Saarland University

Faculty of Natural Sciences and Technology I

Department of Computer Science

Formal Verification of Cryptographic Security Proofs

Matthias Berg

Dissertation

zur Erlangung des Grades
des Doktors der Naturwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Saarbrücken, 2013

Tag des Kolloquiums:	9. August 2013
Dekan:	Prof. Dr. Mark Groves

Prüfungsausschuss

Vorsitzender:	Prof. Dr. Sebastian Hack
Berichterstattende:	Prof. Dr. Michael Backes
	Prof. Dr. Christian Hammer
akademischer Mitarbeiter:	Dr. Dario Fiore

Zusammenfassung

Kryptographische Sicherheitsbeweise manuell zu überprüfen ist mühsam und fehleranfällig. Spielbasierte Beweistechniken ermöglichen einen modularen Beweisaufbau, wobei kryptographische Programme – sog. Spiele – schrittweise so modifiziert werden, dass jeder Schritt einzeln überprüfbar ist. Dieser code-basierte Ansatz erlaubt die computergestützte Verifikation solcher Beweise.

Im ersten Teil dieser Dissertation präsentieren wir Verypto: ein System zur computergestützten formalen Verifikation spielbasierter kryptographischer Sicherheitsbeweise. Auf Grundlage des Theorembeweisers Isabelle entwickelt, bietet Verypto eine formale Sprache, mit der sich kryptographische Eigenheiten wie probabilistisches Verhalten, Orakelzugriffe und polynomielle Laufzeit ausdrücken lassen. Wir beweisen die Korrektheit verschiedener Spieltransformationen und belegen deren Anwendbarkeit durch die Verifikation von Beispielen: Wir zeigen, dass Kompositionen von 1-1 Einwegfunktionen auch einweg sind und dass die ElGamal Verschlüsselung IND-CPA sicher ist.

In einem ähnlichen Projekt entwickelten Barthe et al. das EasyCrypt System, welches Spieltransformationen mit Methoden der Programmanalyse validiert. Im zweiten Teil dieser Dissertation verwenden wir EasyCrypt und verifizieren die Sicherheit der Merkle-Damgård Konstruktion – ein Designprinzip, das vielen Hashfunktionen zugrunde liegt. Wir zeigen die Kollisionsresistenz der Konstruktion und verifizieren, dass sie sich wie ein Zufallsorakel verhält.

Abstract

Verifying cryptographic security proofs manually is inherently tedious and error-prone. The game-playing technique for cryptographic proofs advocates a modular proof design where cryptographic programs called games are transformed stepwise such that each step can be analyzed individually. This code-based approach has rendered the formal verification of such proofs using mechanized tools feasible.

In the first part of this dissertation we present Verypto: a framework to formally verify game-based cryptographic security proofs in a machine-assisted manner. Verypto has been implemented in the Isabelle proof assistant and provides a formal language to specify the constructs occurring in typical cryptographic games, including probabilistic behavior, the usage of oracles, and polynomial-time programs. We have verified the correctness of several game transformations and demonstrate their applicability by verifying that the composition of 1-1 one-way functions is one-way and by verifying the IND-CPA security of the ElGamal encryption scheme.

In a related project Barthe et al. developed the EasyCrypt toolset, which employs techniques from automated program verification to validate game transformations. In the second part of this dissertation we use EasyCrypt to verify the security of the Merkle-Damgård construction – a general design principle underlying many hash functions. In particular we verify its collision resistance and prove that it is indifferentiable from a random oracle.

Background of this Dissertation

This dissertation builds on the following papers that I co-authored as well as the following bachelor's and master's theses that were conducted under my guidance. I contributed to the elaboration of all of them.

- Michael Backes, Matthias Berg, and Dominique Unruh. A formal language for cryptographic pseudocode. In *Logic for Programming, Artificial Intelligence, and Reasoning – LPAR 2008*, volume 5330 of *Lecture Notes in Computer Science*, pages 353–376. Springer, 2008.
- Jonathan Driedger. Formalization of game-transformations. Bachelor's thesis, Saarland University, 2010.
- Malte Skoruppa. Formal verification of ElGamal encryption using a probabilistic lambda-calculus. Bachelor's thesis, Saarland University, 2010.
- Michael Backes, Gilles Barthe, Matthias Berg, Benjamin Grégoire, César Kunz, Malte Skoruppa, and Santiago Zanella Béguelin. Verified security of Merkle-Damgård. In *Computer Security Foundations Symposium – CSF 2012*, pages 354–368. IEEE, 2012.
- Malte Skoruppa. Verifiable security of prefix-free Merkle-Damgård. Master's thesis, Saarland University, 2012.

Acknowledgments

First of all, I wish to express my deep gratitude to Michael Backes. As my advisor, he introduced me to this exciting topic whose elaboration I truly enjoyed. He provided excellent guidance whenever necessary and created the inspiring working atmosphere that is present in his group.

I also want to thank Dominique Unruh for countless discussions and for answering my numerous questions. His support was indispensable to the development of our tool.

I also thank my students Jonathan Driedger and Malte Skoruppa for working out sample applications for our tool. Thank you Malte, for cleaning up the Isabelle code after me and for spending endless nights hacking in Coq.

Thanks go to my co-authors Gilles Barthe, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin for a fruitful collaboration. In particular I want to thank Gilles for inviting us to Madrid when we launched our project, and I thank Santiago for his profound advice during the project.

I wish to thank my former and current colleagues at Saarland University for a vibrant environment that made working here so inspiring. Here, my former office mate Cătălin Hriţcu often provided useful comments. Special thanks go to Bettina Balthasar for her dedication to managing all the bits and pieces that keep the group alive.

Besides work, I am grateful to my friends who always helped to clear my mind through sports and occasional distractions. Last but not least, I deeply thank my family for the steady support throughout my entire life. Thank you for everything!

Contents

Introduction

The security of cryptographic constructions such as encryption schemes, signatures, and hash functions is crucial for the confidentiality, authenticity, and integrity of data. However, the secure design of cryptographic constructions is inherently difficult to accomplish and many flaws in constructions were indeed only found long after they were deployed and believed to be correct [112, 90, 53].

Provable security [82] addresses this problem and advocates to use rigorous mathematical reasoning to infer security properties of cryptographic constructions. Here, constructions and their interaction with a possibly adverse, untrusted environment are explicitly modeled together with the pursued security properties. Proofs are typically conducted by means of a reduction, i.e., the task of breaking the security of a construction is reduced to another objective that is assumably hard to achieve. Such proofs provide solid guarantees on the security of a construction, as they show that the security merely depends on the complexity of – usually well-understood – concise problems. Nevertheless the proofs themselves tend to be quite involved and cumbersome. Often they span tens of pages containing intricate arguments about probability distributions, complexity theory, and the interaction between non-trivial algorithms. Completing such a proof is error-prone as one can easily overlook mistakes or fail to consider special cases [113, 77, 78].

The game-playing technique [42, 114] improves this situation by proposing a general design principle to structure cryptographic proofs. This technique follows a code-based approach where security properties are formulated in terms of probabilistic programs – so-called games. The security reduction in the proof is then performed by stepwise transforming these games into a final game that corresponds to the assumably hard problem. Here each step preserves the probabilistic behavior of the game or only changes it insignificantly, thereby allowing to relate the initial game to the final game in this sequence. The advantage of this technique is that each modification step in this sequence can be analyzed on its own, which enables a modular treatment of the proof.

However, this advantage is diminished by the style in which the games are expressed in practice. Mostly the games are described using words or some pseudo-code which lacks a formal semantics. While this approach helps structuring a proof, it also gives rise to errors resulting from impre-

cise, ambiguous formulations. However, formalizing every tiny detail is also problematic as this entails many mundane proof obligations that distract from the creative parts of the proof. Noticing this nuisance, Halevi [83] calls for the development of a computer-aided tool to take care of these mundane parts of cryptographic proofs. He argues that games should be expressed using a fully specified programming language so that a tool can validate the code modifications that the proof steps perform.

In the first part of this thesis we present our solution to Halevi's problem and introduce Verypto, a formal framework to express cryptographic security proofs. In Verypto, cryptographic games are expressed in a formal language, namely a probabilistic lambda calculus. This language aims at maximal generality in order to express all objects and reasoning patterns that occur in typical cryptographic proofs. It can handle probabilistic behavior, stateful higher-order objects such as oracles, arbitrary data types, and supports event-based reasoning patterns. The formal semantics of the language has been implemented in the proof assistant Isabelle/HOL [102] where we also developed techniques to reason about equivalences and relations between games. This enables us to formally specify security properties of cryptographic constructions and to employ game transformations to verify them. We demonstrate the feasibility of this approach in two case studies. First, we show that the self-composition of an injective one-way function yields another one-way function. Second, we verify that the ElGamal encryption scheme is IND-CPA secure.

In parallel to our development of Verypto another group pursued a similar goal of providing a formal framework for cryptographic proofs and developed CertiCrypt [29, 32], which has been implemented in the proof assistant Coq [63]. It follows a less general approach and sacrifices some expressiveness in comparison to our framework. E.g., it employs a simpler probability model and the language in which the games are formalized is not higher-order. This simpler approach resulted in a fast development of CertiCrypt and the creation of a subsequent framework called EasyCrypt [30], which automates the reasoning behind CertiCrypt using program verification tools.

