

Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

**Formal Verification of ElGamal Encryption
using a Probabilistic Lambda-Calculus**

submitted by

Malte Skoruppa

on January 20, 2010

Supervisor

Prof. Dr. Michael Backes

Advisor

M.Sc. Matthias Berg

Reviewers

Prof. Dr. Michael Backes

Dr. Dominique Unruh

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, January 20, 2010

Malte Skoruppa

Acknowledgments

This work has benefited from the achievements of many people, some of which I would sincerely like to thank.

First, I want to thank Prof. Michael Backes for giving me such an interesting topic in a state-of-the-art research field in Cryptography. I enjoyed it a lot, and am fortunate to have had the possibility to write a topic which really suited me.

I want to thank Matthias Berg for many interesting discussions, in which I learned a lot. Matthias' explanations shed a lot of light onto this topic. He also took amazingly much of his time to suggest many corrections for this thesis. Thank you.

Furthermore I want to thank my fellow student Jonathan Driedger, the work with whom has been enjoyable and a lot of fun. The vivid discussions with Jonathan have always been very enriching.

Last, but not least, I want to sincerely thank my parents for their continued support throughout my studies, which has been of immeasurable value to me. I would never have made it without you.

Abstract

Game-based proofs are a common technique used to verify cryptographic constructions. Such proofs consist of a sequence of games where each transition from one game to the next can be individually verified. However, as more and more increasingly complex proofs are being published, even such transitions are often non-trivial. Moreover, games are frequently described informally or just in some ad-hoc pseudocode and may be understood differently than originally intended by the authors, or underlying assumptions may not be made explicit.

For this reason, Backes et al. developed a new formal language at the chair of Information Security and Cryptography at the Universität des Saarlandes. This language supports most cryptographic primitives typically used in such games and is intended to provide a formal standard to model them. Furthermore it has been implemented on top of the proof assistant Isabelle/HOL, such that it is possible to use Isabelle's logic to formally verify game transitions.

The goal of this thesis is to provide a first application of this language to a real-world cryptographic construction by using it to formally verify the security of the well-known ElGamal encryption scheme. For this, we use the language to model the scheme as well as the desired security properties and the necessary assumptions. Next, we find appropriate game transformations in the language and formally prove their validity. Finally we show how to use these transformations to achieve a fully formalized game-based proof of the security of ElGamal.

Contents

| | |
|---|-----------|
| Notations | 1 |
| 1 Introduction | 3 |
| 2 Related Work | 5 |
| 3 The Framework | 7 |
| 3.1 On the λ -calculus | 7 |
| 3.2 Syntax of the Language | 9 |
| 3.3 Special Operators | 12 |
| 3.3.1 The Lift Operator $\uparrow P$ | 12 |
| 3.3.2 The Substitution Operator $P\{Q\}$ | 13 |
| 3.3.3 The Instantiation Operator P^a | 15 |
| 3.4 Semantics of the Language | 17 |
| 3.4.1 The Reduction Relation | 17 |
| 3.4.2 The Denotation | 20 |
| 3.5 Syntactic Sugar and Uniform Distributions | 23 |
| 3.6 Program Relations | 25 |
| 3.6.1 Denotational Equivalence | 25 |
| 3.6.2 Observational Equivalence | 26 |
| 3.6.3 Computational Indistinguishability | 26 |
| 3.6.4 Properties of these Relations | 27 |
| 4 Cryptographic Basics | 33 |
| 4.1 About Encryption Schemes | 33 |
| 4.2 Chosen-Plaintext Attack | 34 |
| 4.3 The Diffie-Hellman Problem | 36 |
| 4.4 The ElGamal Encryption Scheme | 37 |
| 5 Formalization in Isabelle/HOL | 39 |
| 5.1 The CPA Game | 39 |
| 5.2 The Security Parameter | 40 |
| 5.3 ElGamal | 41 |
| 5.4 The Adversary | 42 |
| 6 Program Transformations | 45 |

Contents

| | | |
|----------|--|-----------|
| 6.1 | Transformations based on Observational Equivalence | 45 |
| 6.1.1 | β -equivalence | 45 |
| 6.1.2 | Expression Propagation | 47 |
| 6.1.3 | Expression Subsumption | 49 |
| 6.1.4 | Nesting Chained Applications | 49 |
| 6.1.5 | Chaining Nested Applications | 52 |
| 6.1.6 | Line Swapping | 53 |
| 6.2 | A Transformation based on Computational Indistinguishability | 56 |
| 6.2.1 | The DDH Assumption | 56 |
| 6.3 | A Transformation based on Denotational Equivalence | 57 |
| 6.3.1 | Multiplication with Random Elements in Cyclic Groups | 57 |
| 7 | The Proof of IND-CPA Security of ElGamal | 63 |
| 7.1 | IND_CPA1 ElGamal to ElGamalGameI | 64 |
| 7.2 | ElGamalGameI to DDHxy | 66 |
| 7.3 | DDHxy to DDHz | 69 |
| 7.4 | DDHz to RandomElGamalGame | 70 |
| 7.5 | Finishing the Proof | 76 |
| 8 | Conclusions | 77 |
| | References | 79 |

Notations (in order of occurrence)

\mathbf{B} The set of basic values of the language, see Def. 3.3.

$\uparrow P$ All free variables in P are increased by one; see Def. 3.5.

$\uparrow_k P$ All free variables in P with indices greater or equal than k are lifted by one; see Def. 3.5.

$P\{V\}$ The value V is substituted into the program P ; see Def. 3.7.

P^a For a program P and a list of values a , the instantiation of all free variables in P with the values in a ; see Def. 3.9.

$P|\sigma|\eta$ For a program P and store σ and event list η , the program state defined by the triple (P, σ, η) ; see Def. 3.12.

$B|\sigma|\eta$ For a set of programs B and store σ and event list η , the set $\{x|\sigma|\eta : x \in B\}$.

\mathcal{PS} The set of all program states $P|\sigma|\eta$, see Definition 3.12.

\mathcal{VS} The set of all value states $V|\sigma|\eta$, see Definition 3.13.

$P|\sigma|\eta \rightsquigarrow \mu$ For a program state $P|\sigma|\eta$ and a measure on program states μ , the statement “ $P|\sigma|\eta$ reduces to μ ”, see Def. 3.14.

δ_x The Dirac measure associated with $x \in X$, defined by $\delta_x(B) = 1 \Leftrightarrow x \in B$ and $\delta_x(B) = 0 \Leftrightarrow x \notin B$, for all $B \in \Sigma_X$.

$\Lambda x. f(x)$ For an element $x \in X$ and a function $f : X \rightarrow Y$, the (unnamed) mathematical function which for x yields the value $f(x)$. Here we use a capital Λ to distinguish mathematical functions from abstractions $\lambda x. f(x)$ of the language.

$f(\mu)$ For a measurable function $f : X \rightarrow Y$ and a measure μ on X , the measure on Y defined by $f(\mu)(B) = \mu(\{x \in X : f(x) \in B\})$, for all $B \in \Sigma_Y$.

$g \cdot \mu$ For a kernel $g : X \rightarrow Y$ and a measure μ on X , the application $g \cdot \mu$ defined by $(g \cdot \mu)(B) := \int g(x)(B) d\mu(x)$, for all $B \in \Sigma_Y$; see Def. 3.16.

$g \circ f$ For a kernel $f : X \rightarrow Y$ and a kernel $g : Y \rightarrow Z$, the kernel composition $g \circ f$ defined by $(g \circ f)(x) := g \cdot (f(x))$, for all $x \in X$; see Def. 3.17.

$\llbracket P|\sigma|\eta \rrbracket$ The denotation of a program P with state σ and event list η , as defined in Def. 3.19.

$\llbracket P \rrbracket$ Shorthand notation for $\llbracket P|\square|\square \rrbracket$, where \square is an empty list.

$T(P|\sigma|\eta)$ For a program state $P|\sigma|\eta$, the probability of termination according to the semantics of the language, see Def. 3.20.

$P|\sigma|\eta \sim Q|\sigma|\eta$ $P|\sigma|\eta$ and $Q|\sigma|\eta$ are denotationally equivalent, see Def. 3.26.

$P \sim Q$ P and Q are denotationally equivalent, see Def. 3.26.

$P \equiv_{obs} Q$ P and Q are observationally equivalent, see Def. 3.28.

$P \equiv_{CIU} Q$ P and Q are CIU-equivalent, see Def. 3.29.

$(P_n)_{n \in \mathbb{N}} \approx_{ind_n} (Q_n)_{n \in \mathbb{N}}$ The families $(P_n)_{n \in \mathbb{N}}$ and $(Q_n)_{n \in \mathbb{N}}$ are computationally indistinguishable in n , see Def. 3.31.

$P \equiv_{obs}^{INV} Q$ P and Q are observationally equivalent under certain invariants.

If we talk of measures on sets X, Y, \dots or measurable functions $X \rightarrow Y$, we assume implicitly the existence of canonical σ -algebras $\Sigma_X, \Sigma_Y, \dots$ on X, Y, \dots , respectively. In particular, as canonical σ -algebras $\Sigma_{\mathbb{R}^+}$ and $\Sigma_{\mathbb{R}}$ on \mathbb{R}^+ and \mathbb{R} , respectively, we always choose the Borel-algebras, i.e. the smallest σ -algebra on \mathbb{R}^+ respectively \mathbb{R} which contains all closed intervals $[a, b]$.

1 Introduction

The main result of this thesis is the proof of the semantic security of ElGamal using game-based transformations in a new formal language developed at the chair of Michael Backes [2]. For this, we find the necessary transformations and prove their validity.

We explain this now in more detail.

The Game-Playing Technique. In modern cryptography, a prominent technique to reason about security properties of cryptographic constructions are so-called *game transformations*. The idea is that one can formulate some desired security property as a probabilistic experiment, which usually models an interaction between a *challenger* and some *adversary*; this experiment is the initial *game*. In such a game, the challenger uses some underlying cryptographic construction and then has to give a certain amount of information to the adversary. The goal is to show that the probability that the adversary may perform a successful attack on the construction (where the meaning of a “successful attack” depends on the context) is in some sense insignificant or *negligible*. To prove this, one transforms the initial game in a series of steps. For each step, one shows that the probability of the outcome of the previous game is changed at most by some negligible amount. The last game usually has a form where it is easy to give an upper bound for the probability of the outcome, or where such a bound is already known or assumed.

The need for more rigor. However, with the increasing complexity and multitude of proofs generated by the cryptographic community, this approach is inherently error-prone. Indeed, as argued by Halevi [17], the algorithms that we deal with are non-trivial (and indeed truly complex), and the proofs concern properties of a sophisticated interaction between these algorithms. On the other hand, the games are usually simply described in just a semi-formal way, using some ad-hoc pseudocode or even just natural language. Thus, ambiguities may be introduced (e.g. some assumptions may not be made explicit), and depending on the interpretation of some game, the correctness of a transformation may not always be easily believed. Thus, Backes et al. [2] implemented a full-fledged language to model cryptographic games in a fully formal way, and which knows several kinds of cryptographic primitives used in such games. It is powerful enough to argue about probabilistic behavior, the usage of oracles, and even continuous probability measures, so that it is even possible to argue about information-theoretic security guarantees. Furthermore, it has been implemented on top of the proof assistant Isabelle/HOL [19], and several kinds of program relations have been formalized, so that it is possible to use the power of Isabelle’s logic to formally verify the validity of the transformations.

Notions of security. Two well-known security properties of encryption schemes are those of *semantic security* and *the indistinguishability of encryptions*. The semantic security of an encryption scheme is the most natural definition that comes to mind, and intuitively states that no adversary should be able to recover any part of a plaintext by looking at the ciphertext. The more technical definition of the indistinguishability of encryptions asserts that it is infeasible for an adversary to distinguish the encryptions of two self-chosen plaintexts. While the first definition seems more natural, the second one is usually far easier to deal with formally. Fortunately, today these definitions are known to be exactly equivalent; that is, semantic security implies indistinguishability of encryptions and vice-versa (see for example [16]). Hence, to show that an encryption scheme is semantically secure, one can show that it has indistinguishable encryptions.

ElGamal. The *ElGamal* encryption scheme, invented in 1985 by Taher Elgamal [13], is a widely used asymmetric encryption scheme based on the difficulty of computing certain discrete logarithms. More precisely, the scheme is well-known to be semantically secure under the assumption that the (decisional) Diffie-Hellman problem is hard to solve in certain cyclic groups. The proof can be beautifully performed using game-based transformations, and is at the same time a good demonstration of this technique.

Goal. As mentioned at the beginning, the goal of this thesis is to show the usefulness of the language to perform game-based cryptographic proofs by using it to formally define the ElGamal encryption scheme, find and formalize appropriate game transformations to show that ElGamal has indistinguishable encryptions, and prove their validity.

Outline. The outline of the thesis is as follows. We begin by presenting some related work in Section 2. In Section 3, we will introduce the language invented by Backes et al., formally define its semantics and define several kinds of program relations. In Section 4, we will define what it means for an encryption scheme to have indistinguishable encryptions and explain the ElGamal encryption scheme, as well as the problem it is based on. The new results proposed in this thesis are given in Sections 5 to 7. In Section 5, we will show how to formalize these definitions in the language. In Section 6, we introduce, formalize and prove several kinds of game-based transformations. Finally, in Section 7 we apply the results explained in the preceding sections to the concrete transformations which prove the indistinguishability of encryptions of ElGamal. We conclude with an outlook in Section 8.

2 Related Work

We point out some related work, in (roughly) chronological order.

In [6], Bellare and Rogaway take a fresh look at the game-playing technique and argue for its usefulness, as it is a widely applicable and easily verifiable technique to perform cryptographic proofs. They present some examples in a pseudo-code language, and discuss that for truly rigorous statements, the language should be fully specified. In [17], Halevi supports this view and furthermore advocates the creation of an automated tool to help with this task. He argues that with the help of such a tool, humans could concentrate on the creative part of proofs, such as finding appropriate transformations, while the tool could help to verify the mundane part, such as checking that some transformation goes through.

CryptoVerif [8, 7] was the first approach to build such a tool relying on the computational model (i.e. proofs by reductions of games) to automatically prove the security of both cryptographic primitives and cryptographic protocols. Blanchet and Pointcheval use it to prove the unforgeability of the Full-Domain Hash signature scheme under the trapdoor-one-wayness of some permutations. However this tool has some limitations as it comes to mathematical reasoning, and is rather suited for security protocols which use high-level primitives such as encryptions or signatures.

In [11], Courant, Daubignard, Ene, Lafourcade and Lakhnech present a different approach for automatically proving several security notions such as IND-CPA and IND-CCA2 security. They define a Hoare logic for a simple programming language, and use it to implement an automated verification procedure. However, this approach has the drawback that they have to sacrifice completeness for soundness, that is, their procedure may fail to prove a secure encryption scheme, while it will never declare one secure which is insecure. For this reason, they do not achieve high assurance e.g. when dealing with exact security.

The framework which comes closest to the one presented in this thesis is CertiCrypt, presented in [4, 3, 24]. It also takes a code-centric approach to formalize game-based transformations. Barthe et al. use an imperative programming language with random assignments, structured datatypes and procedure calls built upon the proof assistant Coq [23]. Their framework provides a set of certified tools to reason about equivalence of probabilistic programs, including a relational Hoare logic, a theory of observational equivalence, verified program transformations and cryptographic techniques such as reasoning about failure events [4]. Thus their framework is similar to ours, although their language is not higher-order and therefore it might be difficult to reason about oracles. Furthermore, their framework does not allow reasoning about continuous probability measures, so they cannot reason about e.g. information theoretic security guarantees. However, a fair amount of proofs have already been achieved in CertiCrypt, such as the unforgeability of FDH

signatures [4, 24], as well as the the IND-CPA security of OAEP [4] and of ElGamal [4, 3], with a proof similar to the one presented in this thesis.

The language that we use in this thesis was first published by Michael Backes, Matthias Berg and Dominique Unruh in [2]. It is actively being developed at the Information Security and Cryptography Group at the Universität des Saarlandes. Recently, Jonathan Driedger submitted his Bachelor's thesis on the formalization of a number of game-based transformations in our language [12], some of which we relied on in this thesis.

3 The Framework

In this section, we will explain the basic framework that we will build on, as published in [2]. More precisely, we will first introduce the language, including its formal syntax and semantics. Afterwards, we can define different kinds of program equivalences that have been formulated in Isabelle/HOL. In this way, we know how to write a game as a program and what a valid transformation is.

The language that has been implemented in Isabelle/HOL is a higher-order functional probabilistic language. It is higher-order, because it has to be able to deal with oracles, and from a language perspective, oracles are higher-order arguments passed to functions. For this reason, it is functional, since functional languages deal with higher-order arguments much more naturally than imperative languages. Furthermore, it has to be probabilistic, since we want to be able to argue about probabilistic behaviors. The language is powerful enough to even deal with continuous probability measures, so that it is possible to reason about, e.g., infinite random tapes for establishing information-theoretic security guarantees. Finally, it also includes a store and references, which is normally not the case in purely functional languages. However, since we want to be able to deal with, e.g., stateful oracles (and simply passing a state around as an object is insufficient, because this might violate secrecy properties, e.g., an oracle might be obliged to output its secret key), we have to extend the language with a store. Similarly, we explicitly include events and an event list in the language, since events are a salient technique used in cryptographic proofs.

As we are dealing with a functional language, it is only natural to model it as a λ -calculus. Hence, in Section 3.1, we will give a quick overview over some general concepts of the λ -calculus. We will proceed by defining the syntax of the language in Section 3.2. Then we will introduce a few special operators in Section 3.3 that are needed in the following sections. In Section 3.4, we will define the semantics of the language, and in Section 3.5 we will introduce some syntactic sugar. Finally, in Section 3.6, we will formalize three kinds of commonly known program relations that justify typical game transformations.

3.1 On the λ -calculus

The λ -calculus is a formal system first published by Alonzo Church in 1936 [10]. It is widely used to model the mechanisms of a functional language, by concentrating on its core aspect: functions. Basically, all computation is reduced to the basic operations of function definition and application. The core syntax of a λ -calculus comprises only three sorts of terms.

| | |
|----------------|--------------------|
| $t ::=$ | <i>terms</i> |
| x | <i>variable</i> |
| $\lambda x. t$ | <i>abstraction</i> |
| $t t$ | <i>application</i> |

A variable x is by itself a term. The abstraction of a variable x from a term t_1 , written $\lambda x. t_1$, is a term. It denotes a function that takes an input x and returns the evaluation of t_1 , where some value v is substituted for x in t_1 . Values are a subset of the set of terms, comprising in such a simple λ -calculus only the variables and the abstractions (and things like e.g. natural numbers in more sophisticated languages). For example, the function

$$\text{factorial} = \lambda n. \text{ if } n=0 \text{ then } 1 \text{ else } n * \text{factorial}(n-1)$$

is a function that takes a natural number n , and returns the factorial of n . Note that here we have defined neither natural numbers, if-then-else constructs, $=$ or $*$ operators or recursion, so this is really just to give an intuition what a λ -term looks like. Finally, the application of a term t_1 to another term t_2 , written $t_1 t_2$, is a term. To continue our example,

$$\text{factorial } 3$$

is an application, and intuitively should evaluate to the value 6.

Indeed, what one now usually does is define an evaluation relation (often called *big-step semantics*), which for each term unambiguously states what it *evaluates to* (if it can be evaluated). Similarly, one can define a *small-step semantics* that well-defines what a term *reduces to* in one step. Note the difference in the meaning of a *reduction* and an *evaluation*: the reduction yields the next term in the course of the evaluation, whereas the evaluation yields its final value. Jumping ahead, we will define similar things in our language. What we will call the *denotation* of a program term will correspond to some kind of big-step semantics, while our *reduction relation* will be a small-step semantics. The notions will still be a bit different though, since our language is probabilistic, so it is not clear a priori what “the next term” in a reduction should be, as a term may reduce to several possible other terms.

Of course, in complex (say interesting) programming languages one additionally introduces a typing system. Furthermore one enriches the language with additional primitives like e.g. natural numbers, and data structures such as pairs or lists. Then one can prove important properties of the language, such as Coincidence (if a program P eventually reduces in a finite number of steps to a value v using the small-step semantics, then the big-steps semantics yields exactly this value v for the program P), the Uniqueness of Types (if a program P has type T and it also has T' , then $T = T'$), Progress (every closed and well-typed term is either a value or reducible), Type Preservation (the type of a program P is preserved under reduction), and Type Safety (if P is a closed and well-typed term, then either there is a value v such that P evaluates to v or the reduction does not terminate; notice this follows from Progress and Type Preservation). Since, in the following, we will

define our own language anyway, we do not get into further details at this point. A much deeper insight into the λ -calculus in general can be found in [20].

One thing is worth noting before we start to define the language, however. This is the so-called *nameless representation of terms* (here the word “names” refers to variables, although generally speaking names can include more than only variables, e.g., names may also refer to constants of the language). We strive for a representation of terms that does not use variable names such as x in a λ -abstraction, because from a computational point of view names are problematic. For example, the terms $\lambda x. x$ and $\lambda y. y$ denote exactly the same function, but they are not equal since the names of the bound variables are different (one says they are “*equal up to α -renaming*”, but it is not immediately clear what this means computationally). Thus, we would prefer to have a canonical representation of terms which does not use names for variables. To accomplish this, instead of using names to refer to λ -binders, we use natural numbers to make variable occurrences point directly to their respective binders; such a number is called a *deBruijn index* (see [20, Sect. 6]). The construct $\text{var } k$, called a *deBruijn variable*, then stands for “the variable bound by the k ’th enclosing λ .”

Example 3.1. The following table shows a few ordinary terms and their corresponding nameless version.

| Ordinary term | Nameless term |
|---------------------------------|--|
| $\lambda x. x$ | $\lambda. \text{var } 0$ |
| $\lambda y. y$ | $\lambda. \text{var } 0$ |
| $\lambda x \lambda y. x (y x)$ | $\lambda. \lambda. \text{var } 1 (\text{var } 0 \text{ var } 1)$ |
| $\lambda x. x (\lambda y. x y)$ | $\lambda. \text{var } 0 (\lambda. \text{var } 1 \text{ var } 0)$ |

It is also important to distinguish *bound variables* and *free variables*. Free variables are variables that are not bound by a λ , but occur freely in a term. Bound variables are variables bound to a λ . In a deBruijn setting, we can say a variable is bound iff its index is strictly less than the number of enclosing λ ’s.

Example 3.2. In the term $\lambda. \lambda. \text{var } 2 (\text{var } 0 \text{ var } 1)$, the variables $\text{var } 0$ and $\text{var } 1$ are bound, but $\text{var } 2$ is free.

Defining a syntax based on the deBruijn notation makes sense from a computational point of view. It also makes the job of defining a substitution operation (in view of a small-step semantics) much easier. Hence, our language is defined using the deBruijn notation. A drawback is that for us humans, these so-called *deBruijn terms* are a lot less readable. For this reason, we will often write ordinary terms with names, when on a low level we are actually dealing with nameless terms. We are comfortable doing so, since any ordinary term can easily and uniquely be mapped to a corresponding nameless term.

3.2 Syntax of the Language

We model the syntax as probabilistic higher-order λ -calculus, extended with references and events, and make use of an iso-recursive type system. We express the syntax using

inductive grammars of terms, values (as a subset of terms), pure values (as a subset of values), types and pure types (as a subset of types).

Since we define the syntax formally and we want to introduce probabilism, we will use some notions from mathematical probability theory, such as σ -algebras and sub-Markov kernels, but do not go into further details at this point. Rather, we give an intuitive idea of what the constructs mean and do. We will come back to these notions from probability theory when we define the semantics of the language in Section 3.4.

We first define the syntax, and proceed with a more detailed explanation.

Definition 3.3 (Syntax). The sets of program terms P , of values V , of pure values V_0 , of types T and of pure types T_0 , are defined by the following grammars, where $n \in \mathbb{N}$, $v \in \mathbf{B}$ (see below), $X \in \Sigma_{\mathbf{B}}$, s denotes strings, and f denotes sub-Markov kernels of type $V_0 \rightarrow (\Sigma_{V_0} \rightarrow \mathbb{R}^+)$.

$$\begin{aligned}
 P & ::= \text{var } n \mid \lambda.P \mid PP \mid \text{fun}(f, P) \mid \text{value } v \mid \text{loc } n \mid \text{ref } P \mid \\
 & \quad !P \mid P := P \mid \text{event } s \mid \text{eventlist} \mid (P, P) \mid \text{fst } P \mid \text{snd } P \mid \\
 & \quad \text{inl } P \mid \text{inr } P \mid \text{case } P P P \mid \text{fold } P \mid \text{unfold } P \\
 \\
 V & ::= \text{value } v \mid \text{var } n \mid (V, V) \mid \lambda.P \mid \text{loc } n \mid \text{inl } V \mid \text{inr } V \mid \text{fold } V \\
 V_0 & ::= \text{value } v \mid (V_0, V_0) \mid \text{inl } V_0 \mid \text{inr } V_0 \mid \text{fold } V_0 \\
 \\
 T & ::= \text{Value}_X \mid T \times T \mid T + T \mid T \rightarrow T \mid \text{Ref } T \mid \mu.T \mid \text{Tvar } n \\
 T_0 & ::= \text{Value}_X \mid T_0 \times T_0 \mid T_0 + T_0 \mid \mu.T_0 \mid \text{Tvar } n
 \end{aligned}$$

Programs P . The first three sorts of program terms are the familiar variables, abstractions and applications, in a deBruijn setting.

Furthermore we introduce probability into the language with the construct $\text{fun}(f, P)$. The function f is a sub-Markov kernel which, intuitively, takes a pure value and returns a measure on pure values. It can be interpreted as applying f to P , thus obtaining a distribution over V_0 . f can be used to express any mathematical (deterministic) function, but also randomized functions (and is not even restricted to discrete probability measures).

The construct $\text{value } v$ is used to introduce an element $v \in \mathbf{B}$ in programs. Here we note that for the sake of extensibility, the language does not fix a set of basic types, but rather assumes a type \mathbf{B} that contains the basic types we expect to need, like a type unit (with a single element unit), the type of real numbers real , or function types like $\text{nat} \rightarrow \text{bool}$.

The next four constructs are used to access the store. To reference the $(n+1)$ -st element of the store, which is represented as a list, one uses the construct $\text{loc } n$. $\text{ref } P$ and $!P$ are used for reference creation and dereferencing, respectively. For two programs P and Q , the construct $P := Q$ denotes the assignment of Q to P .

