique is very common in the field of verifying cryptographic protocols, for showing some problems undecidable [DLMS99].

We assume that a 2-counter machine contains instructions of the form ($i \in \{1, 2\}$):

$$\begin{array}{lll} (1) & q & : N_i := N_i + 1;\ \texttt{goto}\ q' \\ (2) & q & : \texttt{if}\ N_i = 0\ \texttt{then}\ \texttt{goto}\ q' \\ & & \qquad\qquad\quad \texttt{else}\ N_i := N_i - 1;\ \texttt{goto}\ q'' \end{array}$$

where N_1, N_2 denote the two counters. An instruction of type (1) increments the counter i and jumps to another point of the control. An instruction of type (2), tests whether the counter i is 0, and if it is the case it jumps to a control q', otherwise it decrements the counter and jumps to control point q''.

We encode the 2-counter machine into the cryptographic metalanguage so that the problem of determining whether the initial configuration $(q_0, 0, 0)$ of the 2-counter machine does not reach a desired configuration (q_F, n_F^1, n_F^2), for some natural numbers n_F^1, n_F^2, reduces to the contextual equivalence of two programs in the metalanguage. We shall represent a configuration (q, n_1, n_2) by an encrypted message $\{[q, n_1, n_2]\}_k$, where k is a secret key (not known to contexts). In particular, a state q is represented by a natural number in our encoding. (Some syntactic abbreviations are as defined in Figure 2.2.)

First, for every instruction of type (1) we introduce the following function: (We assume that the operations act on the first counter ($i = 1$). The case $i = 2$ is quite similar.)

$$\begin{array}{rcl} f_q & \equiv & \lambda\{x\}_k.\texttt{letopt}\ y \Leftarrow \texttt{getnum}(\pi_1^3(x))\ \texttt{in} \\ & & \qquad\quad \texttt{letopt}\ z_1 \Leftarrow \texttt{getnum}(\pi_2^3(x))\ \texttt{in} \\ & & \qquad\quad \texttt{letopt}\ z_2 \Leftarrow \texttt{getnum}(\pi_3^3(x))\ \texttt{in} \\ & & \qquad\quad \texttt{if}\ (y = q)\ \texttt{then}\ \{[q', z_1 + 1, z_2]\}_k\ \texttt{else}\ \{x\}_k, \end{array}$$

and for every instruction of type (2) we introduce the following function:

$$\begin{aligned}
f_q \quad \equiv \quad & \lambda\{x\}_k.\mathtt{letopt}\ y \Leftarrow \mathtt{getnum}(\pi_1^3(x))\ \mathtt{in} \\
& \qquad \mathtt{letopt}\ z_1 \Leftarrow \mathtt{getnum}(\pi_2^3(x))\ \mathtt{in} \\
& \qquad \mathtt{letopt}\ z_2 \Leftarrow \mathtt{getnum}(\pi_3^3(x))\ \mathtt{in} \\
& \qquad \mathtt{if}\ (y = q)\ \mathtt{then} \\
& \qquad\quad \mathtt{if}\ (z_1 = 0)\ \mathtt{then}\ \{[q', z_1, z_2]\}_k\ \mathtt{else}\ \{[q'', z_1 - 1, z_2]\}_k \\
& \qquad \mathtt{else}\ \{x\}_k.
\end{aligned}$$

For the desired configuration (q_F, n_F^1, n_F^2), we introduce two specific functions

$$\begin{aligned}
f_F^1 \quad \equiv \quad & \lambda\{x\}_k.\mathtt{letopt}\ y \Leftarrow \mathtt{getnum}(\pi_1^3(x))\ \mathtt{in} \\
& \qquad \mathtt{letopt}\ z_1 \Leftarrow \mathtt{getnum}(\pi_2^3(x))\ \mathtt{in} \\
& \qquad \mathtt{letopt}\ z_2 \Leftarrow \mathtt{getnum}(\pi_3^3(x))\ \mathtt{in} \\
& \qquad \mathtt{if}\ (y = q_F\ \mathtt{and}\ z_1 = n_F^1\ \mathtt{and}\ z_2 = n_F^2)\ \mathtt{then}\ \mathtt{n}(0)\ \mathtt{else}\ \{x\}_k, \\
f_F^2 \quad \equiv \quad & \lambda\{x\}_k.\mathtt{letopt}\ y \Leftarrow \mathtt{getnum}(\pi_1^3(x))\ \mathtt{in} \\
& \qquad \mathtt{letopt}\ z_1 \Leftarrow \mathtt{getnum}(\pi_2^3(x))\ \mathtt{in} \\
& \qquad \mathtt{letopt}\ z_2 \Leftarrow \mathtt{getnum}(\pi_3^3(x))\ \mathtt{in} \\
& \qquad \mathtt{if}\ (y = q_F\ \mathtt{and}\ z_1 = n_F^1\ \mathtt{and}\ z_2 = n_F^2)\ \mathtt{then}\ \mathtt{n}(1)\ \mathtt{else}\ \{x\}_k.
\end{aligned}$$

These two functions are designed to return distinguished values (namely 0 and 1) when the machine reaches the desired configuration (q_F, n_F^1, n_F^2). We then define two programs:

$$\begin{aligned}
p_1 \quad & \equiv \nu(k).\langle \{[0,0,0]\}_k,\ f_{q_0},\ \ldots,\ f_{q_m},\ f_F^1 \rangle \\
p_1 \quad & \equiv \nu(k).\langle \{[0,0,0]\}_k,\ f_{q_0},\ \ldots,\ f_{q_m},\ f_F^2 \rangle,
\end{aligned}$$

which are of type $\mathsf{T}(\mathsf{msg} \times (\mathsf{msg} \to \mathsf{opt}[\mathsf{msg}]) \times \cdots \times (\mathsf{msg} \to \mathsf{opt}[\mathsf{msg}]))$. The first components of the two programs act as the initial configuration of the machine. Clearly, contexts can do arbitrary executions of the 2-counter machine: by applying one of the functions $f_{q_0}, \ldots, f_{q_m}$ to a certain message $\{[q, n_1, n_2]\}_k$ in their knowledge, they get another message $\{[q', n_1', n_2']\}_k$ (from the configuration (q, n_1, n_2) to the configuration (q', n_1', n_2')). And they are restricted to the two kinds of instructions defined by the 2-counter machine, because we let the key k be freshly generated so that contexts are not able to crack the machine.

The only possibility that a context can get some distinguished values of the above two programs is that the context gets finally the message $\{[q_F, n_F^1, n_F^2]\}_k$ and apply the two functions f_F^1, f_F^2 to it. In other words, the two programs p_1 and p_2 are not contextually equivalent if and only if a configuration (q_F, n_F^1, n_F^2) is reachable from the initial configuration $(q_0, 0, 0)$. Since the latter problem is not decidable, the problem of determining whether two programs are contextually equivalent is not decidable either.

This encoding is somewhat tricky because we simply represent the numbers in the two counters by numbers in the metalanguage. However, even without integers in the language, i.e., without type nat, the contextual equivalence is still undecidable.

We encode the natural numbers by encryption chains: the representation $\bar{n}$ of the number n is $\{\cdots\{\{k_0\underbrace{\}_k\}_k\cdots\}_k}_{n}$, where k_0 is a plain key representing the number 0 (it does not matter whether k and k_0 are equal). Then the two operations — increment and decrement — of numbers are represented by

$$
\begin{array}{lcl}
\mathtt{S}(\bar{n}) & \equiv & \{\bar{n}\}_k \\
\mathtt{P}(\bar{n}) & \equiv & \mathtt{case}\ \mathtt{dec}(\bar{n}, k)\ \mathtt{of}\ \mathtt{some}(x)\ \mathtt{in}\ x\ \mathtt{else}\ \mathtt{k}(k_0).
\end{array}
$$

A configuration (q, n_1, n_2) is encoded as an encrypted message $\{[k_q, \bar{n}_1, \bar{n}_2]\}_k$, where k_q, k are secret keys (not known to contexts) and k_q represents the state q. The encoding of instructions should be also modified. the representation function for every instruction of type (1) becomes:

$$
\begin{array}{lcl}
f_q & \equiv & \lambda\{x\}_k.\mathtt{letopt}\ y \Leftarrow \mathtt{getkey}(\pi^3_1(x))\ \mathtt{in} \\
& & \qquad\quad \mathtt{letopt}\ z_1 \Leftarrow \pi^3_2(x)\ \mathtt{in} \\
& & \qquad\quad \mathtt{letopt}\ z_2 \Leftarrow \pi^3_3(x)\ \mathtt{in} \\
& & \qquad\quad \mathtt{if}\ (y = k_q)\ \mathtt{then} \\
& & \qquad\qquad \mathtt{some}(\{[k_{q'}, \mathtt{S}(z_1), z_2]\}_k) \\
& & \qquad\quad \mathtt{else}\ \mathtt{some}(\{x\}_k),
\end{array}
$$

and for every instruction of type (2):

$$
\begin{array}{lcl}
f_q & \equiv & \lambda\{x\}_k.\mathtt{letopt}\ y \Leftarrow \mathtt{getkey}(\pi^3_1(x))\ \mathtt{in} \\
& & \qquad\quad \mathtt{letopt}\ z_1 \Leftarrow \pi^3_2(x)\ \mathtt{in} \\
& & \qquad\quad \mathtt{letopt}\ z_2 \Leftarrow \pi^3_3(x)\ \mathtt{in} \\
& & \qquad\quad \mathtt{if}\ (y = k_q)\ \mathtt{then} \\
& & \qquad\qquad \mathtt{letopt}\ z'_1 \Leftarrow \mathtt{getkey}(z_1)\ \mathtt{in} \\
& & \qquad\qquad \mathtt{if}\ (z'_1 = k_0)\ \mathtt{then} \\
& & \qquad\qquad\quad \mathtt{some}(\{[k_{q'}, z_1, z_2]\}_k) \\
& & \qquad\qquad \mathtt{else}\ \mathtt{some}(\{[k_{q''}, \mathtt{P}(z_1), z_2]\}_k) \\
& & \qquad\quad \mathtt{else}\ \mathtt{some}(\{x\}_k).
\end{array}
$$

For checking whether the desired configuration (q_F, n_F^1, n_F^2) is reached, we have to test whether a message is the representation of a particular number n. The test function is defined as follows:

$$
\begin{array}{rcl}
\mathtt{test}_n(x) & \equiv & \mathtt{case}\ \mathtt{dec}(x,k)\ \mathtt{of}\ \mathtt{some}(y_1)\ \mathtt{in} \\
& & \quad \mathtt{case}\ \mathtt{dec}(y_1,k)\ \mathtt{of}\ \mathtt{some}(y_2)\ \mathtt{in} \\
& & \qquad \ldots \\
& & \qquad\quad \mathtt{case}\ \mathtt{dec}(y_{n-1},k)\ \mathtt{of}\ \mathtt{some}(y_n)\ \mathtt{in}\ \mathtt{true} \\
& & \qquad\quad \mathtt{else}\ \mathtt{false} \\
& & \qquad \ldots \\
& & \quad \mathtt{esle}\ \mathtt{false} \\
& & \mathtt{esle}\ \mathtt{false}.
\end{array}
$$