In the second part of this thesis – motivated by the increased interest in hash security during the selection process of the new SHA-3 algorithm – we use EasyCrypt to formally verify security properties of the Merkle-Damgård construction – a general design principle underlying many hash functions. First, we establish its collision resistance, i.e., we verify that it is hard to find distinct messages that hash to the same value, provided that finding colliding inputs to the compression function of the construction is difficult. Second, assuming an ideal compression function, we verify that the Merkle-Damgård construction is indifferentiable from a random oracle. Indifferentiability is a strong security property which states that one cannot distinguish the construction from a random oracle, even if given access to internal components of the construction such as the employed compression function.

Chapter 1

Verypto - Formally Verifying Cryptographic Proofs

1.1 Introduction to Game-based Proofs

Cryptographic security properties are typically formulated in the form of so-called *games*. A game is a probabilistic process that models the potential interaction between an adversary and a cryptographic construction, and the output of the game is used to determine whether the adversary was successful in attacking the construction. Hence, by analyzing the probabilistic behavior of the game, we can deduce security properties of the construction. E.g., if we can show that the game allows no attacks, then the construction is secure with respect to the interaction that is modeled in the game.

A common technique to prove the security of a cryptographic construction is to start with an initial game that models the considered security property and then to modify this game in a sequence of steps such that each step introduces at most a negligible change in the probabilistic behavior of the game. This allows us to relate the behavior of the initial game to the behavior of the final game in this sequence. The goal of this so-called game-playing technique [42, 114] is to reach a final game that corresponds to a complexity hardness assumption or trivially allows no attacks and hence allows us to infer the security of the construction under consideration.

Proofs based on sequences of games have several advantages compared to direct proofs, because the game-based technique leads to well-structured proofs. In contrast to direct proofs, they follow a modular design, i.e., ideally, each step can be verified individually without having to reason about other steps in the sequence. This simplifies the detection of potential mistakes in the proof and helps to determine where these mistakes occur. Furthermore, since the individual steps are typically very simple and since the correctness

of each step can be proven independently, the game-based proof technique is ideally suited for the formal verification of cryptographic security proofs.

In practice, however, the advantages of game-based proofs are diminished by the following limitation: In cryptographic publications, the games are usually described in words, or at best in some ad-hoc pseudo-code. In both cases, no formal semantics of the language used are specified. This leads to several grave disadvantages:

Ambiguity of definitions. As cryptographic games are also used to define security notions, the lack of a formal semantics may result in different interpretations of the details of a given game, and hence in an ambiguous security definition. For example, if a subroutine representing an adversary is invoked twice, it might be unclear whether the adversary may keep state between these two invocations. The author of the security definition may explicitly point out these ambiguities and resolve them; this, however, assumes that the author is aware of all possible interpretations.

Mistakes may be hidden. An error in a proof step may be hard to identify: Since the correctness of a step usually depends on the precise definition of the games, the reader of the proof may not be sure whether the performed modification is indeed incorrect or whether the reader just misinterpreted the definition of the games. Moreover, it may happen that for a sequence of three games A, B, C, depending on the exact definition of B, either the step from A to B, or that from B to C is incorrect. However, if the steps are verified individually, in each case there is some interpretation of the meaning of B that lets the corresponding proof step seem correct.

Unsuited for machine-assisted verification. Finally, if we are interested in machine-assisted verification of cryptographic security proofs, the semantics of the games need to be defined precisely since a computer will not be able to reason about a pseudo-code without semantics.

1.2 Contribution of this Chapter

To overcome these limitations we present Verypto:[1] A formal framework for machine-assisted verification of game-based cryptographic security proofs. We have implemented Verypto in the proof assistant Isabelle/HOL [102]. It provides a language with a formal semantics to express cryptographic games, which yields the following properties:

Expressiveness. The language can express all constructs that usually occur in the specification of cryptographic games, including probabilistic behaviors, the usage of oracles, and potentially continuous probability mea-

[1] For this purpose, this dissertation builds on work published in [14] that I co-authored and on the bachelor's theses [74, 115] which were conducted under my guidance. I contributed to the elaboration of all of them.

sures for reasoning about, e.g., an infinitely long random tape for establishing information-theoretic security guarantees. From a language perspective, oracles are higher-order arguments that are passed to a program. Consequently we have implemented a higher-order functional probabilistic language to deal with sophisticated objects such as oracles. We chose a functional language, because such languages deal with higher-order objects much more naturally than imperative languages. A purely functional language, however, would have been insufficient, because functional reasoning would not allow oracles to keep state between their invocations (and passing on state as explicit inputs would result in secrecy violations, e.g., an oracle would output its secret key to the adversary). Therefore we have included ML-style references in the language. Finally, events constitute a common technique in game-based cryptographic proofs for identifying undesirable behavior. Thus we explicitly support events in the language. The semantics is operational in order to be able to introduce the notion of polynomial runtime.

Simplicity. The definition of the language is kept as simple as possible so that researchers without a strong background in the theory of programming languages can understand at least its intuitive meaning. In particular, the language has a syntax that is readable without a detailed introduction into the language.

Mechanization. We have implemented the language in the proof assistant Isabelle/HOL. Therefore we can use the power of Isabelle's logic to reason about the language itself and to formally verify cryptographic proofs that are written in this language. Moreover, we have formalized several common relations between games in Isabelle/HOL, such as denotational equivalence, observational equivalence, and computational indistinguishability. Using these relations we have verified a set of game transformations with which one can modify games in order to perform a step from one game to another.

Hence using Verypto, one can express game-based cryptographic security proofs in a readable form and formally verify their correctness in a machine-assisted manner. To illustrate Verypto's applicability we have used it to verify the following examples:

We have used Verypto to verify that the composition of a 1-1 one-way function with itself yields another one-way function. This proof heavily employs a game transformation that propagates terms in the language and thereby can move code that is bound to a variable to the site where this variable is used.

Moreover we have used Verypto to verify the IND-CPA security of the ElGamal encryption scheme. This proof uses a combination of game transformations dealing with the reordering of program code, the inlining of code of sub-programs, as well as specialized transformations to model the deci-

sional Diffie-Hellman assumption and to establish properties of randomly selected group elements.

1.3 Design Decisions for Verypto

We strive for defining a language that is powerful enough to express and reason about the constructions and the definitions that are used in cryptography. Since cryptography heavily relies on the use of probabilism, the language necessarily needs to be probabilistic. Moreover, the language should not be restricted to discrete probability measures, since discrete probability measures are not sufficient for reasoning about infinite constructions such as the random selection of an infinitely long random tape, e.g., to reason about information-theoretic security guarantees. Oracles, i.e., objects that can be queried in order to perform tasks such as computing the hash of some string or computing the decryption of some ciphertext, constitute another salient concept in modern cryptography. From a language perspective, oracles are higher-order arguments that are passed to a program. We hence strive for a higher-order functional language – higher-order to deal with sophisticated objects such as oracles, functional since functional languages deal with higher-order objects more naturally than imperative languages.

A purely functional language, however, is insufficient, because functional reasoning would, e.g., not allow oracles to keep state between its invocations. While one can rewrite every program that uses state into a purely functional, equivalent program without state by passing the state around as an explicit object, this approach fails in our setting because it inadequately deals with secrecy properties: Consider an adversary that accesses a decryption oracle which, upon its first query, generates a secret key and subsequently decrypts the queries obtained from the adversary using that key. A purely functional setting with an explicit encoding of state would cause the oracle to return its state, i.e., its secret key, and the adversary had to additionally provide the key in its next query to the oracle. Clearly, this violates the intended secrecy of the key, and there is moreover not even a guarantee that the adversary provides the same, correct state in all of its queries. We remedy this problem by including ML-style references in the language.

To make the language efficiently usable from a programmer's perspective, we need a way to express data structures like lists and trees. Since we do not want to commit ourselves to specific data structures, we include mechanisms that are sufficient for the programmer to define his own type constructors. More precisely, the language has an iso-recursive type system [108] that includes product and sum types. This is enough to express arbitrary data types.

The use of events is a common technique in game-based cryptographic proofs. A game raises an event whenever some – usually unwanted – con-

dition occurs. One can then use this event to identify program branches that show this unwanted condition and transform the game based on the probability that the event occurs. For conveniently reflecting this reasoning, we explicitly include events in the language.

As a result of these design choices that we presented above, we have equipped Verypto with a probabilistic higher-order functional language with references, iso-recursive types, and events. By this Verypto is able to express the constructs that typically occur in cryptographic specifications and security proofs.

1.4 Background on Isabelle/HOL

Isabelle [102] is an interactive proof assistant, i.e., a framework in which one can state mathematical formulas and then interactively manipulate the formulas in order to eventually prove their validity. Since all manipulations are checked by Isabelle, we obtain exceptionally high guarantees in the correctness of such proofs. Isabelle consists of a small core that is used to check the manipulations and of a large tool set from which one can draw in order to perform such manipulations. Since incorrect manipulations are rejected by the core, only the relatively small code base of the core has to be trusted.