Two constructs are available to deal with events. The construct $\text{event } s$ is used to raise an event s , and eventlist returns a list of all previously raised events.

For two programs P and Q , the program (P, Q) represents the pair of P and Q , and the constructs $\text{fst } P$ and $\text{snd } P$ are the first and second projections of a pair P . Sums

(programs that have a sum type) can be constructed using $\text{inl } P$ and $\text{inr } P$ and destructed using $\text{case } P_1 P_2 P_3$.

Finally, the constructs $\text{fold } P$ and $\text{unfold } P$ are used to convert recursive types, which is typical in an iso-recursive setting. For a program P of type $\mu x.T$, the construct $\text{unfold } P$ makes the recursive type explicit by recursively substituting $\mu x.T$ for x in T . The construct $\text{fold } P$ does the inverse. It folds such an explicit type T into a recursive type $\mu x.T'$. A good tutorial on this topic can be found in [14].

As we will in fact only need a small subset of these terms, we do not go into further details here.

Values V . Values are a subset of terms and cannot be reduced any more. Intuitively, for well-typed and closed terms they correspond to those terms whose evaluation is finished. They include, as usual, variables and abstractions. Furthermore $\text{loc } n$ (for $n \in \mathbb{N}$) and value v (for $v \in \mathbf{B}$) are values. A pair (V_1, V_2) of two values V_1 and V_2 is a value, and the constructs $\text{inl } V$, $\text{inr } V$ and $\text{fold } V$, for a value V , are values. Intuitively, the set of pure values V_0 is a subset of the set of values which corresponds to the fixed “datatypes” and do not include the more abstract functions, variables or locations. The pure types are those that can be used for sampling using the $\text{fun}(f, P)$ construct.

Types T . The type constructor Value_X denotes, for a measurable set X , the type of basic values contained in X .

Furthermore there exist some compound types. For some types T and U , the type $T \times U$ denotes the product type, and the type $T + U$ denotes the sum type. $T \rightarrow U$ is the type of all functions from T to U .

$\text{Ref } T$ denotes the type of references of type T .

Finally, the construct $\mu.T$ is used to introduce recursively defined types, as described in [20, Sect. 20]. Recursive types can be used, e.g., to construct the types of lists. Such types describe an infinite set, but have a regular structure. Here μ is a recursive type constructor (like a combination of fix and λ for types), which introduces a binder in deBruijn notation. Accordingly, the constructor $\text{Tvar } n$ denotes the type variable with deBruijn index n , that is, the type variable that belongs to the $(n + 1)$ -st enclosing μ . An example follows.

Intuitively, the set of pure types T_0 is the set of those types in T which correspond to the types of pure values.

Example 3.4. Recall the usual definition of lists using nil and cons . The type of a list of nats is either unit (for the empty list) or a pair type $\text{nat} \times \text{natlist}$, where natlist is again the type of a list of nats . So, using our definition of sum types, we would intuitively define the type natlist as something like

$$\text{natlist} ::= \text{unit} + \text{nat} \times \text{natlist}$$

More generally speaking, for some type X , the type $\text{list } X$ of a list over elements of type X is the sum type of unit and a pair type of the form $X \times \text{list } X$. Now, the μ -binder comes in

handy to define such a type. For readability, similarly as for λ , we do not use the deBruijn notation, but rather write $\mu x.T$, where x is like a variable for types. Intuitively, $\mu x.T$ represents the type obtained by recursively substituting $\mu x.T$ for x in T . Thus, we would define the type list X as

$$\begin{aligned} & \mu x.\text{Value}_{\text{unit}} + X \times x \\ \text{which unfolds to } & \text{Value}_{\text{unit}} + X \times (\mu x.\text{Value}_{\text{unit}} + X \times x) \\ & \text{and so on...} \end{aligned}$$

In DeBruijn notation, the type of list X is then

$$\mu.\text{Value}_{\text{unit}} + X \times \text{Tvar } 0$$

3.3 Special Operators

We will now introduce some special operators, which we will need in the following sections and chapters. These include a lift operator (typical in a deBruijn setting), a substitution operator (to define β -reduction), and finally an instantiation operator (a kind of a generalized substitution operator).

3.3.1 The Lift Operator $\uparrow P$

A lift operator is used to shift deBruijn indices in a term. This is useful in many setups, e.g., in view of defining a substitution operator (see [20, Sect. 6]). We will also need it to formalize certain transformations.

Essentially, the lift operator increases all occurrences of *free* variables in a program term P by one. However, since we do not want to increase bound variables within a term, we need to keep count of how many λ -binders we have already encountered (remember a variable is free iff its index is greater or equal than the number of enclosing λ 's). Hence, we first define the more general lift operator $\uparrow_k P$, which, for a natural number k and a program term P , increases all variables whose index is greater or equal than k by one.

Definition 3.5 (Lift Operator). Let $k \in \mathbb{N}$, and P be a program term. Then the operator $\uparrow_k P$ is defined in Figure 3.1. Furthermore we use the notation $\uparrow P := \uparrow_0 P$.

Notice that only the two first cases are non-trivial:

1. In case we encounter a variable, we increase its index by one iff it is free (that is, if $i \geq k$); otherwise we do not change it
2. In case we encounter a λ -abstraction of a term P , we increase k by one (since we have found a new binder) and recursively descend into P .

In all the other cases, we either apply straightforward recursion, or we found a base case and do not change anything.

$$\begin{aligned}
\uparrow_k \text{ var } i &= \begin{cases} \text{ var } i & \text{if } i < k \\ \text{ var } i + 1 & \text{otherwise} \end{cases} \\
\uparrow_k \lambda. P &= \lambda. \uparrow_{k+1} P \\
\uparrow_k (P_1 P_2) &= (\uparrow_k P_1)(\uparrow_k P_2) \\
\uparrow_k \text{ fun}(f, P) &= \text{ fun}(f, \uparrow_k P) \\
\uparrow_k \text{ value } v &= \text{ value } v \\
\uparrow_k \text{ loc } n &= \text{ loc } n \\
\uparrow_k (\text{ ref } P) &= \text{ ref } (\uparrow_k P) \\
\uparrow_k (!P) &= !(\uparrow_k P) \\
\uparrow_k (P_1 := P_2) &= (\uparrow_k P_1) := \uparrow_k P_2 \\
\uparrow_k \text{ event } e &= \text{ event } e \\
\uparrow_k \text{ eventlist} &= \text{ eventlist} \\
\uparrow_k \text{ fold } P &= \text{ fold } (\uparrow_k P) \\
\uparrow_k \text{ unfold } P &= \text{ unfold } (\uparrow_k P) \\
\uparrow_k (P_1, P_2) &= (\uparrow_k P_1, \uparrow_k P_2) \\
\uparrow_k \text{ fst } P &= \text{ fst } (\uparrow_k P) \\
\uparrow_k \text{ snd } P &= \text{ snd } (\uparrow_k P) \\
\uparrow_k \text{ inl } P &= \text{ inl } (\uparrow_k P) \\
\uparrow_k \text{ inr } P &= \text{ inr } (\uparrow_k P) \\
\uparrow_k \text{ case } P_1 P_2 P_3 &= \text{ case } (\uparrow_k P_1) (\uparrow_k P_2) (\uparrow_k P_3)
\end{aligned}$$

Figure 3.1: Definition of the lift operator

3.3.2 The Substitution Operator $P\{Q\}$

A substitution operator is normally used to substitute a term for a variable in a program P . Thus, it can be used to model β -reduction; consider a program $(\lambda x.P)V$, where x is a variable, P a program and V a value. This term should reduce to the term $[x \mapsto V]P$ (which we call a β -reduction). It means we remove the λx . in front of P , and replace each occurrence of x in P with V .

Similarly, we would like to be able to say that the nameless term $(\lambda.P)V$ reduces to $P\{V\}$, meaning “in P , replace all variables referring to the leading λ in $\lambda.P$ by V ”. Moreover, we will not restrict ourselves to values V , but define the substitution operator such that it can substitute any program term Q for a variable. Thus, our substitution operator $P\{Q\}$ essentially has three jobs:

1. Clearly, replace each occurrence of a variable bound to the leading λ in $\lambda.P$ by the program Q (in P , this is $\text{var } 0$ at the beginning, but since we may encounter λ 's in P , it may become $\text{var } 1$, $\text{var } 2$, etc.)
2. Furthermore, observe that, when we remove the leading λ in $\lambda.P$, then there is one enclosing λ less; hence all free variables in $\lambda.P$ should be shifted down by one, so that they still refer to the same binders after we removed the leading λ .

3. Finally, there may be free variables in Q . Since we substitute Q for certain variables in P , there may be now be several more enclosing λ -binders around Q , but we still want the free variables in Q to reference the same binders. Consequently, we need to shift the free variables in Q as necessary.

Example 3.6. Let $P := \text{var } 0 (\lambda. \text{var } 1 \text{ var } 2)$. Let $V := \lambda. \text{snd var } 1$. We would like to have that $(\lambda. P)V \rightarrow P\{V\}$, i.e. we would like the following term to reduce as shown.

$$\begin{aligned} & (\lambda. \text{var } 0 (\lambda. \text{var } 1 \text{ var } 2))(\lambda. \text{snd var } 1) \\ \rightarrow & (\lambda. \text{snd var } 1)(\lambda. (\lambda. \text{snd var } 2) \text{ var } 1) \end{aligned}$$

All three jobs of the substitution operator can be seen in action here. First, every occurrence of a variable in the original term that referred to the leading λ has been replaced by the value at the right (and we removed this leading λ , which is the job of the β -reduction). This corresponds to job (1). Furthermore, in the second replacement, we changed $\text{snd var } 1$ to $\text{snd var } 2$; this is because since there is one more enclosing λ there, we need to shift the variable to reference the same binder as originally. This is job (3). Finally, the free variable $\text{var } 2$ in the original term has been shifted down by one, since now there is one enclosing λ less than before. This is job (2).

Indeed, designing the substitution operator in such a way means that we get β -reduction “for free”. Intuitively, we will just be able to model β -reduction by saying that for any program P and value V , we have that $(\lambda. P)V \rightarrow P\{V\}$.

Similarly as for the lift operator, we first define a more general operator that keeps track of how many λ 's we encountered so far, so that we know which variables to replace, and how.

Definition 3.7 (Substitution Operator). Let $k \in \mathbb{N}$ and let P and Q be programs. Then the substitution operator $[k \mapsto Q]P$ that replaces $\text{var } k$ in P by Q is defined in Figure 3.2. Furthermore we define the notation $P\{Q\} := [0 \mapsto Q]P$.

As before, only the two first cases are non-trivial, while all others are either straightforward recursion or base cases.

1. In case P has the form $\text{var } i$, there are three possibilities. Either $\text{var } i$ is a free variable, but not the one we want to replace, because the number of enclosing λ 's is strictly less than the variable index i ; in this case, we decrement i . This corresponds to job (2). Or the variable is free and exactly the one we want to replace; in this case we substitute Q for the variable. This is job (1). The last possibility is that the variable is bound, in this case we do not change it.
2. When we encounter a λ , we increment k , as now the variable we are looking for is one larger. Furthermore, we lift Q , so that all its free variables reference the same binders as before. This corresponds to job (3). Then we descend recursively into the body of the abstraction.

$$\begin{array}{lcl}
[k \mapsto Q] \text{ var } i & = & \begin{cases} \text{var } i - 1 & \text{if } k < i \\ Q & \text{if } k = i \\ \text{var } i & \text{otherwise} \end{cases} \\
[k \mapsto Q] \lambda. P & = & \lambda. [k + 1 \mapsto \uparrow Q]P \\
[k \mapsto Q] P_1 P_2 & = & ([k \mapsto Q]P_1) ([k \mapsto Q]P_2) \\
[k \mapsto Q] \text{ fun}(f, P) & = & \text{fun}(f, [k \mapsto Q]P) \\
[k \mapsto Q] \text{ value } v & = & \text{value } v \\
[k \mapsto Q] \text{ loc } n & = & \text{loc } n \\
[k \mapsto Q] \text{ ref } P & = & \text{ref } ([k \mapsto Q]P) \\
[k \mapsto Q] !P & = & ![k \mapsto Q]P \\
[k \mapsto Q] (P_1 := P_2) & = & ([k \mapsto Q]P_1) := [k \mapsto Q]P_2 \\
[k \mapsto Q] \text{ event } e & = & \text{event } e \\
[k \mapsto Q] \text{ eventlist} & = & \text{eventlist} \\
[k \mapsto Q] \text{ fold } P & = & \text{fold } ([k \mapsto Q]P) \\
[k \mapsto Q] \text{ unfold } P & = & \text{unfold } ([k \mapsto Q]P) \\
[k \mapsto Q] (P_1, P_2) & = & ([k \mapsto Q]P_1, [k \mapsto Q]P_2) \\
[k \mapsto Q] \text{ fst } P & = & \text{fst } ([k \mapsto Q]P) \\
[k \mapsto Q] \text{ snd } P & = & \text{snd } ([k \mapsto Q]P) \\
[k \mapsto Q] \text{ inl } P & = & \text{inl } ([k \mapsto Q]P) \\
[k \mapsto Q] \text{ inr } P & = & \text{inr } ([k \mapsto Q]P) \\
[k \mapsto Q] \text{ case } P_1 P_2 P_3 & = & \text{case } ([k \mapsto Q]P_1) ([k \mapsto Q]P_2) ([k \mapsto Q]P_3)
\end{array}$$

Figure 3.2: Definition of the substitution operator

3.3.3 The Instantiation Operator P^a

Sometimes, we may consider a program P with several free variables, and might want to instantiate every such free variable with some value, so as to obtain a closed term. The values which we want to instantiate these variables with may then be represented in some list of values a . To say that we instantiate every free variable in P with a corresponding value from a , we will write P^a .

In practice, we can implement such an instantiation operator nicely using our previously defined substitution operator. For a program P , let us call the *substitution candidates* those variables which would be replaced with some value V if we wrote $P\{V\}$ (i.e. $\text{var } 0$ in a term without any λ -binders, after one λ -binder $\text{var } 1$ etc.). Notice that there must not always necessarily exist a substitution candidate. Furthermore, there may be only one or several substitution candidates in a term. But we can always uniquely determine those variables. Now for a term P and a list of values a , what we want our substitution operator to do is to apply the substitution operator to substitute the first value in a for the substitution candidates in P . Then, since our substitution operator also decrements all free variables by one, there may be new substitution candidates. We want to substitute the second value in a for these new candidates, and so on, until we worked through the

entire list of values a .

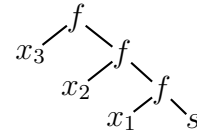
If the list a is long enough (and the values in a do not introduce new free variables themselves), our program P will eventually be closed. Actually, this is the only interesting scenario in which we want to use the instantiation operator. If a was not long enough, or if it introduced new free variables, there may still be free variables in P . Here we are not interested in studying this behavior in detail, since when using the instantiation operator, we will always require that a be such that P^a is closed.

Example 3.8. Consider the term $P := (\lambda.\lambda.\text{var } 1) \text{ var } 0 \text{ var } 2$. Furthermore, let a list of values $a := [\text{value } 4, \text{value } 5, \text{value } 6]$. Let us study the term P^a .

$$\begin{aligned}
 P^a &\stackrel{\text{def}}{=} ((\lambda.\lambda.\text{var } 1) \text{ var } 0 \text{ var } 2)^{[\text{value } 4, \text{value } 5, \text{value } 6]} \\
 &= \left(((\lambda.\lambda.\text{var } 1) \text{ var } 0 \text{ var } 2) \{ \text{value } 4 \} \right)^{[\text{value } 5, \text{value } 6]} \\
 &= ((\lambda.\lambda.\text{var } 1) \text{ value } 4 \text{ var } 1)^{[\text{value } 5, \text{value } 6]} \\
 &= \left(((\lambda.\lambda.\text{var } 1) \text{ value } 4 \text{ var } 1) \{ \text{value } 5 \} \right)^{[\text{value } 6]} \\
 &= ((\lambda.\lambda.\text{var } 1) \text{ value } 4 \text{ var } 0)^{[\text{value } 6]} \\
 &= ((\lambda.\lambda.\text{var } 1) \text{ value } 4 \text{ var } 0) \{ \text{value } 6 \} \\
 &= ((\lambda.\lambda.\text{var } 1) \text{ value } 4 \text{ value } 6)
 \end{aligned}$$

Here, we first substituted value 4 for the substitution candidate. In doing so, we also decremented the free variable var 2. Then, we tried to substitute value 5 for the next candidates; but there were none. However, we still decremented var 1 to var 0, such that there was a new candidate for the next round, where we substituted value 6 for var 0. The $\lambda.\lambda.\text{var } 1$ was never touched, since its variable is bound.

Folding is a recursion scheme which folds a list according to a function f and an initial value s . As an example, consider the expression $\text{foldl } f \ s$ which folds the list $[x_1, x_2, x_3]$ as follows.

$$\text{foldl } f \ s \ [x_1, x_2, x_3] = f(x_3, f(x_2, f(x_1, s)))$$


For further explanations see [22, Sect. 4]. The foldl function can be defined recursively as

$$\text{foldl}: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha \text{ list} \rightarrow \beta$$

$$\begin{aligned}
 \text{foldl } f \ s \ \text{nil} &= s \\
 \text{foldl } f \ s \ (x :: xr) &= \text{foldl } f \ (f \ x \ s) \ xr
 \end{aligned}$$

Now, we can define the instantiation operator, using foldl and the substitution operator, in a straightforward way.

Definition 3.9 (Instantiation Operator). Let P be a program, and a be a list of values. Then the instantiation P^a is defined as

$$P^a := \text{foldl}(\lambda v. \lambda p. p\{v\}) P a$$

3.4 Semantics of the Language

3.4.1 The Reduction Relation

We are about to define the reduction relation \rightsquigarrow , which is the small-step semantics of our language. However, as already mentioned, our language also has to deal with locations and events. For this reason, our reduction relation cannot just map programs to programs, since we have to carry additional information for these constructs. At this point, let us define a few notions.

Definition 3.10 (The store). The *store* of a program, usually denoted by σ , is a list of values. It allows programs to store information in a separate memory using $\text{ref } V$, which inserts V at the end of the list σ . The pointers to these values are denoted by $\text{loc } n$ (the value stored at the n -th index). Dereferencing is also available using $!(\text{loc } n)$, and location update via $\text{loc } n := V$.

Definition 3.11 (The event list). The *event list*, usually denoted by η , is similar to the store in that we carry it around as an extra memory. It is a list of strings. When an event is raised in a program (using event s), the string s is inserted at the end of the list. The construct `eventlist` outputs this list.

Definition 3.12 (Program state). For a program P , a store σ and an event list η , the *program state* is the triple (P, σ, η) , which we write as $P|\sigma|\eta$ by convention. Intuitively, it corresponds to the state of a program run, where P is the remaining code to execute, and σ and η are the current store and event list. We use \mathcal{PS} for the set of all program states.

Definition 3.13 (Value state). A program state of the special form $V|\sigma|\eta$, where V is a value, is called a *value state*. Intuitively, it is a state which cannot be reduced any more. We denote by \mathcal{VS} the set of all value states.

Now we already get a better idea of what our reduction relation should look like. Instead of mapping programs to programs, we would map program states to program states. However, this is still not enough. Since the language is probabilistic, for a program state $P|\sigma|\eta$, there is not always one unique program state $Q|\sigma|\eta$ such that $P|\sigma|\eta$ can be uniquely mapped to it. Instead, a program state may reduce to several different other program states. Hence, our reduction relation will in fact map program states to measures on program states. Let $P|\sigma|\eta$ be a program state. By applying the reduction relation to $P|\sigma|\eta$, we obtain a measure which for a set of program states B yields the probability that $P|\sigma|\eta$ reduces to one of the program states in B .

We now present the reduction rules, and proceed with explanatory comments.

Definition 3.14. Let E be an evaluation context (see below), σ a store, η an event list, P a program, V a value, and f a sub-Markov kernel from V_θ to V_θ (also explained below). The semantic reduction rules are defined as follows.

$$\begin{aligned}
 E ::= & \square \mid \text{fun}(f, E) \mid (E, P) \mid (V, E) \mid EP \mid VE \mid \text{ref } E \mid !E \mid (E := P) \mid (V := E) \\
 & \mid \text{fst } E \mid \text{snd } E \mid \text{fold } E \mid \text{unfold } E \mid \text{inl } E \mid \text{inr } E \mid \text{case } E P P \mid \text{case } V E P \\
 & \mid \text{case } V V E
 \end{aligned}$$

$$\begin{array}{lll}
 E[(\lambda.P)V]|\sigma|\eta & \rightsquigarrow & \delta_{E[P\{V\}]|\sigma|\eta} & \text{APPL} \\
 E[\text{ref } V]|\sigma|\eta & \rightsquigarrow & \delta_{E[\text{loc } (|\sigma|)]|\sigma@[V]|\eta} & \text{REF} \\
 E[!(\text{loc } l)]|\sigma|\eta & \rightsquigarrow & \delta_{E[\sigma_l]|\sigma|\eta} & \text{DEREF}, \text{ if } l < |\sigma| \\
 E[\text{loc } l := V]|\sigma|\eta & \rightsquigarrow & \delta_{E[\text{value } \text{unit}]|\sigma[l:=V]|\eta} & \text{ASSIGN}, \text{ if } l < |\sigma| \\
 E[\text{fst}(V, V')]|\sigma|\eta & \rightsquigarrow & \delta_{E[V]|\sigma|\eta} & \text{FST} \\
 E[\text{snd}(V, V')]|\sigma|\eta & \rightsquigarrow & \delta_{E[V']|\sigma|\eta} & \text{SND} \\
 E[\text{fun}(f, V)]|\sigma|\eta & \rightsquigarrow & (\Lambda x. E[x]|\sigma|\eta)(fV) & \text{FUN} \\
 E[\text{event } s]|\sigma|\eta & \rightsquigarrow & \delta_{E[\text{value } \text{unit}]|\sigma|\eta@[s]} & \text{EV} \\
 E[\text{eventlist}]|\sigma|\eta & \rightsquigarrow & \delta_{E[\bar{\eta}]|\sigma|\eta} & \text{EVLIST} \\
 E[\text{case } (\text{inl } V) V_L V_R]|\sigma|\eta & \rightsquigarrow & \delta_{E[V_L V]|\sigma|\eta} & \text{CASEL} \\
 E[\text{case } (\text{inr } V) V_L V_R]|\sigma|\eta & \rightsquigarrow & \delta_{E[V_R V]|\sigma|\eta} & \text{CASER} \\
 E[\text{unfold } (\text{fold } V)]|\sigma|\eta & \rightsquigarrow & \delta_{E[V]|\sigma|\eta} & \text{FOLD}
 \end{array}$$

Figure 3.3: Reduction rules

Fig. 3.3, shows all the reduction rules for our language. First, note that there are two different kinds of specifications: the definition of evaluation contexts E on the one hand, and the definition of the semantic reduction relation \rightsquigarrow on the other hand. Let us consider these in more detail, one after the other.

The definition of evaluation contexts gives us, intuitively, the specification of the *structural* rules (also sometimes referred to as *congruence* rules or *descent* rules), and they tell us *where* to evaluate a term. However, they are not literally rules, but rather specify how an evaluation context can be built. For a better understanding, we first introduce the notion of a *context*.

Definition 3.15 (Context). A context C is a program with zero or more holes \square . For the program obtained by inserting a program P at the holes in C , we write $C[P]$. Mathematically, a context can be seen as a function mapping programs to programs. Contexts are defined by a grammar analogous to that of program terms (see Definition 3.3), but with the additional construct \square .

Now, an *evaluation context* is a special context with the additional property that the hole is at a position where the program should be evaluated first, according to a call-by-value strategy. Formally, they are defined by the inductive grammar E given at the top of Fig. 3.3. For example, when evaluating the application $P_1 P_2$, we first evaluate P_1 until it is a value; when it is a value, we begin to evaluate P_2 until it is a value. This behavior

emerges from the fact that we can build evaluation contexts using the specifications EP and VE , respectively. That is, either the evaluation takes place in the first part, or given the first part is a value, it takes place in the second part. Evaluation contexts constitute an elegant and compact way to specify such structural rules.

Now let's have a look at the second kind of rules. They are called *computation* rules, because they are the ones where the real computation takes place (they are also commonly known as *proper* rules or *top-level* rules). For programs of a certain form, we can explicitly give the distribution that they reduce to. More precisely, we can do this for any evaluation context which is applied to a primitive construct of our language. This is what the reduction relation \rightsquigarrow does. The reduction relation is a relation (partially) defined on the set of program states, which yields for these program states $P|\sigma|\eta$ a sub-probability measure μ on program states.

The most used measure is the so-called *Dirac measure*. For an element $x \in X$, we write δ_x to denote the Dirac measure (on X) associated with x . We use it to model the deterministic reduction steps. For a program state $P|\sigma|\eta$, $\delta_{P|\sigma|\eta}$ is the distribution that returns 1 for a set $B \subseteq \mathcal{PS}$ iff $P|\sigma|\eta \in B$, and 0 otherwise. Hence, for the reduction relation, when we have $Q|\sigma'|\eta' \rightsquigarrow \delta_{P|\sigma|\eta}$, this intuitively tells us that the probability that $Q|\sigma'|\eta'$ reduces to the program state $P|\sigma|\eta$ is 1. Again in the example of an application, when we have a program state $(\lambda.P)V|\sigma|\eta$ (i.e., both sides of the application are values), the probability that this program state reduces to the program state $P\{V\}|\sigma|\eta$ is 1. Observe that we require a special form of the value at the left. When we cannot reduce a program term because there are no rules for it, we say that the execution got *stuck*. This cannot happen with well-typed terms (our language fulfills the properties of Progress and Preservation).

Finally, it remains to understand the rule FUN, i.e. the rule for the probabilistic reduction steps. For this we need to introduce some more notions. First, by the construction $\Lambda x.E[x]|\sigma|\eta$, we mean the mathematical function $g : P \rightarrow \mathcal{PS}$,

$$g(x) := E[x]|\sigma|\eta$$

where x is a program, and σ and η are fixed (they are the same as in the previous state, as this rule does not affect them).