Then for the desired configuration (q_F, n_F^1, n_F^2), we introduce two specific functions

$$
\begin{array}{rcl}
f_F^i & \equiv & \lambda\{x\}_k.\mathtt{letopt}\ y \Leftarrow \mathtt{getkey}(\pi_1^3(x))\ \mathtt{in} \\
& & \qquad \mathtt{letopt}\ z_1 \Leftarrow \pi_2^3(x)\ \mathtt{in} \\
& & \qquad \mathtt{letopt}\ z_2 \Leftarrow \pi_3^3(x)\ \mathtt{in} \\
& & \qquad \mathtt{if}\ (y = k_{q_F}\ \mathtt{and}\ \mathtt{test}_{n_F^1}(z_1)\ \mathtt{and}\ \mathtt{test}_{n_F^2}(z_2)) \\
& & \qquad \mathtt{then}\ \mathtt{some}(\mathtt{k}(k_i))\ \mathtt{else}\ \mathtt{some}(\{x\}_k) \qquad (i = 1, 2),
\end{array}
$$

where k_1, k_2 are two different keys that contexts can distinguish. As in the former encoding, the two functions here may return distinguished values — k_1 and k_2 when the machine reaches the desired configuration (q_F, n_F^1, n_F^2). We then define two programs:

$$
\begin{array}{rcl}
p_1 & \equiv & \nu(k, k_0, k_1, k_2, k_{q_0}, \ldots, k_{q_m}).\langle k_1, k_2, \{[k_{q_0}, k_0, k_0]\}_k,\ f_{q_0},\ \ldots,\ f_{q_m},\ f_F^1\rangle \\
p_2 & \equiv & \nu(k, k_0, k_1, k_2, k_{q_0}, \ldots, k_{q_m}).\langle k_1, k_2, \{[k_{q_0}, k_0, k_0]\}_k,\ f_{q_0},\ \ldots,\ f_{q_m},\ f_F^2\rangle,
\end{array}
$$

which are of type $\mathsf{key} \times \mathsf{key} \times \mathsf{msg} \times (\mathsf{msg} \to \mathsf{opt}[\mathsf{msg}]) \times \cdots \times (\mathsf{msg} \to \mathsf{opt}[\mathsf{msg}])$. The third components of the two programs act as the initial configuration of the machine. It is clear that in this encoding, contexts can still do arbitrary executions of the 2-counter machine. And the only possibility that a context can distinguish the two programs p_1 and p_2 is that the context gets the message $\{[k_{q_F}, \bar{n}_F^1, \bar{n}_F^2]\}_k$, i.e., the configuration (q_F, n_F^1, n_F^2) is reached. Hence for the same reason as in the former encoding, the problem of determining whether two programs are contextually equivalent in the cryptographic metalanguage (without type nat) is not decidable.

Chapitre 8

Conclusion

L'idée d'utiliser le lambda-calcul et les techniques connexes pour vérifier les protocoles cryptographiques a été proposée en premier par Sumii et Pierce [SP01]. Bien que la vérification des protocoles cryptographiques dans le cadre du lambda-calcul soit plus difficile que dans la plupart des autres modèles, par exemple le Spi-calcul ou le modèle de Dolev-Yao, nous obtenons dans cette thèse une compréhension profonde et entière du rôle des fonctions d'ordre supérieur. Bien que pour le moment, les fonctions d'ordre supérieur soient rarement utilisées pour modéliser les protocoles qui comportent principalement de l'échange de messages, il est naturellement possible que les protocoles futurs échangent des programmes, par exemple des algorithmes cryptographiques.

L'avantage principal de l'approche par le lambda-calcul est de permettre d'utiliser des techniques puissantes telles que les relations logiques pour prouver des propriétés de sécurité, notamment la propriété de secret. De plus, la génération dynamique de clés est un mécanisme crucial dans les protocoles cryptographiques, auquel la plupart des modèles formels n'accordent pas une importance suffisante. La génération de clés était bien étudiée par Pitts et Stark dans le nu-calcul [PS93a] et elle remonte aux origines des travaux de Moggi sur le lambda-calcul computationnel [Mog89, Mog91]. Naturellement, nous adoptons des techniques de leurs travaux pour vérifier les protocoles cryptographiques. De ce point de vue, ce que font Sumii et Pierce consiste à étendre les travaux de Pitts et Stark à un langage plus riche — le lambda-calcul cryptographique. Le coeur de leurs travaux est l'utilisation des relations logiques pour prouver l'équivalence contextuelle dans leur langage.

Pitts et Stark définissent une relation logique opérationnelle pour le nu-calcul, qu'ils prouvent correcte par rapport à l'équivalence contextuelle et complète pour les types du premier ordre. Les relations logiques diverses de Sumii et Pierce peuvent être considérées comme des extensions de la relation logique opérationnelle. Pourtant, toutes leurs relations logiques sont syntaxiques, et elles se fondent sur une sémantique opérationnelle, ce qui rend relativement difficiles l'extension et l'adaptation de leurs relations logiques à un langage plus riche. À la place de la sémantique opérationnelle, notre étude est basée purement sur des modèles dénotationnels. Ceci nous permet d'adapter facilement nos résultats à des langages plus riches, à condition que ces langages puissent être interprétés dans le même modèle.

Ce dernier chapitre est une conclusion de cette thèse. En particulier, nous résumons les résultats présentés dans la thèse et nous proposons des directions susceptibles d'être suivies pour approfondir encore le sujet.

The idea of using lambda-calculus and relevant techniques to verify cryptographic protocols was first proposed by Sumii and Pierce in 2001 [SP01]. While verifying cryptographic protocols in suitable lambda-calculi is harder than in most formal models like Spi-calculus or Dolev-Yao model, we gain in this thesis a thorough understanding of the role of higher-order functions. This may come in handy in the future. Although for the moment, higher-order functions are rarely used in modeling protocols, which are usually sets of message exchanges, it is very possible that future protocols will involve exchanges of codes, e.g., cryptographic algorithms.

The main advantage of this lambda-calculus approach is to make use of some powerful techniques such as logical relations, to prove desired security properties, notably the secrecy property. Furthermore, the mechanism of fresh key generation plays a crucial role in cryptographic protocols while most formal models do not stress it enough. Since fresh key generation has been well studied by Pitts and Stark in the nu-calculus [PS93a], which is again traced to Moggi's work on the computational lambda-calculus [Mog89, Mog91], it is very natural to adopt techniques from their works in protocol verification. Indeed, on the aspect of fresh key generation, what Sumii and Pierce did is to extend Pitts and Stark's work in a richer language — the cryptographic lambda-calculus. The heart of their work is to prove contextual equivalence in this language, through logical relations.

Pitts and Stark define an operational logical relation for the nu-calculus, which is proved sound w.r.t. the contextual equivalence, and complete for types up to first order. Sumii and Pierce's various logical relations for the cryptographic lambda-calculus are indeed extensions of this operational logical relation. However, all these logical relations are syntactic and rely largely on the operational semantics, which makes them hard to extend and to fit in richer languages. Instead, we rest on purely denotational models, not only because logical relations are originally developed on semantics, but also by doing so we are allowed to easily extend our work to richer languages, provided that the language can be interpreted in the same model.

8.1 Résumé des résultats

This thesis contributes mainly to the theoretical aspect of cryptographic lambda-calculi, on the following four points:

Categories

First of all, we would like to underline the categories over which we derive logical relations.

The crucial mechanism of dynamic key generation can be nicely modeled in Moggi's framework of the computational lambda-calculus, through monads. Thanks to Stark's work, we can in particular make use of his category for the nu-calculus — the functor category $\mathcal{S}et^{\mathcal{I}}$. This model is comprehensive enough for describing computations of key generation: every computation consists of a value and a set of freshly generated keys. It also shows how keys are involved and interfere with programs as computations go along. Defining logical relations over this model is relatively easy — we can simply follow the categorical construction of logical relations and this was done by Goubault-Larrecq, Lasota and Nowak. In particular, Goubault-Larrecq et al. provide a natural way to derive logical relations for monadic types.

However, although the category $\mathcal{S}et^{\mathcal{I}}$ is adequately perfect for modeling dynamic key generation, it is indeed not sufficient when we consider relations between programs. The reason is that keys must be classified when two programs are taken into account — some keys can be manipulated by both programs while others not. But in the model $\mathcal{S}et^{\mathcal{I}}$, we do not consider any classification of keys for any given set of keys. In fact, compared with Pitts and Stark's operational logical relation, the logical relation defined by Goubault-Larrecq et al. over the category $\mathcal{S}et^{\mathcal{I}}$ is proved too weak in the sense that it fails in relating certain contextually equivalent programs related by Pitts and Stark's [ZN03]. Our observation is that, as far as relations between programs are concerned, a better category to consider is $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$, where in $\mathcal{I}^{\rightarrow}$, we do have some classification of keys. Precisely, every object in the category $\mathcal{I}^{\rightarrow}$ represents a selection of those non-secret keys (keys that can be accessed by any program) from a given set of keys. However, the category $\mathcal{I}^{\rightarrow}$ does not explain properly how these selections evolve as more and more keys are generated. The classification of keys in $\mathcal{I}^{\rightarrow}$ is so coarse that logical relations derived over $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ are still too weak. We then refine the classification by the category $\mathcal{PI}^{\rightarrow}$ and show that $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ is the right category that one should consider when studying relations between programs with fresh key generation.

Both these categories — $\mathcal{S}et^{\mathcal{I}^{\rightarrow}}$ and $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ — satisfy the categorical requirements for deriving logical relations, so we follow the general construction to derive sound logical relations for the cryptographic metalanguage, being careful that the Basic Lemma should hold.

Contextual equivalence

Another contribution of this thesis is the denotational notion of contextual equivalence (in the framework of lambda-calculus) for cryptographic protocols. Intuitively, the meaning of contextual equivalence is very clear: two programs are contextually equivalent if there is no context that can distinguish them. This concept should not be difficult to understand and standard definitions (both syntactic and denotational) in the simply-typed lambda-calculus are simple. However, it turns out that these standard definitions do not fit in some richer typed lambda-calculi. This is because a proper notion of contexts becomes very crucial when we have more syntactic components, since contexts are closely related to the syntax. Essentially, a context should be neither too "weak" nor too "strong". For example, in a standard typed lambda-calculus, we do not have types for computations, so it is sufficient to take contexts of any observation type. But in the computational lambda-calculus, we must allow contexts to compare computations, for this purpose, we should allow contexts themselves to do some computations, i.e., the type of contexts should be the corresponding computation types of observation types.

As for cryptographic protocols, the key point is that contexts must represent honestly the power of attackers. In particular, they should be able to get access to those non-secret keys, as well as a set of secret cipher-texts. These arguments must be represented in the definition. While they are probably not easy to define in a syntactic way, the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$ allows us to represent properly these points in semantics. Finally, we arrive at what we believe should be the proper definition of the contextual equivalence for cryptographic protocols (Definition 6.2, Chapter 6), still over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$.

Completeness

Completeness is important for logical relations but it is usually difficult to achieve, because for contextual equivalence, we do not care about the internal structure of different programs, while for logical relations, we do. This is probably the reason why usually we only have completeness for first-order functions (for higher-order functions, contexts may throw away some information about the function structure). This becomes even worse when we introduce computation types in the language, since programs may consequently have more complicated structure, notably those programs of "computations of computation" (type $\mathsf{TT}\tau$) or "computations of function" (type $\mathsf{T}(\tau \rightarrow \tau'))$. For logical relations derived over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, we prove their completeness for a certain subset of first-order types.

In fact, as for completeness, a better notion is that of lax logical relation. Lax logical relations do not require to be constructed by strict induction on types, hence it allows us to achieve completeness of logical relations for any higher order type, by relaxing the restrictions on relations for certain types, notably function types and computation types. Again over the category $\mathcal{S}et^{\mathcal{PI}^{\rightarrow}}$, we define a lax logical relation that is lax at function types and monadic types, but can remain strict at other types. It is sound, and complete at all types.

Decidability

In the field of formal verification of protocols, a critical criterion for verification techniques is decidability. However, it is in general undecidable to determine whether two programs are contextually equivalent in the cryptographic metalanguage. We have shown this by encoding the 2-counter machine in the metalanguage and reducing the probelm of reachability to that of deciding contextual equivalence.

On the other hand, because logical relations can identify the contextual equivalence for a set of types and it is decidable, for another set of types, whether two values are related by certain logical relation, determining the equivalence between programs of certain types is still decidable. We explore the decidability of the problem whether two values (of certain types) are related by logical relations derived over the category $\mathcal{S}et^{\mathcal{PI}^{\to}}$ and show that relations for two kinds of types $\mathsf{key} \to \tau$ and $\mathsf{T}\tau$ are decidable, provided that the relation for type τ is decidable. Then we conclude that it is decidable whether two programs of certain types are contextually equivalent (Corollary 7.11, Chapter 7).