Isabelle is generic in the sense that it can handle different object logics. For our purpose we employ Isabelle/HOL, which is Isabelle in the setting of *higher-order logic*. See Section 1.4.1 below for a brief introduction to higher-order logic. Furthermore, since our formalization of Verypto extensively builds upon concepts from measure theory, we base our implementation on the formalization given in [109], which provides basic concepts from measure theory including the Lebesgue integral.

The formalization of Verypto in Isabelle/HOL consists of approximately 40000 lines of code. A selection of important definitions, lemmas, and theorems can be found in Appendix I. In order to enable the reader to relate the content of this chapter to its formal development in Appendix I, we annotate definitions and statements with a reference to their corresponding formalization in Isabelle.

Disclaimer. Even though formally verified proofs provide high guarantees on their correctness, Isabelle provides mechanisms to selectively skip the verification of parts of a proof. While this mechanism may introduce errors in a proof, it also allows for a faster development, because trivial but cumbersome proof steps can be skipped.

Since Verypto builds upon measure theory, proofs in our development often require us to establish the measurability of the involved mathematical objects. We have regularly skipped the verification of measurability in order to continue with the more interesting parts in our development. Even though

these gaps in the proofs may give rise to errors in our formalization, we believe that this problem can be safely ignored. Most mathematical objects are indeed measurable and it is unlikely to construct non-measurable objects by accident.

1.4.1 Higher-Order Logic

In (simply typed) higher-order logic (HOL) every value in a statement has to be assigned some type. Types may be elementary types (e.g., Booleans, integers) or higher-order types. Examples of higher-order types are $\alpha\,\mathsf{set}$, which denotes the type of sets of elements of type α (where α may again be an elementary or higher-order type), $\alpha \to \beta$, which is the type of all functions taking values of type α to values of type β, or $\alpha \times \beta$, which is the type of all pairs in which the first component has type α and the second has type β. For example, in the expression $1 \in \mathbb{N}$, we have that 1 has type nat (naturals), $\mathbb{N}$ has type $\mathsf{nat\,set}$, and $\in$ is a function of type $\mathsf{nat} \to \mathsf{nat\,set} \to \mathsf{bool}$ (written in infix notation). An example for a statement that does not typecheck is $M \in M$, as $\in$ necessarily has type $\alpha \to \alpha\,\mathsf{set} \to \mathsf{bool}$ for some α, and thus M needs to have types α and $\alpha\,\mathsf{set}$ simultaneously for some α. Major advantages of HOL are its quite simple logic as well as the possibility to automatically infer types of expressions. Statements written in HOL are usually shorter and easier to read than their counterparts in untyped logics. HOL also allows to state and prove theorems in a polymorphic way: Instead of giving a concrete type to each variable, we can use type variables $\alpha, \beta, \ldots$ in the statement of the theorem; the intended meaning is that this theorem holds for any instantiation of these type variables with concrete types.

An addition to the type system of HOL there are also Haskell-style type classes implemented in Isabelle/HOL. A type class introduces constants and constraints that a type has to satisfy. For example, a type class $\mathsf{semigroup}$ might require that for a type α, there is a constant $\circ$ of type $\alpha \to \alpha \to \alpha$ and it holds that $\circ$ fulfills the semigroup axioms. A type α that fulfills the constraints of a type class is called an instance of that type class. Namely, the type nat is an instance of the type class $\mathsf{semigroup}$. The advantage of type classes comes into play when considering polymorphic statements. The type variables in these statements can then be restricted to a given type class; the statement is then expected to hold for all instantiations of the type variables that satisfy the constraints of the type class. Consider the statement $x \circ (x \circ x) = (x \circ x) \circ x$, where x may have any type α. In general, this statement is wrong as there are types in which $\circ$ is not associative. If we restrict α to the type class $\mathsf{semigroup}$, however, the statement becomes true. The main advantage of type classes is that important side conditions (as being a semigroup) can be captured automatically using type inference and do not need to be stated explicitly, leading to shorter and more readable statements.

1.5 Mathematical Background

1.5.1 Notation

Let $\mathbb{N}$ be the natural numbers including 0, $\mathbb{R}$ the real numbers, $\mathbb{R}^+$ the non-negative real numbers including 0, $\mathbb{B}$ the set of Booleans, and let $\mathbb{U}$ denote the unit set containing the single element *unit*. The powerset of X is denoted by 2^X. $X \times Y$ is the Cartesian product of X and Y, and $X \to Y$ is the function space from X to Y. For a set Y, we write $f^{-1}(Y)$ for $\{x \mid f(x) \in Y\}$. By $\lambda x.\, p(x)$ we denote the anonymous function mapping x to $p(x)$. We write $[a_1, \ldots, a_n]$ for lists (finite sequences) and [] for the empty list. Given lists a, b, by $a@b$ we denote the concatenation of a and b. By $x::a$ we denote the list a with the element x prepended, and $|a|$ denotes the length of list a. Given a list a with $|a| > n$, we write a_n for the n-th element (counting from 0) in a and $a[n := x]$ for replacing the n-th element in a with x.

We further summarize the notation introduced in Section 1.5.2 below: We write Σ_X for the canonical σ-algebra over X and $A^{\complement}$ for the corresponding set complement $X \setminus A$. With $f(\mu)$ we denote the projection of distribution μ using function f, and $\forall x \leftarrow \mu.\, P(x)$ denotes the fact that $P(x)$ holds almost surely with respect to the distribution μ. The unit kernel is written as δ, where $\delta(x)$ is the Dirac measure of x. $L \circ K$ denotes the composition of two kernels, and $L \cdot \mu$ denotes the application of kernel L to distribution μ. The kernel $\downarrow_E$ is used to restrict distributions to the event E and $\overline{f}$ is the deterministic kernel constructed from function f.

1.5.2 A Primer in Probability theory

We give a compact overview of measures and probability theory in this section. For a detailed overview, we refer to a standard textbook on probability theory, e.g., [84]. A *σ-algebra* over a set X is a set $\Sigma_X \subseteq 2^X$ such that the empty set $\emptyset \in \Sigma_X$, and $A \in \Sigma_X \Rightarrow (A^{\complement}) \in \Sigma_X$, and for any sequence $(A_i)_{i \in \mathbb{N}}$ with $A_i \in \Sigma_X$ we have that $\bigcup_i A_i \in \Sigma_X$. The smallest σ-algebra containing all $A \in G$ for some set G is called the σ-algebra generated by G. We assume that there is some canonical σ-algebra Σ_X associated with each set X and call the sets $A \in \Sigma_X$ *measurable sets*. For countable X, $\Sigma_X = 2^X$. $\Sigma_{\mathbb{R}}$ is generated by all intervals $[a, b]$ with $a, b \in \mathbb{R}$ (Borel-algebra); $\Sigma_{\mathbb{R}^+}$ is defined analogously. We assume that $\Sigma_{X \times Y}$ is generated by all $A \times B$ with $A \in \Sigma_X$, $B \in \Sigma_Y$. $\Sigma_{X \to Y}$ is generated by all sets $\{f \mid f(x) \in A\}$ for $x \in X$, $A \in \Sigma_Y$. I.1, p.109 I.8, p.109 I.9, p.109 I.12, p.110

A *measure* over X is a function $\mu : \Sigma_X \to \mathbb{R}^+$ with the following properties: $\mu(\emptyset) = 0$ and for a pairwise disjoint sequence $(A_i)_{i \in \mathbb{N}}$ with $A_i \in \Sigma_X$, we have $\mu(\bigcup_i A_i) = \sum_i \mu(A_i)$ (countable additivity). We call μ a *subprobability measure* if $\mu(X) \leq 1$. Intuitively, $\mu(E)$ denotes the probability that some value $x \in E$ is chosen. A subprobability measure can be used to model the output of a program that does not terminate with probability 1; then I.14, p.110

$\mu(X)$ is the probability of termination. Abusing notation, we write 0 for the measure $\lambda E.\, 0$. The words measure and *distribution* are synonyms.

I.15, p.110 A function $X \to Y$ is called *measurable* if for all $E \in \Sigma_Y$, we have $f^{-1}(E) \in \Sigma_X$. We can apply a measurable function f to a measure μ by
I.22, p.110 defining $f(\mu)(E) \stackrel{\text{def}}{=} \mu(f^{-1}(E))$. Intuitively, $f(\mu)$ is the distribution of $f(x)$ when x is chosen according to μ.

If μ is a measure over X and $P : X \to \mathbb{B}$ is a measurable function, we
I.26, p.110 write $\forall x \leftarrow \mu.\, P(x)$ to denote $\exists A \in \Sigma_X.\, \mu(A^{\complement}) = 0 \wedge \forall x \in A.\, P(x)$. In this case we also say that $P(x)$ holds *for almost all* $x \in \mu$. Intuitively, this means a value x sampled according to μ satisfies $P(x)$.

For any measurable function $f : X \to \mathbb{R}$ and any distribution μ over
I.42, p.111 X, we write $\int f(x)\, d\mu(x)$ for the *Lebesgue integral*. Intuitively, $\int f(x)\, d\mu(x)$ is the expected value of $f(x)$ if x is chosen according to the distribution μ. The integral over a measurable set A is written as $\int_A f(x)\, d\mu(x)$. We refer to [84] for the formal definition of the integral. If f is bounded from below and above, then the integral always exists (and is finite).