Next, we explain the construction fV . For this, we have to explain, first of all, the notion of a sub-Markov kernel. A *kernel* from X to Y is a function $f : X \rightarrow (\Sigma_Y \rightarrow \mathbb{R}^+)$ such that

1. for all $x \in X$ the function

$$\Sigma_Y \rightarrow \mathbb{R}^+, \quad B \mapsto f(x)(B)$$

defines a measure on Y , and,

2. for all $B \in \Sigma_Y$, the function

$$X \rightarrow \mathbb{R}^+, \quad x \mapsto f(x)(B)$$

is measurable¹.

We call f a *Markov kernel* or a *sub-Markov kernel* if for all $x \in X$, it holds that $f(x)$ is a probability measure, respectively a sub-probability measure. For a more in-depth overview over probability theory and especially a formal definition of kernels, please see [5, §36]. Intuitively, a (sub-)Markov kernel from X to Y is a function which, for all $x \in X$, returns a (sub-)probability measure on Y . In this thesis, we will see two concrete such kernels:

- The sub-Markov kernel *step* from \mathcal{PS} to \mathcal{PS} (a kind of extended \rightsquigarrow , totally defined over the set \mathcal{PS}), which we will use to define denotations of programs, and which intuitively will yield for each program state a sub-probability measure on program states
- The Markov kernel *uniform* from V_0 to V_0 , which we will use to model uniform random selection of numbers in a range $[0, \dots, n - 1]$, for some natural number n

Accordingly, the construction fV in the reduction rule FUN denotes a sub-probability measure

$$\mu : \Sigma_{V_0} \rightarrow \mathbb{R}^+,$$

so that $\text{fun}(f, V)|\sigma|\eta$ reduces to the distribution $g(\mu)$, which we now explain.

In general, for a measurable function $g : X \rightarrow Y$ and a measure μ on X , we denote by $g(\mu)$ the measure on Y defined by

$$g(\mu)(B) = \mu(\{x \in X : g(x) \in B\}),$$

for all $B \in \Sigma_Y$. In other words, $g(\mu)(B) = \mu(g^{-1}(B))$.

In our case, for g and μ as explained above, $g(\mu)$ is accordingly the sub-probability measure on \mathcal{PS} such that

$$g(\mu)(B) = \mu(\{x \in V_0 : g(x) \in B\}) = \mu(\{x \in V_0 : E[x]|\sigma|\eta \in B\}),$$

for all $B \in \Sigma_{\mathcal{PS}}$. A concrete example for the application of the rule FUN will be given in Section 3.5, where μ will be a certain uniform distribution, more precisely, the Markov kernel *uniform* whose precise description will be given in that section.

Now that the reduction relation \rightsquigarrow has been defined (which corresponds to a small-step semantics), we will extend it to obtain the notion of a denotation.

3.4.2 The Denotation

The denotation of a program state $P|\sigma|\eta$ will be, in some sense, a big-step semantics of our language. However, as opposed to a typical big-step semantics, it does not evaluate program states to values, but rather to measures on program states. We would like to

¹A function $h : X \rightarrow Y$ is called *measurable* if for all $B \in \Sigma_Y$, the inverse image $h^{-1}(B)$ is contained in Σ_X . Recall that $\Sigma_{\mathbb{R}^+}$ denotes the Borel-algebra.

think of the denotation as an n -fold application of the reduction relation to a state $P|\sigma|\eta$, until the program term P has been reduced to a value and thus cannot be reduced any more. However, while a single reduction step is well-defined, it is not yet clear how we should apply several reduction steps successively, since they do not map program states to program states, but program states to measures on program states. Moreover, the reduction relation is only partially defined on the set of program states, and for example, we do not yet have the notion that “a value evaluates to itself”. To solve these problems, we need to introduce some more notions of probability theory.

First of all, if $f : X \rightarrow \mathbb{R}$ is a measurable function, and μ is a measure on X , then by probability theory we have the notion of the (*measure*) *integral* of f with respect to the measure μ

$$\int f(x) d\mu(x),$$

sometimes, if we want to make clear the domain of integration X , also written as

$$\int_X f(x) d\mu(x).$$

Remark. If μ is of the special form

$$\mu(B) = \sum_{x \in A \cap B} c_x \quad (B \in \Sigma_X)$$

for a fixed finite subset A of X and numbers $c_x > 0$, then

$$\int_X f(x) d\mu(x) = \sum_{x \in A} f(x) c_x. \quad (3.1)$$

Special cases of such measures which will be important in the sequel are Dirac measures respectively uniform distributions with carrier (see Definition 3.23 below).

For a formal definition of the measure integral in the general case see [18]. Intuitively, if we think of μ as a probability measure, the integral $\int f(x) d\mu(x)$ represents the expected value of $f(x)$ when x is chosen according to the probability defined by μ .

Definition 3.16 (Kernel application). Let g be a kernel from X to Y , and let μ be a measure on X . Then we define the measure $g \cdot \mu$ on Y by the equation

$$(g \cdot \mu)(B) := \int g(x)(B) d\mu(x) \quad (B \in \Sigma_Y).$$

Definition 3.17 (Kernel composition). Let f be a kernel from X to Y , and g a kernel from Y to Z . The *composition* $g \circ f$ is the kernel from X to Z defined by

$$(g \circ f)(x) := g \cdot (f(x)) \quad (x \in X).$$

Remark. Note that according to Definition 3.16, we have, for the composition $g \circ f$, the formula

$$(g \circ f)(x)(C) = \int g(y)(C) df_x(y) \quad (x \in X, C \in \Sigma_Z),$$

where, for the sake of readability, we write f_x for the measure $f(x)$.

We are now able to define the sub-Markov kernel *step* which is an extension of the reduction relation to the complete set \mathcal{PS} (whereas the reduction relation of Definition 3.14 is only defined on a proper subset of \mathcal{PS}). This will enable us to perform the reduction relation successively. Namely, using kernel composition, we can then inductively define the sub-Markov kernels $step_n$ ($n \in \mathbb{N}$), which represent the reductions after n successive steps. We will see that this sequence actually stabilizes, and the denotation will then be defined as the supremum of $step_n$.

Definition 3.18 (Step). Let $P|\sigma|\eta$ be a program state. We define

$$step(P|\sigma|\eta) := \begin{cases} \mu & \text{if } P|\sigma|\eta \rightsquigarrow \mu \\ \delta_{P|\sigma|\eta} & \text{otherwise} \end{cases}$$

Furthermore, we define sub-Markov kernels $step_n$ from \mathcal{PS} to \mathcal{PS} inductively by

$$\begin{aligned} step_0(P|\sigma|\eta) &:= \delta_{P|\sigma|\eta} \\ step_{n+1}(P|\sigma|\eta) &:= step \circ step_n(P|\sigma|\eta) \end{aligned}$$

(where the right-hand side of the last equation is the composition of sub-Markov kernels according to Definition 3.17).

Definition 3.19 (Denotation). Let $P|\sigma|\eta$ be a program state. The denotation $\llbracket P|\sigma|\eta \rrbracket$ of $P|\sigma|\eta$ is defined as

$$\llbracket P|\sigma|\eta \rrbracket := \sup_n \bar{\mu}^n,$$

where $\bar{\mu}^n$ is the sub-probability measure on \mathcal{PS} defined by

$$\bar{\mu}^n(A) := \mu^n(A \cap \mathcal{VS}) \quad (A \in \Sigma_{\mathcal{PS}}).$$

Here we use $\mu^n = step_n(P|\sigma|\eta)$. Recall that \mathcal{VS} is the subset of \mathcal{PS} of all program states of the form $V|\sigma|\eta$, where V is a value. Furthermore, we define the denotation of a program P as $\llbracket P \rrbracket := \llbracket P|\square|\square \rrbracket$.

The denotation of a program state $P|\sigma|\eta$ models our intuitive notion of the distribution on \mathcal{VS} generated by the execution of the program P with store σ and event list η . In particular, we can use the denotation of a program to model the *probability of termination*.

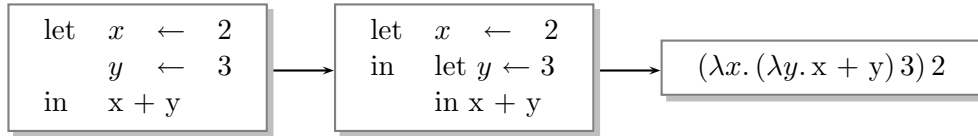
Definition 3.20 (Probability of termination). Let $P|\sigma|\eta$ be a program state. The probability of termination of $P|\sigma|\eta$ is defined as $T(P|\sigma|\eta) := \llbracket P|\sigma|\eta \rrbracket(\mathcal{PS})$.

Note that the definition of the probability of termination conforms to our intuition. Namely, since the denotation of a program is a sub-probability measure on \mathcal{PS} , applying it to the set of *all* program states yields the probability that the program evaluates to any program state (whose first element is a value) at all. If it is 1, it means the program always terminates, since it *always* evaluates to a value. By contrast, if it is 0, it means it never evaluates to any value, i.e. the program diverges.

3.5 Syntactic Sugar and Uniform Distributions

We now define some syntactic sugar. First of all, we explain the *let* construct. This will enable us to write applications of the form $(\lambda x.P)Q$, in a more easily readable way. We use *let* constructs only for terms with names, because nameless *let* constructs would not increase readability. For the named λ -calculus, any *let* construct can be unambiguously mapped to a named λ -term and vice versa. Furthermore, we can nest arbitrarily many *let* constructs one after the other. We explain the *let* construct by an example.

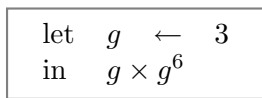
Example 3.21. The following program adds the natural numbers 2 and 3:



Note that we implicitly used additional syntactic sugar here. Namely, whenever we are dealing with a natural number n , we will just write n instead of the construct `value n`. Furthermore, we assume the existence of an infix operator $+$. This can, of course, be rewritten to the application of a program `plus` which performs addition of natural numbers, applied to x and y .

Similarly, we will also use group multiplication and exponentiation operators when dealing with (cyclic) groups. Assume we are working in some cyclic group G , whose order $|G|$ is known in the context. Then for two variables x and y that are bound to natural numbers, we will just write $x \times y$ to denote the application of a program `group_multiplication` that takes *three* arguments: the order of the group, as well as x and y . Here the first argument is implicit and will always be known in the context. For a generator g of G , and another natural number x , we will just write g^x , which means we apply a program `group_exponentiation` to $|G|$, g and x .

Example 3.22. Consider the unit group $(\mathbb{Z}/7\mathbb{Z})^*$. Its order equals $\varphi(7) = 6$, and 3 is a generator of the group. Assume the order of the group is known in the context. The following program yields the value 3.



Finally, we introduce uniform random selection as a variant of the standard let construct. What we want is to be able to write something like “let $x \leftarrow_R 7$ ” to select uniformly at random a value from the range $[0, \dots, 6]$. Clearly we will have to use the probabilistic construct `fun`. For this we introduce a Markov kernel *uniform* that, for any pure value of the form value n , yields an appropriate distribution over pure values. Here is the formal definition.

Definition 3.23 (Uniform distribution with carrier). Let A be a finite subset of some set X . The *uniform distribution on X with carrier A* is the probability measure $\mu_{X,A}$ on X such that

$$\mu_{X,A}(B) = \frac{|A \cap B|}{|A|}$$

for all $B \in \Sigma_X$.

Remark. We shall later need the following fact. If μ is the uniform distribution on X with carrier A and if $f : X \rightarrow \mathbb{R}$ is a measurable function, then by equation 3.1 we have

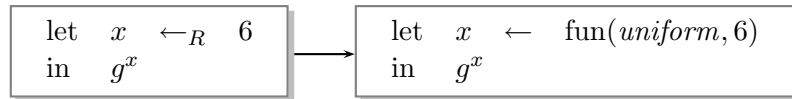
$$\int_X f(x) d\mu(x) = \frac{1}{|A|} \sum_{x \in A} f(x).$$

Definition 3.24 (Markov kernel uniform). The Markov kernel *uniform* : $V_o \rightarrow (\Sigma_{V_o} \rightarrow \mathbb{R}^+)$ is the Markov kernel from V_o to V_o such that *uniform*(l), for a natural number l , equals the uniform distribution on V_o with carrier $[0, \dots, l - 1]$, and such that *uniform*(l) = δ_l otherwise.

The definition of *uniform*(l) for pure values l which are not natural numbers given here is arbitrary, since we shall use *uniform*(l) only for natural numbers l . Thus any other definition of *uniform*(l) for non-natural l would also be sensible.

Using *uniform*, we can define “let $x \leftarrow_R n$ ”, for a natural number n , as syntactic sugar for “let $x \leftarrow \text{fun}(\text{uniform}, n)$ ”. We continue the previous example.

Example 3.25. Assume we know a generator g and the order of the group $(\mathbb{Z}/7\mathbb{Z})^*$ in the context. Then, for given σ and η , the following program yields a uniform distribution on \mathcal{PS} with carrier $[1, \dots, 6]|\sigma|\eta$ (see Notations).



Lastly, we will often use, for a natural number n , the symbol 1^n to denote a (unary) string of length n , i.e. we set

$$\begin{aligned} 1^0 &= \text{nil} \\ 1^{n+1} &= \text{value unit} :: 1^n \end{aligned}$$

3.6 Program Relations

Now that we know how to write programs in the language and how to interpret them, it remains to understand how we can model game transformations. We will see several possible transformations in Section 6. All these are based on one of three equivalence relations on programs: the denotational equivalence, the observational equivalence, or the computational indistinguishability. We will first formally define these equivalences. Then we will see a few properties of these relations and explain how they relate to each other.

3.6.1 Denotational Equivalence

In a sense, the denotational equivalence of two programs is the strongest equivalence relation that we consider (we will make this more precise in Section 3.6.4). It means that the denotations of two programs P and Q are exactly the same. That is, they evaluate to exactly the same distribution over program states.

Definition 3.26 (Denotational equivalence). We say that two programs states $P|\sigma|\eta$ and $Q|\sigma'|\eta'$ are *denotationally equivalent* and write $P|\sigma|\eta \sim Q|\sigma'|\eta'$, iff it holds that

$$\llbracket P|\sigma|\eta \rrbracket = \llbracket Q|\sigma'|\eta' \rrbracket.$$

We say P and Q are denotationally equivalent, and simply write $P \sim Q$, if it holds that $\llbracket P \rrbracket = \llbracket Q \rrbracket$.

This is a strong relation indeed. Consider for example the two programs $P := (\lambda x. x)$ and $Q := (\lambda x. (\lambda y. y) x)$. Both compute the identity function, but they are not denotationally equivalent. Indeed, we have that $\llbracket P \rrbracket = \delta_{P|\square|\square}$ and $\llbracket Q \rrbracket = \delta_{Q|\square|\square}$ and so $\delta_{P|\square|\square} \neq \delta_{Q|\square|\square}$, because $P \neq Q$. But, on the other hand, for all $v \in V$, it holds that $(Pv) \sim (Qv)$, because $\llbracket Pv \rrbracket = \delta_{v|\square|\square}$, and also $\llbracket Qv \rrbracket = \delta_{v|\square|\square}$.

A powerful tool for reasoning about denotational equivalence which has been shown at the chair of Michael Backes is the so-called *chaining of denotations*. Consider a program $E[P]$, where E is an evaluation context, and P is a program. According to the rules defined in Def. 3.14, first P will be reduced until it is a measure on values V (more precisely, a measure on program states restricted to values in the sense of the definition), then $E[V]$ will be evaluated. The chaining rule tells us that we can compute the denotations of P and $E[V]$ separately, and then use kernel application (see Def. 3.16) to compute the final denotation of $E[P]$.

Theorem 3.27 (Chaining denotations). *Let E be an evaluation context, P a program, σ a store and η an event list. Define*

$$g(v|\sigma'|\eta') := \llbracket E[v]|\sigma'|\eta' \rrbracket$$

for $v|\sigma'|\eta' \in \mathcal{VS}$. Note that $g : \mathcal{VS} \rightarrow (\Sigma_{\mathcal{PS}} \rightarrow \mathbb{R}^+)$ is a sub-Markov kernel. Then it holds that

$$\llbracket E[P]|\sigma|\eta \rrbracket = g \cdot \llbracket P|\sigma|\eta \rrbracket$$

Proof. Shown by Michael Backes, Matthias Berg and Dominique Unruh, see [2]. \square

3.6.2 Observational Equivalence

The observational equivalence of two programs P and Q is a slightly weaker notion than that of denotational equivalence. Intuitively, it states that P and Q are observationally equivalent if their behavior is the same in all contexts that bind all free variables of P and Q which are closed themselves and contain no locations outside of the store. Let's make this precise. We say that a program state $P|\sigma|\eta$ is *fully closed* iff P and σ contain no free variables, and P and σ contain no locations greater or equal than $|\sigma|$.

Definition 3.28 (Observational equivalence). We say that two programs P and Q are observationally equivalent, and write $P \equiv_{obs} Q$, if for all contexts C such that $C[P]|\square|\square$ and $C[Q]|\square|\square$ are fully closed, we have that $T(C[P]|\square|\square) = T(C[Q]|\square|\square)$.

To model the statement that “ P and Q behave the same in all contexts”, we just consider the probability of termination of P and Q under all contexts. If P and Q can be distinguished, then there is a context C such that $C[P]$ and $C[Q]$ have a different probability of termination. By contraposition, if there is no such context, then P and Q cannot be distinguished.

Remark. Because of the intuition that this relation means that P and Q have the same behavior in all contexts, observational equivalence is also often referred to as *contextual equivalence* in the theory of programming languages.

Often it is difficult to work with the definition of observational equivalence directly, because of the universal quantification over arbitrary contexts, which makes it hard to argue about the semantic behavior of the resulting programs. Fortunately, there is yet another powerful tool that helps us with this problem: the CIU Theorem. Intuitively, it tells us that it suffices to look at all *evaluation* contexts, but in exchange we must also consider all possible instantiations of the free variables of P and Q , as well as all possible stores and event lists, all of which could have been modelled by a general context.

Definition 3.29 (CIU Equivalence). Let P and Q be programs. We say that P and Q are *CIU-equivalent* if for all evaluation contexts E , all list of values a , all stores σ and all event lists η such that $E[P^a]|\sigma|\eta$ and $E[Q^a]|\sigma|\eta$ are fully closed it holds that

$$T(E[P^a]|\sigma|\eta) = T(E[Q^a]|\sigma|\eta).$$

Theorem 3.30 (CIU Theorem). *If P and Q are CIU-equivalent, then they are observationally equivalent.*

Proof. Shown by Backes et al. [2] □

3.6.3 Computational Indistinguishability

The last notion that we look at is that of *computational indistinguishability*, well-established in cryptography. Loosely speaking, it tells us that two objects are computationally indistinguishable if they cannot be distinguished by any *efficient* algorithm (where “efficient”

essentially means that its runtime is polynomially bounded). A bit more precisely, the notion of computational indistinguishability deals with two sequences of objects (in our case, programs), $(P_n)_{n \in \mathbb{N}}$ and $(Q_n)_{n \in \mathbb{N}}$. We say that these sequences are computationally indistinguishable if for sufficiently large n , the probability that any probabilistic polynomial-time decider (i.e., an algorithm that returns true or false) accepts P_n but not Q_n is negligible in n . A function f is said to be negligible if it falls faster than the inverse of any polynomial, that is, if for every positive polynomial $p(\cdot)$ and sufficiently large n , $f(n) < 1/p(n)$. A deep insight into this subject can be found in [15].

Definition 3.31 (Computational indistinguishability). Let \mathcal{S}_{true} be the set $\{true\}|\sigma|\eta$, i.e. the set of all program states whose first element is *true*. Define $\Pr[P] := \llbracket P \rrbracket(\mathcal{S}_{true})$ (that is, the probability that P evaluates to true). Two families $(P_n)_{n \in \mathbb{N}}$ and $(Q_n)_{n \in \mathbb{N}}$ are said to be computationally indistinguishable, iff for all polynomial-time programs D the function $f : \mathbb{N} \rightarrow [0, 1]$, defined by

$$f(n) := |\Pr[D(1^n, P_n)] - \Pr[D(1^n, Q_n)]|$$

is negligible.

3.6.4 Properties of these Relations

We are now interested in how exactly the above relations are related to each other. When we restrict our attention to closed programs P and Q , we will see that the denotational equivalence implies the observational equivalence, and the observational equivalence implies the computational indistinguishability. In this sense, denotational equivalence is the relation with the highest granularity, while computational indistinguishability is the coarsest relation on programs that we consider.

Theorem 3.32. *Let P and Q be programs. Assume that for all lists of values a , stores σ and event lists η such that $P^a|\sigma|\eta$ and $Q^a|\sigma|\eta$ are fully closed, $P^a|\sigma|\eta$ and $Q^a|\sigma|\eta$ are denotationally equivalent. Then P and Q are observationally equivalent.*

Proof. We obtain this result using the CIU Theorem and the chaining rule for denotations. For all evaluations contexts E , lists of values a , stores σ and event lists η we have

$$\begin{aligned} T(E[P^a]|\sigma|\eta) &= \llbracket E[P^a]|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(by Definition 3.20)} \\ &= (\Lambda(v, \sigma', \eta'). \llbracket E[v]|\sigma'|\eta' \rrbracket) \cdot \llbracket P^a|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(chaining rule)} \\ &= (\Lambda(v, \sigma', \eta'). \llbracket E[v]|\sigma'|\eta' \rrbracket) \cdot \llbracket Q^a|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(by assumption)} \\ &= \llbracket E[Q^a]|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(chaining rule)} \\ &= T(E[Q^a]|\sigma|\eta) && \text{(by Definition 3.20)} \end{aligned}$$

We conclude that $P \equiv_{CIU} Q$, which implies $P \equiv_{obs} Q$. □

Remark. Intuitively we often think of the denotational equivalence as the strongest relation because there are closed programs which are not denotationally equivalent, but still observationally equivalent. The programs $\lambda x. x$ and $\lambda x. (\lambda y. y) x$ seen in Section 3.6.1 constitute an example of such a case.

However, the requirement that P and Q be closed is really necessary here. For programs P and Q which contain free variables, the implication does not hold. Consider for example the programs $P := (\text{var } 0) \text{Value}_{\text{True}}$ and $Q := (\text{var } 0) \text{Value}_{\text{False}}$. P and Q are denotationally equivalent because their evaluation gets “stuck” so that they can never reduce to any value. Hence their denotation is the measure on \mathcal{PS} which always returns 0. But P and Q are not observationally equivalent. Consider for example the context $C := (\lambda. \square)(\lambda. \text{var } 0)$, such that $C[P]$ evaluates to $\text{Value}_{\text{True}}$ and $C[Q]$ evaluates to $\text{Value}_{\text{False}}$ (clearly we can extend this to a context which has a different termination behavior in either case).

Theorem 3.33. *Let $(P_n)_{n \in \mathbb{N}}$ and $(Q_n)_{n \in \mathbb{N}}$ be two families of programs. Assume that for all natural numbers n , P_n and Q_n are observationally equivalent. Then $(P_n)_{n \in \mathbb{N}}$ and $(Q_n)_{n \in \mathbb{N}}$ are computationally indistinguishable.*

Proof. Assume for contradiction that $(P_n)_{n \in \mathbb{N}}$ and $(Q_n)_{n \in \mathbb{N}}$ are not computationally indistinguishable. Then there exist a fixed polynomial-time decider D and a fixed natural number n such that

$$\Pr[D(1^n, P_n)] \neq \Pr[D(1^n, Q_n)] \quad (3.2)$$

according to Definition 3.31. Recall that D is a function which always outputs true or false. Now we can use it to define a context F which has a different termination behavior for P_n and Q_n , namely

$$F := \text{if } D(1^n, \square) \text{ then } \Omega \text{ else } 0$$

Here we use Ω for some diverging term. We can construct diverging terms e.g. by using recursion (to define recursion we can use the fold and unfold constructs of the language, or yet the store). We use an if-then-else construct here, which can also be translated into our language with the case, inl and inr constructs. Here we do not go further into these technical details.

Since by equation 3.2 we have $\llbracket D(1^n, P_n) \rrbracket(\mathcal{S}_{\text{true}}) \neq \llbracket D(1^n, Q_n) \rrbracket(\mathcal{S}_{\text{true}})$, we obtain that

$$\llbracket F[P_n] \rrbracket(\mathcal{PS}) \neq \llbracket F[Q_n] \rrbracket(\mathcal{PS}).$$

But this means that $P_n \not\equiv_{\text{obs}} Q_n$. This is a contradiction to our assumption. Hence we conclude that $(P_n)_{n \in \mathbb{N}} \approx_{\text{ind}_n} (Q_n)_{n \in \mathbb{N}}$, which proves the claim. \square

We will now look into a few additional properties of these relations. We will speak about reflexivity, symmetry and transitivity. In view of program transformations, the symmetry of an equivalence relation is useful, since when we know that P and Q are in a relation, we can transform P into Q and vice-versa. The transitivity is clearly useful when we want to apply several transformations in a row, since it tells us that if P can be transformed into Q and Q into R , then P can be transformed into R . Furthermore we introduce

another property of relations. Often it is the case that we are dealing with big programs, and we only want to transform some small subterm, especially since we want to keep the transformations as simple as possible. Hence for a relation between programs, it is useful to know that, when two programs P and Q are in a relation, then for all contexts C it holds that $C[P]$ and $C[Q]$ are also in the relation.

Definition 3.34 (Compatibility). Let $R \subseteq P \times P$ be a binary relation on program terms. We call R *compatible* if it holds that for all contexts C , we have $(C[s], C[t]) \in R$ whenever $(s, t) \in R$.

Theorem 3.35 (Properties of denotational equivalence). *The relation \sim is reflexive, symmetric and transitive. That is, it is an equivalence relation.*

Proof. Since denotational equivalence is based on the equality of distributions over program states, and we know that the $=$ relation is an equivalence relation, the claim is trivial. \square

Remark. Denotational equivalence is not compatible. Consider the following counter-example. Let $P := \text{Value}_{\text{Unit}}$ and $Q := (\lambda x.x)\text{Value}_{\text{Unit}}$. We have $P \sim Q$. Now consider the context $C := \lambda y.\square$. Since our reduction relation does not evaluate λ -abstractions, $C[P] \approx C[Q]$.

However, we can at least state that for all *evaluation* contexts E , if $P \sim Q$ then it also holds that $E[P] \sim E[Q]$. This follows easily from the chaining rule for denotations.