8.2 Perspectives

We have concentrated ourselves on proving the secrecy property of security protocols, by means of logical relations. Since there are many other security properties like authenticity, anonymity, etc., it is natural to ask: is it possible to prove security properties other than secrecy in this framework? The general idea is to prove the equivalence of a real system and an ideal (secure) one. But it is also possible to introduce other techniques of manipulating message terms into our framework, in particular symbolic techniques based on term algebra. This is feasible because the message type is defined as a base type in our metalanguage and all cryptographic primitives operate only on messages. Such a treatment of messages is quite similar as in Spi-calculus, where messages (terms) and processes are define separately. Indeed, there are already attempts of developing symbolic techniques in the framework of cryptographic process calcului [FA01, AL00, Bor01]. Again, separating (to some extent) messages from the whole language allows us to extend easily the language with other primitives satisfying specific algebraic properties [CDL05], and consequently to reason about specific protocols. It would be also interesting to define proper categorical notions corresponding to these algebraic properties.

Another direction is on the proof techniques for contextual equivalence. We have presented the technique of logical relations, but recursion is never considered in our language, neither in the nu-calculus nor in the cryptographic lambda-calculus. Dealing with recursion would be challenging for defining logical relations. In particular, the presence of recursive functions requires us to switch to domain theoretical models. Bisimulations present no difficulties with recursion. There are notions of applicative bisimulations that are sound and complete in typed lambda-calculi with full universal, existential and recursive types [SP05], but without monadic types. It would be interesting to extend this notion to some kind of "monadic bisimulation" in the computational lambda-calculus. Furthermore, all these applicative bisimulations are defined syntactically and there is no known underlying mathematical theory. A more ambitious direction could be looking for a mathematical model for deriving bisimulations in lambda-calculi, like those categorical models for deriving logical relations [MS93, GLLN02].

Annexe A

Règles de raisonnement du métalangage cryptographique

Stark defines an equational logic of Horn clauses for reasoning about terms of his computational metalanguage [Sta94]. This equational logic can be easily extended for giving the semantics of the cryptographic metalanguage and reasoning about terms, since the cryptographic metalanguage is simply an extension of Stark's computational metalanguage. Rules of the logic are given in the form defined at the end of Section 2.2.

The detailed rules are given in following figures.

Horn clauses:

$$\frac{}{\Gamma; \Phi \vdash \phi}\,(\phi \in \Phi) \qquad \frac{\Gamma; \Phi \vdash \phi \qquad \Gamma; \Psi, \phi \vdash \psi}{\Gamma; \Phi \cup \Psi \vdash \psi}$$

Equality:

$$\frac{\Gamma \vdash a : \tau}{\Gamma \vdash a = a} \qquad \frac{\Gamma \vdash a_1, a_2 : \tau}{\Gamma; a_1 = a_2 \vdash a_2 = a_1}$$

$$\frac{\Gamma \vdash a_1, a_2, a_3 : \tau}{\Gamma; a_1 = a_2, a_2 = a_3 \vdash a_1 = a_3}$$

Figure A.1: Rules for equational reasoning in the cryptographic metalanguage (I)

Congruence:

$$\frac{}{\Gamma \vdash x = x}\,(x : \tau \in \Gamma) \qquad \frac{}{\Gamma \vdash \mathtt{true} = \mathtt{true}} \qquad \frac{}{\Gamma \vdash \mathtt{false} = \mathtt{false}}$$

$$\frac{\Gamma \vdash b, b' : \mathsf{bool} \qquad \Gamma \vdash t_1, t_1', t_2, t_2' : \tau}{\Gamma; b = b', t_1 = t_1', t_2 = t_2' \vdash \mathtt{if}\ b\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 = \mathtt{if}\ b'\ \mathtt{then}\ t_1'\ \mathtt{else}\ t_2'}$$

$$\frac{}{\Gamma \vdash \mathtt{new} = \mathtt{new}} \qquad \frac{}{\Gamma \vdash i = i}\,(i = 0, 1, 2, \ldots)$$

$$\frac{\Gamma \vdash t_1, t_1', \ldots, t_n, t_n' : \mathsf{nat}}{\Gamma; t_1 = t_1', \ldots, t_n = t_n' \vdash \mathtt{nat\_op}_n(t_1, \ldots, t_n) = \mathtt{nat\_op}_n(t_1', \ldots, t_n')}$$

$$\frac{\Gamma \vdash t_1, t_1' : \tau_1 \qquad \Gamma \vdash t_2, t_2' : \tau_2}{\Gamma; t_1 = t_1', t_2 = t_2' \vdash \langle t_1, t_2 \rangle = \langle t_1', t_2' \rangle} \qquad \frac{\Gamma \vdash t, t' : \tau_1 \times \tau_2}{\Gamma; t = t' \vdash \mathtt{proj}_1(t) = \mathtt{proj}_1(t)}$$

$$\frac{\Gamma \vdash t, t' : \tau_1 \times \tau_2}{\Gamma; t = t' \vdash \mathtt{proj}_2(t) = \mathtt{proj}_2(t)} \qquad \frac{\Gamma \vdash t, t' : \tau}{\Gamma; t = t' \vdash \mathtt{some}(t) = \mathtt{some}(t')}$$

$$\frac{\Gamma \vdash t_1, t_1' : \mathsf{opt}[\tau] \qquad \Gamma, x : \tau; \Phi \vdash t_2 = t_2' : \tau'\ (x \notin \mathit{fv}(\Phi)) \qquad \Gamma \vdash t_3, t_3' : \tau'}{\Gamma; t_1 = t_1', t_3 = t_3' \vdash \mathtt{case}\ t_1\ \mathtt{of}\ \mathtt{some}(x)\ \mathtt{in}\ t_2\ \mathtt{else}\ t_3 = \mathtt{case}\ t_1'\ \mathtt{of}\ \mathtt{some}(x)\ \mathtt{in}\ t_2'\ \mathtt{else}\ t_3'}$$

$$\frac{\Gamma \vdash t, t' : \tau}{\Gamma \vdash \mathtt{val}(t) = \mathtt{val}(t')} \qquad \frac{\Gamma \vdash t_1, t_1' : \mathsf{T}\tau \qquad \Gamma, x : \tau; \Phi \vdash t_2 = t_2' : \mathsf{T}\tau'}{\Gamma; \Phi, t_1 = t_1' \vdash \mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2 = \mathtt{let}\ x \Leftarrow t_1'\ \mathtt{in}\ t_2'}\,(x \notin \mathit{fv}(\Phi))$$

$$\frac{\Gamma \vdash f, f' : \tau \to \tau' \qquad \Gamma \vdash t, t' : \tau}{\Gamma; f = f', t = t' \vdash f t = f' t'} \qquad \frac{\Gamma, x : \tau; \Phi \vdash t = t' : \tau'}{\Gamma; \Phi \vdash \lambda x.t = \lambda x.t'}\,(x \notin \mathit{fv}(\Phi))$$

$$\frac{\Gamma \vdash t_1, t_1' : \mathsf{msg} \qquad \Gamma \vdash t_2, t_2' : \mathsf{key}}{\Gamma; t_1 = t_1', t_2 = t_2' \vdash \mathtt{enc}(t_1, t_2) = \mathtt{enc}(t_1', t_2')}$$

$$\frac{\Gamma \vdash t_1, t_1' : \mathsf{msg} \qquad \Gamma \vdash t_2, t_2' : \mathsf{key}}{\Gamma; t_1 = t_1', t_2 = t_2' \vdash \mathtt{dec}(t_1, t_2) = \mathtt{dec}(t_1', t_2')} \qquad \frac{\Gamma \vdash t_1, t_1', t_2, t_2' : \mathsf{msg}}{\Gamma; t_1 = t_1', t_2 = t_2' \vdash \mathtt{p}(t_1, t_2) = \mathtt{p}(t_1', t_2')}$$

$$\frac{\Gamma \vdash t, t' : \mathsf{msg}}{\Gamma; t = t' \vdash \mathtt{fst}(t) = \mathtt{fst}(t')} \qquad \frac{\Gamma \vdash t, t' : \mathsf{msg}}{\Gamma; t = t' \vdash \mathtt{snd}(t) = \mathtt{snd}(t')}$$

$$\frac{\Gamma \vdash t, t' : \mathsf{nat}}{\Gamma; t = t' \vdash \mathtt{n}(t) = \mathtt{n}(t')} \qquad \frac{\Gamma \vdash t, t' : \mathsf{msg}}{\Gamma; t = t' \vdash \mathtt{getnum}(t) = \mathtt{getnum}(t')}$$

$$\frac{\Gamma \vdash t, t' : \mathsf{key}}{\Gamma; t = t' \vdash \mathtt{k}(t) = \mathtt{k}(t')} \qquad \frac{\Gamma \vdash t, t' : \mathsf{msg}}{\Gamma; t = t' \vdash \mathtt{getkey}(t) = \mathtt{getkey}(t')}$$

Figure A.2: Rules for equational reasoning in the cryptographic metalanguage (II)

Functions:

$$\beta \quad \frac{\Gamma, x : \tau_1 \vdash t_2 : \tau_2 \qquad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash (\lambda x^{\tau_1}.t_2)t_1 = t_2[t_1/x]} \qquad \eta \quad \frac{\Gamma \vdash f : \tau_1 \to \tau_2}{\Gamma \vdash f = \lambda x^{\tau_1}.fx}$$

Booleans:

$$\frac{\Gamma; \Phi, b = \texttt{true} \vdash \phi \qquad \Gamma; \Phi, b = \texttt{false} \vdash \phi}{\Gamma; \Phi \vdash \phi} \qquad \frac{\Gamma; \Phi \vdash \texttt{true} = \texttt{false}}{\Gamma; \Phi \vdash \phi}$$

$$\frac{\Gamma \vdash t, t' : \tau}{\Gamma \vdash \texttt{if true then } t \texttt{ else } t' = t} \qquad \frac{\Gamma \vdash t, t' : \tau}{\Gamma \vdash \texttt{if false then } t \texttt{ else } t' = t'}$$

Products:

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{proj}_i(\langle t_1, t_2 \rangle) = t_i} (i = 1, 2) \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \langle \texttt{proj}_1(t), \texttt{proj}_2(t) \rangle = t}$$

Options:

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma, x : \tau \vdash t_2 : \tau' \qquad \Gamma \vdash t_3 : \tau'}{\Gamma \vdash \texttt{case some}(t_1) \texttt{ of some}(x) \texttt{ in } t_2 \texttt{ else } t_3 = t_2[t_1/x]}$$

$$\frac{\Gamma, x : \tau \vdash t_2 : \tau' \qquad \Gamma \vdash t_3 : \tau'}{\Gamma \vdash \texttt{case } \bot_\tau \texttt{ of some}(x) \texttt{ in } t_2 \texttt{ else } t_3 = t_3}$$

Messages:

$$\frac{\Gamma \vdash k, k' : \mathsf{key}}{\Gamma, \texttt{dec}(\texttt{enc}(\texttt{n}(1), k), k') = \texttt{some}(1) \vdash k = k'} \qquad \frac{\Gamma \vdash t : \mathsf{msg} \qquad \Gamma \vdash k : \mathsf{key}}{\Gamma \vdash \texttt{dec}(\texttt{enc}(t, k), k) = \texttt{some}(t)}$$

$$\frac{\Gamma \vdash t_1, t_2 : \mathsf{msg}}{\Gamma \vdash \texttt{fst}(\texttt{p}(t_1, t_2)) = \texttt{some}(t_1)} \qquad \frac{\Gamma \vdash t_1, t_2 : \mathsf{msg}}{\Gamma \vdash \texttt{snd}(\texttt{p}(t_1, t_2)) = \texttt{some}(t_2)}$$

$$\frac{\Gamma \vdash t : \mathsf{nat}}{\Gamma \vdash \texttt{getnum}(\texttt{n}(t)) = \texttt{some}(t)} \qquad \frac{\Gamma \vdash t : \mathsf{key}}{\Gamma \vdash \texttt{getkey}(\texttt{k}(t)) = \texttt{some}(t)}$$

Figure A.3: Rules for equational reasoning in the cryptographic metalanguage (III)