A function $K : X \to (\Sigma_Y \to \mathbb{R}^+)$ is called a *kernel* from X to Y, if for all $x \in X$ we have that $K(x)$ is a measure and if for all $E \in \Sigma_Y$ we have that the function $K_E : X \to \mathbb{R}^+$, $K_E(x) \stackrel{\text{def}}{=} K(x)(E)$ is measurable. See [39] for a detailed introduction to kernels. Intuitively, a kernel is a *probabilistic function* from X to Y that assigns every input $x \in X$ a distribution over
I.31, p.111 outputs in Y. We call K a *submarkov kernel* if $K(x)$ is a subprobability measure for all $x \in X$. The *unit kernel* δ is defined as $\delta(x)(E) \stackrel{\text{def}}{=} 1$ if $x \in$
I.38, p.111 E, and $\delta(x)(E) \stackrel{\text{def}}{=} 0$ otherwise. The measure $\delta(x)$ is also called the *Dirac measure* of x, which assigns probability 1 to x. We say a kernel K is *invariant* on V if for all $v \in V$, $K(v) = \delta(v)$. Given a set V, we define the *restriction*
I.41, p.111 *kernel* $\downarrow_V$ by $\downarrow_V(x) \stackrel{\text{def}}{=} \delta(x)$ if $x \in V$ and $\downarrow_V(x) \stackrel{\text{def}}{=} 0$ if $x \notin V$. Given a function
I.55, p.112 $f : X \to Y$, the *deterministic kernel* $\overline{f}$ is defined as $\overline{f}(x) \stackrel{\text{def}}{=} \delta(fx)$.

Given a kernel L from Y to Z and a measure μ on Y, we can apply
I.43, p.112 L to μ by defining $(L \cdot \mu)(E) \stackrel{\text{def}}{=} \int L(x)(E) d\mu(x)$. Intuitively, $L \cdot \mu$ is the distribution resulting from choosing x according to μ and then applying the probabilistic function L. E.g, the application $\downarrow_V \cdot \mu$ sets the probability of
I.52, p.112 all events outside V to 0, i.e., $(\downarrow_V \cdot \mu)(A) = \mu(A \cap V)$. Given a kernel K from X to Y, we can define the *(kernel) composition* $L \circ K$ by $(L \circ K)(x) \stackrel{\text{def}}{=}$
I.44, p.112 $L \cdot (K(x))$. If L and K are submarkov kernels, then so is their composition $L \circ K$.

I.24, p.110 For distributions μ, ν, we write $\mu \leq \nu$ iff for all E, $\mu(E) \leq \nu(E)$ (i.e., $\leq$
I.36, p.111 is defined pointwise). For kernels, $\leq$ is also defined pointwise.

1.5.3 Limits of Kernels

In this section we will investigate properties of the repeated self-composition of submarkov kernels. In particular, given a set of specific conditions, we will see in Theorem 1.9 how to split the limit of the self-composition of some

kernel M into a composition of limits of self-compositions of some kernels L and K. The sole purpose of this rather specific theorem is to prove another theorem in Section 1.11.3, in which we deal with the consecutive evaluation of programs (Theorem 1.48).

Note that we have skipped the verification of Theorem 1.9 in Isabelle due to its heavy usage of measure theoretic concepts (see the disclaimer in Section 1.4 for a discussion). Instead we will present a detailed manual proof of Theorem 1.9 in the following.

Lemma 1.1. *Fix a set $A \in \Sigma_X$ and kernels K and L from X to Y with $K(x) \sim L(x)$ for all $x \in A$ where $\sim \in \{\leq, \geq, =\}$. Then $K \cdot \mu \sim L \cdot \mu$ for all distributions μ with $\mu(A^{\complement}) = 0$.*

Proof. Let $E \in \Sigma_Y$. It holds

$$\begin{aligned}(K \cdot \mu)(E) &= \int_A K(x)(E)d\mu(x) + \underbrace{\int_{A^{\complement}} K(x)(E)d\mu(x)}_{=0} \\ &\sim \int_A L(x)(E)d\mu(x) + \overbrace{\int_{A^{\complement}} L(x)(E)d\mu(x)} \\ &= (L \cdot \mu)(E)\end{aligned}$$

□

Lemma 1.2. *If K is invariant on V, then $\downarrow_V \circ K \geq \downarrow_V$.*

Proof. For $v \in V$, $(\downarrow_V \circ K)(v) = \downarrow_V(v)$. For $v \notin V$, $(\downarrow_V \circ K)(v) \geq 0 = \downarrow_V(v)$. □

Let K^n denote the composition $K \circ K \circ \ldots \circ K$ (n-times). The previous lemma entails that $\downarrow_V \circ K^n$ is monotonically increasing in n and, given that K is a submarkov kernel, it also has an upper bound. Hence the following limit exists:

Definition 1.3 (Kernel-Limit)**.** *Let a submarkov kernel K with domain and range Ω be given where K is invariant on $V \subseteq \Omega$, and let K^n denote the n-times composition $K \circ K \circ \ldots \circ K$. We define $\lim_V K \stackrel{def}{=} \lim_n \downarrow_V \circ K^n$.* I.61, p.113

Condition 1.4 (Conditions for Theorem 1.9)**.** *Let K and L be submarkov kernels with domain and range Ω. Let $U, V, D_K \subseteq \Omega$ be measurable sets. Assume:*

- *$U \subseteq D_K$.*
- *$V \cap D_K = \varnothing$.*
- *K is invariant on U.*
- *L is invariant on V.*
- *For all x, $K(x)(D_K^{\complement}) = 0$.*

- *For all x, $L(x)(D_K) = 0$.*
- *For all $x \in D_K \setminus U$, $L(x) = 0$.*
- *For all $x \in D_K^{\complement}$, $K(x) = 0$.*

Let $M(x) \stackrel{def}{=} \begin{cases} K(x) & \text{if } x \in D_K \setminus U, \\ L(x) & \text{otherwise.} \end{cases}$.

Lemma 1.5. *Under Condition 1.4, we have $L \circ K = L \circ \downarrow_U \circ K$.*

Proof. Fix an $x \in \Omega$. Let $\mu_1 \stackrel{\text{def}}{=} K(x)$. Then $\mu_1(D_K^{\complement}) = 0$. Since $\downarrow_U(x') = \downarrow_{D_K^{\complement} \cup U}(x')$ for all $x' \in D_K$, by Lemma 1.1 it holds $\downarrow_U \cdot \mu_1 = \downarrow_{D_K^{\complement} \cup U} \cdot \mu_1$. Furthermore, for any $x' \in D_K^{\complement} \cup U$, it holds $L \circ \downarrow_{D_K^{\complement} \cup U}(x') = L(x')$, and for any $x' \in D_K \setminus U$, it holds $L \circ \downarrow_{D_K^{\complement} \cup U}(x') = L \cdot 0 = 0 = L(x')$. Hence $L \circ \downarrow_{D_K^{\complement} \cup U} = L$ and

$$(L \circ K)(x) = L \cdot (\downarrow_{D_K^{\complement} \cup U} \cdot \mu_1) = L \cdot (\downarrow_U \cdot \mu_1) = (L \circ \downarrow_U \circ K)(x). \qquad \square$$

Lemma 1.6. *Under Condition 1.4, we have $\lim_V L \circ \lim_U K = \lim_n \downarrow_V \circ L^n \circ K^n$.*

Proof. We calculate as follows:

$$\begin{aligned} \lim_V L \circ \lim_U K(x) &= (\lim_n \downarrow_V \circ L^n) \circ (\lim_m \downarrow_U \circ K^m)(x) \\ &= \lim_n \downarrow_V \circ L^n \cdot \lim_m (\downarrow_U \circ K^m(x)) \\ &= \lim_n \lim_m \downarrow_V \circ L^n \cdot (\downarrow_U \circ K^m(x)) \\ &= (\lim_n \lim_m \downarrow_V \circ L^n \circ \downarrow_U \circ K^m)(x) \end{aligned}$$

Hence $\lim_V L \circ \lim_U K = \lim_n \lim_m N_{n,m}$ with $N_{n,m} \stackrel{\text{def}}{=} \downarrow_V \circ L^n \circ \downarrow_U \circ K^m$. By Lemma 1.2, $N_{n,m}$ is increasing in n and m. Hence $\lim_n \lim_m N_{n,m} = \lim_n N_{n,n} = \lim_n \downarrow_V \circ L^n \circ \downarrow_U \circ K^n \stackrel{1.5}{=} \lim_n \downarrow_V \circ L^n \circ K^n$. $\square$

Lemma 1.7. *Under Condition 1.4, we have $\downarrow_V \circ M^{2n} \geq \downarrow_V \circ L^n \circ K^n$ for all $n \in \mathbb{N}$.*

Proof. The case $n = 0$ is trivial, so we can assume $n \geq 1$. By Condition 1.4, we have $M(x) = K(x)$ for $x \in D_K \setminus U$ and $K(x) = 0$ for $x \in D_K^{\complement}$. Hence $M(x) \geq K(x)$ for $x \in U^{\complement}$ and hence $(\downarrow_V \circ M^{i+1})(x) \geq (\downarrow_V \circ M^i \circ K)(x)$ for all $x \in U^{\complement}$ and $i \geq 0$.