Theorem 3.36 (Properties of observational equivalence). *The relation \equiv_{obs} is reflexive, symmetric, transitive and compatible, i.e. it is a compatible equivalence relation.*

Proof. The three first properties again follow because the observational equivalence is based on the equality of real numbers (in the range $[0, 1]$).

For the compatibility, consider two programs P and Q such that $P \equiv_{\text{obs}} Q$. This means that for all contexts C such that $P[\square]\square$ and $Q[\square]\square$ are fully closed, we have that

$$T(C[P]\square\square) = T(C[Q]\square\square). \quad (3.3)$$

Our goal is to show that for all contexts C' , it holds that $C'[P] \equiv_{\text{obs}} C'[Q]$, which means that for all contexts C such that $C[C'[P]]\square\square$ and $C[C'[Q]]\square\square$ are fully closed, we have that

$$T(C[C'[P]]\square\square) = T(C[C'[Q]]\square\square)$$

We fix such contexts C and C' . We now set $C_0 := C[C']$. Note that $C_0[P] = C[C'[P]]$ and $C_0[Q] = C[C'[Q]]$. Furthermore, $C_0[P]\square\square$ and $C_0[Q]\square\square$ are fully closed since we required that C be such that $C[C'[P]]\square\square$ and $C[C'[Q]]\square\square$ are fully closed. Hence by equation 3.3, we obtain

$$T(C_0[P]\square\square) = T(C_0[Q]\square\square).$$

This proves the claim. \square

Theorem 3.37 (Properties of computational indistinguishability). *The relation \approx_{ind_n} is reflexive, symmetric and transitive, i.e. it is an equivalence relation.*

Proof. The claims for reflexivity and symmetry are trivial.

For the transitivity, consider sequences of programs $A = (A_n)_{n \in \mathbb{N}}$, $B = (B_n)_{n \in \mathbb{N}}$ and $C = (C_n)_{n \in \mathbb{N}}$, and assume that $A \approx_{ind_n} B$ and that $B \approx_{ind_n} C$. By this, we know respectively that for a fixed polynomial-time decider D , the functions

$$\begin{aligned} f_{AB}(n) &:= |P_D(A) - P_D(B)| \\ f_{BC}(n) &:= |P_D(B) - P_D(C)| \end{aligned}$$

are negligible. Here (and throughout the proof), for the sake of readability we use the notation $P_D(P) = \Pr[D(1^n, P_n)]$ according to Definition 3.31 for a decider D and a program P , and make the security parameter n implicit. Our goal is to show that the function $f_{AC} : \mathbb{N} \rightarrow [0, 1]$,

$$f_{AC}(n) := |P_D(A) - P_D(C)|$$

is negligible too.

First note that the function $f_{ABC} : \mathbb{N} \rightarrow [0, 1]$, defined by $f_{ABC}(n) := f_{AB}(n) + f_{BC}(n)$ is also negligible. In the following we use the triangle inequality, which states that for any two real numbers x and y , we have $|x + y| \leq |x| + |y|$. We compute

$$\begin{aligned} f_{AC}(n) &= |P_D(A) - P_D(C)| \\ &= |P_D(A) - P_D(B) + P_D(B) - P_D(C)| \\ &\leq |P_D(A) - P_D(B)| + |P_D(B) - P_D(C)| \\ &= f_{AB}(n) + f_{BC}(n) \\ &= f_{ABC}(n). \end{aligned}$$

We showed that $f_{AC}(n) \leq f_{ABC}(n)$. But since f_{ABC} is negligible, then so is f_{AC} . Hence we conclude that $A \approx_{ind_n} C$, which finishes the proof. \square

Remark. Computational indistinguishability is not compatible. This is because in the definition of compatibility, we quantify over *all* possible contexts, and in particular also contexts whose runtime is not polynomially bounded. Thus for a non-polynomial time context C , even if we have $P \approx_{ind_n} Q$ for some programs P and Q , it must not necessarily hold that $C[P]$ and $C[Q]$ are also computationally indistinguishable.

However it is not difficult to see that we can establish a similar property in the limit of polynomial-time contexts, which in practice should be sufficient in many cases. But since we will not need such a property in the remaining of this thesis, we do not go into further details here.

In practice, all of the knowledge obtained in this section will be very helpful when dealing with game-based transformations. We can successively transform one game into another and know by transitivity that the initial and the final game are in the same relation. Most often we will use observational equivalence, where we can even state that if only two subprograms are observationally equivalent then the whole two games are also observationally equivalent. If, somewhere along these transformations, we use a transformation of computational indistinguishability, we will only be able to state that the initial and final games

are computationally indistinguishable (since the relation \approx_{ind_n} is coarser than the relation \equiv_{obs}). The notion of denotational equivalence is too strong in general (and we could mostly only use it for the whole games, since it is not compatible), but will sometimes come in as a helpful lemma to conclude observational equivalence of two programs.

4 Cryptographic Basics

In this section, we introduce the basic cryptographic concepts that we want to formalize in Isabelle/HOL in Section 5. We will first explain a few general ideas about public-key encryption schemes. Then we discuss security notions, especially the *indistinguishability of encryptions* and therefore introduce the *Chosen-Plaintext Attack* in Section 4.2. We will proceed with an explanation of the Diffie-Hellman problem, on which the ElGamal encryption scheme is based, in Section 4.3. Finally we will define ElGamal in Section 4.4. Since these are very basic and well-established concepts, we will rather give a rough overview than a thorough discussion. For the definitions we mainly follow [1], but we concentrate on the uniform case, as this is what we will use in the sequel. Further information on these concepts can also be found in a standard textbook about the basics of cryptography, e.g. [9].

4.1 About Encryption Schemes

Encryption, the classical object of cryptography, is the art of transforming private or confidential information in such a way as to make it unreadable without the prior knowledge of some secret. Formally, we can model an encryption scheme with message space \mathcal{M} and ciphertext space \mathcal{C} as a triple of algorithms (K, E, D) that fulfill the following properties:

- K is a *randomized* key generation algorithm that takes a security parameter 1^n and returns a key pair $(e, d) \in \mathcal{K}$. Here e is the encryption key, and d is the decryption key. We denote by \mathcal{K} the key space of the encryption scheme.
- E is an encryption algorithm that takes a tuple $(1^n, e, m)$ of a security parameter 1^n , an encryption key e and a plaintext $m \in \mathcal{M}$, and returns a ciphertext $c \in \mathcal{C}$. This algorithm may be randomized or stateful.
- D is a decryption algorithm that takes a tuple $(1^n, d, c)$ of a security parameter 1^n , a decryption key d and a ciphertext $c \in \mathcal{C}$. It returns a plaintext $m \in \mathcal{M}$. This algorithm is usually deterministic. We further require that for all security parameters 1^n , key pairs $(e, d) \in \mathcal{K}$ and messages $m \in \mathcal{M}$, it holds that $D(1^n, d, E(1^n, e, m)) = m$.

In the case of *private-key* (or *symmetric*) encryption schemes we have that $e = d$, or at least that d is easily computable from e . By contrast, for *public-key* (or *asymmetric*) encryption schemes, which are our scope of interest here, it is infeasible (with acceptable effort) to compute d from e . In this case, we call e the *public key* and d the *private key*.

Asymmetric encryption schemes are useful because it is not necessary to share any a priori secret (like a commonly known secret key) for two parties to communicate in a possibly insecure network. People make their public key publicly known. Thus, everyone in the network can encrypt messages for an intended recipient using their public key, and only the recipient can decrypt this message, because only he owns the corresponding secret key. Note that this only works in the passive case, i.e. when an adversary can only eavesdrop on messages sent over the network. In the active case, i.e. when an adversary can even alter the messages, here we also have to deal with the issue of authenticity, but we do not discuss that here. Finally, one can quickly show that the encryption algorithm must necessarily be randomized, as otherwise it cannot even fulfill basic security properties.

4.2 Chosen-Plaintext Attack

As mentioned in the introduction, two well-established security properties are those of *semantic security*, meaning that no adversary can deduce even partial information about a plaintext by looking at a ciphertext, while the notion of *indistinguishability of ciphertexts* intuitively asserts that even if an adversary can choose two messages and see the encryption of one of them, he cannot decide which message was encrypted better than by pure guessing. Interestingly, these two properties are in fact equivalent (see e.g. [16]). This is nice, because to show that an encryption scheme fulfills the very natural definition of semantic security, one can show that it has indistinguishable encryptions, which is usually much easier.

Now, in cryptography, one usually models such security properties as *games*, that is, an interaction between an adversary and some “challenger”, and shows that the probability for the adversary to “win” the game is negligible. There are several kinds of attacks known in cryptography, yielding different security notions. We usually differentiate the *capabilities* of the adversary, i.e. what he is allowed to do to break an encryption scheme, and the *goal* that we aim at. Here our goal is the security property of *indistinguishable ciphertexts*, and we model the capabilities of the adversary with the well-known *Chosen-Plaintext Attack* (CPA). The game depicted in Figure 4.1, called the CPA game, formalizes this scenario.

Definition 4.1 (CPA Challenger). Let $\mathcal{PE} = (K, E, D)$ be an encryption scheme, let n be a security parameter and let $b \in \{0, 1\}$ be a bit. We define the CPA challenger as follows.

- First it creates a key pair $(e, d) \leftarrow K(1^n)$, and outputs $(1^n, e)$ to the adversary.
- Then, it receives two plaintexts $m_1, m_2 \in \mathcal{M}$ such that $|m_1| = |m_2|$ from the adversary, and it computes and outputs the tuple $(1^n, c)$ to the adversary, where c is the ciphertext $E(1^n, e, m_{b+1})$.

Hence there are two interactions with the adversary. At first, he obtains a tuple $(1^n, e)$, where 1^n is the security parameter in unary representation and the encryption key e , and so he may encrypt arbitrarily many messages of his choice, before sending two messages m_1 and m_2 of his choice to the challenger. Then, the challenger sends to the adversary the tuple $(1^n, E(1^n, e, m_{b+1}))$ of the security parameter (in unary representation) 1^n and the

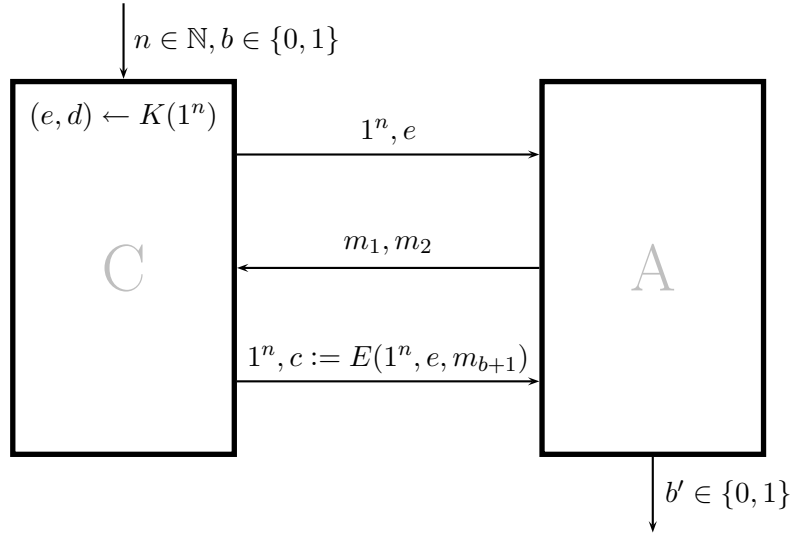


Figure 4.1: The CPA game

encryption of m_{b+1} . The adversary now has to guess b , that is, which message has been encrypted. Of course, he may again encrypt messages of his choice as he sees fit. This is why encryption should be randomized, otherwise his task would clearly be trivial. Finally the adversary outputs his guess b' .

Definition 4.2 (Indistinguishable encryptions under CPA). For a security parameter n , let CPA1_n be the CPA game where $b = 0$ and CPA2_n the game where $b = 1$. Furthermore let us denote by $\Pr[\text{CPA1}_A^{\mathcal{PE}} = b']$ and $\Pr[\text{CPA2}_A^{\mathcal{PE}} = b']$ the probability of the events that the respective games output bit b' , for an efficient uniform adversary A and an encryption scheme \mathcal{PE} . In this notation we omit the security parameter n for readability. Now, given a public-key encryption scheme $\mathcal{PE} = (K, E, D)$, we say that \mathcal{PE} has *indistinguishable encryptions under CPA* if for all efficient (i.e., probabilistic polynomial-time) adversaries A , the function

$$f(n) := |\Pr[\text{CPA1}_A^{\mathcal{PE}} = 1] - \Pr[\text{CPA2}_A^{\mathcal{PE}} = 1]|$$

is negligible (i.e. the probability that the adversary guesses correctly is at most negligibly better than $1/2$). This is equivalent to saying that for all efficient adversaries A , the families of games $(\text{CPA1}_n)_{n \in \mathbb{N}}$ and $(\text{CPA2}_n)_{n \in \mathbb{N}}$ are computationally indistinguishable.

Here we consider the uniform case, that is, every algorithm takes a security parameter 1^n . In the non-uniform case, we define an encryption scheme as a family of algorithms $(K_n, E_n, D_n)_{n \in \mathbb{N}}$. We can construct such an encryption scheme using a uniform encryption scheme by using the sequence $(K_n, E_n, D_n)_{n \in \mathbb{N}} := (K(1^n), E(1^n), D(1^n))_{n \in \mathbb{N}}$. Defining the notion of security over a family of algorithms is typical in cryptography, and we use it to speak about negligible functions or computational indistinguishability. This seems pretty natural anyway, as the algorithms are often already defined in function of some security parameter n (for example, n could be the length of the key). The security parameter n is

given in unary representation, as we measure the efficiency of the algorithms in the *length* of the input. See [15] for a deeper insight.

4.3 The Diffie-Hellman Problem

The encryption scheme that we are going to look at, ElGamal, is based on the Diffie-Hellman problem, which is derived from the difficult problem of computing discrete logarithms in certain cyclic groups. Computing such logarithms is a very old and well-known mathematical problem from number theory.

The difficulty of breaking the security of ElGamal can be reduced to the difficulty of solving the Diffie-Hellman problem. It is a good idea to base a cryptographic construction on the difficulty of a well-known mathematical problem, as many mathematicians around the world have worked and failed on this problem. As long as it is unsolved, the construction stays secure. If it is solved (and the security of the construction breaks down), the solution becomes quickly known and people can act accordingly. By contrast, if a construction is based on some very special and not so well-known problem, it could happen that only a small group of people (e.g. a secret service) finds the solution, and people may not even learn that they are using an encryption scheme that has become insecure.

The problem of computing discrete logarithms is conjectured to be hard in many cyclic groups. Often, it is used in unit groups $(\mathbb{Z}/p\mathbb{Z})^*$ for a prime number p (but it is also conjectured to be hard in many other cyclic groups). Let g be a generator of such a group mod p . Then for each number $h \in \{1, 2, \dots, p-1\}$ there exists an exponent $x \in \{0, 1, 2, \dots, p-2\}$ such that

$$h \equiv g^x \pmod{p}$$

We call x the *discrete logarithm* to the base g of h . As of today, no efficient algorithms are known to solve this problem.

A related problem is the *Computational Diffie-Hellman problem* (CDH). In a cyclic group G with a generator g (where computing discrete logarithms is conjectured to be hard), the problem is the following. Given only g , g^x and g^y , for elements x and y distributed uniformly at random in $\{1, \dots, |G|\}$, compute the value g^{xy} . Clearly, if one could compute discrete logarithms in G , the task would be easy. Hence, the CDH problem is at most as difficult as computing discrete logarithms. The converse direction is an open problem in the general case.

Finally, the *Decisional Diffie-Hellman problem* (DDH) is a slightly weaker form of this problem. Here, the task is as follows. Given g , g^x , g^y and g^z , decide whether $g^z = g^{xy}$. Here x and y are again elements distributed uniformly at random in $\{0, \dots, |G|-1\}$, whilst with probability $1/2$ we have $z = xy$ and with probability $1/2$, z is also an element distributed uniformly at random in $\{0, \dots, |G|-1\}$. Clearly, a solution to the CDH problem in general implies a solution to the DDH problem. While the Decisional Diffie-Hellman problem can be efficiently solved in unit groups $(\mathbb{Z}/p\mathbb{Z})^*$, it is still conjectured to be hard in many cyclic groups in general. Because this assumption will play an important role later on, similar as for the chosen-plaintext attack we formally define a game to model it.

Definition 4.3 (DDH Challenger). Let $(G_n)_{n \in \mathbb{N}}$ be a sequence of cyclic groups with respective generators g_n , and let $b \in \{0, 1\}$ be a bit. Let n be a security parameter. First, the DDH Challenger selects uniformly at random elements $x, y, z \in [0, \dots, |G_n| - 1]$. If $b = 0$, it outputs to the adversary the *DDH tuple* $(1^n, g_n^x, g_n^y, g_n^{xy})$, otherwise it outputs the tuple $(1^n, g_n^x, g_n^y, g_n^z)$.

The task of an adversary interacting with the DDH challenger is to guess b , as above. But the DDH assumption tells us that there exists a sequence of cyclic groups such that this is not feasible for efficient adversaries.

Assumption 4.4 (Hardness of the DDH problem for $(G_n)_{n \in \mathbb{N}}$). *We assume that the sequence of cyclic groups $(G_n)_{n \in \mathbb{N}}$ fulfills the following property. Similarly as in the CPA game, let $DDHxy_n$ be the DDH game where $b = 0$, and $DDHz_n$ the DDH game where $b = 1$. For an adversary A interacting with the DDH challenger, let $\Pr[DDHxy_A = b']$ and $\Pr[DDHz_A = b']$ be the probability that the adversary outputs b' in the respective games. Then the function*

$$f(n) := |\Pr[DDHxy_A = 1] - \Pr[DDHz_A = 1]|$$

is negligible for all efficient uniform adversaries A . That is, we assume the two games $DDHxy$ and $DDHz$ to be computationally indistinguishable.

4.4 The ElGamal Encryption Scheme

The ElGamal encryption scheme, which gets its name from its inventor, Taher Elgamal (though it really has a meaning: the name *Elgamal* is derived from the Arabic *al djamal*, which literally means “of beauty”), is a well-known encryption scheme based on the hardness of the DDH problem, and under this assumption is known to have indistinguishable encryptions under CPA.

We work with a sequence of cyclic groups $(G_n)_{n \in \mathbb{N}}$ where the DDH problem is hard, as in Assumption 4.4. Furthermore assume that there is a corresponding sequence of canonical generators $(g_n)_{n \in \mathbb{N}}$ such that for all $n \in \mathbb{N}$, we have $\langle g_n \rangle = G_n$ (i.e. there is a pointwise relation between the two sequences). Let $n \in \mathbb{N}$ be a security parameter. We describe below the ElGamal encryption scheme $\mathcal{PE}^{\text{ElGamal}} = (K, E, D)$ in the scenario where Bob wants to send a message to Alice.

Key generation. Alice uses the key generation algorithm $K(1^n)$, which selects a value x uniformly at random from $\{0, \dots, |G_n| - 1\}$ and computes $h := g_n^x$. It sets $e := h$ and $d := x$ and outputs the key pair (e, d) . Alice can now send her public key to Bob.

Encryption. The message space \mathcal{M}_n is the set G_n . To encrypt a message $m \in \mathcal{M}_n$, Bob uses the encryption algorithm $E(1^n, e, m)$, which chooses an element y uniformly at random from $\{0, \dots, |G_n| - 1\}$. It computes $i := g_n^y$, $c' := m \times h^y$ and outputs the ciphertext $c := (i, c')$. Hence the ciphertext space \mathcal{C}_n is the set G_n^2 .

Decryption. When Alice receives the cipher c from Bob, she can recover the plaintext m as follows, knowing her secret key d . She uses the decryption algorithm $D(1^n, d, c)$, which computes and outputs $m := c' \times (i^x)^{-1}$.

The correctness of the scheme can be easily verified, as we have

$$\begin{aligned} D(1^n, d, c) &= D(1^n, (g_n, x), (g_n^y, m \times h^y)) \\ &= (m \times h^y) \times ((g_n^y)^x)^{-1} \\ &= m \times g_n^{xy} \times g_n^{-yx} \\ &= m \end{aligned}$$

For a total break of this scheme (that is, for recovering the secret exponent x from the public key g_n^x), one clearly needs to be able to compute discrete logarithms in G_n . To recover the plaintext from an encryption, one needs to be able to solve the Computational Diffie-Hellman problem. But as we will see, to ensure the indistinguishability of encryptions, we even have to assume the hardness of the Decisional Diffie-Hellman problem.

5 Formalization in Isabelle/HOL

In Section 3, we saw how to formally write programs in the cryptographic language in Isabelle/HOL, and learned about program relations. Then in Section 4, we recalled a few basic constructions from cryptography. In this section, we are going to plug the concepts introduced in these two chapters together. We are going to formally define the CPA game and the ElGamal encryption scheme in the language in Isabelle/HOL. Together, they will serve as a starting point of our proof.

5.1 The CPA Game

Following Definition 4.1, to model the CPA challenger, we model two CPA games: one in which we encrypt the message m_1 and another in which we encrypt the message m_2 . Apart from this, these two games are perfectly identical. We parameterize these games with an encryption scheme (since the challenger uses a key generation algorithm and an encryption algorithm), an adversary and the security parameter. As explained in Definition 4.2, given an encryption scheme, the goal is to show that the first CPA game and the second CPA game are computationally indistinguishable in the security parameter. This entails that the encryption scheme is CPA-secure.

We define the first CPA game IND_CPA1 as follows.

```
IND_CPA1 ES A A' n ==  
let (e, d) ← :encGen ES: 1n  
    (m1, m2, a) ← :A: (1n, e)  
    c ← :encEnc ES: (1n, e, m1)  
in :A': (1n, m1, m2, a, e, c)
```

Figure 5.1: Definition of IND_CPA1

The game takes as arguments an encryption scheme ES (which defines a key generation algorithm $encGen$, an encryption algorithm $encEnc$ and a decryption algorithm $encDec$), an adversary defined by the functions A and A' , and a security parameter n .

First it uses the key generation algorithm $encGen$ as defined by the encryption scheme. This algorithm should return a pair (e, d) where e represents the (public) encryption key, and d the (secret) decryption key.

Then it calls the function A (the first interaction with the adversary) and gives it the security parameter 1^n (in unary representation) as well as the public key e . The adversary

returns two messages m_1 and m_2 , where we require that $|m_1| = |m_2|$, and a string a , in which it can store additional information. We will say more about how we precisely model the adversary in Section 5.4.

Now, the game encrypts the first message m_1 using the encryption function defined by the encryption scheme ES , passing it the security parameter 1^n and the obvious arguments e and m_1 needed for encryption. The encryption algorithm returns a ciphertext c . For instance, using the ElGamal encryption scheme, we will obtain a ciphertext $c = (i, c')$, as described in Section 4.4.

Finally, it calls the function A' , which corresponds to the second interaction with the adversary. It again parameterizes it with the security parameter 1^n , and also passes the two original messages m_1 and m_2 , the string a (into which A might have written additional information), the public key e , and the ciphertext c . The output of the game is the output of the function A' .

We define the second CPA game IND_CPA2 analogously. As the name suggests, the only difference is that here the message m_2 is encrypted in place of the message m_1 .

| |
|--|
| <pre> IND_CPA2 ES A A' n == let (e, d) ← :encGen ES: 1ⁿ (m₁, m₂, a) ← :A: (1ⁿ, e) c ← :encEnc ES: (1ⁿ, e, m₂) in :A': (1ⁿ, m₁, m₂, a, e, c) </pre> |
|--|

Figure 5.2: Definition of IND_CPA2

The goal of the adversary is to determine which message was encrypted. Clearly, the adversary can only use the information it is given to determine which of the two functions IND_CPA1 and IND_CPA2 called it, in other words, it can only use the arguments with which we call it. But these are almost identical in the two games, except for the ciphertext c of the encrypted message m_1 or m_2 , which is the only sensitive information the adversary is given. Recall that the output of the game is the output of the function A' . Hence, if for some encryption scheme ES and for all proper adversaries (A, A') we can show that the programs IND_CPA1 ES A A' n and IND_CPA2 ES A A' n are computationally indistinguishable in n , this means the ciphertexts cannot be distinguished with non-negligible probability. That is, the encryption scheme ES has indistinguishable encryptions under chosen-plaintext attack.

5.2 The Security Parameter

Throughout the whole proof, in all the games we are working with a fixed family of cyclic groups $(G_n)_{n \in \mathbb{N}}$, in which we assume the DDH problem to be hard, as defined in Section 4.3.

We use the security parameter n as the index of the cyclic group G_n which we are working with. Intuitively, we may think, for instance, that the greater the security parameter, the

greater the order of G_n . Note that this does not mean that the order of the group is equal to the security parameter, it just depends on it.

We will often need a fixed generator of our group G_n . For this purpose, we will use a function that, given a security parameter (in unary representation) 1^n , returns a fixed generator of the cyclic group G_n . More formally, we define a deterministic function g , which takes a bitstring of length n , as defined in Section 3.5. It returns an element of the set G_n such that $\langle g(1^n) \rangle = G_n$, for all $n \in \mathbb{N}$. Here we assume that the elements of any cyclic group G_n can be represented as bitstrings in our language. We consider g as a global constant, i.e. it can be directly accessed in all the games. For the sake of readability, we also use the notation

$$g_n := g(1^n).$$

In this manner, we can use g_n to denote a fixed generator of the group G_n .

5.3 ElGamal

As introduced in Section 4.1, an encryption scheme ES consists of a tuple (K, E, D) where K is the key generation algorithm, E the encryption algorithm, and D the decryption algorithm. In Isabelle/HOL, we define an encryption scheme analogously as a record of functions $encGen$ (the key generation algorithm), $encEnc$ (the encryption algorithm), and $encDec$ (the decryption algorithm).

Hence the definition of our ElGamal encryption scheme in Isabelle/HOL is as follows.

```
ElGamal ==
{  encGen  =  ElGamalGen
   encEnc  =  ElGamalEnc
   encDec  =  ElGamalDec }

```

Figure 5.3: The ElGamal encryption scheme

With our description of the ElGamal encryption scheme from Section 4.4, the corresponding formal games for key generation, encryption and decryption are fairly straightforward. In the remaining of this section, we will precisely define these programs. First we consider the key generation algorithm.

```
ElGamalGen ==
λ unaryn. let x ←R |⟨g(unaryn)⟩|
           in (g(unaryn)x, x)

```

Figure 5.4: The ElGamal key generation algorithm

The key generation algorithm expects a security parameter 1^n , which we call `unaryn` here. It selects a random element x in the range $[0, \dots, p-1]$ where $p = |\langle g_n \rangle|$ is the order

of G_n and returns a tuple consisting of the element g_n^x (consequently distributed uniformly at random in the set G_n) and x itself. They correspond to the public and the secret key used in the algorithms for encryption and decryption.