Computations:

$$\frac{\Gamma \vdash t : \mathsf{T}\tau}{\Gamma \vdash \mathtt{let}\ x \Leftarrow t\ \mathtt{in}\ \mathtt{val}(x) = t} \qquad \frac{\Gamma \vdash t, t' : \tau}{\Gamma; \mathtt{val}(t) = \mathtt{val}(t') \vdash t = t'}$$

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma, x : \tau \vdash t' : \mathsf{T}\tau'}{\Gamma \vdash \mathtt{let}\ x \Leftarrow \mathtt{val}(t)\ \mathtt{in}\ t' = t'[t/x]}$$

$$\frac{\Gamma \vdash t : \mathsf{T}\tau \qquad \Gamma, x : \tau \vdash t' : \mathsf{T}\tau' \qquad \Gamma, x' : \mathsf{T}\tau' \vdash t'' : \mathsf{T}\tau''}{\Gamma \vdash \mathtt{let}\ x' \Leftarrow (\mathtt{let}\ x \Leftarrow t\ \mathtt{in}\ t')\ \mathtt{in}\ t'' = \mathtt{let}\ x \Leftarrow t\ \mathtt{in}\ (\mathtt{let}\ x' \Leftarrow t'\ \mathtt{in}\ t'')}$$

Generating keys:

(DROP)
$$\frac{\Gamma \vdash t : \mathsf{T}\tau}{\Gamma \vdash t = \mathtt{let}\ k \Leftarrow \mathtt{new}\ \mathtt{in}\ t}\ (k : \mathsf{key} \notin \Gamma)$$

(SWAP)
$$\frac{\Gamma, k, k' : \mathsf{key} \vdash t : \mathsf{T}\tau}{\Gamma \vdash \mathtt{let}\ k \Leftarrow \mathtt{new}\ \mathtt{in}\ \mathtt{let}\ k' \Leftarrow \mathtt{new}\ \mathtt{in}\ t = \mathtt{let}\ k' \Leftarrow \mathtt{new}\ \mathtt{in}\ \mathtt{let}\ k \Leftarrow \mathtt{new}\ \mathtt{in}\ t}$$

(FRESH)
$$\frac{\Gamma \vdash k : \mathsf{key} \qquad \Gamma, k' : \mathsf{key}; \Phi, \mathtt{dec}(\mathtt{enc}(1, k), k') = \mathtt{error} \vdash t = t'}{\Gamma; \Phi \vdash \mathtt{let}\ k' \Leftarrow \mathtt{new}\ \mathtt{in}\ t = \mathtt{let}\ k' \Leftarrow \mathtt{new}\ \mathtt{in}\ t'}$$

where `error` is the syntatic abbreviation as defined in Figure 2.2.

Figure A.4: Rules for equational reasoning in the cryptographic metalanguage (VI)

Annexe B

Complétude des relations logiques monadiques

Completeness (w.r.t. the contextual equivalence) is an important concern of logical relations, We say that a logical relation is *complete*, if and only if any pair of contextually related programs can be related by this logical relation. However, completeness is rather difficult to achieve. In simply-typed lambda-calculi, logical relations are only complete for types up to first order in general. When we have monadic types, things become much subtler, even for first-order types. As shown in the discussion at the beginning of Chapter 6, it is very difficult to get a general result on completeness for all monads, because particular monads (and corresponding logical relations) usually have specific properties, which are quite different. Furthermore, since contexts must be involved in the discussion, language constants play an important role in discussions of completeness and these constants vary widely for different kinds of computations. So in this appendix, we shall investigate the completeness of monadic logical relations for a set of concrete monads, namely partial computations, exceptions, non-determinism and state transformers.

Recall the syntax of the computational lambda-calculus. We have in particular a unary type constructor T to construct types for computations, and two relevant constants

$$t ::= \ldots \mid \mathtt{val}(t) \mid \mathtt{let}\; x \Leftarrow t \;\mathtt{in}\; t,$$

with corresponding typing rules

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathtt{val}(t) : \mathsf{T}\tau}\,(\mathit{Val}) \qquad \frac{\Gamma \vdash t_1 : \mathsf{T}\tau \quad \Gamma, x : \tau \vdash t_2 : \mathsf{T}\tau'}{\Gamma \vdash \mathtt{let}\; x \Leftarrow t_1 \;\mathtt{in}\; t_2 : \mathsf{T}\tau'}\,(\mathit{Let})$$

We shall restrict ourselves to types up to *first order* in the computational lambda-calculus, i.e., those given by the grammar

$$\tau^1 ::= b \mid \mathsf{T}\tau^1 \mid b \to \tau^1,$$

where $b \in \Sigma$ ranges over a set of base types. In certain cases, e.g., the non-determinism monad, we will only consider a subset of types up to first order which we call *weak first-order types*. They are given by the following grammar:

$$\tau_w^1 ::= b \mid \mathsf{T}b \mid b \to \tau_w^1. \tag{B.1}$$

As we have seen in Section 3.1.2, semantics of the two computation constructs can be given in full generality in a categorical setting. In particular, the interpretation of terms in the computational lambda-calculus must satisfy the following equations:

$$[\![\mathtt{let}\ x \Leftarrow \mathtt{val}(t_1)\ \mathtt{in}\ t_2]\!]\rho = [\![t_2[t_1/x]]\!]\rho \tag{B.2}$$

$$[\![\mathtt{let}\ x \Leftarrow t\ \mathtt{in}\ \mathtt{val}(x)]\!]\rho = [\![t]\!]\rho \tag{B.3}$$

$$[\![\mathtt{let}\ x_2 \Leftarrow (\mathtt{let}\ x_1 \Leftarrow t_1\ \mathtt{in}\ t_2)\ \mathtt{in}\ t_3]\!]\rho = [\![\mathtt{let}\ x_1 \Leftarrow t_1\ \mathtt{in}\ \mathtt{let}\ x_2 \Leftarrow t_2\ \mathtt{in}\ t_3]\!]\rho \tag{B.4}$$

Indeed, every term of a monadic type can be written in some canonical form (respecting these equations):

Definition 1 (Computational canonical form). *A term t of a monadic type $\mathsf{T}\tau$ in the computational λ-calculus is said to be a* computational canonical term *if it is of the form*

$$\mathtt{let}\ x_1 \Leftarrow t_1\ \mathtt{in} \cdots \mathtt{let}\ x_n \Leftarrow t_n\ \mathtt{in}\ \mathtt{val}(u) \quad (n = 0, 1, 2, \ldots)$$

where u is a term of type τ, $x_1, \ldots, x_n$ are variables and every t_i ($i = 1, \ldots, n$) is a weak head normal form, i.e., $t_i = u_i w_{i1} \cdots w_{ik_i}$ and each u_i is either a variable or a constant.

Proposition B.1. *For every term t of a monadic type $\mathsf{T}\tau$ in the computational λ-calculus, there exists a computational canonical term t' such that $[\![t']\!]\rho = [\![t]\!]\rho$, for every valid interpretation $[\![\_]\!]\rho$ (i.e., interpretations satisfying the equations (B.2-B.4)).*

Proof. The computational λ-calculus is strongly normalizing [BBdP98], so we consider the β-normal form of term t and prove it by induction on t.

If t is a variable or a constant, then according to the equation (B.3)

$$[\![t]\!]\rho = [\![\mathtt{let}\ x \Leftarrow t\ \mathtt{in}\ \mathtt{val}(x)]\!]\rho,$$

where x is not free in t.

If t is an application $t_1 t_2 \cdots t_n$, then t_1 is of a functions type and it must be a variable or a constant (it cannot be a λ-abstraction since t is β-normal). Similarly, t is equivalent to the term $\mathtt{let}\ x \Leftarrow t\ \mathtt{in}\ \mathtt{val}(x)$.

If t is a trivial computation $\mathtt{val}(t')$, it is already in the computational canonical form.

If t is a sequential computation $\mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2$, by induction, there are computational canonical terms for both t_1 and t_2, namely

$$\mathtt{let}\ x_1^1 \Leftarrow t_1^1\ \mathtt{in} \cdots \mathtt{let}\ x_m^1 \Leftarrow t_m^1\ \mathtt{in}\ \mathtt{val}(u_1)$$

and

$$\mathtt{let}\ x_1^2 \Leftarrow t_1^2\ \mathtt{in} \cdots \mathtt{let}\ x_n^2 \Leftarrow t_n^2\ \mathtt{in}\ \mathtt{val}(u_2),$$

where $x_1^1, \ldots, x_m^1, x_1^2, \ldots, x_n^2$ are not free in t. Replace t_1 and t_2 with these terms in t and we get

$$\begin{array}{rcl}
[\![t]\!]\rho & = & [\![\mathtt{let}\ x \Leftarrow (\mathtt{let}\ x_1^1 \Leftarrow t_1^1\ \mathtt{in} \cdots \mathtt{let}\ x_m^1 \Leftarrow t_m^1\ \mathtt{in}\ \mathtt{val}(u_1))\ \mathtt{in}\ t_2]\!]\rho \\
& = & [\![\mathtt{let}\ x_1^1 \Leftarrow t_1^1\ \mathtt{in} \\
& & \quad \mathtt{let}\ x \Leftarrow (\mathtt{let}\ x_2^1 \Leftarrow t_2^1\ \mathtt{in} \cdots \mathtt{let}\ x_m^1 \Leftarrow t_m^1\ \mathtt{in}\ \mathtt{val}(u_1))\ \mathtt{in}\ t_2]\!]\rho \\
& = & \quad \ldots \quad \ldots \\
& = & [\![\mathtt{let}\ x_1^1 \Leftarrow t_1^1\ \mathtt{in} \cdots \mathtt{let}\ x_m^1 \Leftarrow t_m^1\ \mathtt{in}\ \mathtt{let}\ x \Leftarrow \mathtt{val}(u_1)\ \mathtt{in}\ t_2]\!]\rho \\
& = & [\![\mathtt{let}\ x_1^1 \Leftarrow t_1^1\ \mathtt{in} \cdots \mathtt{let}\ x_m^1 \Leftarrow t_m^1\ \mathtt{in}\ \mathtt{let}\ x \Leftarrow \mathtt{val}(u_1)\ \mathtt{in} \\
& & \quad \mathtt{let}\ x_1^2 \Leftarrow t_1^2\ \mathtt{in} \cdots \mathtt{let}\ x_n^2 \Leftarrow t_n^2\ \mathtt{in}\ \mathtt{val}(u_2)]\!]\rho.
\end{array}$$

Because all $t_1^1, \ldots, t_m^1, t_1^2, \ldots, t_n^2$ and $\mathtt{val}(u)$ are weak head normal forms, so the last term in the above equation is computational canonical. □

Now define the contextual equivalence in the computational lambda-calculus (we consider here a set-theoretical model of the computational lambda-calculus):

Definition B.1 (Contextual equivalence in the computational lambda-calculus). *In the computational lambda-calculus, two closed terms t_1, t_2, of the same type τ, are* contextually equivalent*, written as $t_1 \approx_\tau t_2$, if and only if, whatever the term $\mathbb{C}$ such that $x : \tau \vdash \mathbb{C} : \mathsf{T}o$ ($o \in \mathbf{Obs}$) is derivable,*

$$[\![\mathbb{C}]\!][x \mapsto [\![t_1]\!]] = [\![\mathbb{C}]\!][x \mapsto [\![t_2]\!]].$$

In a set-theoretical model, a value $a \in [\![\tau]\!]$ is *definable* if and only if there is a term t such that $\vdash t : \tau$ is derivable and $a = [\![t]\!]$. We then define a relation $\sim_\tau$, for every type τ, by: for every pair of values $a_1, a_2 \in [\![\tau]\!]$, $a_1 \sim_\tau a_2$ if and only if a_1, a_2 are definable and $a_1 \approx_\tau a_2$.