As K is invariant on U, for $u \in U$ we have $(\downarrow_V \circ M^i \circ K)(u) = (\downarrow_V \circ M^i)(u) \stackrel{1.2}{\leq} (\downarrow_V \circ M^{i+1})(u)$. Together, we have $\downarrow_V \circ M^i \circ K \leq \downarrow_V \circ M^{i+1}$ for all $i \geq 0$. Hence $\downarrow_V \circ M^i \circ K^{j+1} \leq \downarrow_V \circ M^{i+1} \circ K^j$ for all $i, j \geq 0$. By induction, $\downarrow_V \circ M^{2n} \geq \downarrow_V \circ M^n \circ K^n$ follows.

Furthermore, by Condition 1.4, $M(x) = L(x)$ for $x \in D_K^{\complement} \cup U$, and $L(x) = 0$ for $x \in D_K \setminus U$. Hence $M(x) \geq L(x)$ for all x. Thus $\downarrow_V \circ M^{2n} \geq \downarrow_V \circ M^n \circ K^n \geq \downarrow_V \circ L^n \circ K^n$. $\square$

Lemma 1.8. *Under Condition 1.4,* $(\downarrow_V \circ M^n)(x) \leq (\downarrow_V \circ L^n \circ K^n)(x)$ *for all* $x \in D_K$ *and* $n \in \mathbb{N}$.

Proof. Let $X \stackrel{\text{def}}{=} ((D_K \setminus U) \times \{0\}) \cup (U \times \mathbb{N})$. We define the kernel K' on X as

$$K'(a,i) \stackrel{\text{def}}{=} \begin{cases} (\lambda a.(a,0))(K(a)) & \text{if } (a,i) \in (D_K \setminus U) \times \{0\}, \\ \delta(a, i+1) & \text{if } (a,i) \in U \times \mathbb{N}. \end{cases}$$

Further, we define $L^*(a,n) \stackrel{\text{def}}{=} L^n(a)$ and $\pi(a,n) = \delta(a)$.

For $u \in U$, we have $(M \circ L^*)(u,0) = M(u) = L(u) = L^*(u,1) = (L^* \circ K')(u,0)$. Also, for $u \in U$ and $i \geq 1$, we have $(M \circ L^*)(u,i) = (M \circ L^i)(u)$. Since $M(x) = L(x)$ for $x \in D_K^{\complement}$ and $L^i(u)(D_K) = 0$, by Lemma 1.1 we have $(M \circ L^i)(u) = (L \circ L^i)(u) = L^*(u, i+1) = (L^* \circ K')(u,i)$.

For $x \in D_K \setminus U$, we have $(M \circ L^*)(x,0) = M(x) = K(x) = (\pi \circ K')(x,0)$. Moreover, $L^*(x,0) = \pi(x,0)$, and $K'(x,0)((D_K \times \{0\})^{\complement}) = 0$, hence by Lemma 1.1 we have $(\pi \circ K')(x,0) = (L^* \circ K')(x,0)$.

Summarizing, we have $(M \circ L^*)(x) = (L^* \circ K')(x)$ for all $x \in X$. Furthermore, for all $y \in X$ and all $i \geq 0$, $K'^i(y)(X^{\complement}) = 0$. Hence, by Lemma 1.1 we have $(M \circ L^* \circ K'^i)(y) = (L^* \circ K' \circ K'^i)(y) = (L^* \circ K'^{i+1})(y)$. Hence, $(M^{j+1} \circ L^* \circ K'^i)(y) = (M^j \circ L^* \circ K'^{i+1})(y)$ for all $i, j \geq 0$. By induction, we get $(M^n \circ L^*)(y) = (L^* \circ K'^n)(y)$ for $y \in X$. Hence for any $x \in D_K$, we have $(\downarrow_V \circ M^n)(x) = (\downarrow_V \circ M^n \circ L^*)(x,0) = (\downarrow_V \circ L^* \circ K'^n)(x,0)$.

Let $X_j \stackrel{\text{def}}{=} \{(x,i) \in X : i \leq j\}$. By definition of K', for $x \in X_j$ we have that $K'(x)(X_{j+1}^{\complement}) = 0$. By induction and using Lemma 1.1, we have $K'^n(x,0)(X_n^{\complement}) = 0$.

For $(x,i) \in X_n$, we have $(\downarrow_V \circ L^*)(x,i) = (\downarrow_V \circ L^i)(x) \stackrel{1.2}{\leq} (\downarrow_V \circ L^n)(x) = (\downarrow_V \circ L^n \circ \pi)(x,i)$. Together with $K'^n(x,0)(X_n^{\complement}) = 0$ and Lemma 1.1 we have $(\downarrow_V \circ L^* \circ K'^n)(x,0) \leq (\downarrow_V \circ L^n \circ \pi \circ K'^n)(x,0)$.

We have $(\pi \circ K')(x,i) = K(x) = (K \circ \pi)(x,i)$, hence $(\pi \circ K'^n)(x,i) = (K^n \circ \pi)(x,i)$, and therefore $(\downarrow_V \circ L^n \circ \pi \circ K'^n)(x,0) = (\downarrow_V \circ L^n \circ K^n \circ \pi)(x,0) = (\downarrow_V \circ L^n \circ K^n)(x)$.

Combining the results above, we get $(\downarrow_V \circ M^n)(x) \leq (\downarrow_V \circ L^n \circ K^n)(x)$ for all $x \in D_K$. □

Theorem 1.9. *Under Condition 1.4,* M *is a submarkov kernel invariant on* V *and* $\lim_V M(x) = (\lim_V L) \circ (\lim_U K)(x)$ *for all* $x \in D_K$. I.62, p.113

Proof. Fix $x \in D_K$. Then

$$\begin{aligned} \lim_V M(x) &\stackrel{1.8}{\leq} \lim_n (\downarrow_V \circ L^n \circ K^n)(x) \\ &\stackrel{1.7}{\leq} \lim_n (\downarrow_V \circ M^{2n})(x) \\ &= \lim_V M(x). \end{aligned}$$

Hence $\lim_V M(x) = \lim_n (\downarrow_V \circ L^n \circ K^n)(x) \stackrel{1.6}{=} (\lim_V L) \circ (\lim_U K)(x)$. □

1.6 Syntax of the Language

We now introduce the syntax of the language used by Verypto. To express
I.63, p.113 elementary objects in our language, we assume a set $\mathbf{B}$ of basic values. It contains the basic types we expect to need, but should not be considered fixed, as it can easily be extended in order to support other elementary objects. It includes the type $\mathbb{U}$ with a single element *unit*, the type of real numbers $\mathbb{R}$, but also functions of type $\mathbb{N} \to \mathbb{B}$ mapping natural numbers to Booleans, which can be used to encode infinite random tapes.

Following the considerations of Section 1.3, the syntax models a probabilistic higher-order lambda calculus with references and events in an iso-recursive setting. We first give the syntax in the following definition and then proceed with explanatory comments on the syntax.

I.67, p.113 **Definition 1.10** (Programs)**.** *We define programs P, pure values V_0, and*
I.68, p.114 *values V by the following grammar, where $n \in \mathbb{N}$, $v \in \mathbf{B}$, s denotes strings,*
I.69, p.114 *and f denotes submarkov kernels from V_0 to V_0:*

$$
\begin{aligned}
P &::= \mathsf{value}\, v \mid \mathsf{var}\, n \mid \lambda P \mid PP \mid \mathsf{fun}(f, P) \mid \\
&\quad\ \mathsf{loc}\, n \mid \mathsf{ref}\, P \mid !P \mid P := P \mid \mathsf{event}\, s \mid \mathsf{eventlist} \mid \\
&\quad\ (P, P) \mid \mathsf{fst}\, P \mid \mathsf{snd}\, P \mid \mathsf{inl}\, P \mid \mathsf{inr}\, P \mid \mathsf{case}\ P\ P\ P \mid \mathsf{fold}\, P \mid \mathsf{unfold}\, P \\
V_0 &::= \mathsf{value}\, v \mid (V_0, V_0) \mid \mathsf{inl}\, V_0 \mid \mathsf{inr}\, V_0 \mid \mathsf{fold}\, V_0 \\
V &::= \mathsf{value}\, v \mid (V, V) \mid \mathsf{inl}\, V \mid \mathsf{inr}\, V \mid \mathsf{fold}\, V \mid \mathsf{var}\, n \mid \lambda P \mid \mathsf{loc}\, n
\end{aligned}
$$

The construct $\mathsf{value}\, v$ is used to introduce an element $v \in \mathbf{B}$ in programs. Programs use de Bruijn notation [72] for variables. With $\mathsf{var}\, n$ we denote variables with de Bruijn index n, i.e., variables belonging to the $(n+1)$-st enclosing λ-binder and with λP we denote the function with body P.[2] See Section 1.6.1 for more details on the de Bruijn notation. Function application is written $P_1 P_2$ where P_1 is the function and P_2 is its argument. Store locations are denoted by $\mathsf{loc}\, n$, reference creation by $\mathsf{ref}\, P$, dereferencing by $!P$, and assignment by $P_1 := P_2$. Events are raised using the construct $\mathsf{event}\, s$. The list of previously raised events is accessible using $\mathsf{eventlist}$. The language also provides pairs (P_1, P_2) and their projections $\mathsf{fst}\, P$ and $\mathsf{snd}\, P$. Sums are constructed using $\mathsf{inl}\, P$ and $\mathsf{inr}\, P$ and destructed using the $\mathsf{case}\ P_1\ P_2\ P_3$ construct. In an iso-recursive setting (in contrast to an equi-recursive setting) the folding and unfolding of recursive types are made explicit using the constructions $\mathsf{fold}\, P$ and its inverse $\mathsf{unfold}\, P$ [108]. See Section 1.8 for further details.