Now, we define the encryption algorithm as follows.

```

ElGamalEnc ==
λ (unaryn, gx, m). let y ←R |⟨g(unaryn)⟩|
                  in  (g(unaryn)y, m × gxy)

```

Figure 5.5: The ElGamal encryption algorithm

It takes a security parameter 1^n , a public key gx and a message m . Similarly as in the key generation algorithm, it selects a random element y from the range $[0, \dots, p-1]$, where $p = |G_n|$, and returns as a ciphertext the tuple consisting of the elements g_n^y and $m \times gx^y$, as explained in Section 4.4.

Although we will in fact not need the decryption algorithm in the remaining of this thesis, we define it here for the purpose of clarity and completeness.

```

ElGamalDec ==
λ (unaryn, x, i, c'). c' × i-x

```

Figure 5.6: The ElGamal decryption algorithm

Note that the parameter unaryn, which corresponds to the security parameter 1^n , is implicitly used in the multiplication and exponentiation operations, since we are always working with the cyclic group G_n (see Section 3.5 on syntactic sugar). We simply multiply the encrypted message with the inverse of the element that we used for encryption, yielding the original message.

5.4 The Adversary

Intuitively, it is clear we have to parameterize a CPA game with a CPA adversary. Technically, we represent the adversary as a pair of functions (A, A') , since there are two calls of the adversary in the CPA game. In the first interaction, we ask from the adversary two messages m_1 and m_2 . In the second interaction, after having encrypted one of these two messages, we send the encryption of one of the two messages to the adversary. The two functions A and A' are the functions which correspond to these two interactions.

We will now formalize more precisely the behavior of A and A' . First, two important assumptions about both A and A' should be made explicit:

1. Their runtime is *polynomially bounded*, since we are only interested in efficient adversaries. Technically, we say they are probabilistic polynomial-time adversaries.

2. They are *fully closed*; this is important, since we do not want our adversaries to be able to read variables from the outer context, otherwise it would be trivial for an adversary to determine which message has been encrypted. Furthermore they are not allowed to access the eventlist, nor are they allowed to read or write the store at indexes other than those which they allocated themselves (we might even forbid that they access the store or the eventlist altogether, but then we would have to show that this does not lessen their power).

Now we take a closer look at the two functions.

- A takes as arguments
 - 1^n , the security parameter in unary representation and
 - e , the public key used for the encryption.

A returns two messages m_1, m_2 , and a string a . Here we assume that our adversary is a valid CPA adversary for ElGamal. In particular, we assume that $m_1, m_2 \in G_n$. In this case, we do not even need to require that $|m_1| = |m_2|$. The string a is used for the communication between A and A' . In the context, we later only use a as an additional argument for A' , so A can encode any information it wants and pass it to A' via the string a . This is necessary because we want to consider a single (stateful) adversary, but model it with two functions that are both closed (and so cannot read variables from each other). By allowing the function A to generate an arbitrary string a which we pass unaltered to the function A' , the adversary has the possibility to “remember” any computations it may have performed in its first call, when we call it a second time.

- A' takes as arguments
 - 1^n , the security parameter in unary representation,
 - m_1 , the first plaintext chosen by A ,
 - m_2 , the second plaintext chosen by A ,
 - a , used to store additional information computed by A ,
 - e , the public key used for encryption and
 - c , the ciphertext of either m_1 or m_2 ; the task of A' is to determine which one it is.

Technically, note that it is not strictly necessary to pass m_1, m_2 or e to A' since A could encode all of this information and write it into the string a , thus implicitly passing it to A' . However, we only use a for “internal” computations performed by A , and not for obvious information that A' will need. Note that this is simply a design decision. A' tries to guess which of the two messages m_1 or m_2 has been encrypted and outputs a single bit b , meaning that its guess is that the ciphertext c is an encryption of the plaintext m_{b+1} .

6 Program Transformations

In this chapter, we are going to look into several kinds of program transformations, all of which will be needed in the proof in Chapter 7. Recall from Section 3.6 that all transformations are based on one of three equivalence relations. The relation we will use most often is the observational equivalence. Furthermore, we will see one transformation based on the computational indistinguishability and finally we will see a rather involved transformation based on the denotational equivalence.

6.1 Transformations based on Observational Equivalence

6.1.1 β -equivalence

The first program transformation that we are going to look at is an easy one. We will show that two β -equivalent programs are also observationally equivalent. Let us first get an intuitive notion of what we want to achieve and then properly formalize the property that we will prove.

Example 6.1. The two programs shown in Figure 6.1 compute the same value. Intuitively, the left-hand program deterministically reduces to the right-hand program, i.e. the right-hand program corresponds to the left-hand program one step further in its execution. Of course we expect such two programs to be observationally equivalent.



Figure 6.1: A β -reduction step

Note that the following more general observation is rather clear. Loosely speaking, when we consider a program P that deterministically reduces to a program Q , then we have that P and Q are denotationally equivalent. In fact, this theorem is a typical theorem in the design of functional programming languages and is usually called *Coincidence* (of the big-step semantics and the small-step semantics). Here, because our language is probabilistic, the notion is a little bit different, but we can still state the following

Theorem 6.2 (Coincidence). *Let $P|\sigma|\eta$ and $Q|\sigma'|\eta'$ be program states such that $P|\sigma|\eta \rightsquigarrow \delta_{Q|\sigma'|\eta'}$. Then $P|\sigma|\eta \sim Q|\sigma'|\eta'$.*

Proof. The proof is straight-forward by computing the denotations of $P|\sigma|\eta$ and $Q|\sigma'|\eta'$. First, by the definition of $step_0$, we know that

$$step_0(Q|\sigma'|\eta') = \delta_{Q|\sigma'|\eta'}$$

Now we compute one step of the program state $P|\sigma|\eta$.

$$\begin{aligned} step_1(P|\sigma|\eta) &= step \circ step_0(P|\sigma|\eta) && \text{(by definition of } step_{n+1}\text{)} \\ &= step \cdot (step_0(P|\sigma|\eta)) && \text{(by definition of } \circ\text{)} \\ &= step \cdot \delta_{P|\sigma|\eta} && \text{(by definition of } step_0\text{)} \\ &= \int_{\mathcal{PS}} step(x)(A) d\delta_{P|\sigma|\eta} && \text{(by definition of } \cdot\text{)} \\ &= step(P|\sigma|\eta) && \text{(by equation 3.1)} \\ &= \delta_{Q|\sigma'|\eta'} && \text{(since } P|\sigma|\eta \rightsquigarrow \delta_{Q|\sigma'|\eta'}\text{)} \end{aligned}$$

We have shown that $step_0(Q|\sigma'|\eta') = step_1(P|\sigma|\eta)$. Recall that by Definition 3.18 we have $step_{n+1}(A) = step \circ step_n(A)$, for $A \in \mathcal{PS}$. Thus we easily get by induction on n that $step_n(Q|\sigma'|\eta') = step_{n+1}(P|\sigma|\eta)$. By Definition 3.19 of the denotation of program states, we obtain $\llbracket P|\sigma|\eta \rrbracket = \llbracket Q|\sigma'|\eta' \rrbracket$. \square

However, here the observational equivalence does not follow trivially. More precisely, the observational equivalence of two programs P and Q , where P reduces to Q , such as two β -equivalent programs, follows only under the assumption that both $P|\sigma|\eta$ and $Q|\sigma|\eta$ are fully closed for all stores and eventlists (see Section 3.6). However, we would like to state at least that two β -equivalent programs are observationally equivalent even if they do not fulfill these restrictions. Fortunately, the following lemma has already been shown and comes in very helpful.

Lemma 6.3. *For all programs P , values V , lists of values a and stores σ and event lists η the following equation holds true:*

$$\llbracket ((\lambda.P)V)^a|\sigma|\eta \rrbracket = \llbracket (P\{V\})^a|\sigma|\eta \rrbracket$$

Proof. Proven by Matthias Berg in Isabelle/HOL. \square

We shall need later the following fact.

Corollary 6.4. *By the chaining rule for denotations (see Theorem 3.27), we obtain furthermore that for all evaluation contexts E , it holds*

$$\llbracket E[((\lambda.P)V)^a]|\sigma|\eta \rrbracket = \llbracket E[(P\{V\})^a]|\sigma|\eta \rrbracket$$

Note that the simple equality $\llbracket (\lambda.P)V|\sigma|\eta \rrbracket = \llbracket P\{V\}|\sigma|\eta \rrbracket$ already follows from Theorem 6.2, as we have $(\lambda.P)V|\sigma|\eta \rightsquigarrow \delta_{P\{V\}|\sigma|\eta}$ by Definition 3.14. But in Lemma 6.3, we are additionally allowed to instantiate the two programs with arbitrary lists of values a . This makes it very easy to prove the observational equivalence of two β -equivalent programs.

Theorem 6.5 (β -equivalence). *Let P be a program, and let V be a value. Then we have $(\lambda.P)V \equiv_{obs} P\{V\}$.*

Proof. We show that $(\lambda.P)V \equiv_{CIU} P\{V\}$. For all evaluation contexts E , lists of values a and stores σ and event lists η such that $E[(\lambda.P)V]^a|\sigma|\eta$ and $E[(P\{V\})^a|\sigma|\eta$ are fully closed, we have

$$\begin{aligned}
 & T(E[(\lambda.P)V]^a|\sigma|\eta) \\
 &= \llbracket E[(\lambda.P)V]^a|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(by Definition 3.20)} \\
 &= (\Lambda(v, \sigma', \eta'). \llbracket E[v]|\sigma'|\eta' \rrbracket) \cdot \llbracket ((\lambda.P)V)^a|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(chaining rule)} \\
 &= (\Lambda(v, \sigma', \eta'). \llbracket E[v]|\sigma'|\eta' \rrbracket) \cdot \llbracket (P\{V\})^a|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(by Lemma 6.3)} \\
 &= \llbracket E[(P\{V\})^a|\sigma|\eta] \rrbracket(\mathcal{PS}) && \text{(chaining rule)} \\
 &= T(E[(P\{V\})^a|\sigma|\eta]) && \text{(by Definition 3.20)}
 \end{aligned}$$

Thus we obtain that $(\lambda.P)V \equiv_{CIU} P\{V\}$. It follows that $(\lambda.P)V \equiv_{obs} P\{V\}$, which proves the Theorem. \square

6.1.2 Expression Propagation

The next very useful equivalence that we look at is that of *expression propagation*. It is described in depth and formally proven in [12]. Here we only give the essential ideas. Intuitively, expression propagation models the following transformation. When we consider an application $(\lambda x.P)Q$ where P and Q are programs, then the program $[x \mapsto Q]P$ resulting from replacing every occurrence of x in P by Q should be observationally equivalent to the program $(\lambda x.P)Q$, given some restrictions on Q . Indeed, such a transformation corresponds to a reduction step in a call-by-name setup, and is in fact a generalization of the β -equivalence discussed in Section 6.1.1.

Example 6.6. In Figure 6.2, the first program only performs addition of a few natural numbers. We may want to transform it into the last program. This can be achieved by a double application of expression propagation. First we replace all occurrences of x by its assignment, then we do the same for y . Note that the assignments themselves disappear respectively, as would be the case for β -reduction in a call-by-name setup.

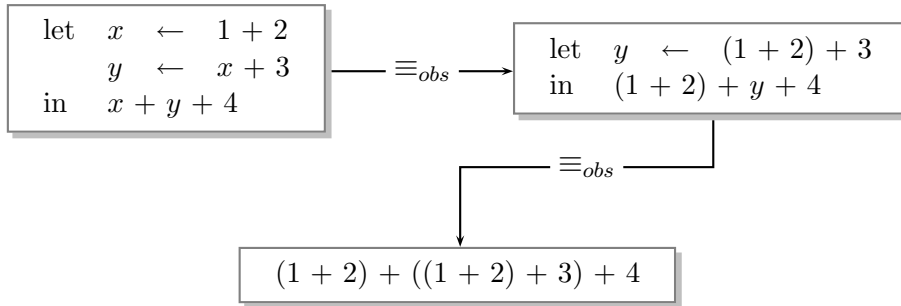


Figure 6.2: Two consecutive applications of expression propagation

However, when we model this equivalence, namely the observational equivalence of $(\lambda.P)Q$ and $P\{Q\}$ as explained above, we have to be careful. In fact, it turns out that this equivalence does not hold unconditionally. The following problems arise, and we have to formulate the theorem such as to exclude these cases. Recall that we call the *substitution candidates* those variable which are replaced by the substitution operator (see Section 3.3.3), e.g. x in the named version of the transformation described at the beginning of this section.

1. The program Q may be randomized. Then, if there are more than one substitution candidates in P , it might yield different outcomes at each place where it was substituted into the program $P\{Q\}$. Clearly, this is not the case in the program $(\lambda.P)Q$, where Q is executed only once.
 - We require that on every possible input, either Q deterministically reduces to a value q , or it does not terminate at all. Allowing Q to not terminate at all on some inputs makes this Theorem more realistic in practice, since otherwise we would have to ensure that Q terminates on every possible input, even on inputs outside of its domain.
2. Q could read or write the store or the event list. In either case, it clearly makes a difference whether we execute Q only once at the beginning or possibly several times later on, especially since we explicitly want to give P the ability to use the store and the event list. However, we need to require that Q cannot do so.
 - We require that the value q to which Q deterministically evaluates (if it terminates) is independent of the store and the event list, and does not alter them.
3. Finally, it might be that Q diverges, but P does not use the variable to which Q is assigned. Then, $(\lambda.P)Q$ diverges too, but $P\{Q\}$ does not (provided that P does not already diverge itself).
 - We require that whenever Q does not terminate, $P\{Q\}$ does not terminate either.

Theorem 6.7 (Expression propagation). *Let P and Q be programs. Assume that both of the following two conditions hold.*

1. For all lists of values a such that Q^a is closed, either
 - a) there exists a unique value q such that for all stores σ and event lists η such that $Q^a|\sigma|\eta$ is fully closed, $\llbracket Q^a|\sigma|\eta \rrbracket = \llbracket q|\sigma|\eta \rrbracket$, or
 - b) $\llbracket Q^a|\sigma|\eta \rrbracket = \mathbf{o}$, where we use \mathbf{o} to denote the measure on \mathcal{PS} defined by $\mathbf{o}(A) = 0$, for all $A \in \Sigma_{\mathcal{PS}}$.
2. For all lists of values a , stores σ and event lists η , we have $\llbracket (P\{Q\})^a|\sigma|\eta \rrbracket = \mathbf{o}$ whenever $\llbracket Q^a|\sigma|\eta \rrbracket = \mathbf{o}$.

Then the equation $(\lambda.P)Q \equiv_{\text{obs}} P\{Q\}$ holds true.

Proof. For the proof, we refer to the Bachelor thesis of Jonathan Driedger [12]. □

6.1.3 Expression Subsumption

The notion of *expression subsumption* denotes the transformation which corresponds to the exact opposite of expression propagation. Intuitively, in a program P , if some expression Q occurs zero or more times, we may want to simply assign the program Q to some fresh variable x at the beginning of P , and then replace every occurrence of Q in P with x . Consider the following

Example 6.8. The following game sends an image of some function f to two adversaries A_1 and A_2 . The adversaries try to guess something (e.g. the preimage) and their output is bound to the variables x and y respectively. Finally the game checks if both adversaries computed the same guess. It seems unnecessary to compute the function $f(123)$ twice here, especially if we assume for instance that the computation is expensive. We may want to transform it as in the second game, where the computation takes place only once.

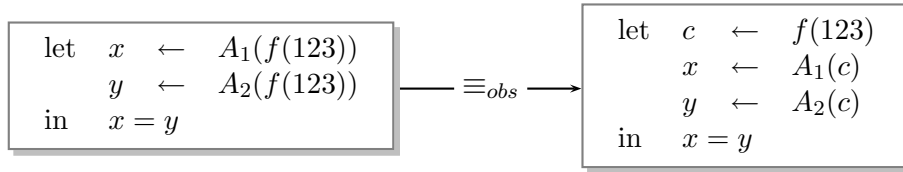


Figure 6.3: An application of expression subsumption

At first glance this transformation seems somewhat surprising as we have to make up a fresh name for a variable. However, recall from Section 3.1 that we are actually dealing with nameless terms and that we only write names here for readability. Hence this problem does not arise, as on a low level, our representation of variables is canonical.

The validity of this transformation follows immediately from Theorem 6.7 under the restrictions given there, as the observational equivalence is a symmetric (equivalence) relation. Often there are several possible ways to apply expression subsumption. In the above program, we could also just have subsumed the first computation of $f(123)$ in the call of A_1 , or just the second in the call of A_2 . We could choose, as all of these transformations are valid. This is because here, the obtained program can always be represented in the form $P\{Q\}$, as required in Theorem 6.7. Indeed, it could be that P already contains the expression Q zero or more times itself, before we substitute Q for the substitution candidates in P .

6.1.4 Nesting Chained Applications

We are now going to look at yet another very useful transformation. To explain the goal of this section, we first explain the notions of *nested lets* and of *chained lets*.

As already mentioned in Section 3.5, we can use several let constructs one after the other in the following manner.



Figure 6.4: Nested lets

The two above programs are identical, as the left program is just syntactic sugar for the right program. We refer to this construction of one let construct within another as *nested lets*, or *nested applications*. In the low-level syntax, we would write the above program as

$$(\lambda x. (\lambda y. y + 3) (x + 2)) 1$$

Here we can observe that the inner application $(\lambda y. y + 3) (x + 2)$ is indeed nested within the outer application, i.e. the whole term.

We can also chain such let constructs. Consider the following program.

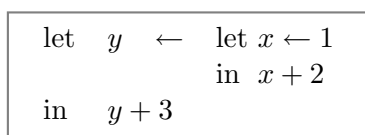


Figure 6.5: Chained lets

Here we chained the two lets one after the other. We will refer to such a construction as *chained lets* or *chained applications*. In the low-level syntax, this program becomes

$$(\lambda y. y + 3)((\lambda x. x + 2)1)$$

This is really another program as the one described in Figure 6.4, but we would still expect it to be observationally equivalent to the program in Figure 6.5. Hence, the following example models an application of the transformation that we call *nesting chained lets* which transforms a chained let into a nested let.

Example 6.9. The first two programs in Figure 6.6 should be observationally equivalent by the transformation described in this section, while the last program is just identical to the second one by syntactic sugar.

The converse direction of the transformation does not hold unconditionally. In the second or third program of Figure 6.6, consider the expression $y + 3$. We also expect that we can transform it into the first program, and indeed we can, but this only works because this expression does not contain the variable x . If it did (e.g. if we had $y + x$ instead of $y + 3$), then the programs would certainly not be observationally equivalent, as the variable x would be free in the first program, but not in the second program.

Hence our formalization will be as follows. We describe a chained let by the construction $(\lambda.R)((\lambda.Q)P)$, where P , Q and R are programs. The program obtained by transforming

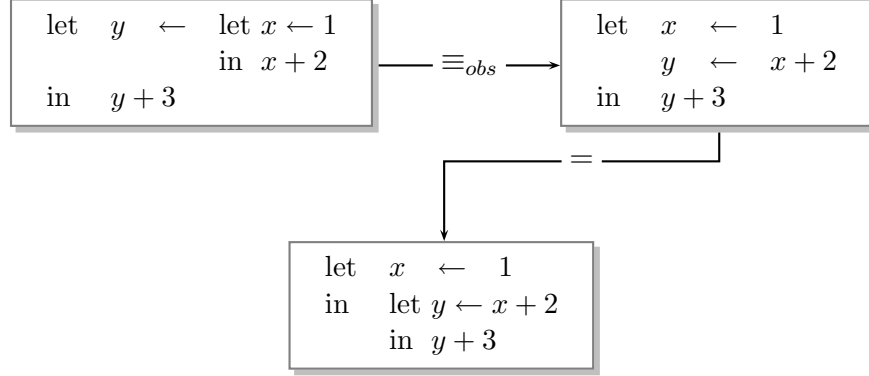


Figure 6.6: Nesting chained lets

it as above then has the form $(\lambda.(\lambda. \uparrow_1 R)Q)P$. Here the lift operator is useful to model that R cannot access the variable bound to the result of P (as is the case in the original program), since we shift all variables with deBruijn index greater or equal than 1 in R by 1, so that no variable references the outer λ . Furthermore, in this manner, all other potentially free variables in R are bound to the same λ 's in the context as they were in the original program. Thus, this formalization is sensible, and we can state the following

Theorem 6.10. *Let P , Q and R be programs. Then the following identity holds true.*

$$A := (\lambda.R)((\lambda.Q)P) \equiv_{obs} (\lambda.(\lambda. \uparrow_1 R)Q)P =: B$$

Proof. We show that $A \equiv_{CIU} B$. For all evaluation contexts E , lists of values a , stores σ and event lists η such that $E[A^a]|\sigma|\eta$ and $E[B^a]|\sigma|\eta$ are fully closed, we have

$$\begin{aligned}
 & T(E[(\lambda.(\lambda. \uparrow_1 R)Q)P]^a|\sigma|\eta) \\
 &= \llbracket E[(\lambda.(\lambda. \uparrow_1 R)Q)P]^a|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(by Definition 3.20)} \\
 &= \llbracket E[(\lambda.(\lambda. \uparrow_1 R)Q)\square]^a\llbracket P^a \rrbracket|\sigma|\eta \rrbracket(\mathcal{PS}) && (*, \text{ see below}) \\
 &= \left(\Lambda(v, \sigma', \eta'). \llbracket E[(\lambda.(\lambda. \uparrow_1 R)Q)v]^a|\sigma'|\eta' \rrbracket \right) \cdot \llbracket P^a|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(chaining rule)} \\
 &= \left(\Lambda(v, \sigma', \eta'). \llbracket E[(\lambda.R)((\lambda.Q)v)]^a|\sigma'|\eta' \rrbracket \right) \cdot \llbracket P^a|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(by Lemma 6.11)} \\
 &= \llbracket E[(\lambda.R)((\lambda.Q)\square)]^a\llbracket P^a \rrbracket|\sigma|\eta \rrbracket(\mathcal{PS}) && \text{(chaining rule)} \\
 &= \llbracket E[(\lambda.R)((\lambda.Q)P)]^a|\sigma|\eta \rrbracket(\mathcal{PS}) && (**, \text{ see below}) \\
 &= T(E[(\lambda.R)((\lambda.Q)P)]^a|\sigma|\eta) && \text{(by Definition 3.20)}
 \end{aligned}$$

For (*), we use that $E' := E[(\lambda.(\lambda. \uparrow_1 R)Q)\square]^a$ is another evaluation context, such that $E'[P^a] = E[(\lambda.(\lambda. \uparrow_1 R)Q)P^a]^a = E[(\lambda.(\lambda. \uparrow_1 R)Q)P]^a$. Here note that in the term in the middle the substitution a is applied to P twice, but since we assume that a closes the entire term, the second substitution has no effect on P . The argument for (**) is analogous.

We obtain that $(\lambda.R)((\lambda.Q)P) \equiv_{CIU} (\lambda.(\lambda. \uparrow_1 R)Q)P$, which proves the theorem. \square

Lemma 6.11. *In Theorem 6.10 we used that for all evaluation contexts E , lists of values a , programs R and Q , values v and stores σ and event lists η the following equation holds true.*

$$\llbracket E[\langle (\lambda.(\lambda. \uparrow_1 R)Q)v \rangle^a] | \sigma | \eta \rrbracket = \llbracket E[\langle (\lambda.R)((\lambda.Q)v) \rangle^a] | \sigma | \eta \rrbracket$$

Proof. We compute

$$\begin{aligned} & \llbracket E[\langle (\lambda.(\lambda. \uparrow_1 R)Q)v \rangle^a] | \sigma | \eta \rrbracket \\ &= \llbracket E[\langle ((\lambda. \uparrow_1 R)Q)\{v\} \rangle^a] | \sigma | \eta \rrbracket && \text{(by Corollary 6.4)} \\ &= \llbracket E[\langle (\lambda. \uparrow_1 R)\{v\}(Q\{v\}) \rangle^a] | \sigma | \eta \rrbracket && \text{(by Definition 3.7)} \\ &= \llbracket E[\langle (\lambda.R)(Q\{v\}) \rangle^a] | \sigma | \eta \rrbracket && \text{(*, see below)} \\ &= \llbracket E[\langle (\lambda.R)((\lambda.Q)v) \rangle^a] | \sigma | \eta \rrbracket && \text{(by Corollary 6.4)} \end{aligned}$$

It remains to show for (*) that we have $(\lambda. \uparrow_1 R)\{v\} = \lambda.R$. First observe that for all programs P , we have that $\uparrow P$ contains no substitution candidates, since all variables are lifted by 1. Thus we have that $(\uparrow P)\{v\} = P$ by Definitions 3.5 and 3.7 of the lift operator and the substitution operator. Here note that the lift operator shifts all free variables up by 1, and the substitution operator shifts them down by 1. Now if P has the form $\lambda.R$, we obtain that

$$\lambda.R = (\uparrow (\lambda.R))\{v\} = (\lambda. \uparrow_1 R)\{v\}$$

This proves the lemma. \square

6.1.5 Chaining Nested Applications

As already mentioned, under sensible side conditions we can also apply the previous transformation in the reverse direction. That is, when we consider a nested application of the form $(\lambda x.(\lambda y.R)Q)P$, we can transform it into a chained application of the form $(\lambda y.R)((\lambda x.Q)P)$, provided that x does not occur free in R .

Intuitively, we can think of a “big” program A that first uses a program P to compute a value which is then bound to a variable x in A . Then this result is used in the computation of another variable y . If we know that x is not used in the remaining of the program A , then we can just compute the variable y directly and “hide” the variable x in the computation of the variable y . We continue the example from the last section.

Example 6.12. This transformation is analogous to the previous example, only in the reverse direction. It only works because the variable x does not occur free in the expression $y + 3$ in the first program.