We shall investigate completeness in a strong sense and aim at finding a logical relation $(\mathcal{R}_\tau)_{\tau\ \mathrm{type}}$ such that if $\vdash t_1 : \tau$ and $\vdash t_2 : \tau$ are derivable, for any type τ up to first order, then

$$t_1 \approx_\tau t_2 \implies [\![t_1]\!]\ \mathcal{R}_\tau\ [\![t_2]\!]$$

Or, shortly: $\sim_\tau \subseteq \mathcal{R}_\tau$. Let us induce a logical relation $(\mathcal{R}_\tau)_{\tau\ \mathrm{type}}$ by $\mathcal{R}_b = \sim_b$, for any base type b. Then the proof would go by induction over τ, to show $\sim_\tau\ \subseteq\ \mathcal{R}_\tau$ for an arbitrary monad

T and every first-order type τ. Cases $\tau = b$ and $\tau = b \to \tau'$ go identically as in the proof for simply-typed lambda-calculus (see Chapter 6). The difficult case is $\tau = \mathsf{T}\tau'$, i.e.,

$$\sim_\tau \subseteq \mathcal{R}_\tau \implies \sim_{\mathsf{T}\tau} \subseteq \mathcal{R}_{\mathsf{T}\tau} \tag{B.5}$$

We did not find any general way to prove this for arbitrary monad. Instead, in the following subsections we show it for particular ones. In fact, (B.5) does not always hold for all first-order type. For certain concrete monads, we must have further restrictions on τ.

There is also another subtle point – notice that it is even not true in general that relations on $\mathsf{T}o$ (o an observation type) is partial identity. Fortunately, this difficulty can be solved in general, under some mild assumptions on *T*, fulfilled by all the monads investigated in the sequel.

At the heart of the difficulty of showing (B.5), one finds an issue of definability at monadic types. By definition, an element c of $[\![\mathsf{T}\tau]\!]$ is definable if and only if there is a close term t such that $\vdash t : \mathsf{T}\tau$ is derivable and $[\![t]\!] = c$. But this definition states nothing on the connection between the definablity of a computation and its corresponding "value". Intuitively, if an element of $[\![\mathsf{T}\tau]\!]$ is definable, either it corresponds to a computation which "returns a definable value" (necessarily of type τ), or there is a specific constant in the language which defines this value. This is of course informal. We shall make this argument precise and formal for each monad, in Propositions B.3, B.6, B.10 and B.13. Interestingly, all of these propositions can be spelled out as $\mathsf{def}_{\mathsf{T}\tau} \subseteq T\mathsf{def}_\tau$, where by $\mathsf{def}_\tau \subseteq [\![\tau]\!]$ we mean the subset of definable elements of $[\![\tau]\!]$. But even if stated easily in general, this fact needs substantially different proofs for different monads.

Before moving on to discussions of concrete monads, we first define a *$\mathcal{P}$-form* for closed terms, parameterized by a predicate $\mathcal{P}$ on terms.

Definition B.2. *For any predicate $\mathcal{P}$ on terms, we say that a closed term (necessarily of a computation type $\mathsf{T}\tau$ for some τ) is in $\mathcal{P}$-*form *if and only if it is of the form*

$$\texttt{let}\ x_1 \Leftarrow t_1\ \texttt{in} \cdots \texttt{let}\ x_n \Leftarrow t_n\ \texttt{in}\ \texttt{val}(u) \quad (n = 0, 1, 2, \ldots),$$

where $\mathcal{P}$ is a predicate on closed terms, $t_i = u_i w_{i1} \cdots w_{ik_i}$ ($1 \leq i \leq n$), u_i is either a variable x_l ($1 \leq l \leq i-1$) or a closed term such that $\mathcal{P}(u_i)$ holds, w_{im} ($1 \leq m \leq k_i$) is a term whose free variables must be in $\{x_1, \ldots, x_{i-1}\}$ and u is any term of type τ with free variables in $\{x_1, \ldots, x_n\}$.

We then define, for every monad, a predicate $\mathbf{Cond}$ on closed terms. These predicates impose a restriction on constants, and we show that reasonable constants in these concrete monads all satisfy the corresponding predicates.

B.1 Partial computation

In the case of partial computations, the semantics of the monadic types and the `val` and `let` constructs are given by:

$$\begin{array}{rcl} [\![\mathsf{T}\tau]\!] & = & [\![\tau]\!] \cup \{\perp\}, \\ [\![\mathtt{val}(t)]\!]\rho & = & [\![t]\!]\rho, \\ [\![\mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2]\!]\rho & = & \begin{cases} [\![t_2]\!]\rho[x \mapsto [\![t_1]\!]\rho] & \text{if } [\![t_1]\!]\rho \neq \perp \\ \perp & \text{if } [\![t_1]\!]\rho = \perp \end{cases}, \end{array}$$

where $\perp$ denotes all non-terminating computations. Logical relations at monadic types are given by [GLLN02]:

$$c_1\ \mathcal{R}_{\mathsf{T}\tau}\ c_2 \Longleftrightarrow c_1\ \mathcal{R}_\tau\ c_2 \text{ or } c_1 = c_2 = \perp \tag{B.6}$$

Let $\mathbf{Cond}$ be the smallest set of closed terms such that, for any closed term t of type $\tau_1 \to \cdots \to \tau_n \to \mathsf{T}\tau$, $\mathbf{Cond}(t)$ holds if and only if: for any closed terms $\vdash t_1 : \tau_1, \cdots, \vdash t_n : \tau_n$, $[\![t]\!]([\![t_1]\!], \cdots, [\![t_n]\!])$ is either:

- equal to $\perp$, or
- in $[\![\tau]\!]$ and definable by some closed term t' of type τ; if τ is of the form $\tau'_1 \to \cdots \to \tau'_m \to \mathsf{T}\tau'$, then $\mathbf{Cond}(t')$ holds.

We assume that for any constant $\mathtt{d}$, $\mathbf{Cond}(\mathtt{d})$ holds. We also assume that there is, for every τ, a constant Ω_τ of type $\mathsf{T}\tau$ such that $[\![\Omega_\tau]\!] = \perp$. Clearly $\mathbf{Cond}(\Omega_\tau)$ holds.

Lemma B.2. *For any closed term t (of type $\mathsf{T}\tau$) in $\mathbf{Cond}$-form, $[\![t]\!]$ is either $\perp$, or a definable value at type τ.*

Proof. Because t is of the $\mathbf{Cond}$-form,

$$t \equiv \mathtt{let}\ x_1 \Leftarrow t_1\ \mathtt{in} \cdots \mathtt{let}\ x_n \Leftarrow t_n\ \mathtt{in}\ \mathtt{val}(u), \qquad (n = 0, 1, \ldots).$$

We reason by induction on n:

- In the base case ($n = 0$): $[\![\mathtt{val}(u)]\!] = [\![u]\!]$. It is obvious that $[\![t]\!]$ is definable at type τ (by the term u in particular).
- For any $n \geq 1$,

  $$[\![t_1]\!] = [\![u_1 w_{11} \cdots w_{1k_1}]\!] = [\![u_1]\!]([\![w_{11}]\!], \cdots, [\![w_{1k_1}]\!]),$$

  where $u_1, w_{11}, \cdots, w_{1k_1}$ are all closed terms and $\mathbf{Cond}(u_1)$ holds, so $[\![t_1]\!]$ is either equal to $\perp$ or definable at type τ_1 (note that t_1 is of type $\mathsf{T}\tau_1$). If $[\![t_1]\!] = \perp$, then the denotation of the whole term is $\perp$, i.e., $[\![t]\!] = \perp$. If $[\![t_1]\!] \neq \perp$, suppose $[\![t_1]\!]$ is defined by a closed term t'_1 (of type τ_1). Because $\mathbf{Cond}(u_1)$ holds, so does $\mathbf{Cond}(t'_1)$, then $\mathbf{Cond}(u_2[t'_1/x_1])$, ..., $\mathbf{Cond}(u_n[t'_1/x_1])$ hold as well (because $u_2[t'_1/x_1], \ldots, u_n[t'_1/x_1]$ are either t'_1 or a constant). Let $t'_i = t_i[t'_1/x_1]$ ($2 \leq i \leq n$), then

$$\begin{aligned} & [\![\texttt{let}\ x_1 \Leftarrow t_1\ \texttt{in}\ \texttt{let}\ x_2 \Leftarrow t_2\ \texttt{in} \cdots \texttt{let}\ x_n \Leftarrow t_n\ \texttt{in}\ \texttt{val}(u)]\!] \\ = \;& [\![\texttt{let}\ x_2 \Leftarrow t_2\ \texttt{in} \cdots \texttt{let}\ x_n \Leftarrow t_n\ \texttt{in}\ \texttt{val}(u)]\!][x_1 \mapsto [\![t'_1]\!]] \\ = \;& \left[\!\!\left[\texttt{let}\ x_2 \Leftarrow t'_2\ \texttt{in} \cdots \texttt{let}\ x_n \Leftarrow t'_n\ \texttt{in}\ \texttt{val}(u[t'_1/x_1])\right]\!\!\right]. \end{aligned}$$

Clearly, $\texttt{let}\ x_2 \Leftarrow t'_2\ \texttt{in} \cdots \texttt{let}\ x_n \Leftarrow t'_n\ \texttt{in}\ \texttt{val}(u[t'_1/x_1])$ is again in $\mathbf{Cond}$-form, so by induction, its denotation is either $\perp$ or a value which is definable at type τ. □

Proposition B.3. *A value $c \in [\![\mathsf{T}\tau]\!]$ is definable if and only if, either c is definable at type τ, or $c = \perp$, i.e.: $\mathsf{def}_{\mathsf{T}\tau}(c) \Longleftrightarrow \mathsf{def}_\tau(c)$ or $c = \perp$.*

Proof. The "if" direction: For any value $c \in [\![\mathsf{T}\tau]\!]$, if $c = \perp$, it is obvious (Ω_τ defines it); if $c \in [\![\tau]\!]$ and $\mathsf{def}_\tau(c)$ holds, suppose c is defined by some closed term t of type τ, then c is also definable at type $\mathsf{T}\tau$ (by the term $\texttt{val}(t)$), i.e., $\mathsf{def}_{\mathsf{T}\tau}(c)$ holds.

The "only if" direction: Suppose that there is a value $c \in [\![\mathsf{T}\tau]\!]$ which is definable by some closed term t of type $\mathsf{T}\tau$. Consider the computational canonical form of t:

$$u_t \equiv \texttt{let}\ x_1 \Leftarrow t_1\ \texttt{in} \cdots \texttt{let}\ x_n \Leftarrow t_n\ \texttt{in}\ \texttt{val}(u), \qquad (n = 0, 1, \ldots)$$

where $t_i = y_i w_{i1} \cdots w_{ik_i}$ ($1 \leq i \leq n$), y_i is either a constant or a variable x_l ($1 \leq l \leq i-1$, if $i \geq 2$), and w_{im} ($1 \leq m \leq k_i$) is a term with free variables all in $\{x_1, \cdots, x_{i-1}\}$. u_t is in the $\mathbf{Cond}$-form, because for any constant $\mathtt{d}$, $\mathbf{Cond}(\mathtt{d})$ holds. Hence by Lemma B.2, the denotation of term t (the value c) is either $\perp$ or a definable value of type τ. □

Lemma B.4. *For any logical relation $(\mathcal{R}_\tau)_{\tau \text{ type}}$, $\sim_\tau \subseteq \mathcal{R}_\tau \implies \sim_{\mathsf{T}\tau} \subseteq \mathcal{R}_{\mathsf{T}\tau}$.*

Proof. We assume that $\sim_\tau \subseteq \mathcal{R}_\tau$. Take any two elements $(c_1, c_2) \notin \mathcal{R}_{\mathsf{T}\tau}$. There are two cases:

- $c_1, c_2 \in [\![\tau]\!]$ but $(c_1, c_2) \notin \mathcal{R}_\tau$, then $c_1 \not\sim_\tau c_2$. If one of these two values is not definable at type τ, by Proposition B.3, it is not definable at type $\mathsf{T}\tau$ either. If both values are definable at type τ but are not contextually equivalent, then there is a context $x : \tau \vdash \mathbb{C} : \mathsf{T}o$ such that $[\![\mathbb{C}]\!][x \mapsto c_1] \neq [\![\mathbb{C}]\!][x \mapsto c_2]$. Thus, the context $y : \mathsf{T}\tau \vdash \texttt{let}\ x \Leftarrow y\ \texttt{in}\ \mathbb{C} : \mathsf{T}o$ can distinguish c_1 and c_2 (as two values of type $\mathsf{T}\tau$).