Programs of the form $\mathsf{value}\, v$, $\mathsf{var}\, n$, λP, and $\mathsf{loc}\, n$ are considered values. If V_1 and V_2 are values, then so are (V_1, V_2), $\mathsf{inl}\, V_1$, $\mathsf{inr}\, V_1$, and $\mathsf{fold}\, V_1$. Pure

[2] Note that we use a different lambda symbol when expressing anonymous mathematical functions as in $\lambda x.\, x$; functions in our language use the lambda symbol as in λP.

values V_0 are used to represent data structures in the language. Pure values are values that do not contain variables, λ-abstractions, or locations.

Probabilism is introduced by the construct $\mathsf{fun}(f, P)$, where f is a sub-markov kernel from V_0 to V_0. It is interpreted as applying f to P, yielding a (sub-)probability measure on pure values. This construct is truly expressive: It allows for expressing every (deterministic) mathematical function f by using the deterministic kernel $\overline{f} = \lambda x.\, \delta(fx)$, but also every probabilistic function such as a coin-toss; moreover, it is not even limited to discrete probability measures. Using this construct we can, for example, express the random selection of infinite random tapes. We implemented the language in the proof assistant Isabelle/HOL [102]. The restriction to submarkov kernels on pure values is due to the fact that the datatype for P must not contain functions with argument type P in Isabelle/HOL. The extension Isabelle/HOLCF [101] which includes Scott's logic for computable functions allows such datatypes, but we decided not to introduce this additional domain-theoretic complexity.

We define the σ-algebra over programs Σ_P as the σ-algebra generated by the set of program rectangles $\dot{P}$, where $\dot{P}$ is defined inductively by including $\{\mathsf{value}\, v \mid v \in B\}$ for all $B \in \Sigma_{\mathbf{B}}$, $\{\mathsf{var}\, n\}$ for all $n \in \mathbb{N}$, $\{\mathsf{event}\, s\}$ for all strings s, $\{\mathsf{eventlist}\}$, $\{\mathsf{fun}(f, P) \mid P \in A \wedge f \in F\}$ for all $A \in \dot{P}$ and arbitrary F, $\{P_1 P_2 \mid P_1 \in A_1 \wedge P_2 \in A_2\}$ for all $A_1, A_2 \in \dot{P}$, $\{(P_1, P_2) \mid P_1 \in A_1 \wedge P_2 \in A_2\}$ for all $A_1, A_2 \in \dot{P}$, and analogously for the other cases. I.99, p.117

We define the set of contexts C similarly to the set of programs. A context is a program that may contain holes, denoted by $\Box$. These holes mark places where a program or another context can be inserted.

Definition 1.11 (Contexts)**.** *The syntax of contexts C is defined by the following grammar, where $n \in \mathbb{N}$, $v \in \mathbf{B}$, s denotes strings, and f denotes submarkov kernels from V_0 to V_0:* I.114, p.119

$$\begin{aligned} C ::= {} & \Box \mid \mathsf{value}\, v \mid \mathsf{var}\, n \mid \lambda C \mid CC \mid \mathsf{fun}(f, C) \mid \\ & \mathsf{loc}\, n \mid \mathsf{ref}\, C \mid !C \mid C := C \mid \mathsf{event}\, s \mid \mathsf{eventlist} \mid \\ & (C, C) \mid \mathsf{fst}\, C \mid \mathsf{snd}\, C \mid \mathsf{inl}\, C \mid \mathsf{inr}\, C \mid \mathsf{case}\ C\ C\ C \mid \mathsf{fold}\, C \mid \mathsf{unfold}\, C \end{aligned}$$

Given contexts C_1 and C_2, we write $C_1[C_2]$ to denote the term resulting from replacing every hole $\Box$ in C_1 with C_2.

Note that programs P can be seen as contexts without holes. We will also use the notation $C[P]$ to denote the replacement of every hole $\Box$ in C with program P. I.116, p.120

1.6.1 De Bruijn Notation

The language introduced in Definition 1.10 uses a nameless representation for terms [72], i.e., variables are not named and binders do not introduce

$$
\begin{array}{lcl}
\mathcal{FV}(\mathsf{var}\, n) & = & \{\mathsf{var}\, n\} \\
\mathcal{FV}(\lambda P) & = & \{\mathsf{var}\,(n-1) \mid \mathsf{var}\, n \in \mathcal{FV}(P) \wedge n > 0\} \\
\mathcal{FV}(\mathsf{value}\, v) & = & \{\} \\
\mathcal{FV}(\mathsf{loc}\, n) & = & \{\} \\
\mathcal{FV}(\mathsf{event}\, s) & = & \{\} \\
\mathcal{FV}(\mathsf{eventlist}) & = & \{\} \\
\mathcal{FV}(\mathsf{ref}\, P) & = & \mathcal{FV}(P) \\
\mathcal{FV}(\mathsf{fun}(f, P)) & = & \mathcal{FV}(P) \\
\mathcal{FV}(!P) & = & \mathcal{FV}(P) \\
\mathcal{FV}(\mathsf{fst}\, P) & = & \mathcal{FV}(P) \\
\mathcal{FV}(\mathsf{snd}\, P) & = & \mathcal{FV}(P) \\
\mathcal{FV}(\mathsf{inl}\, P) & = & \mathcal{FV}(P) \\
\mathcal{FV}(\mathsf{inr}\, P) & = & \mathcal{FV}(P) \\
\mathcal{FV}(\mathsf{fold}\, P) & = & \mathcal{FV}(P) \\
\mathcal{FV}(\mathsf{unfold}\, P) & = & \mathcal{FV}(P) \\
\mathcal{FV}(P_1 P_2) & = & \mathcal{FV}(P_1) \cup \mathcal{FV}(P_2) \\
\mathcal{FV}((P_1, P_2)) & = & \mathcal{FV}(P_1) \cup \mathcal{FV}(P_2) \\
\mathcal{FV}(P_1 := P_2) & = & \mathcal{FV}(P_1) \cup \mathcal{FV}(P_2) \\
\mathcal{FV}(\mathsf{case}\ P_1\ P_2\ P_3) & = & \mathcal{FV}(P_1) \cup \mathcal{FV}(P_2) \cup \mathcal{FV}(P_3)
\end{array}
$$

Figure 1.12: The definition of the set of the free variables of a term.

new names either. Instead, variables are represented by a natural number n called a *de Bruijn index*. This index refers to the binder associated with the variable. Namely, variable $\mathsf{var}\, n$ refers to the $(n+1)$-st enclosing λ-binder in the syntax tree. For example, the term $\lambda(\mathsf{var}\, 0)$ denotes the identity function.

Using a nameless representation for terms circumvents the cumbersome treatment of α-equivalent terms, i.e., terms that are equal up to the renaming of bound variables. For example the named terms $\lambda x.\, x$ and $\lambda y.\, y$ both denote the identity function, but use different variable names. Usually such terms that only differ in the names of bound variables are treated as equal. By using a nameless representation as explained above, this treatment is made explicit as both terms are represented by the same nameless term $\lambda(\mathsf{var}\, 0)$.

In the following we will give a series of definitions and operators related to the nameless representation of terms. In particular, a variable in some term is called *free*, if there is no associated λ-binder in its enclosing syntax tree. This is the case if the number of λ-binders enclosing the variable is smaller or equal than its de Bruijn index. A variable that is not free is called *bound*. Note that if a term contains multiple instances of the same variable, these instances might have different de Bruijn indices. For example the term $\lambda((\lambda(\mathsf{var}\, 3))\mathsf{var}\, 2)$ contains two instances of the same free variable. Therefore we identify free variables by the number of additional λ-binders that are necessary to bind them. Namely, if variable $\mathsf{var}\, n$ in term P is

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

Figure 1.13: The definition of the operator $\uparrow_k$ to lift free variables of a term.

enclosed by m λ-binders where $m \leq n$, we say that variable $\mathsf{var}\,(n-m)$ is free in P. We write $\mathcal{FV}(P)$ for the set of free variables of term P. For the example above this means that $\mathcal{FV}\,(\lambda((\lambda(\mathsf{var}\,3))\mathsf{var}\,2)) = \{\mathsf{var}\,1\}$. A formal definition of $\mathcal{FV}(P)$ is given in Figure 1.12. Note that only the first two rules in this definition are non-trivial. Namely, the set of free variables of variable $\mathsf{var}\, n$ contains the variable itself, and the set of free variables of a λ-term contains the free variables of the function's body (except for $\mathsf{var}\,0$) with their corresponding de Bruijn indices decreased by one. The other rules just descend. We call a term P *variable closed*, iff $\mathcal{FV}(P) = \{\}$, i.e., it has no free variables.