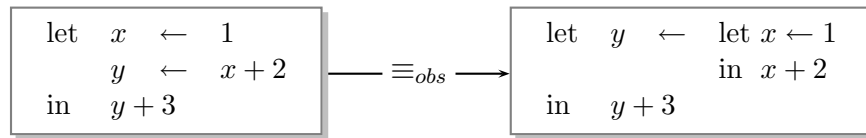


Figure 6.7: Chaining nested lets

The validity of this transformation follows directly from Theorem 6.10, since this theorem asserts the observational equivalence of both programs, which is a symmetric relation. Indeed, the side condition that R cannot access x in the program $(\lambda x.(\lambda y.R)Q)P$ is automatically ensured by the fact that R cannot access the variable $\text{var } 1$ in the program $(\lambda.(\lambda. \uparrow_1 R)Q)P$ according to our formalization for the nameless form, as the lift operator lift all variables starting from $\text{var } 1$ up by 1. We make this observation explicit here, since it is interesting to note that we need the side condition only for this direction, and we will indeed use both directions later on.

6.1.6 Line Swapping

Often, it is quite helpful if we are allowed to swap two consecutive lines in a program. Clearly, such a transformation does not hold under all circumstances. Intuitively, the two lines should not be able to influence each other. For instance, if the first line computes a value for some variable x which is used in the next line, certainly these two lines cannot be swapped. Similarly, if both lines access the store or the eventlist (and at least one of the lines changes them), the result might change. However we expect that there are situations where the order of two lines does not matter, and we might like to swap them.

Example 6.13. Consider the first two lines in the two programs at the top of Figure 6.8. Note that with *lines* here we refer more precisely to nested applications, but in this case the term “lines” conforms well to what we think of as a line in a program. We expect these two programs to be observationally equivalent. Indeed, here it should not make any difference which of the variables a and b is assigned its value first.

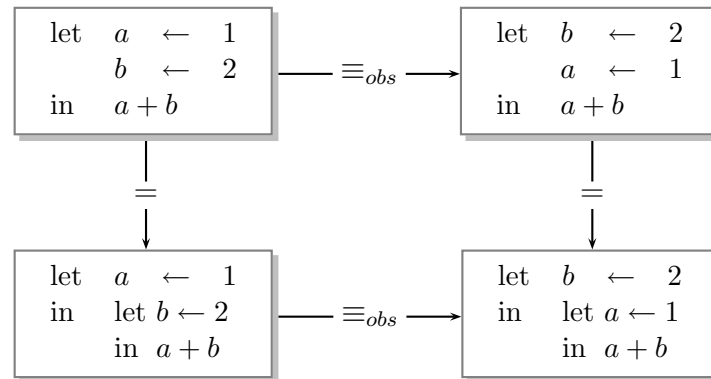


Figure 6.8: Line swapping

First we are going to formulate a slightly weaker version of the theorem asserting that for all programs A and B , under sensible conditions it holds that the programs $(\lambda a.(\lambda b.(a, b))B)A$ and $(\lambda b.(\lambda a.(a, b))A)B$ are observationally equivalent. The proof is pretty involved, but fortunately has already been proven by Backes et al. Then we will generalize the theorem to programs of the form $(\lambda a.(\lambda b.C)B)A$ and $(\lambda b.(\lambda a.C)A)B$, for all programs C .

Now we want to formalize the conditions for the first theorem. First we need that one of the programs A and B is not allowed to raise events or issue the command eventlist, i.e. it cannot access the eventlist in any way whatsoever. It does not matter which of the two programs we restrict in this way, since the transformation is symmetric. Furthermore, they are not allowed to access the same places in the store, or places outside of the store. For this, we introduce the notion of *accessible locations*.

Definition 6.14 (Accessible locations). Let A be a program. Let $L(A)$ be the set of locations in A (syntactically). The set of *accessible locations* $\text{Loc}(A, \sigma)$ of a program state $A|\sigma|\eta$ is defined as the smallest set of locations reachable from A in $A|\sigma|\eta$, namely

$$\text{Loc}(A, \sigma) := \bigcap \{l \mid L(A) \cup \bigcup_{i \in l} L(\sigma[i]) \subseteq l\}.$$

Now we can state the theorem with the appropriate side conditions.

Theorem 6.15. *Let A and B be programs and let σ be a store. Assume that*

1. $\text{Loc}(A, \sigma) \cap \text{Loc}(B, \sigma) = \emptyset$,
2. A, B, σ do not contain locations greater or equal than $|\sigma|$ and
3. B does not contain events or the command eventlist.

Then the following equivalence holds:

$$(\lambda a. (\lambda b. (a, b)) B) A \equiv_{obs} (\lambda b. (\lambda a. (a, b)) A) B.$$

More precisely, in the nameless representation we write

$$P_{AB} := (\lambda. (\lambda. (\text{var } 1, \text{var } 0)) \uparrow B) A \equiv_{obs} (\lambda. (\lambda. (\text{var } 0, \text{var } 1)) \uparrow A) B := P_{BA}.$$

Proof. The most difficult part of the proof has already been performed at the chair of Michael Backes (see [2]). The main idea is to show that under the given conditions, the programs P_{AB} and P_{BA} are similar in the sense that their denotations can be computed from each other. More precisely, since, in P_{AB} , first A is executed and then B , while in P_{BA} it is the opposite, one can show that the denotations $\llbracket P_{AB}|\sigma|\eta \rrbracket$ and $\llbracket P_{BA}|\sigma|\eta \rrbracket$ are identical up to reordering of the store. This kind of equivalence is then called *L-renaming equivalence*. The fact that L-renaming equivalence implies observational equivalence has also already been performed. \square

We now show how to generalize this theorem. For this we first need an auxiliary function *swap* which, loosely speaking, “swaps” in a program C the free variables $\text{var } 0$ and $\text{var } 1$, more precisely those two variables which are bound respectively to the results of the two preceding lines in the context. This is exactly what we need for line swapping, since in the resulting program after a line swap, the first two λ ’s bound respectively to the two swapped lines will also be swapped.

Definition 6.16. Let $k \in \mathbb{N}$, and P be a program term. We define the $swap_k$ operator by recursion on P . The k is used to keep track of how many λ 's have been encountered so far in a subterm, similar as we did for the lift operator. Indeed, the only cases which are non-trivial are the cases where P is a variable or an abstraction. They are shown in Figure 6.9. All the other cases are analogous to the definition of the lift operator in Figure 3.1. We now define the notation $swap P := swap_0 P$.

$$\begin{aligned} swap_k(\text{var } i) &= \begin{cases} \text{var } i + 1 & \text{if } i = k \\ \text{var } i - 1 & \text{if } i = k + 1 \\ \text{var } i & \text{otherwise} \end{cases} \\ swap_k(\lambda. P) &= \lambda. swap_{k+1}(P) \end{aligned}$$

Figure 6.9: Definition of $swap_k$

Theorem 6.17. Let A , B and C be programs, and let σ be a store. Assume the same conditions on A , B and σ as in Theorem 6.15. Then the following equivalence holds:

$$P_{ABC} := (\lambda.(\lambda. C) \uparrow B)A \equiv_{obs} (\lambda.(\lambda. swap C) \uparrow A)B =: P_{BAC}$$

Proof. The idea is to transform the initial program into a form where Theorem 6.15 can be applied. This is possible by defining an appropriate program C' such that we can in some way substitute C' for C in P_{ABC} and then use this new form to apply the Theorem. In this proof, by abuse of the language, we use the lift operator in named terms. Here we are in fact thinking of nameless terms, but giving the variables names instead of dealing with a deBruijn representation makes the programs much more readable.

We define the program C' as follows:

$$C' := \lambda x_1. (\lambda x_2. (\lambda x_3. \uparrow_2 C))(\text{fst } x_1)(\text{snd } x_1)$$

What this program does is to take a pair and “split it up” by passing the first and second projections of this pair to $\uparrow_2 C$. Here the \uparrow_2 operator models the fact that C cannot access the pair x_1 itself. The intuition is that C' is a program which is similar to C , but it expects a pair (x_2, x_3) instead of expecting these variables successively. Using this program, we can apply the transformations shown in Figure 6.10.

For (1), the transformation appropriately inserts the program C' in place of C . Here the double lift operator is needed to express the fact that C' cannot access x_a or x_b . This step is mostly computational. It follows from the observational equivalence of β -equivalent terms, and similar lemmas that assert the observational equivalences $\text{fst}(x_a, x_b) \equiv_{obs} x_a$ and $\text{snd}(x_a, x_b) \equiv_{obs} x_b$.

The argument for (2) essentially uses an interesting observation which states that for two evaluation contexts E_1 and E_2 , if it holds for all values v that $E_1[v] \equiv_{obs} E_2[v]$ this implies that for all programs P , we have that $E_1[P] \equiv_{obs} E_2[P]$, which itself eventually follows from the chaining rule.

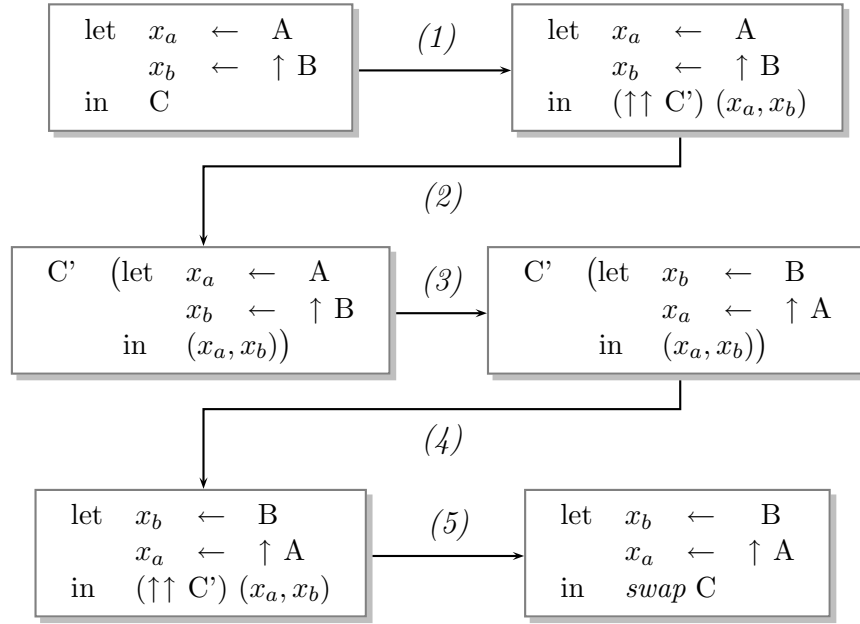


Figure 6.10: How to generalize line swapping

For (3), we apply Theorem 6.15. Recall that here already, in the nameless representation the pair $(\text{var } 1, \text{var } 0)$ has been changed to $(\text{var } 0, \text{var } 1)$. The argument for (4) is analogous to (2), but is used for the converse direction.

For (5), the argument is also similar to (1), but here, moreover, since the pair has been swapped we need to apply the *swap* function to C .

As one can see, the transformations are not difficult to understand, but it is a tedious task to perform all the computational steps to formally prove the observational equivalence. It is with such kinds of tasks where we can hope to obtain much help from Isabelle's logic to prove the validity of the transformations. Indeed, the transformations described in this proof have been performed by Matthias Berg in Isabelle/HOL. \square

6.2 A Transformation based on Computational Indistinguishability

6.2.1 The DDH Assumption

Recall the Decisional Diffie-Hellman problem from Section 4.3. We stated that, for some sequence of cyclic groups $(G_n)_{n \in \mathbb{N}}$ with generators g_n , the tuples of the form $(1^n, g_n^x, g_n^y, g_n^{xy})$ and $(1^n, g_n^x, g_n^y, g_n^z)$ are assumed to be computationally indistinguishable in n , if x, y and z are elements distributed uniformly at random in $[0, \dots, |G_n| - 1]$.

We would now like to model a corresponding program transformation in our language. More precisely, we will consider two games DDH_xy and DDH_z, which, as the name suggest,

will model the two DDH games as described in Definition 4.3. Both games will be parameterized with an efficient uniform decider D and a security parameter n , that is, they will be treated as constants within the program term. In Chapter 7, we will be able to prove the IND-CPA security of ElGamal only under the assumption that the two games defined in Figure 6.11 are computationally indistinguishable in n .

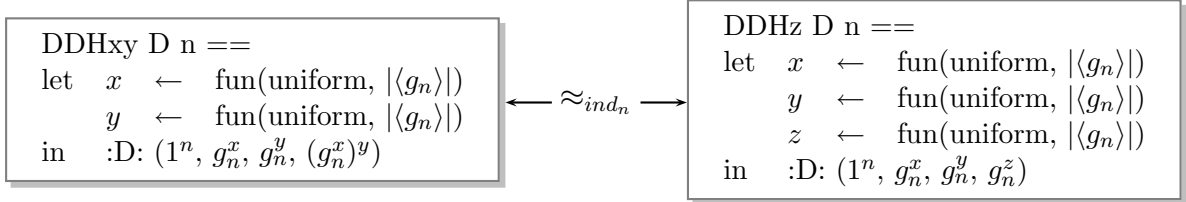


Figure 6.11: Definition of the DDH games DDHxy and DDHz

Formally, we define the following assumption.

Assumption 6.18. *Let D be a probabilistic polynomial-time program that always returns a bit. Let n be a natural number. We assume the following program terms to be computationally indistinguishable in n :*

$$(\lambda x. (\lambda y. :D: (1^n, g_n^x, g_n^y, g_n^{xy})) \mu) \mu \equiv_{obs} (\lambda x. (\lambda y. (\lambda z. :D: (1^n, g_n^x, g_n^y, g_n^z)) \mu) \mu) \mu$$

where we use the notation $\mu := \text{fun}(\text{uniform}, |\langle g_n \rangle|)$.

6.3 A Transformation based on Denotational Equivalence

6.3.1 Multiplication with Random Elements in Cyclic Groups

The last transformation that we look at is rather mathematical and concerns cyclic groups. Let G be a cyclic group, and g a generator of G .

Consider a program P that selects a value x uniformly at random from $[0, \dots, |G| - 1]$, i.e. from the set of exponents of G , and then returns the value g^x . It is intuitively clear (and will be shown below) that the value to which it evaluates is distributed uniformly at random in the set of values of the group G . Next, let m be some constant in G . Now consider a program Q that also selects a value x uniformly at random from the set of exponents of G , but then returns the value $m \times g^x$. We expect (and we will in fact show below) that Q evaluates again to a value distributed uniformly at random in G . Since both programs evaluate to the same distribution over values, we expect them to be denotationally equivalent. The goal of this section is to give a formal statement and proof that this holds indeed true.

For stating the main result of this section we need some notations. Let $(G_n)_{n \in \mathbb{N}}$ denote a sequence of cyclic groups, and, for each n , fix a generator g_n of G_n . We assume that the

elements of G_n can be represented as bitstrings in our language. For a fixed N and a fixed constant $m \in G_N$, we consider the programs P and Q defined in Figure 6.12.

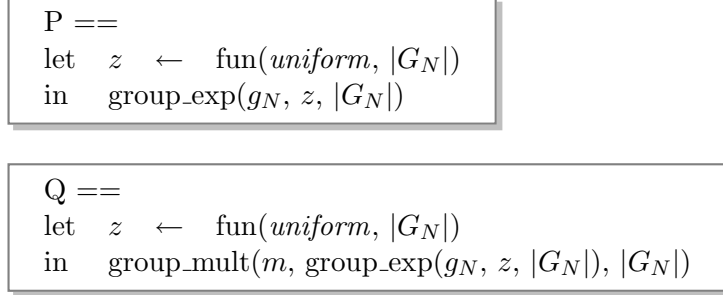


Figure 6.12: Definition of P and Q

We shall assume from now on:

Assumption 6.19. *Let $P|\sigma|\eta$ be a program state. We denote by $\mu_P^{(n)}$ the distribution over program states $step_n(P|\sigma|\eta)$, that is, the distribution of $P|\sigma|\eta$ after n steps. Assume that for a natural number j , the distribution $\mu_P^{(j)}$ is of the form*

$$\mu_P^{(j)}(A) = \frac{1}{l} |\{v \in [0, \dots, l-1] : \text{group_exp}(g_N, v, l) | \sigma | \eta \in A\}| \quad (A \in \Sigma_{\mathcal{PS}}),$$

where g_N is a generator of G_n and $l = |G_N|$. Then we assume that there is a natural number k such that the distribution $\mu_P^{(j+k)}$, i.e. the distribution of $P|\sigma|\eta$ after $j+k$ steps, has the form

$$\mu_P^{(j+k)}(A) = \frac{1}{l} |\{v \in [0, \dots, l-1] : g_N^v | \sigma | \eta \in A\}| \quad (A \in \Sigma_{\mathcal{PS}}).$$

Similarly, if the distribution $\mu_P^{(j)}$ has the form

$$\mu_P^{(j)}(A) = \frac{1}{l} |\{v \in [0, \dots, l-1] : \text{group_mult}(m, \text{group_exp}(g_N, v, l), l) | \sigma | \eta \in A\}|$$

for $A \in \Sigma_{\mathcal{PS}}$, where g_N is a generator of G_n , $l = |G_N|$ and m is an element of G_n , we assume that there is a natural number k' such that the distribution after $j+k'$ steps has the form

$$\mu_P^{(j+k')}(A) = \frac{1}{l} |\{v \in [0, \dots, l-1] : mg_N^v | \sigma | \eta \in A\}| \quad (A \in \Sigma_{\mathcal{PS}}).$$

That is, we assume that under the given conditions, the program `group_exp` always computes the value g_N^v in a finite number of steps k . This means that the program `group_exp` always performs the same number of steps for values from the specified domains, i.e. it always has a “worst-case” running time. The argument for the program `group_mult` is analogous. This assumption is not unrealistic, as we could always perform a number of

skips (i.e. steps which do not do anything) in the cases where the algorithms perform better, so that they always performs the same number of steps. This assumption eases the proof a lot.

Recall from Section 3.4.2 that the denotation of a program is a sub-probability measure μ on the set of program states \mathcal{PS} , which assumes the value 0 for sets $A \in \Sigma_{\mathcal{PS}}$ which do not contain elements of \mathcal{VS} . Finally, recall from section Notations that $G_N|\sigma|\eta$, for fixed σ and η denotes the set $\{x|\sigma|\eta : x \in G_N\}$. We then have:

Theorem 6.20. *For a fixed N , let P and Q be the programs defined in Figure 6.12, and let σ and η be a store and an event list, respectively. Let μ be the uniform distribution on program states with carrier $G_N|\sigma|\eta$ (i.e. $\mu(A) = |A \cap (G_N|\sigma|\eta)|/|G_N|$ for $A \in \Sigma_{\mathcal{PS}}$, see Definition 3.23). Then:*

$$\llbracket P|\sigma|\eta \rrbracket = \llbracket Q|\sigma|\eta \rrbracket = \mu.$$

Proof. For the proof we calculate the denotations of $P|\sigma|\eta$ and $Q|\sigma|\eta$ by following the very definition of a denotation of a program state. Let $\mu_P^{(n)} := \text{step}_n(P|\sigma|\eta)$. Recall, that by Definition 3.18 we have

$$\begin{aligned} \mu_P^{(0)} &= \text{step}_0(P|\sigma|\eta) = \delta_{P|\sigma|\eta} \\ \mu_P^{(n+1)} &= \text{step}_{n+1}(P|\sigma|\eta) = (\text{step} \circ \text{step}_n)(P|\sigma|\eta) \\ &= \text{step} \cdot (\text{step}_n(P|\sigma|\eta)) = \text{step} \cdot \mu_P^{(n)}. \end{aligned}$$

Using Definition 3.16 of the application of a kernel to a measure we can rewrite the second formula in the form

$$\mu_P^{(n+1)}(A) = \int_{\mathcal{PS}} \text{step}(x)(A) d\mu_P^{(n)}(x) \quad (A \in \Sigma_{\mathcal{PS}})$$

We calculate $\mu_P^{(n)}(A)$. First of all, we deduce from the preceding formula and the formula for $\mu_P^{(0)}(A)$ that

$$\mu_P^{(1)}(A) = \int_{\mathcal{PS}} \text{step}(x)(A) d\delta_{P|\sigma|\eta}(x) = \text{step}(P|\sigma|\eta)(A).$$

The second identity follows from equation 3.1. We shall show in Lemma 6.21 below that

$$\text{step}(P|\sigma|\eta)(A) = \frac{1}{l} |A \cap \{(\lambda z.\text{group_exp}(g_N, z, l))v|\sigma|\eta : v \in [0, \dots, l-1]\}|, \quad (6.1)$$

where $l = |G_N|$. Hence we obtain

$$\begin{aligned} \mu_P^{(2)}(A) &= \int_{\mathcal{PS}} \text{step}(x)(A) d\text{step}(P|\sigma|\eta)(x) \\ &= \frac{1}{l} \sum_{v \in [0, \dots, l-1]} \text{step}((\lambda z.\text{group_exp}(g_N, z, l))v|\sigma|\eta)(A), \end{aligned}$$

where the second identity follows again from equation 3.1. By the definition of *step* and by the reduction rule APPL from Definition 3.14, we have

$$\text{step}((\lambda z.\text{group_exp}(g_N, z, l))v|\sigma|\eta) = \delta_{R_v|\sigma|\eta},$$

where we use $R_v := \text{group_exp}(g_N, v, l)$. Therefore the last formula for $\mu_P^{(2)}(A)$ becomes

$$\mu_P^{(2)}(A) = \frac{1}{l} \sum_{v \in [0, \dots, l-1]} \delta_{R_v|\sigma|\eta}(A) = \frac{1}{l} |\{v \in [0, \dots, l-1] : R_v|\sigma|\eta \in A\}|.$$

By Assumption 6.19 we obtain

$$\begin{aligned} \mu_P^{(2+k)}(A) &= \frac{1}{l} |\{v \in [0, \dots, l-1] : g_N^v|\sigma|\eta \in A\}| \\ &= \frac{1}{l} |\{g \in G_N : g|\sigma|\eta \in A\}| \\ &= \frac{1}{l} |A \cap (G_N|\sigma|\eta)| \\ &= \mu(A). \end{aligned}$$

But then

$$\begin{aligned} \mu_P^{(3+k)}(A) &= \int_{\mathcal{PS}} \text{step}(x)(A) d\mu(x) = \frac{1}{l} \sum_{x \in G_N|\sigma|\eta} \text{step}(x)(A) \\ &= \frac{1}{l} \sum_{x \in G_N|\sigma|\eta} \delta_x(A) = \frac{1}{l} |A \cap (G_N|\sigma|\eta)| = \mu(A). \end{aligned}$$

Inductively we hence see that the sequence $\mu_P^{(n)}$ gets stationary for $n \geq 2+k$. Summing up we have

$$\mu_P^{(n)} = \begin{cases} \delta_{P|\sigma|\eta} & \text{if } n = 0, \\ \text{step}(P|\sigma|\eta) & \text{if } n = 1, \\ \mu & \text{if } n \geq 2+k. \end{cases}$$

The denotations $\mu_P^{(n)}$ for $2 \leq n < 2+k$ are the intermediate distributions over program states generated by the program *group_exp*. Finally, $\llbracket P|\sigma|\eta \rrbracket(A) = \sup \bar{\mu}_P^{(n)}(A)$ (see 3.19), where $\bar{\mu}_P^{(n)}(A) := \mu_P^{(n)}(A \cap \mathcal{VS})$. Since, for $n < 2+k$, the restrictions of $\bar{\mu}_P^{(n)}$ to $\Sigma_{\mathcal{VS}}$ are identically zero we therefore find $\llbracket P|\sigma|\eta \rrbracket = \mu$ as claimed (here, we use that for $2 \leq n < 2+k$, the intermediate steps generated by the program *group_exp* are never value states).

The proof of $\llbracket Q|\sigma|\eta \rrbracket = \mu$ is analogous. Here, for calculating the measures $\mu_Q^{(n)} := \text{step}_n(Q|\sigma|\eta)$ we have to replace (6.1) by

$$\begin{aligned} &\text{step}(Q|\sigma|\eta)(A) \\ &= \frac{1}{l} |A \cap \{(\lambda z.\text{group_mult}(m, \text{group_exp}(g_N, z, l), l))v|\sigma|\eta : v \in [0, \dots, l-1]\}|. \end{aligned}$$

(see Lemma 6.21). By a similar calculation as above (and the application of Assumption 6.19) we then obtain

$$\mu_Q^{(2+k')}(A) = \frac{1}{l} |\{v \in [0, \dots, l-1] : mg_N^v | \sigma | \eta \in A\}|.$$

But if v runs through $[0, \dots, l-1]$ then mg_N^v runs through G_N , and therefore we obtain here similarly $\mu_Q^{(2+k')}(A) = \frac{1}{l} |A \cap (G_N | \sigma | \eta)|$. This proves the theorem. \square

Lemma 6.21. *In Theorem 6.20 we used the following. Let ν_1 be the uniform distribution on program states with carrier*

$$\{(\lambda z. \text{group_exp}(g_N, z, l))v | \sigma | \eta : v \in [0, \dots, l-1]\}$$

where g_N is a generator of G_N and $l = |G_N|$. Let ν_2 be the uniform distribution on program states with carrier

$$\{(\lambda z. \text{group_mult}(m, \text{group_exp}(g_N, z, l), l))v | \sigma | \eta : v \in [0, \dots, l-1]\}$$

where m is a value from G_N . Then we have, respectively,

$$\begin{aligned} \text{step}(P | \sigma | \eta)(A) &= \nu_1, \\ \text{step}(Q | \sigma | \eta)(A) &= \nu_2. \end{aligned}$$

Proof. We will show that

$$\begin{aligned} P | \sigma | \eta &\rightsquigarrow \nu_1, \\ Q | \sigma | \eta &\rightsquigarrow \nu_2. \end{aligned}$$

The claim then follows by the definition of the sub-Markov kernel *step*. We define the evaluation context

$$E[\square] := (\lambda z. \text{group_exp}(g_N, z, l))\square.$$

According to the reduction rule FUN as in Definition 3.14 (reduction rules of the language), we have

$$E[\text{fun}(\text{uniform}, l)] | \sigma | \eta \rightsquigarrow \mu' := g(\text{uniform}(l)),$$

where $g : V_0 \rightarrow \mathcal{PS}$, $g(x) = E[x] | \sigma | \eta$. Thus by definition of $g(\mu)$ for a measure μ (see Section 3.4.1) we have

$$\mu'(B) = \mu_l(\{v \in V_0 : E[v] | \sigma | \eta \in B\}),$$

where we use $\mu_l = \text{uniform}(l)$. We have to show that μ' is the uniform distribution on program states with carrier $\{(\lambda z. \text{group_exp}(g_N, z, l))v | \sigma | \eta : v \in [0, \dots, l-1]\}$. Indeed we have:

$$\begin{aligned} \mu'(B) &= \mu_l(\{v \in V_0 : E[v] | \sigma | \eta \in B\}) \\ &= \frac{1}{l} |\{v \in V_0 : E[v] | \sigma | \eta \in B\} \cap [0, \dots, l-1]| \\ &= \frac{1}{l} |\{v \in [0, \dots, l-1] : E[v] | \sigma | \eta \in B\}| \\ &= \frac{1}{l} |\{v \in [0, \dots, l-1] : (\lambda z. \text{group_exp}(g_N, z, l))v | \sigma | \eta \in B\}| \end{aligned}$$

The proof of the statement for Q is analogous: replace in the preceding proof the evaluation context $E[\square]$ by $F[\square] := (\lambda z.\text{group_mult}(m, \text{group_exp}(g_N, z, l), l))\square$. This proves the lemma. \square

Note that here, we have only shown the denotational equivalence of the two programs under the assumption that m is a value in G_N . However, the games that we will consider later will not in fact use such a value m . Instead, m will just be some free variable in the term. In this case, it is not possible to conclude the observational equivalence of the two programs. This is because when we consider all possible instantiations of the variable m by quantifying over all list of values a which we use to close the term (see Definition 3.29 of CIU equivalence), then m will also be instantiated with values which are not elements of G_N . What we need here is an extended version of the CIU Theorem, which allows us to impose certain invariants by restricting the list of values a to appropriate values.