- $c_1 \in [\![\tau]\!]$ and $c_2 = \perp$ (or symmetrically, $c_1 = \perp$ and $c_2 \in [\![\tau]\!]$, then the context $\texttt{let}\ x \Leftarrow y\ \texttt{in}\ \texttt{val}(\texttt{true})$ can be used to distinguish them.

In both cases, $c_1 \not\sim_{\mathsf{T}\tau} c_2$, hence $\sim_{\mathsf{T}\tau} \subseteq \mathcal{R}_{\mathsf{T}\tau}$. $\square$

Theorem B.5. *Logical relations for the partial computation monad are complete up to first-order types, in the strong sense that there exists an observational logical relation $(\mathcal{R}_\tau)_{\tau \text{ type}}$ such that for any closed terms t_1, t_2 of a type τ^1 up to first order,*

$$t_1 \approx_{\tau^1} t_2 \quad \Longrightarrow \quad ([\![t_1]\!], [\![t_2]\!]) \in \mathcal{R}_{\tau^1}$$

Proof. Take the logical relation $(\mathcal{R}_\tau)_{\tau\ type}$ induced by $\mathcal{R}_b = \sim_b$, for any base type b. It can be proved by induction on types that $\sim_{\tau^1} \subseteq \mathcal{R}_{\tau^1}$ for any type τ^1 up to first order (using Lemma B.4 for the induction case of monadic types). $\square$

B.2 Exceptions

The exception monad is seen as generalization of the partial computation monad. The semantics of monadic types and of the `val` and `let` constructs are given by

$$\begin{array}{rcl} [\![\mathsf{T}\tau]\!] & = & [\![\tau]\!] \cup E \\ [\![\mathtt{val}(t]\!]\rho & = & [\![t]\!]\rho \\ [\![\mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2]\!]\rho & = & \begin{cases} [\![t_2]\!]\rho[x \mapsto [\![t_1]\!]\rho] & \text{if } [\![t_1]\!]\rho \notin E \\ [\![t_1]\!]\rho & \text{if } [\![t_1]\!]\rho \in E \end{cases} \end{array}$$

where E is a fixed set of exceptions. Logical relations at monadic types are given by [GLLN02]:

$$c_1\ \mathcal{R}_{\mathsf{T}\tau}\ c_2 \Longleftrightarrow c_1\ \mathcal{R}_\tau\ c_2\ \text{ or }\ c_1 = c_2 \in E$$

Let $\mathbf{Cond}$ be the smallest set of closed terms such that, for any closed term t of type $\tau_1 \to \cdots \to \tau_n \to \mathsf{T}\tau$, $\mathbf{Cond}(t)$ holds if and only if, for any terms $\vdash t_1 : \tau_1, \cdots, \vdash t_n : \tau_n$, $[\![t]\!]([\![t_1]\!], \cdots, [\![t_n]\!])$ is either:

- an exception e in E, or
- in $[\![\tau]\!]$ and definable by some closed term t' of type τ, and if τ is again of the form $\tau'_1 \to \cdots \to \tau'_m \to \mathsf{T}\tau'$, then $\mathbf{Cond}(t')$ holds.

We assume that for any constant $\mathtt{d}$, $\mathbf{Cond}(\mathtt{d})$ holds. We also assume that there is, for every type τ and every exception $e \in E$, a constant $\mathtt{raise}^e_\tau$ of type $\mathsf{T}\tau$ such that $[\![\mathtt{raise}^e_\tau]\!] = e$. Clearly, $\mathbf{Cond}(\mathtt{raise}^e_\tau)$ holds.

Proposition B.6. *A value $c \in [\![\mathsf{T}\tau]\!]$ is definable at type $\mathsf{T}\tau$, if and only if, either $c \in [\![\tau]\!]$ and c is definable at type τ, or $c = e$ for some $e \in E$.*

Proof. Similarly as in proofs of Lemma B.2 and Proposition B.3, this is proved by induction on the computational canonical form of terms. □

Lemma B.7. *For any logical relation* $(\mathcal{R}_\tau)_{\tau\ \mathrm{type}}$, $\sim_\tau \subseteq \mathcal{R}_\tau \implies \sim_{\mathsf{T}\tau} \subseteq \mathcal{R}_{\mathsf{T}\tau}$.

Proof. We assume that $\sim_\tau \subseteq \mathcal{R}_\tau$. Take any pair of computations $(c_1, c_2) \notin \mathcal{R}_{\mathsf{T}\tau}$. There are three cases where $(c_1, c_2) \notin \mathcal{R}_{\mathsf{T}\tau}$:

- $c_1, c_2 \in [\![\tau]\!]$ but $(c_1, c_2) \notin \mathcal{R}_\tau$, then $c_1 \not\sim_\tau c_2$. Suppose both values are definable at type τ, otherwise by Proposition B.3, they must not be definable at type $\mathsf{T}\tau$. Similar as in the proof of Lemma B.4, we can build a context that distinguishes c_1 and c_2 as values of type $\mathsf{T}\tau$, from the context that distinguishes c_1 and c_2 as values of type τ.

- $c_1 \in [\![\tau]\!], c_2 \in E$. Consider the following context:

$$y : \mathsf{T}\tau \vdash \mathtt{let}\ x \Leftarrow y\ \mathtt{in}\ \mathtt{val}(\mathtt{true}) : \mathsf{Tbool}.$$

  When y is substituted by c_1 and c_2, the context evaluates to different values, namely, a boolean and an exception.

- $c_1, c_2 \in E$ but $c_1 \neq c_2$. Try the same context as in the second case, which will evaluate to two different exceptions that can be distinguished.

In all the three cases, we have $c_1 \not\sim_{\mathsf{T}\tau} c_2$, hence $\sim_{\mathsf{T}\tau} \subseteq \mathcal{R}_{\mathsf{T}\tau}$. □

Theorem B.8. *Logical relations for the exception monad are complete up to first-order types, in the strong sense that there exists an observational logical relation* $(\mathcal{R}_\tau)_{\tau\ type}$ *such that for any closed terms* t_1, t_2 *of any type* τ^1 *up to first order,*

$$t_1 \approx_{\tau^1} t_2 \quad \Longrightarrow \quad [\![t_1]\!]\ \mathcal{R}_{\tau^1}\ [\![t_2]\!]$$

Proof. Take the logical relation $(\mathcal{R}_\tau)_{\tau\ type}$ induced by $\mathcal{R}_b = \sim_b$, for any base type b. We prove by induction on types that $\sim_{\tau^1} \subseteq \mathcal{R}_{\tau^1}$ for any type τ^1 up to first order. The induction step at monadic types is in particular proved in Lemma B.7. □

It is interesting to note that our proof of completeness does not require any language primitive able to distinguish a normal value from an exception, or between two different exceptions. It is because our contextual equivalence is defined at the level of semantics instead of syntax. In practice this means that, even though the language itself does not provide any mechanism to capture exceptions, we can still observe them and tell the difference when programs throw exceptions.

B.3 Non-determinism

In the case of non-determinism, the semantics of monadic types and relevant constructs are defined by

$$\begin{array}{rcl} [\![\mathsf{T}\tau]\!] & = & \mathbb{P}_{\mathit{fin}}[\![\tau]\!] \\ [\![\mathtt{val}(t)]\!]\rho & = & \{[\![t]\!]\rho\} \\ [\![\mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2]\!]\rho & = & \bigcup_{a\in[\![t_1]\!]\rho} [\![t_2]\!]\rho[x\mapsto a] \end{array}$$

where $\mathbb{P}_{\mathit{fin}}(S)$ is the set of finite subsets of S. Logical relations at monadic types are given by [GLLN02]:

$$\begin{array}{rcl} c_1\ \mathcal{R}_{\mathsf{T}\tau}\ c_2 & \iff & (\forall a_1 \in c_1.\ \exists a_2 \in c_2.\ a_1\ \mathcal{R}_\tau\ a_2) \\ & & \quad \&\ \ (\forall a_2 \in c_2.\ \exists a_1 \in c_1.\ a_1\ \mathcal{R}_\tau\ a_2) \end{array} \tag{B.7}$$

Let $\mathbf{Cond}$ be the smallest set of closed terms such that, for any closed term t of type $\tau_1 \to \cdots \to \tau_n \to \mathsf{T}\tau$, $\mathbf{Cond}(t)$ holds if and only if:

- for any closed terms $\vdash t_1 : \tau_1, \cdots, \vdash t_n : \tau_n$, $[\![t]\!]([\![t_1]\!], \cdots, [\![t_n]\!])$ is a finite set where each element is definable at type τ (by a closed term t'), and,
- if τ is again of the form $\tau_1' \to \cdots \to \tau_m' \to \mathsf{T}\tau'$, then, for every t', $\mathbf{Cond}(t')$ holds.

We assume that for any constant $\mathtt{d}$, $\mathbf{Cond}(\mathtt{d})$ holds. We also assume that there is, for every τ, a constant $+_\tau$ of type $\tau \to \tau \to \mathsf{T}\tau$ and a constant $\emptyset_\tau$ of type $\mathsf{T}\tau$ such that for any $a_1, a_2 \in [\![\tau]\!]$, $[\![+_\tau]\!](a_1, a_2) = \{a_1\} \cup \{a_2\}$ and $[\![\emptyset_\tau]\!] = \emptyset$. Obviously, $\mathbf{Cond}(+_\tau)$ and $\mathbf{Cond}(\emptyset_\tau)$ hold.

Lemma B.9. *For any closed term t (of type $\mathsf{T}\tau$) in $\mathbf{Cond}$-form, $[\![t]\!]$ is a finite set of definable values of type τ.*

Proof. Because t is of the $\mathbf{Cond}$-form,

$$t \equiv \mathtt{let}\ x_1 \Leftarrow t_1\ \mathtt{in} \cdots \mathtt{let}\ x_n \Leftarrow t_n\ \mathtt{in}\ \mathtt{val}(u), \qquad (n = 0, 1, 2, \ldots).$$

We reason by induction on n:

- In the base case ($n = 0$): $[\![\mathtt{val}(u)]\!] = \{[\![u]\!]\}$. It is obvious that $[\![u]\!]$ is definable at type τ (by the term u in particular).
- For any $n \geq 1$,

$$[\![t_1]\!] = [\![u_1 w_{11} \cdots w_{1k_1}]\!] = [\![u_1]\!]([\![w_{11}]\!], \cdots, [\![w_{1k_1}]\!]),$$

where $u_1, w_{11}, \cdots, w_{1k_1}$ are all closed terms and $\mathbf{Cond}(u_1)$ holds, so every element of $[\![t_1]\!]$ is definable at type τ_1 (note that t_1 is of type $\mathsf{T}\tau_1$). Suppose that for every $a \in [\![t_1]\!]$, there is a closed term t_1^a such that $[\![t_1^a]\!] = a$. Because $\mathbf{Cond}(u_1)$ holds, for every $a \in [\![t_1]\!]$, $\mathbf{Cond}(t_1^a)$ holds as well, hence $\mathbf{Cond}(u_2[t_1^a/x_1])$, ..., $\mathbf{Cond}(u_n[t_1^a/x_1])$ hold (because $u_2[t_1^a/x_1], \ldots, u_n[t_1^a/x_1]$ are either t_1^a or a constant). Let $t_i^a = t_i[t_1^a/x_1]$ ($2 \leq i \leq n$), then

$$\begin{aligned} & [\![\texttt{let}\ x_1 \Leftarrow t_1\ \texttt{in}\ \texttt{let}\ x_2 \Leftarrow t_2\ \texttt{in} \cdots \texttt{let}\ x_n \Leftarrow t_n\ \texttt{in}\ \texttt{val}(u)]\!] \\ = & \bigcup_{a \in [\![t_1]\!]} [\![\texttt{let}\ x_2 \Leftarrow t_2\ \texttt{in} \cdots \texttt{let}\ x_n \Leftarrow t_n\ \texttt{in}\ \texttt{val}(u)]\!][x_1 \mapsto a] \\ = & \bigcup_{a \in [\![t_1]\!]} [\![\texttt{let}\ x_2 \Leftarrow t_2^a\ \texttt{in} \cdots \texttt{let}\ x_n \Leftarrow t_n^a\ \texttt{in}\ \texttt{val}(u[t_1^a/x_1])]\!]. \end{aligned}$$

Clearly, for every $a \in [\![t_1]\!]$, $\texttt{let}\ x_2 \Leftarrow t_2^a\ \texttt{in} \cdots \texttt{let}\ x_n \Leftarrow t_n^a\ \texttt{in}\ \texttt{val}(u[t_1^a/x_1])$ is again in $\mathbf{Cond}$-form, so by induction, its denotation is a finite set of definable values of type τ, and so is the union of all these sets, since $[\![t_1]\!]$ is also finite. $\square$

Proposition B.10. *A value $c \in [\![\mathsf{T}\tau]\!]$ is definable at type $\mathsf{T}\tau$, if and only if, for any $a \in c$, a is definable at type τ.*

Proof. By considering the computational canonical form of those terms defining the value c and applying Lemma B.9. $\square$

However, for the non-determinism monad, we are not able to achieve the completeness of logical relations for any type up to first order. We assume that for every non-observable base type b, there is an equality test constant $\texttt{test}b : b \to b \to \mathsf{bool}$ (clearly, $\mathbf{Cond}(\texttt{test}b)$ holds). We then show that logical relations for the non-determinism monad are complete for weak first-order types.