I.92, p.117

When substituting a term P for a variable in some other term, the de Bruijn indices of the free variables of P may need to be increased in order to avoid *capturing*. Namely, if the variable to be substituted is enclosed by λ-binders, we must lift the free variables of P so that they remain free after the substitution. The operator $\uparrow_k$ as defined in Figure 1.13 lifts all free variables with de Bruijn index $\geq k$ in some term by one. Again, only the first two rules in this definition are non-trivial. A variable $\mathsf{var}\, n$ is only lifted if $n \geq k$ and to lift a λ-term, we lift all free variables of the function's body with de Bruijn index $\geq k+1$. The other rules just descend.

I.86, p.115

In a named λ-calculus the term $(\lambda x.\, P)V$ usually reduces to the term

$$
\begin{array}{rcl}
(\mathsf{var}\, n)\{P/_k\} & = & \begin{cases} \mathsf{var}\, n-1 & \text{if } n > k \\ P & \text{if } n = k \\ \mathsf{var}\, n & \text{if } n < k \end{cases} \\
(\lambda P_1)\{P/_k\} & = & \lambda(P_1\{\uparrow_0 P/_{k+1}\}) \\
(P_1 P_2)\{P/_k\} & = & (P_1\{P/_k\})(P_2\{P/_k\}) \\
(\mathsf{fun}(f, P_1))\{P/_k\} & = & \mathsf{fun}(f, P_1\{P/_k\}) \\
(\mathsf{value}\, v)\{P/_k\} & = & \mathsf{value}\, v \\
(\mathsf{loc}\, n)\{P/_k\} & = & \mathsf{loc}\, n \\
(\mathsf{ref}\, P_1)\{P/_k\} & = & \mathsf{ref}\, (P_1\{P/_k\}) \\
(!P_1)\{P/_k\} & = & !(P_1\{P/_k\}) \\
(P_1 := P_2)\{P/_k\} & = & (P_1\{P/_k\}) := (P_2\{P/_k\}) \\
(\mathsf{event}\, s)\{P/_k\} & = & \mathsf{event}\, s \\
(\mathsf{eventlist})\{P/_k\} & = & \mathsf{eventlist} \\
(\mathsf{fold}\, P_1)\{P/_k\} & = & \mathsf{fold}\, (P_1\{P/_k\}) \\
(\mathsf{unfold}\, P_1)\{P/_k\} & = & \mathsf{unfold}\, (P_1\{P/_k\}) \\
((P_1, P_2))\{P/_k\} & = & (P_1\{P/_k\}), (P_2\{P/_k\}) \\
(\mathsf{fst}\, P_1)\{P/_k\} & = & \mathsf{fst}\, (P_1\{P/_k\}) \\
(\mathsf{snd}\, P_1)\{P/_k\} & = & \mathsf{snd}\, (P_1\{P/_k\}) \\
(\mathsf{inl}\, P_1)\{P/_k\} & = & \mathsf{inl}\, (P_1\{P/_k\}) \\
(\mathsf{inr}\, P_1)\{P/_k\} & = & \mathsf{inr}\, (P_1\{P/_k\}) \\
(\mathsf{case}\ P_1\ P_2\ P_3)\{P/_k\} & = & \mathsf{case}\ (P_1\{P/_k\})\ (P_2\{P/_k\})\ (P_3\{P/_k\})
\end{array}
$$

Figure 1.14: The definition of the operator $\{\cdot/_k\}$ to substitute free variables with a term.

$P\{V/x\}$, i.e., the occurrences of variable x in P are substituted with the value V. Such a step is called a *β-reduction*. For our nameless representation of terms, we introduce a substitution operator $\{\cdot/_k\}$ which can be used analogously, i.e., the operator is defined such that the nameless term $(\lambda P)V$ will β-reduce to the term $P\{V/_0\}$. Since the β-reduction removes the λ-binder enclosing P, the substitution operator must adapt the free variables of P accordingly. The formal definition of $\{\cdot/_k\}$ is given in Figure 1.14. As before, only the first two rules in this definition are non-trivial. Namely, the operator $\{P/_k\}$ substitutes the free variable $\mathsf{var}\, k$ with term P, while decreasing free variables with greater de Bruijn indices by one. To substitute the free variable $\mathsf{var}\, k$ in a λ-term, we substitute the variable $\mathsf{var}\,(k+1)$ in the function's body. Because the function's body is enclosed by a λ-binder, we additionally need to lift all free variables of P in order to avoid their capturing. The other rules just descend. I.88, p.115

Many statements about programs involve the lift operator $\uparrow_k$ and the substitution operator $\{\cdot/_k\}$. Therefore we need rules that allow us to reason about combinations of these two operators. The following lemma introduces several such rules that allow us to commute the order in which these operators are applied to programs.

Lemma 1.15. *Let P_1, P_2, P_3 be programs and assume $k \leq l$. Then the following properties hold:*

- $\uparrow_k(\uparrow_l P_1) = \uparrow_{l+1}(\uparrow_k P_1)$ I.103, p.118
- $(\uparrow_k P_1)\{P_2/_k\} = P_1$ I.104, p.118
- $\uparrow_k(P_1\{P_2/_l\}) = (\uparrow_k P_1)\{\uparrow_k P_2/_{l+1}\}$ I.105, p.118
- $\uparrow_l(P_1\{P_2/_k\}) = (\uparrow_{l+1} P_1)\{\uparrow_l P_2/_k\}$ I.106, p.118
- $(P_1\{P_2/_{l+1}\})\{P_3/_k\} = (P_1\{\uparrow_l P_3/_k\})\{P_2\{P_3/_k\}/_l\}$ I.107, p.118
- $(P_1\{P_2/_k\})\{P_3/_l\} = (P_1\{\uparrow_k P_3/_{l+1}\})\{P_2\{P_3/_l\}/_k\}$ I.108, p.118

To handle contexts, we will use the same notation as above to denote the lifting and substitution of free variables in contexts, i.e., we define the operators $\uparrow_k$ and $\{\cdot/_k\}$ analogously for contexts by introducing additional rules for holes $\square$: The lifting of a hole is defined as $\uparrow_k(\square) \stackrel{\text{def}}{=} \square$ and the substitution of a hole is defined as $\square\{C/_k\} \stackrel{\text{def}}{=} \square$. Note that there is a fundamental difference between inserting into a context $C[P]$ and substituting into a context $C\{P/_k\}$. Namely, the operator $\{P/_k\}$ performs a *capture avoiding* substitution, i.e., the free variables of P are lifted as needed so that they remain free after the substitution, whereas the insertion $C[P]$ is a mere syntactical replacement of every hole $\square$ in C with P, i.e., λ-binders that enclose holes in C might capture free variables of P. In case P contains no free variables, we show the following connection between context insertion and substitution: I.119, p.121 I.120, p.121

Lemma 1.16. *Let P and P' be programs with $\mathcal{FV}(P) = \{\}$. We define the context $C_{P'} \stackrel{\text{def}}{=} P'\{\square/_0\}$. Then it holds $C_{P'}[P] = P'\{P/_0\}$.* I.126, p.123

1.7 Semantics of the Language

We now define the semantics of our language. In order to handle the notion of polynomial runtime, the semantics is operational, based on a small-step reduction relation $\rightsquigarrow$. Since the language contains references and is probabilistic, this relation maps program states to distributions over program states. Here, a program state also includes a list of previously raised events.

I.134, p.123 **Definition 1.17** (Program State and Reduction)**.** *A* program state *is a triple $P|\sigma|\eta$ of a program P, a list σ of values (the store), and a list η of strings (the events raised so far). If the program P is a value, we also call the triple a* value state*. We denote the set of all program states by Ω and the set of all value states by Val.*

The reduction relation $\rightsquigarrow$ *is a relation between program states and sub-probability measures over program states. It is defined by the inference rules given in Figure 1.18.*

The inference rules in the lower part of Figure 1.18 have no real computational content, they merely define which subterm should be evaluated first. Such rules are called *congruence rules* [108]. They enforce that a term's subterms are evaluated from left to right until they are values. E.g., the rule App1 formalizes that a state $P_1P_2|\sigma|\eta$ containing the application P_1P_2 can be reduced by reducing the state $P_1|\sigma|\eta$, which contains only the term P_1, to a measure μ. The function $\lambda(P_1'|\sigma'|\eta').\, P_1'P_2|\sigma'|\eta'$ is then used to project the measure μ, i.e., this function reintroduces the application of P_2 in the states of μ. Similarly, the rule App2 formalizes that it is also possible to reduce the latter part of an application, but only if its former part is a value. Note that there is no rule to reduce a value. Hence, the rules App1 and App2 enforce that applications must be evaluated from left to right.

In contrast to these congruence rules, the rules in the upper half of Figure 1.18 define the computational behavior of the language, i.e., instead of determining *where* to perform a reduction step, they specify *how* this step is performed. They characterize a set of reducible expressions, so-called *redexes*, and the corresponding measure they reduce to. Such rules are called *computation rules* [108]. Together with the congruence rules they enforce a *call-by-value* evaluation strategy for the language.