Michael Backes, Matthias Berg and Dominique Unruh are currently working on such a version of the CIU Theorem with invariants. For this proof, what one would have to do is to impose the invariant that m is only instantiated with values which are elements of G_N . Then, one could use the above Theorem as a Lemma to conclude the observational equivalence of the two programs under this invariant. However, since this would require a huge expansion of the framework and the language, and since this version of the CIU Theorem is still in an experimental phase, we do not go into further details here. Rather, we will later on assume that the above programs are observationally equivalent under the invariant that m only gets instantiated with elements of G_N , which certainly holds true.

7 The Proof of IND-CPA Security of ElGamal

As outlined in Chapter 5, our goal is to show that for all efficient adversaries (A, A') , the two games $\text{IND_CPA1}_{\text{ElGamal } A, A', n}$ and $\text{IND_CPA2}_{\text{ElGamal } A, A', n}$ are computationally indistinguishable in n . The main goal of this section is to present the game transformations this can be achieved with.

Roughly speaking, we will transform the first CPA game, within which we encrypt m_1 , into a computationally indistinguishable program where m_1 is not used any longer for the calculation of the ciphertext. We will call this game RandomElGamalGame . To do this, we will rely on the DDH assumption and use all of the transformations presented in Chapter 6.

By applying the same transformations to the second CPA game, we can see that this game is also computationally indistinguishable from the game RandomElGamalGame . By that we will be able to conclude that (under the DDH assumption) the first and the second CPA game are computationally indistinguishable. Figure 7.1 shows a high-level overview

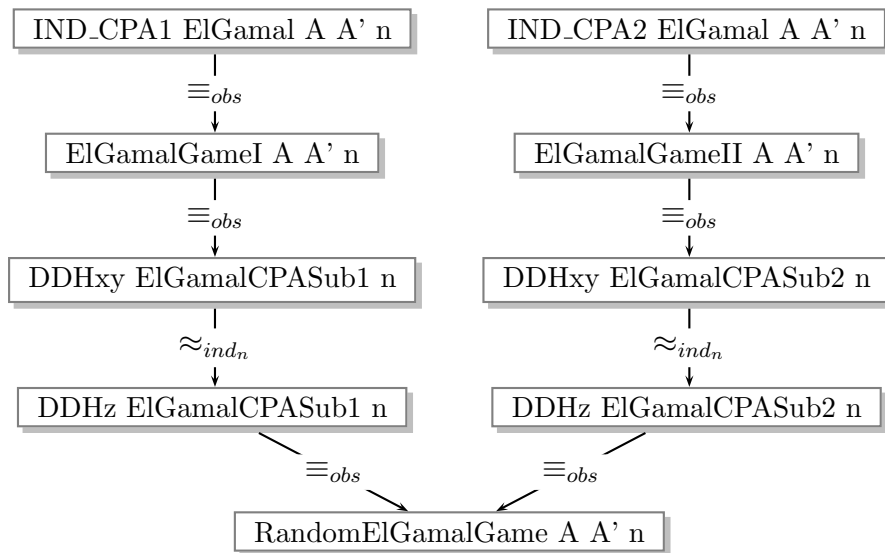


Figure 7.1: High-level overview

of the transformations. Here, for reasons of space, we only give the names of the games, but not all their definitions. The definitions of the games on the left-hand side are depicted in the figures at the beginning of the next sections of this chapter, respectively. Indeed,

to get from one game to the next, most often several transformations are needed, which is why we call this overview *high-level*. The next sections describe respectively these high-level transformations in detail. Furthermore here we note the following. Often, we will apply a transformation which asserts the observational equivalence of two *subterms* of some program term. By the compatibility of observational equivalence (see Theorem 3.36), we then know that the two entire games are also observationally equivalent. Yet, because we want to concentrate on other facts, we will not mention this each time. We note here once and for all in this chapter that if two subterms are observationally equivalent, then so are the whole terms.

As for the games on the right-hand side in Figure 7.1, the transformations are perfectly analogous to those on the left-hand side, as the definitions of the games are analogous, the only difference being that where we use m_1 for encryption in the left games, we use m_2 in the right games. Hence, when we know how to perform the transformations for the left-hand side, the right-hand side becomes trivial. For this reason, in this chapter we will only consider the left-hand side.

7.1 IND_CPA1 ElGamal to ElGamalGame1

Let us first get an intuitive notion of what we want to do in this section.

We start by parameterizing the game IND_CPA1 (see Figure 5.1) with the ElGamal encryption scheme, as defined in Figure 5.3. Recall that in Chapter 5 we have seen definitions of:

1. the CPA game IND_CPA1,
2. the ElGamal key generation algorithm ElGamalGen and
3. the ElGamal encryption algorithm ElGamalEnc.

Now, according to these definitions, the game IND_CPA1 calls a key generation algorithm `encGen ES` which it passes a security parameter, and expects a tuple (e, d) as return value. Furthermore it calls an encryption algorithm `encEnc ES`, which it passes the tuple $(1^n, e, m_1)$, and expects some value c . Our definitions of ElGamalGen and ElGamalEnc are defined such that they behave exactly in this way. Hence, we can substitute these function calls with the definitions of ElGamalGen and ElGamalEnc. We obtain a well-typed program.

In doing so, we get, loosely speaking, a “modularized” game where the calls of the key generation and encryption algorithms are closed functions. In this section, our goal is to transform this game into a more “monolithic” form which describes a CPA game using the ElGamal encryption scheme, as we might have written it if we had defined this game directly. We will call this game ElGamalGame1.

For our first transformation, we just apply the definitions of the key generation and encryption algorithms to their arguments which corresponds to the transformation shown in Figure 7.2. This transformation follows essentially by applying Theorem 6.5 (β -equivalence).

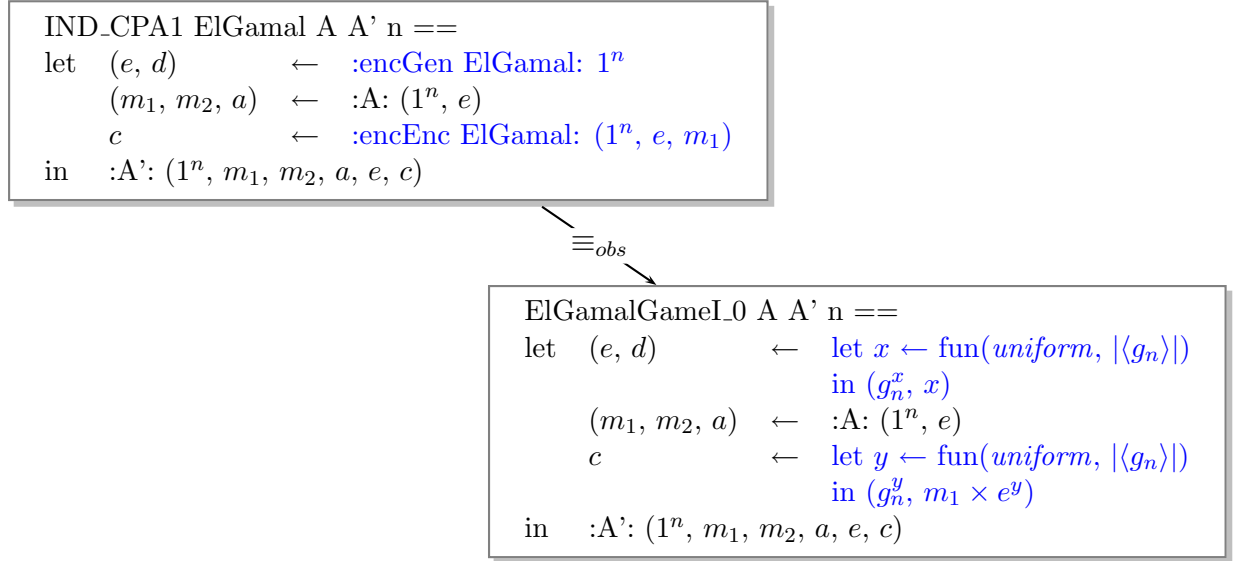


Figure 7.2: Inserting definitions and applying arguments

Next, we want to integrate these subroutines into the main program. Thereby, we get a more “intuitive” and readable form of a CPA game using ElGamal. Formally, we want to make the following transformation.

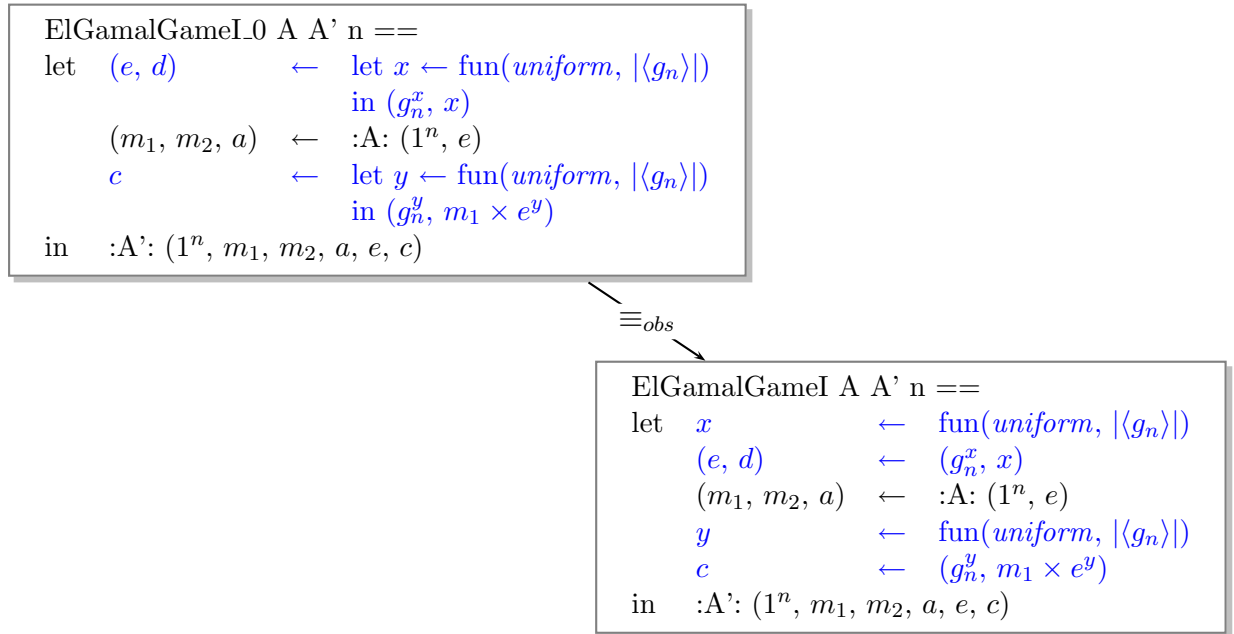


Figure 7.3: Nesting chained applications

This transformation follows by applying Theorem 6.10 (nesting chained applications).

7.2 ElGamalGameI to DDHxy

Recall that the proof of the IND-CPA security of ElGamal relies on the assumed hardness of the DDH problem. Hence, on our way to transforming the game such that the computation of the ciphertext is independent of m_1 , we expect to need this assumption. In Section 6.2.1, we saw the associated program transformation that models the DDH assumption. In this section, we are going to transform our program into such a form that it matches this transformation, which we will then apply in the next section.

At first we are going to propagate the expression (e, d) , that is, we are going to replace every occurrence of e and d by the values we assigned to these variables, and eliminate the assignment itself. Indeed, Theorem 6.7 (expression propagation) tells us that the following transformation yields two observationally equivalent programs.

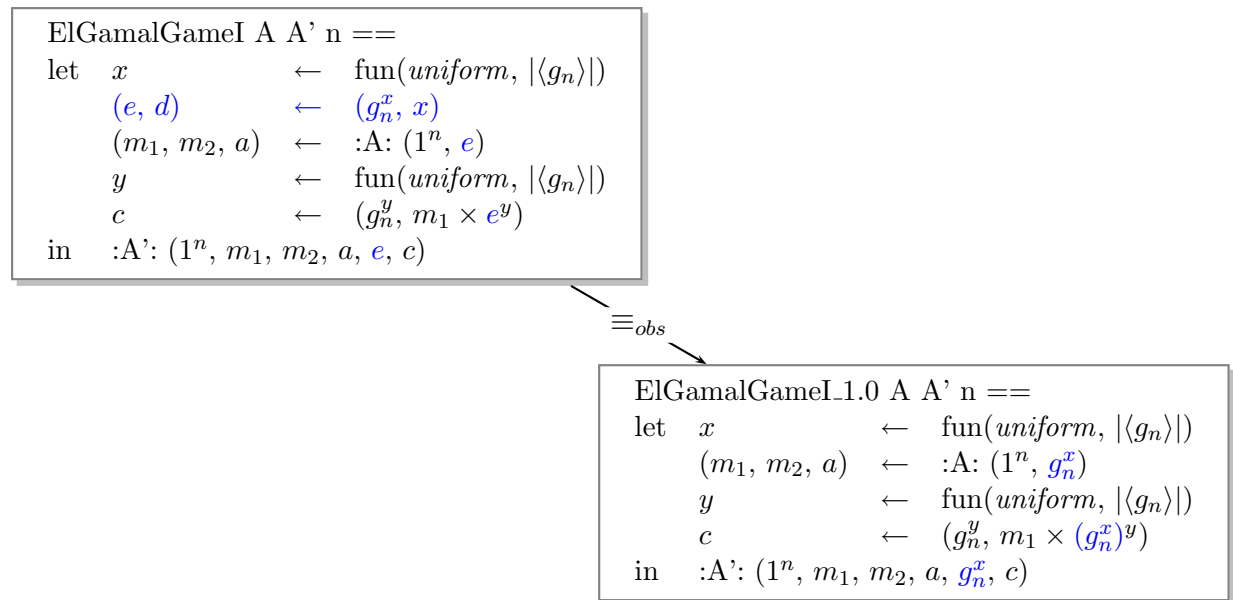
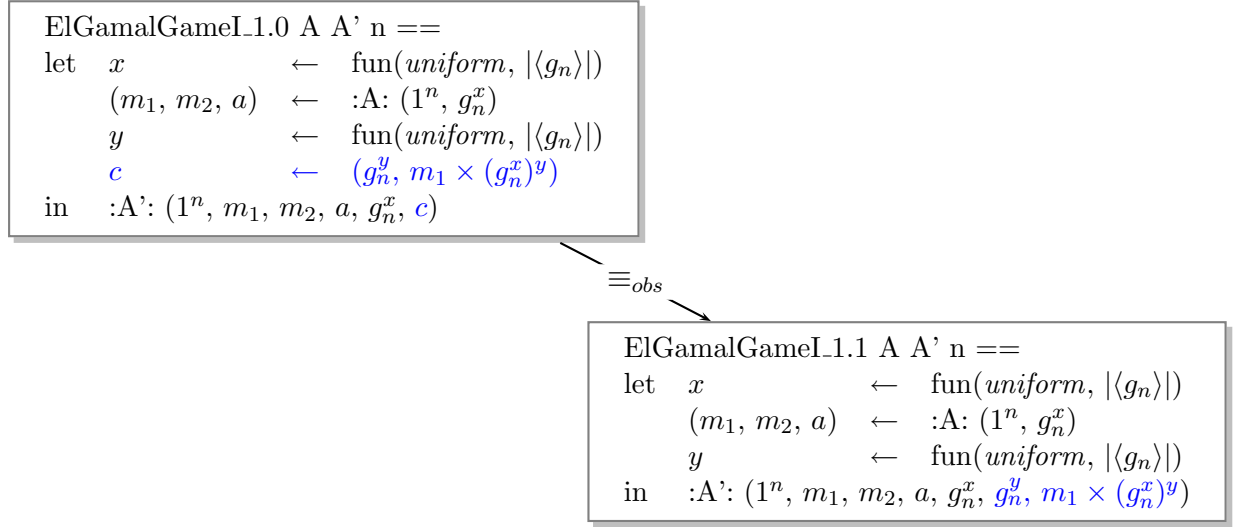


Figure 7.4: Expression propagation of (e, d)

Here on a low-level we additionally have to deal with uncurrying of pairs and simple equivalences such as $\text{fst}(p, v) \equiv_{obs} p$, for a program p and a value v . This is a rather technical (but not difficult) part which concerns the pretty-print syntax, not further detailed here. We now similarly propagate the ciphertext c .

Figure 7.5: Expression propagation of c

This transformation again holds by Theorem 6.7. Here, note that the parameters for A' are internally represented as nested pairs. That is, an expression of the form (a, b, c) is syntactic sugar for the expression $(a, (b, c))$. Thus, it is not necessary to put parentheses around $g_n^y, m_1 \times (g_n^x)^y$ here, or in other words, these parentheses are implicit.

To apply the DDH assumption as explained in Section 6.2.1, the program should have a form where first, two random values are selected uniformly at random from the range of exponents of $|G_n|$, then the values $(1^n, g^x, g^y, (g^x)^y)$ are passed to a probabilistic polynomial-time function. To bring our program into such a form, only two steps remain. First, we are going to swap two lines such that our program selects the two random values for the variables x and y at the very beginning of the execution.

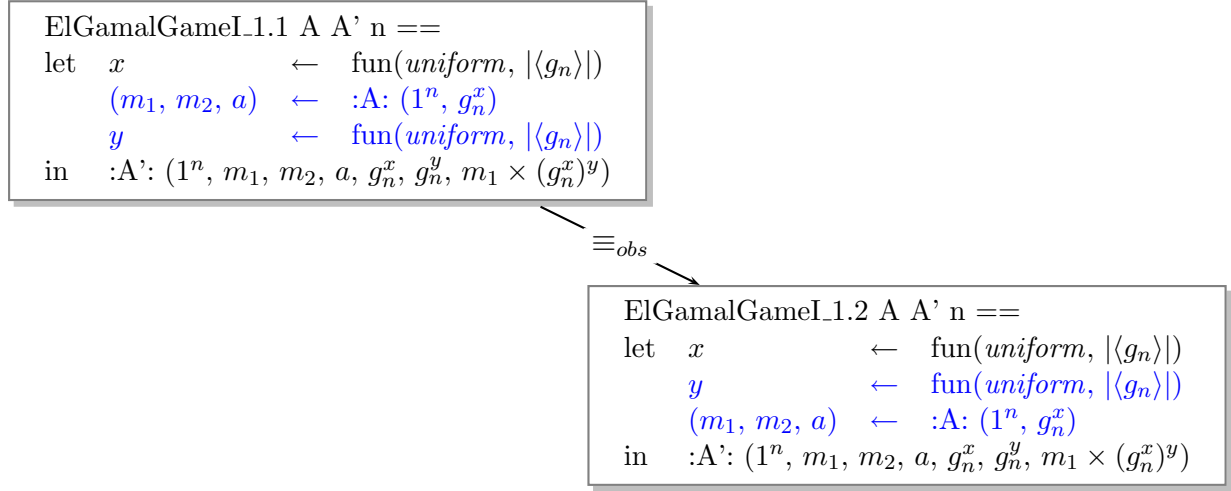


Figure 7.6: Swapping lines

This transformation is valid by Theorem 6.17 (line swapping). Second, we need to abstract the remaining of our program into a closed, probabilistic polynomial-time function that will act as the decider. Since we will need it several times, for the sake of readability we define it here once, and will only refer to it by name from then on. We call it `ElGamalCPASub1`.

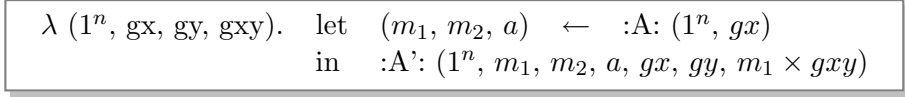


Figure 7.7: Definition of `ElGamalCPASub1`

Observe that this function corresponds indeed to the last two lines of the previous game, namely `ElGamalGameI_1.2`, except that all occurrences of the expressions g^x , g^y and $(g^x)^y$ have been replaced by the variables gx , gy and gxy , which are bound at the beginning of this function. In fact, the term `ElGamalCPASub1 (1^n, g^x, g^y, (g^x)^y)` is β -equivalent to these last two lines. Hence, we can perform the following transformation using Theorem 6.5 (β -equivalence).

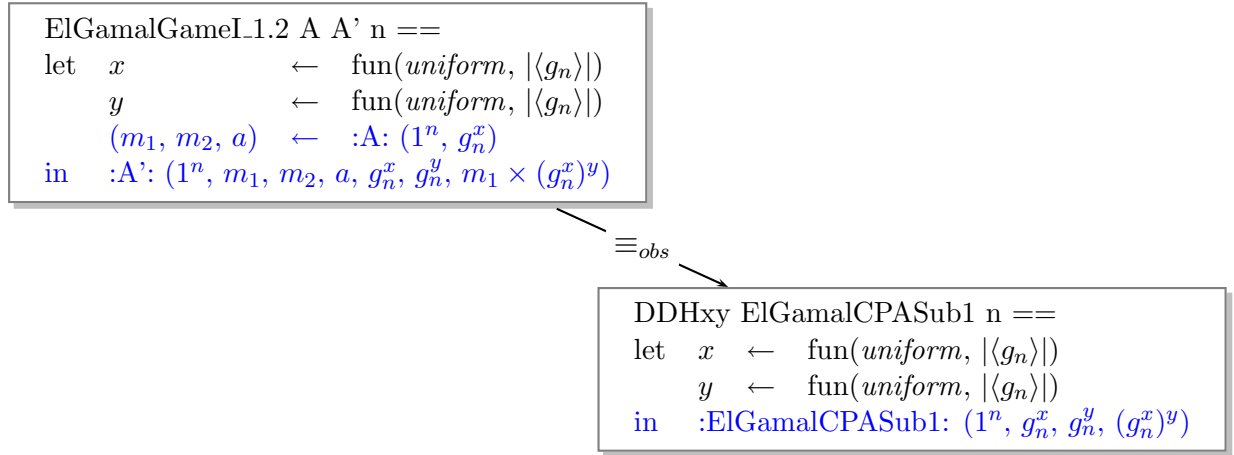


Figure 7.8: The IND_CPA1 ElGamal game as DDH subroutine

A few things should be mentioned here. First, the reason why we call this function ElGamalCPASub1 is because it acts as a “subroutine” for the DDH game which implements the first CPA game for ElGamal. We append a trailing “1” to make it clear that the message m_1 is being encrypted here. In contrast, if we started with IND_CPA2, we would get another function ElGamalCPASub2. The only difference would be that the last parameter for A' was $m_2 \times g^{xy}$ instead of $m_1 \times g^{xy}$.

Second, to apply the DDH assumption we required that the decider be a closed, probabilistic polynomial-time function. The function ElGamalCPASub1 is indeed closed. The variables A and A' are not variables bound in the context, but rather they are parameters for the game itself. That is, they are treated as constants within the term. Hence, in the sense of the language, the function is closed. It is also probabilistic polynomial-time, since we required of the functions A and A' that they also be probabilistic polynomial-time.

7.3 DDH_{xy} to DDH_z

Our next transformation is based on the assumed hardness of the DDH problem. Indeed, at this point we can apply Assumption 6.18. Our program has exactly the required form. Furthermore, as we already observed, the condition that ElGamalCPASub1 be a closed, probabilistic polynomial-time function is also fulfilled. Thus, ElGamalCPASub1 just acts as a probabilistic polynomial-time decider for the DDH game.

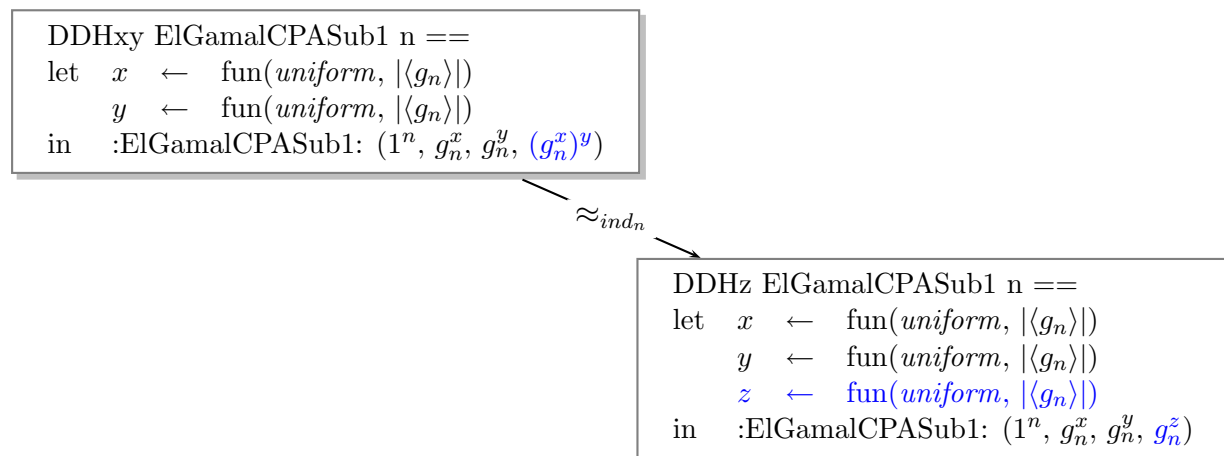


Figure 7.9: Application of the DDH assumption

This transformation is special in two ways. First, it is justified by the relation of computational indistinguishability, while all our other transformations are based on the observational equivalence. Since computational indistinguishability is a coarser relation than observational equivalence (cf. Section 3.6.4), this means that at the end, we will only be able to state that our initial and final games, namely IND_CPA1 and IND_CPA2 (with the appropriate parameters), are computationally indistinguishable. It is not surprising that this is the case. Indeed, we cannot show that these two games are observationally equivalent, since this would imply that ElGamal would be information-theoretically secure, which it is not. Observational equivalence between such two games is a very strong statement that is only fulfilled by encryption schemes like the one-time pad.