Theorem B.11. *Logical relations for the non-determinism monad are complete up to weak first-order types (as defined in (B.1)), in the strong sense that there exists an observational logical relation $(\mathcal{R}_\tau)_{\tau\ \text{type}}$ such that for any closed terms t_1, t_2 of a weak first-order type τ_w^1,*

$$t_1 \approx_{\tau_w^1} t_2 \implies [\![t_1]\!]\ \mathcal{R}_{\tau_w^1}\ [\![t_2]\!]$$

Proof. Take the logical relation $(\mathcal{R}_\tau)_{\tau\ \text{type}}$ induced by $\mathcal{R}_b = \sim_b$, for any base type b. We prove by induction on types that $\sim_{\tau_w^1} \subseteq \mathcal{R}_{\tau_w^1}$ for any weak first-order type τ_w^1.

Cases b and $b \to \tau_w^1$ go identically as in normal typed lambda-calculi (Chapter 6). For monadic types $\mathsf{T}b$, suppose that $(c_1, c_2) \notin \mathcal{R}_{\mathsf{T}b}$, which means that either there is a value in c_1 such that no value of c_2 is related to it, or there is such a value in c_2. We assume that every value in c_1 and c_2 is definable (otherwise it is obvious that $c_1 \not\sim_{\mathsf{T}b} c_2$ because at least one of them is not definable, according to Proposition B.10). Suppose there is a value $a \in c_1$ such that no value in c_2 is related to it, and a can be defined by a closed term t of type b. Then the following context can distinguish c_1 and c_2:

$$x : \mathsf{T}\tau \vdash \mathtt{let}\ y \Leftarrow x\ \mathtt{in}\ \mathtt{test}_b(y, t) : \mathsf{Tbool}$$

since every value in c_2 is not contextually equivalent to a, hence not equal to a. $\square$

B.4 State transformers

In the case of the state monad, the semantics of the monadic types and constructs are defined by:

$$\begin{array}{rcl}
[\![\mathsf{T}\tau]\!] & = & ([\![\tau]\!] \times St)^{St} \\
[\![\mathtt{val}(t)]\!]\rho & = & s \longmapsto ([\![t]\!]\rho, s) \\
[\![\mathtt{let}\ x \Leftarrow t_1\ \mathtt{in}\ t_2]\!]\rho & = & s \longmapsto [\![t_2]\!]\rho[x \mapsto a_1](s_1) \\
 & & \qquad \text{where } a_1 = \pi_1([\![t_1]\!]\rho(s)), s_1 = \pi_2([\![t_2]\!]\rho(s))
\end{array}$$

where St is a finite set of states. Logical relations at monadic types are given by [GLLN02]:

$$c_1\ \mathcal{R}_{\mathsf{T}\tau}\ c_2 \iff \forall s \in St.\ \pi_1(c_1 s)\ \mathcal{R}_\tau\ \pi_1(c_2 s)\ \ \&\ \ \pi_2(c_1 s) = \pi_2(c_2 s)$$

Let $\mathbf{Cond}$ be the smallest set of closed terms such that, for any closed term t of type $\tau_1 \to \cdots \to \tau_n \to \mathsf{T}\tau$, $\mathbf{Cond}(t)$ holds if and only if,

- for any closed terms $\vdash t_1 : \tau_1, \cdots, \vdash t_n : \tau_n$, $[\![t]\!]([\![t_1]\!], \cdots, [\![t_n]\!])$ is a function such that for any $s \in St$, $[\![t]\!]([\![t_1]\!], \cdots, [\![t_n]\!])(s) = (a, s')$ where $s' \in St$ and a is definable at type τ (by some closed term t'), and
- if τ is of the form $\tau_1' \to \cdots \to \tau_m' \to \mathsf{T}\tau'$, then $\mathbf{Cond}(t')$ holds.

We assume that for any constant $\mathtt{d}$, $\mathbf{Cond}(\mathtt{d})$ holds. Let unit be the base type which contains only a dummy value $*$. We assume that there is, for each $s \in St$, a constant $\mathtt{update}_s$ of type Tunit such that for any $s' \in St$, $[\![\mathtt{update}_s]\!](s') = (*, s)$. This constant does nothing but change the current state to s. Clearly, $\mathbf{Cond}(\mathtt{update}_s)$ holds.

Lemma B.12. *For any closed term t (of type $\mathsf{T}\tau$) in $\mathbf{Cond}$-form, and for any $s \in St$, $\pi_1([\![t]\!]s)$ is definable at type τ.*

Proof. Because t is of the $\mathbf{Cond}$-form,

$$t \equiv \mathtt{let}\ x_1 \Leftarrow t_1\ \mathtt{in} \cdots \mathtt{let}\ x_n \Leftarrow t_n\ \mathtt{in}\ \mathtt{val}(u), \qquad (n = 0, 1, 2, \ldots).$$

We reason by induction on n:

- In the base case ($n = 0$), for every $s \in St$, $[\![\mathtt{val}(u)]\!]s = ([\![u]\!], s)$. It is obvious that $[\![u]\!]$ is definable at type τ (by the term u in particular).

- For any $n \geq 1$,

$$[\![t_1]\!] = [\![u_1 w_{11} \cdots w_{1k_1}]\!] = [\![u_1]\!]([\![w_{11}]\!], \cdots, [\![w_{1k_1}]\!]),$$

where $u_1, w_{11}, \cdots, w_{1k_1}$ are all closed terms and $\mathbf{Cond}(u_1)$ holds, so for every $s \in St$, $\pi_1([\![t_1]\!](s))$ is definable at type τ_1 (note that t_1 is of type $\mathsf{T}\tau_1$). Suppose that for every $s \in St$, t_1^s is a closed term of type τ_1 such that $\pi_1([\![t_1]\!](s)) = [\![t_1^s]\!]$. Because $\mathbf{Cond}(u_1)$ holds, $\mathbf{Cond}(t_1^s)$ holds as well, hence $\mathbf{Cond}(u_2[t_1^s/x_1])$, ..., $\mathbf{Cond}(u_n[t_1^s/x_1])$ hold (because $u_2[t_1^s/x_1], \ldots, u_n[t_1^s/x_1]$ are either t_1^s or a constant). For every $s \in St$, let $t_i^s = t_i[t_1^s/x_1]$ ($2 \leq i \leq n$), then

$$\begin{aligned}
& [\![\mathtt{let}\ x_1 \Leftarrow t_1\ \mathtt{in}\ \mathtt{let}\ x_2 \Leftarrow t_2\ \mathtt{in} \cdots \mathtt{let}\ x_n \Leftarrow t_n\ \mathtt{in}\ \mathtt{val}(u)]\!](s) \\
= \ & [\![\mathtt{let}\ x_2 \Leftarrow t_2\ \mathtt{in} \cdots \mathtt{let}\ x_n \Leftarrow t_n\ \mathtt{in}\ \mathtt{val}(u)]\!][x \mapsto [\![t_1^s]\!]](s') \\
= \ & \left[\!\left[\mathtt{let}\ x_2 \Leftarrow t_2^s\ \mathtt{in} \cdots \mathtt{let}\ x_n \Leftarrow t_n'\ \mathtt{in}\ \mathtt{val}(u[t_1^s/x_1])\right]\!\right](s')
\end{aligned}$$

where $s' = \pi_2([\![t_1]\!](s))$. Clearly, for every $a \in [\![t_1]\!]$,

$$\mathtt{let}\ x_2 \Leftarrow t_2^s\ \mathtt{in} \cdots \mathtt{let}\ x_n \Leftarrow t_n^s\ \mathtt{in}\ \mathtt{val}(u[t_1^s/x_1])$$

is again in $\mathbf{Cond}$-form, so by induction, its denotation, when applied to any state, is a pair of a definable value at type τ and a state in St. □

Proposition B.13. *If a value $c \in [\![\mathsf{T}\tau]\!]$ is definable at type $\mathsf{T}\tau$, then, for any $s \in St$, $\pi_1(cs)$ is definable at type τ.*

Proof. By considering the computational canonical form of corresponding terms. □

Lemma B.14. *For any logical relation $(\ \mathcal{R}_\tau\ )_{\tau\ \text{type}}$, $\sim_\tau \subseteq \mathcal{R}_\tau \implies \sim_{\mathsf{T}\tau} \subseteq \mathcal{R}_{\mathsf{T}\tau}$.*

Proof. We assume that $(c_1, c_2) \notin \mathcal{R}_{\mathsf{T}\tau}$, so there exists some $s_0 \in St$ such that

- either $(\pi_1(c_1 s_0), \pi_1(c_2 s_0)) \notin \mathcal{R}_\tau$. Then by induction $\pi_1(c_1 s_0) \not\sim_\tau \pi_1(c_2 s_0)$. If $\pi_1(c_i s_0)$ ($i = 1, 2$) is not definable, then by Proposition B.13, c_i is not definable either. If both $\pi_1(c_1 s_0)$ and $\pi_1(c_2 s_0)$ are definable, but $\pi_1(c_1 s_0) \not\approx_\tau \pi_1(c_2 s_0)$, then there is a context $x : \tau \vdash \mathbb{C} : \mathsf{T}o$ such that $[\![\mathbb{C}]\!][x \mapsto \pi_1(c_1 s_0)] \neq [\![\mathbb{C}]\!][x \mapsto \pi_1(c_2 s_0)]$, i.e., for some state $s_0' \in St$,

$$[\![\mathbb{C}]\!][x \mapsto \pi_1(c_1 s_0)](s_0') \neq [\![\mathbb{C}]\!][x \mapsto \pi_1(c_1 s_0)](s_0')$$

Now we can use the following context

$$y : \mathsf{T}\tau \vdash \mathtt{let}\ x \Leftarrow y\ \mathtt{in}\ \mathtt{let}\ z \Leftarrow \mathtt{update}_{s'_0}\ \mathtt{in}\ \mathbb{C} : To,$$

Let $f_i = [\![\mathtt{let}\ x \Leftarrow y\ \mathtt{in}\ \mathtt{do}\ \mathbb{C}\ \mathtt{at}\ s'_0]\!][y \mapsto c_i]$ $(i = 1, 2)$, then for any $s \in St$,

$$\begin{aligned} f_i(s) &= \left[\!\!\left[\mathtt{let}\ z \Leftarrow \mathtt{update}_{s'_0}\ \mathtt{in}\ \mathbb{C}\right]\!\!\right][x \mapsto \pi_1(c_i s)](\pi_2(c_i s)) \\ &= [\![\mathbb{C}]\!][x \mapsto \pi_1(c_i s)](s'_0), \qquad (i = 1, 2). \end{aligned}$$

$f_1 \neq f_2$, because when applied to the state s_0, they will return two different pairs, so the above context can distinguish the two values c_1 and c_2;

- or $\pi_2(c_1 s_0) \neq \pi_2(c_2 s_0)$. We use the context

  $$y : \mathsf{T}\tau \vdash \mathtt{let}\ x \Leftarrow y\ \mathtt{in}\ \mathtt{val}(\mathtt{true}) : \mathsf{Tbool},$$

  then

  $$[\![\mathtt{let}\ x \Leftarrow y\ \mathtt{in}\ \mathtt{val}(\mathtt{true})]\!][y \mapsto c_i] = \lambda s.(\mathtt{true}, \pi_2(c_i s)) \quad (i = 1, 2)$$

  These two functions are not equal since they return different results when applied to the state s_0.