Second, this transformation is only justified by an assumption, namely the DDH assumption. We do not show it. We know that this assumption is not true in general (i.e., for any cyclic group), but the assumption just says that we work with some sequence of cyclic groups such that the games are computationally indistinguishable. If someone discovered a solution for the DDH problem in general, the IND_CPA security of ElGamal would break down. Furthermore, since this would also imply that this transformation is not valid, this proof would rely on a false assumption and thus would not be sensible. This is what we would expect, since we know that ElGamal yields indistinguishable encryptions only under the DDH assumption.

7.4 DDHz to RandomElGamalGame

We have applied the DDH assumption to our program. Now, it seems pretty clear that no adversary A' could discern which message, m_1 or m_2 , we used for encryption. It gets the value $m_1 \times g_n^z$ as a parameter, but since z is sampled uniformly at random from the set of exponents $[0, \dots, |G_n| - 1]$, the value g_n^z is distributed uniformly at random in the cyclic

group G_n . Hence, so is $m_1 \times g_n^z$, if m_1 is some variable bound to a value in G_n , which is a restriction that we imposed on the adversary A .

To show this, we would like to transform this game into an observationally equivalent game where m_1 is not even used for the encryption any more, i.e. where, instead of computing the value $m_1 \times g_n^z$, we just compute g_n^z . Since both look like values sampled uniformly at random from G_n to any outside observer who does not know z , intuitively this should not make any difference. Indeed, the variable z is only used once in the computation of the ciphertext, and nowhere else. In particular, the functions A and A' are closed and thus cannot access z . Furthermore, since we want to keep all of our transformations as small and as general as possible, we will now do the following. First, in several steps, we will transform our game such that a subterm will look like the program which we saw in Section 6.3.1. We will then argue that these two programs are observationally equivalent under a certain invariant, namely that m_1 only gets instantiated with values in G_n .

First, we just “reintegrate” the function `ElGamalCPASub1` back into the main program. This step is analogous to the last step before we applied the DDH assumption, but in the other direction. It is valid by Theorem 6.5 (β -equivalence).

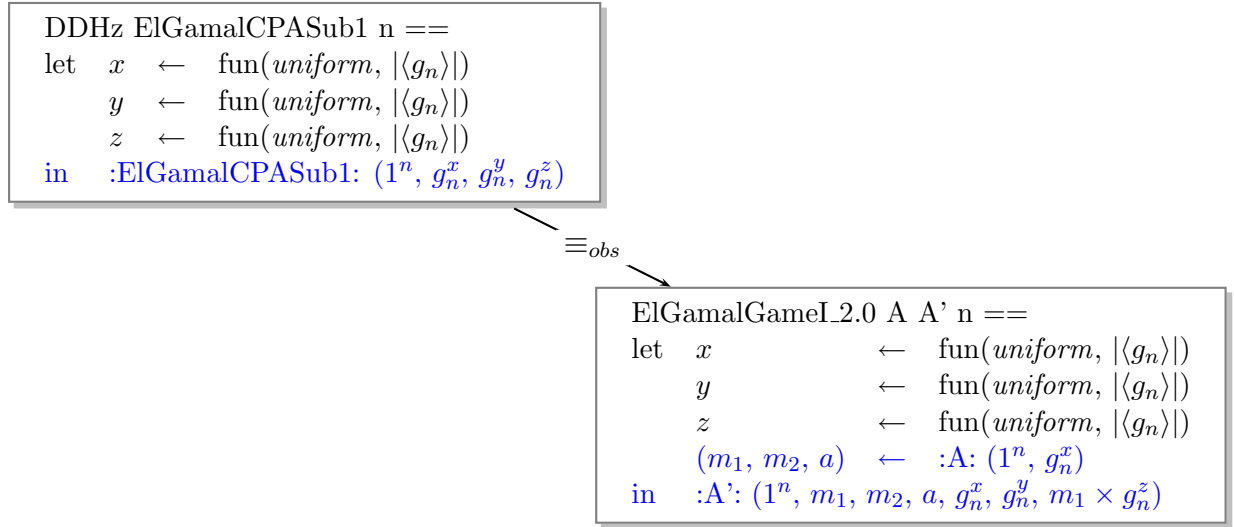


Figure 7.10: Reintegrating the DDH subroutine

To argue about the value $m_1 \times g_n^z$, we first use expression subsumption to introduce a new, fresh variable c' that is assigned the value $m_1 \times g_n^z$, and we use this variable whenever we encounter the value $m_1 \times g_n^z$ in the remaining of the program. The validity of this transformation follows from Theorem 6.7 (expression propagation) and is explained in more detail in Section 6.1.3 (expression subsumption). Note that the name c' is arbitrary here, since we are actually dealing with nameless terms. The reason why we choose this name is that it matches our intuition that this variable corresponds to the second part of the ciphertext $c = (i, c')$.

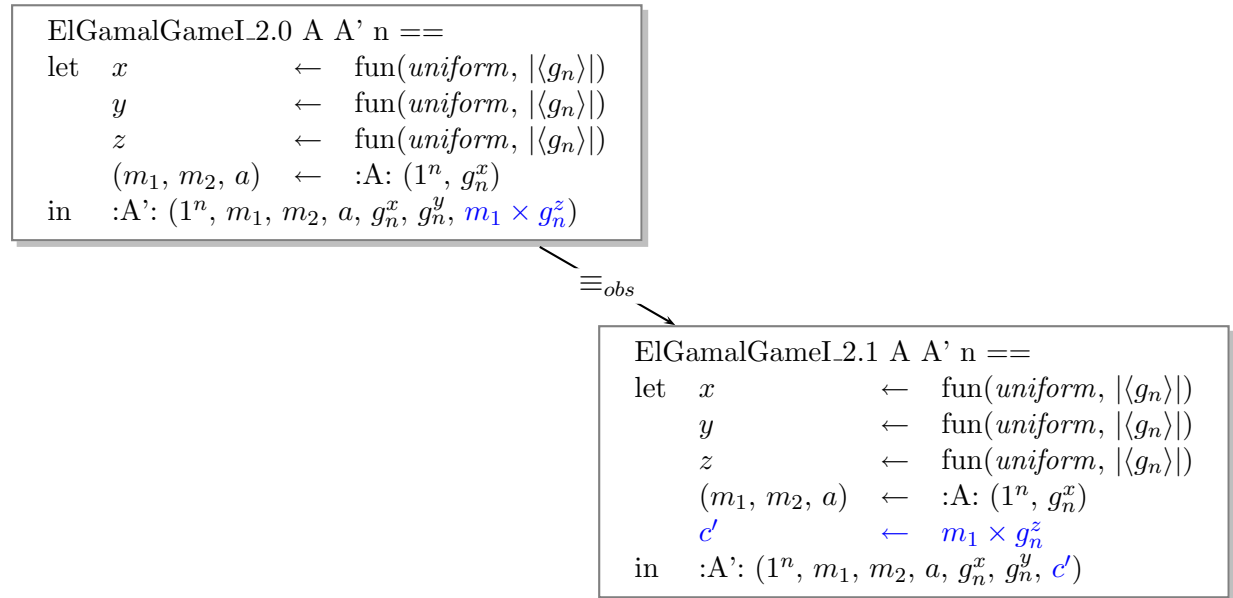


Figure 7.11: Expression subsumption of $m_1 \times g_n^z$

We observe that the reason why the expression $m_1 \times g_n^z$ looks like a value sampled uniformly at random from G_n is twofold. First, m_1 is a variable bound to a value in G_n , as it is bound to the λ which corresponds to the return value of A . But we required in Section 5.4 that A be such that the values m_1 and m_2 which it returns are values in G_n . Second, the value g_n^z is distributed uniformly at random in G_n . Ultimately, this is the case because z is a value distributed uniformly at random in $[0, \dots, |G_n| - 1]$, the set of exponents of G_n . Hence the line that samples z from $[0, \dots, |G_n| - 1]$ is, intuitively, important to make a statement about the value $m_1 \times g_n^z$. We swap two lines in our program to bring these two lines next to each other. The validity of this transformation follows by Theorem 6.17 (line swapping).

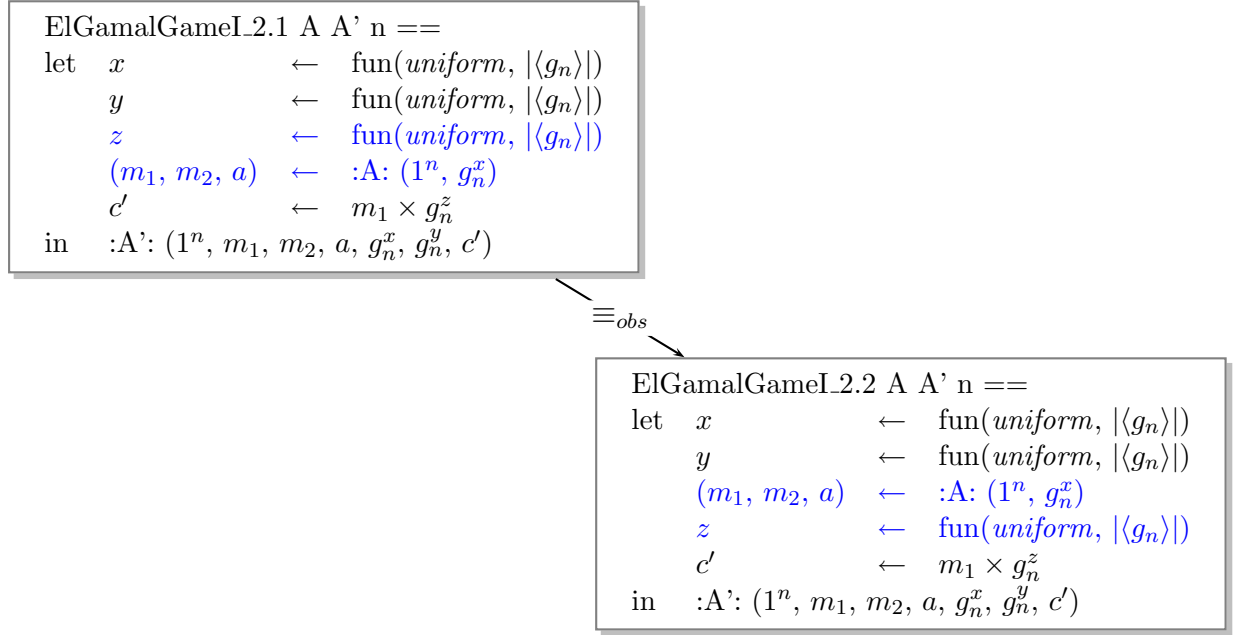


Figure 7.12: Swapping lines

Now, only one step remains to obtain a form which matches the program discussed in Section 6.3.1. We are going to chain the nested let construct of the variable z into the computation of c' . We refer to this transformation as chaining nested applications, and its validity holds by Theorem 6.10. The condition that z is not used in the remaining of the program is fulfilled. In the new form of the program, this becomes obvious.

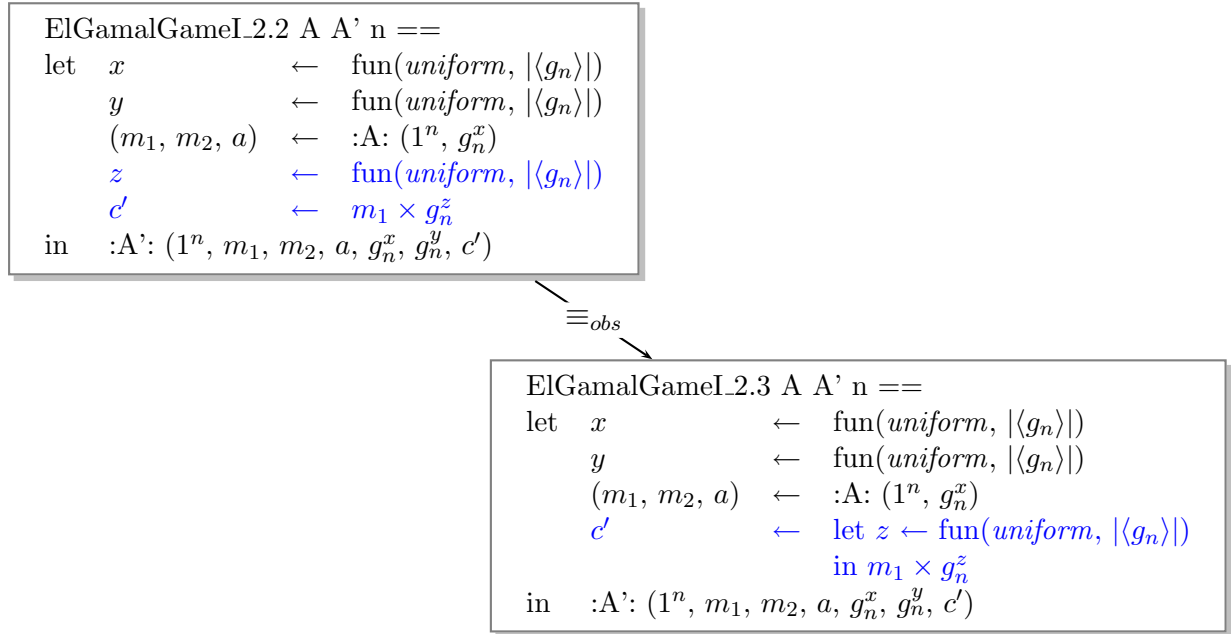


Figure 7.13: Chaining a nested application

Finally, we have the program in the form that we want it in. Recall that we showed in Theorem 6.20 that if m_1 is a value in G_n , then the two highlighted subterms in Figure 7.14 are denotationally equivalent. Since the function g and the security parameter n are regarded as constants within the program term, and the variable z is bound, the terms would be closed, and hence, observationally equivalent. However, here m_1 is not a value in G_n , but rather a free variable bound by the context. For this reason, it is impossible to show the observational equivalence of the two highlighted subterms, simply because it does not hold true. This is because to show the observational equivalence of two programs, we have to consider all possible instantiations of free variables. Here in particular we have to consider all possible instantiations of the variable m_1 . But the observational equivalence only holds true if we instantiate m_1 only with values that are elements of G_n , which is indeed the case here as we required this of A . As discussed in Section 6.3.1, here we need an extended version of the CIU Theorem, which allows us to impose certain invariants. More precisely, here we need the invariant that m_1 only gets instantiated with values from G_n . As mentioned in said section, such an extended version of the CIU Theorem is being worked on at the chair of Michael Backes. Here, we just assume that under this invariant, the two program terms are observationally equivalent. We write \equiv_{obs}^{INV} to denote this kind of equivalence. By this, we can eliminate the multiplication with the constant m_1 in the computation of c' .

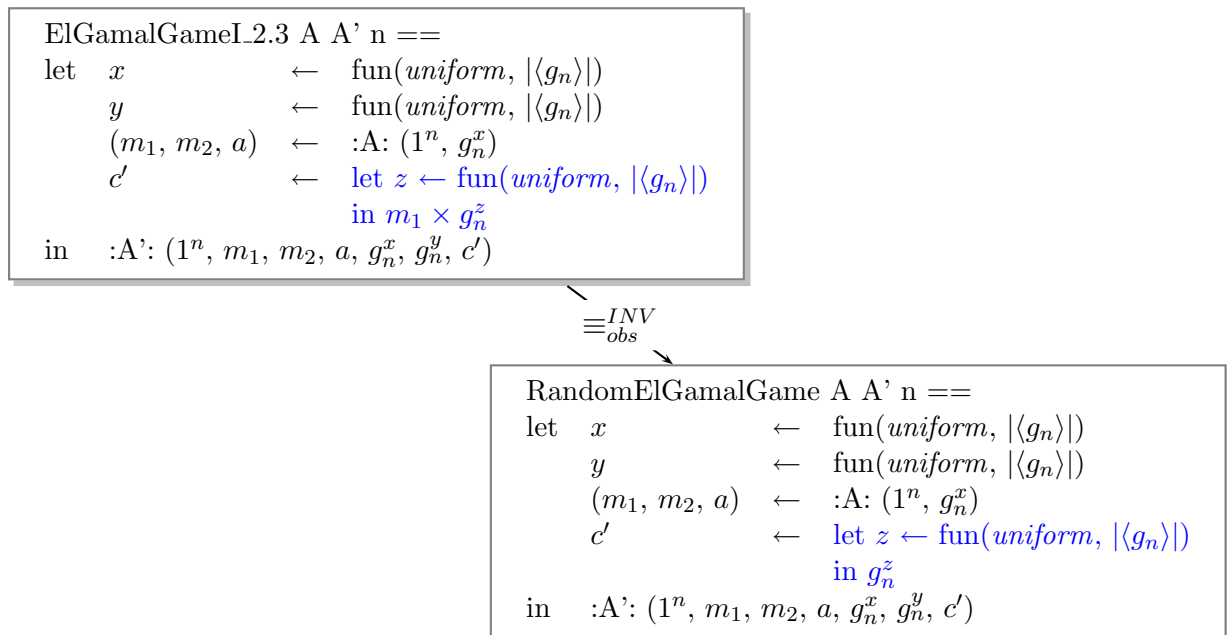


Figure 7.14: Multiplying with random elements in cyclic groups

This was the last transformation, and we are done. We have transformed the initial game into a game where m_1 is not even used in the computation of the ciphertext anymore. To do this we had to rely on the DDH assumption, which stated that one of our transformations involved two programs that are (assumed to be) computationally indistinguishable.

7.5 Finishing the Proof

By looking at all the transformations described in this chapter, it becomes clear that we can perform the very same steps on the program $\text{IND_CPA2 ElGamal } A' n$. The difference is that we would encrypt the message m_2 where in the games described in this chapter we encrypted m_1 . Hence we could get rid of this message in the computation of the ciphertext in the same manner as we did for m_1 . Thus, both of the games $\text{IND_CPA1 ElGamal } A' n$ and $\text{IND_CPA2 ElGamal } A' n$ are computationally indistinguishable from $\text{RandomElGamalGame } A' n$. By the transitivity of computational indistinguishability (see Theorem 3.37), we conclude that $\text{IND_CPA1 ElGamal } A' n$ and $\text{IND_CPA2 ElGamal } A' n$ are also computationally indistinguishable from each other, which is what we wanted to show. The following figure clarifies this idea.

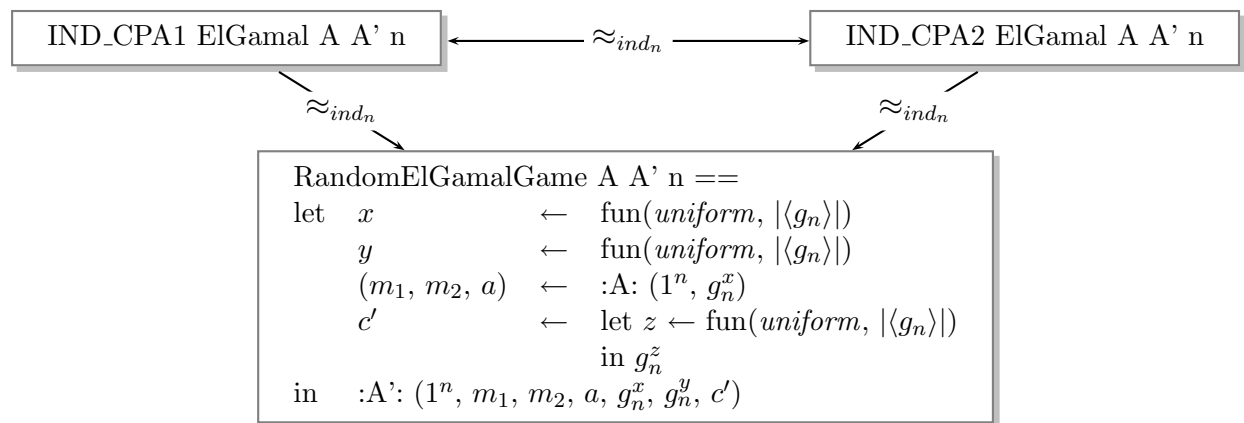


Figure 7.15: Result: $\text{IND_CPA1 ElGamal } A' n \approx_{ind_n} \text{IND_CPA2 ElGamal } A' n$

8 Conclusions

As discussed in Section 5.1, the fact that the two CPA games are computationally indistinguishable from each other yields the conclusion that the ElGamal encryption scheme has indistinguishable encryptions under chosen-plaintext attack assuming the DDH problem is hard in certain cyclic groups, which was the main goal of this thesis. We learned about the cryptographic language that has been implemented on top of Isabelle/HOL, how to formalize program relations written in this language, and made several important properties of these relations explicit. We recalled some basic cryptographic definitions and showed how to formalize these definitions in the language. We formalized several important and useful game transformations and proved their validity. Finally we showed how to transform the CPA games for ElGamal to prove their computational indistinguishability under the DDH assumption, and we explained how and why these transformations are valid and on which program relations they were based. This thesis demonstrates the usefulness of the language to model cryptographic constructions, as well as the ability to formally state and prove game-based transformations. It was the first application of this new language to a well-known real-world encryption scheme and hopefully will serve as a base for many more such applications.

Future Work

This thesis describes how to verify the security of the ElGamal encryption scheme in our new probabilistic λ -calculus for cryptographic proofs. This language has been implemented on top of the proof assistant Isabelle/HOL, and clearly the next step would be to prove all the statements of this thesis with it. This is a challenging task, albeit a lot of work has already been done. The language, the special operators and the relations we considered in Chapter 3 have already been defined in Isabelle/HOL. However some of the properties, like the transitivity of computational indistinguishability or the relation between observational equivalence and computational indistinguishability still have to be made explicit. While this may be less difficult, what should prove to be quite challenging is to verify all the game transformations from Chapter 6 in Isabelle/HOL. For instance, in the case of line swapping, while the result that L-renaming equivalence implies observational equivalence has already been performed on paper, it has not yet been shown in Isabelle/HOL. The expression propagation uses a pretty involved proof too that may be tedious to verify in a proof assistant. The proof for the multiplication with random elements in cyclic groups will also be difficult, especially as the measure theory which is used (see [21]) does not yet support such complex calculations, so it would need to be extended first. Furthermore

the observational equivalence with invariants would have to be implemented. Then the formalizations of the games from Chapter 5 would have to be defined, and the game transformations from Chapter 7 would have to be verified. This also involves intricate technical details. For instance, proving that the games are indeed all probabilistic-polynomial time (to reason about computational indistinguishability) is hard to verify formally, and for the time being would first have to be used as an additional assumption.

Thus, a lot of work remains to do, but already a lot of significant steps have been achieved. Using a proof assistant can increase our trust in proofs, but some extra effort is involved. A difficulty in designing such a framework is to design it in such a way as to make this extra effort not exceedingly cumbersome. For this, it is helpful to have a huge database of small lemmas which take care of many of the technical details, an elaborate theory of program equivalences, and many transformations that can be reused over and over. The research on this field is an interesting and state-of-the-art topic, and it will be fascinating to see what future developments are awaiting us.

References

- [1] Michael Backes. Cryptography, lecture notes of the core theory lecture. <http://www.infsec.cs.uni-sb.de/teaching/SS08/Cryptography/>, 2008. Universität des Saarlandes.
- [2] Michael Backes, Matthias Berg, and Dominique Unruh. A formal language for cryptographic pseudocode. In *LPAR*, pages 353–376, November 2008.
- [3] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt. In *5th International Workshop on Formal Aspects in Security and Trust, FAST 2008*, volume 5491 of *Lectures Notes in Computer Science*, pages 1–19. Springer, 2009.
- [4] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 90–101, New York, NY, USA, 2009. ACM.
- [5] Heinz Bauer. *Probability theory*. Walter de Gruyter & Co., Hawthorne, NJ, USA, 1996. Translator-Burckel, Robert B.
- [6] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. In *Advances in Cryptology EUROCRYPT 2006*. Springer-Verlag, 2006.
- [7] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Trans. Dependable Secur. Comput.*, 5(4):193–207, 2008.
- [8] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 2006.
- [9] Johannes Buchmann. *Introduction to Cryptography*. Springer, 2002.
- [10] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *Amer. J. Math.*, 58:345–363, 1936.
- [11] Judicaël Courant, Marion Daubignard, Cristian Ene, Pascal Lafourcade, and Yassine Lakhnech. Towards automated proofs for asymmetric encryption schemes in the random oracle model. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 371–380, New York, NY, USA, 2008. ACM.

- [12] Jonathan Driedger. Formalization of Game-Transformations, Jan 2010. Bachelor's Thesis.
- [13] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
- [14] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed: (functional pearl). In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 221–231, New York, NY, USA, 2000. ACM.
- [15] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, New York, NY, USA, 2000.
- [16] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.
- [17] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. <http://eprint.iacr.org/>.
- [18] Paul R. Halmos. *Measure theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1978. Volume 18 of *Graduate Texts in Mathematics*.
- [19] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [20] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
- [21] Stefan Richter. Formalizing integration theory, with an application to probabilistic algorithms, 2003.
- [22] Gert Smolka. *Programmierung – eine Einführung in die Informatik mit Standard ML*. R. Oldenbourg, München, Germany, 2008.
- [23] Coq D. Team. The Coq proof assistant reference manual, version 8.2, August 2009.
- [24] S. Zanella Béguelin, B. Grégoire, G. Barthe, and F. Olmedo. Formally certifying the security of digital signature schemes. In *30th IEEE Symposium on Security and Privacy, S&P 2009*, 2009. To appear.