In both cases, $c_1 \not\sim_{\mathsf{T}\tau} c_2$, hence $\sim_{\mathsf{T}\tau} \subseteq \mathcal{R}_{\mathsf{T}\tau}$. ☐

Note that in the above proof, we assume that contexts can distinguish different states. For this purpose, the language must provide some mechanism to read the current state. Since usually a state is just a set of variables with values assigned to them, such a "state access" mechanism is just retrieving values of these variables.

Theorem B.15. *Logical relations for the state monad are complete up to first-order types, in the strong sense that there exists an observational logical relation $(\mathcal{R}_\tau)_{\tau\ type}$ such that for any closed terms t_1, t_2 of any type τ^1 up to first order,*

$$t_1 \approx_{\tau^1} t_2 \implies [\![t_1]\!]\ \mathcal{R}_{\tau^1}\ [\![t_2]\!]$$

Proof. Take the logical relation $(\mathcal{R}_\tau)_{\tau\ type}$ induced by $\mathcal{R}_b = \sim_b$, for any base type b. We prove by induction on types that $\sim_{\tau^1} \subseteq \mathcal{R}_{\tau^1}$ for any type τ^1 up to first order. The induction step at monadic types is proved by Lemma B.14. ☐

Bibliographie

[Aba99] Martín Abadi. Security protocols and specifications. In *Foundations of Software Science and Computation Structures : Second International Conference (FOSSACS)*, volume 1578 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1999.

[Abr90] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, 1990.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th Annual Symposium on Principles of Programming Languages (POPL)*, pages 104–115. ACM Press, January 2001.

[AG97] Martín Abadi and Andrew D. Gordon. Reasoning about cryptographic protocols in the Spi calculus. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR)*, volume 1243 of *Lecture Notes in Computer Science*, pages 59–73. Springer-Verlag, 1997.

[AG98] Martín Abadi and Andrew D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5(4) :267–303, 1998.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols : The Spi calculus. *Journal of Information and Computation*, 148(1) :1–70, 1999.

[AHS90] Jiří Adámek, Horst Herrlich, and George E. Strecker. *Abstract and Concrete Categories*. Pure and applied mathematics. John Wiley and Sons, New York, 1990.

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures : An Introduction to Category Theory for the Working Computer Scientist*. The MIT Press, Cambridge, MA, 1991.

[AL00] Roberto Amadio and Denis Lugiez. On the rechability problems in cryptographic protocols. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR)*, volume 1877 of *Lecture Notes in Computer Science*, pages 380–394. Springer-Verlag, 2000.

[Bar80] Hank P. Barendregt. *The Lambda-Calculus : Its Syntax and Semantics*. North-Holland, 1980.

[Bar91] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1991.

[BBdP98] P. N. Benton, G. M. Bierman, and V. C. V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2) :177–193, March 1998.

[Bec69] Jon Beck. Distributive laws. In B. Eckmann, editor, *Seminar on Triples and Categorical Homology Theory, ETH, Zürich, 1966/67*, volume 80 of *Lecture Notes in Mathematics*, pages 119–140. Springer-Verlag, Berlin, 1969.

[BN02] Johannes Borgström and Uwe Nestmann. On bisimulations for the spi calculus. In Hélène Kirchner and Christophe Ringeissen, editors, *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 2422 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2002.

[BNP99] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Proof techniques for cryptographic processes. In *Proceedings of the 14th Symposium on Logic in Computer Science (LICS)*, pages 157–166. IEEE Computer Society, July 1999.

[Bor01] Michele Boreale. Symbolic trace analysis of cryptographic protocols. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of *Lecture Notes in Computer Science*, pages 667–681. Springer-Verlag, 2001.

[BW90] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, London, 1990.

[CDL05] Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 2005. To appear.

[CJ97] John A. Clark and Jeremy L. Jacob. A survey of authentication protocol literature. Version 1.0, University of York, Department of Computer Science, November 1997.

[CS02] Hubert Comon and Vitaly Shmatikov. Is it possible to decide whether a cryptographic protocol is secure or not ? *Journal of Telecommunications and Information Technology*, 4, 2002.

[DES] DES. The data encryption standard. FIPS PUB 46.

[DK02] Hans Delfs and Helmut Knebl. *Introduction to Cryptography — Principles and Applications*. Information Security and Cryptography. Springer-Verlag, 2002.

[DLMS99] Nancy Durgin, Pat D. Lincoln, John C. Mitchell, and Andre Scedrov. Undecidability of bounded security protocols. In *Proceedings of the FLOC Workshop on Formal Methods in Security Protocols*, 1999.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2) :198–208, 1983.

[FA01] Marcelo Fiore and Martín Abadi. Computing symbolic models for verifying cryptographic protocols. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 160–173. IEEE Computer Society, June 2001.

[FS90] Peter J. Freyd and Andre Scedrov. *Categories, Allegories*, volume 39 of *North-Holland Mathematical Library*. North-Holland, Amsterdam, 1990.

[GJ02] Andrew D. Gordon and Alan S. A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Proceedings International Software Security Symposium*, volume 2609 of *Lecture Notes in Computer Science*, pages 263–282. Springer-Verlag, 2002.

[GJ03a] Andrew D. Gordon and Alan S. A. Jeffrey. Authenticity by typing for security protocols. *Journal Computer Security*, 11(4) :451–521, 2003.

[GJ03b] Andrew D. Gordon and Alan S. A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300 :379–409, 2003.

[GJ04] Andrew D. Gordon and Alan S. A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal Computer Security*, 12(3/4) :435–484, 2004.

[GLLN02] Jean Goubault-Larrecq, Sławomir Lasota, and David Nowak. Logical relations for monadic types. In *Proceedings of the 16th International Workshop of Computer Science Logic (CSL)*, volume 2471 of *Lecture Notes in Computer Science*, pages 553–568. Springer-Verlag, 2002.

[GLLNZ04] Jean Goubault-Larrecq, Sławomir Lasota, David Nowak, and Yu Zhang. Complete lax logical relations for cryptographic lambda-calculi. In *Proceedings of the 18th International Workshop of Computer Science Logic (CSL)*, volume 3210 of *Lecture Notes in Computer Science*, pages 400–414. Springer-Verlag, 2004.

[Gor98] Andrew D. Gordon. Operational equivalences for untyped and polymorphic object calculi. In Andrew D. Gordon and Andrew M. Pitts, editors, *Higher-Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 9–54. Cambridge University Press, 1998.

[Gor99] Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, 228(1–2) :5–47, October 1999.

[GR96] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 386–395. ACM Press, January 1996.

[HLS03] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. *Journal of Computer Security*, 11(2) :217–244, 2003.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus : Programming with security and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, 1998.

[Laf87] Yves Lafont. *Logiques, Catégories and Machines*. Ph. D. dissertation, Université Paris 7, September 1987.

[Low95] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3) :131–133, 1995.

[Low96a] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[Low96b] Gavin Lowe. Some new attacks upon security protocols. In *Proceedings of the 9th IEEE Computer Security Foundations Workshop (CSFW)*, pages 162–169. IEEE Computer Society, June 1996.

[LS86] Joachim Lambek and Philip J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge studies in advanced mathematics. Cambridge University Press, 1986.

[LSV] LSV. Security protocols open repository. http ://www.lsv.ens-cachan.fr/spore/.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, New York, 1971.

[Mea00] Catherine Meadows. Open issues in formal methods for cryptographic protocol analysis. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX)*, pages 237–250. IEEE Computer Society Press, January 2000.

[Mea03] Catherine Meadows. Formal methods for cryptographic protocol analysis : Emerging issues and trends. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Series in Computer Science. Prentice Hall International, 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems : The π-Calculus*. Cambridge University Press, 1999.

[Min61] Marvin L. Minsky. Recursive unsolvability of Post's problem of "tag" and other topics in the theory of Turing machines. *Annals of Mathematics, Second Series*, 74(3) :437–455, 1961.

[Mit96] John C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[MM91] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2) :99–124, 1991.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science (LICS)*, pages 14–23. IEEE Computer Society Press, June 1989.

[Mog90] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, LFCS, Department of Computer Science, University of Edinburgh, 1990.

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

[Mor68] James H. Morris. *Lambda-Calculus Models of Programming Languages*. Ph. D. dissertation, Massachusetts Institute of Technology, December 1968. Report No. MAC–TR–57.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1) :1–77, September 1992.

[MR92] Qingming Ma and John C. Reynolds. Types, abstraction and parametric polymorphism, part 2. In *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics (MFPS)*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40. Springer-Verlag, Berlin, 1992.

[MS93] John C. Mitchell and Andre Scedrov. Notes on sconing and relators. In E. Boerger et al., editor, *Proceedings of the 6th International Workshop on Computer Science Logic (CSL)*, volume 702 of *Lecture Notes in Computer Science*, pages 352–378. Springer-Verlag, 1993.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanston, editors. *Handbook of Applied Cryptography*. CRC Press, 1996.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12) :993–999, December 1978.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[Pit98] Andrew M. Pitts. Existential types : Logical relations and operational equivalence. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, 1998.

[Pit00] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10 :321–359, 2000.

[Pit03] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2) :165–193, 2003.

[Plo80] Gordon D. Plotkin. Lambda-definability in the full type hierarchy. In J[onathan] P. Seldin and J. R[oger] Hindley, editors, *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, 1980.

[PPST00] Gordon Plotkin, John Power, Donald Sannella, and Robert Tennent. Lax logical relations. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1853 of *Lecture Notes in Computer Science*, pages 85–102. Springer-Verlag, 2000.

[PS93a] Andrew Pitts and Ian Stark. Observable properties of higher order functions that dynamically create local names, or : What's *new* ? In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, number 711 in Lecture Notes in Computer Science, pages 122–141. Springer-Verlag, 1993.

[PS93b] Andrew Pitts and Ian Stark. On the observable properties of higher-order functions that dynamically create local names (preliminary report). In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, number YALEU/DCS/RR-968 in Research Report, pages 31–45. Yale University Department of Computer Science, 1993.

[RS99] Peter Y. A. Ryan and Steve A. Schneider. Process algebra and non-interference. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW)*, pages 214–227. IEEE Computer Society, June 1999.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1) :5–19, January 2003.

[SP01] Eijiro Sumii and Benjamin Pierce. Logical relations for encryption. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 256–272. IEEE Computer Society, June 2001.

[SP03] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4) :521–554, 2003.

[SP04] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004.

[SP05] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[Sta94] Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994. Also available as Technical Report 363, University of Cambridge Computer Laboratory.

[Sta96] Ian Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1) :77–107, 1996.

[SW01] Davide Sangiorgi and David Walker. *The Pi-Calculus — A Theory of Mobile Processes*. Cambridge University Press, 2001.

[Ten81] Robert D. Tennent. *Principles of Programming Languages*. Prentice-Hall International, 1981.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages : An Introduction.* Foundations of Computing series. MIT Press, February 1993.

[ZN03] Yu Zhang and David Nowak. Logical relations for dynamic name creation. In *Proceedings of the 17th International Workshop of Computer Science Logic and the 8th Kurt Gödel Colloqium (CSL & KGL)*, volume 2803 of *Lecture Notes in Computer Science*, pages 575–588. Springer-Verlag, 2003.