The rule Beta defines the β-reduction using the substitution operator $\{\cdot/_k\}$ we introduced in Figure 1.14. Namely, a state containing an application of a λ-term λP to a value V reduces to the Dirac measure of the state with the substituted term $P\{V/_0\}$. The function construct $\mathsf{fun}(f, V_0)|\sigma|\eta$, where V_0 is a pure value, is reduced using rule Fun. The distribution is obtained by applying f to V_0 and projecting the state into the resulting measure, namely we reduce to $(\lambda x.\, x|\sigma|\eta)(f(V_0))$. This is the only computation rule that introduces probabilistic behavior. All other rules are deterministic in the sense that they reduce to Dirac measures.

$$(\lambda P)V|\sigma|\eta \rightsquigarrow \delta(P\{V/_0\}|\sigma|\eta) \qquad \textsc{Beta}$$

$$\mathsf{fun}(f, V_0)|\sigma|\eta \rightsquigarrow (\lambda x.\, x|\sigma|\eta)(f(V_0)) \qquad \textsc{Fun}$$

$$\mathsf{ref}\, V|\sigma|\eta \rightsquigarrow \delta(\mathsf{loc}\,(|\sigma|)|\sigma@[V]|\eta) \qquad \textsc{Ref}$$

$$!(\mathsf{loc}\, n)|\sigma|\eta \rightsquigarrow \delta(\sigma_n|\sigma|\eta) \quad \text{if } n < |\sigma| \qquad \textsc{Deref}$$

$$\mathsf{loc}\, n := V|\sigma|\eta \rightsquigarrow \delta(\mathsf{value}\, unit|\sigma[n := V]|\eta) \quad \text{if } n < |\sigma| \qquad \textsc{Ass}$$

$$\mathsf{event}\, s|\sigma|\eta \rightsquigarrow \delta(\mathsf{value}\, unit|\sigma|\eta@[s]) \qquad \textsc{Ev}$$

$$\mathsf{eventlist}|\sigma|\eta \rightsquigarrow \delta(\overline{\eta}|\sigma|\eta) \qquad \textsc{EvList}$$

$$\mathsf{fst}\,(V_1, V_2)|\sigma|\eta \rightsquigarrow \delta(V_1|\sigma|\eta) \qquad \textsc{Fst}$$

$$\mathsf{snd}\,(V_1, V_2)|\sigma|\eta \rightsquigarrow \delta(V_2|\sigma|\eta) \qquad \textsc{Snd}$$

$$\mathsf{case}\ (\mathsf{inl}\, V)\ V_1\ V_2|\sigma|\eta \rightsquigarrow \delta(V_1 V|\sigma|\eta) \qquad \textsc{CaseL}$$

$$\mathsf{case}\ (\mathsf{inr}\, V)\ V_1\ V_2|\sigma|\eta \rightsquigarrow \delta(V_2 V|\sigma|\eta) \qquad \textsc{CaseR}$$

$$\mathsf{unfold}\,(\mathsf{fold}\, V)|\sigma|\eta \rightsquigarrow \delta(V|\sigma|\eta) \qquad \textsc{Fold}$$

$$\textsc{App1:}\ \frac{P_1|\sigma|\eta \rightsquigarrow \mu}{P_1 P_2|\sigma|\eta \rightsquigarrow (\lambda(P_1'|\sigma'|\eta').\, P_1' P_2|\sigma'|\eta')\mu} \qquad \textsc{Fold1:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{\mathsf{fold}\, P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, \mathsf{fold}\, P'|\sigma'|\eta')\mu}$$

$$\textsc{App2:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{V P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, V P'|\sigma'|\eta')\mu} \qquad \textsc{Unfold1:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{\mathsf{unfold}\, P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, \mathsf{unfold}\, P'|\sigma'|\eta')\mu}$$

$$\textsc{Ref1:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{\mathsf{ref}\, P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, \mathsf{ref}\, P'|\sigma'|\eta')\mu} \qquad \textsc{Pair1:}\ \frac{P_1|\sigma|\eta \rightsquigarrow \mu}{(P_1, P_2)|\sigma|\eta \rightsquigarrow (\lambda(P_1'|\sigma'|\eta').\, (P_1', P_2)|\sigma'|\eta')\mu}$$

$$\textsc{Deref1:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{!P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, !P'|\sigma'|\eta')\mu} \qquad \textsc{Pair2:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{(V, P)|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, (V, P')|\sigma'|\eta')\mu}$$

$$\textsc{Ass1:}\ \frac{P_1|\sigma|\eta \rightsquigarrow \mu}{P_1 := P_2|\sigma|\eta \rightsquigarrow (\lambda(P_1'|\sigma'|\eta').\, P_1' := P_2|\sigma'|\eta')\mu} \qquad \textsc{Fst1:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{\mathsf{fst}\, P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, \mathsf{fst}\, P'|\sigma'|\eta')\mu}$$

$$\textsc{Ass2:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{V := P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, V := P'|\sigma'|\eta')\mu} \qquad \textsc{Snd1:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{\mathsf{snd}\, P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, \mathsf{snd}\, P'|\sigma'|\eta')\mu}$$

$$\textsc{Inl1:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{\mathsf{inl}\, P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, \mathsf{inl}\, P'|\sigma'|\eta')\mu} \qquad \textsc{Inr1:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{\mathsf{inr}\, P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, \mathsf{inr}\, P'|\sigma'|\eta')\mu}$$

$$\textsc{Fun1:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{\mathsf{fun}(f, P)|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, \mathsf{fun}(f, P')|\sigma'|\eta')\mu}$$

$$\textsc{Case1:}\ \frac{P_1|\sigma|\eta \rightsquigarrow \mu}{\mathsf{case}\ P_1\ P_2\ P_3|\sigma|\eta \rightsquigarrow (\lambda(P_1'|\sigma'|\eta').\, \mathsf{case}\ P_1'\ P_2\ P_3|\sigma'|\eta')\mu}$$

$$\textsc{Case2:}\ \frac{P_2|\sigma|\eta \rightsquigarrow \mu}{\mathsf{case}\ V\ P_2\ P_3|\sigma|\eta \rightsquigarrow (\lambda(P_2'|\sigma'|\eta').\, \mathsf{case}\ V\ P_2'\ P_3|\sigma'|\eta')\mu}$$

$$\textsc{Case3:}\ \frac{P|\sigma|\eta \rightsquigarrow \mu}{\mathsf{case}\ V_1\ V_2\ P|\sigma|\eta \rightsquigarrow (\lambda(P'|\sigma'|\eta').\, \mathsf{case}\ V_1\ V_2\ P'|\sigma'|\eta')\mu}$$

Figure 1.18: The small-step reduction rules of the language. The computation rules are defined in the upper part, whereas the congruence rules of the language are given below.

The rules concerning the store are Ref, Deref, and Assign. The state $\mathsf{ref}\, V|\sigma|\eta$ reduces to $\delta(\mathsf{loc}\,(|\sigma|)|\sigma@[V]|\eta)$, i.e., the value V is appended to the store and its location $|\sigma|$ is returned. Assuming $n < |\sigma|$, the state $!(\mathsf{loc}\, n)|\sigma|\eta$ reduces to the n-th element of σ, namely $\delta(\sigma_n|\sigma|\eta)$ and the assignment $\mathsf{loc}\, n := V|\sigma|\eta$ replaces the n-th element of σ by V and returns the value *unit*, namely $\delta(\mathsf{value}\, \mathit{unit}|\sigma[n := V]|\eta)$.

If an event is raised, as in $\mathsf{event}\, s|\sigma|\eta$, the event is appended to the event list η and the *unit* value is returned (rule Ev), namely $\delta(\mathsf{value}\, \mathit{unit}|\sigma|\eta@[s])$. The program state $\mathsf{eventlist}|\sigma|\eta$ returns the list of previously raised events η (rule EvList), namely it reduces to $\delta(\overline{\eta}|\sigma|\eta)$. Here $\overline{\eta}$ is the representation of η in the program syntax, i.e., a list of strings.

The pair destructors $\mathsf{fst}\,(V_1, V_2)|\sigma|\eta$ and $\mathsf{snd}\,(V_1, V_2)|\sigma|\eta$ reduce to the first and the second component of the pair, respectively, namely $\delta(V_1|\sigma|\eta)$ (rule Fst) and $\delta(V_2|\sigma|\eta)$ (rule Snd). The rules CaseL and CaseR specify the behavior of the case-construct. If the case-construct is applied to a value $\mathsf{inl}\, V$ as in $\mathsf{case}\ (\mathsf{inl}\, V)\ V_1\ V_2|\sigma|\eta$, then the value V is given as an argument to V_1, namely $\delta(V_1 V|\sigma|\eta)$. Likewise the program state $\mathsf{case}\ (\mathsf{inr}\, V)\ V_1\ V_2|\sigma|\eta$ reduces to $\delta(V_2 V|\sigma|\eta)$. Finally, rule Fold states that an unfold applied to a fold cancels out, namely the program state $\mathsf{unfold}\,(\mathsf{fold}\, V)|\sigma|\eta$ reduces to $\delta(V|\sigma|\eta)$.

Note that the rules in Figure 1.18 are mutually exclusive, i.e., for each term at most one rule applies. The following lemma captures that a program state cannot reduce to two different measures.

I.143, p.125 **Lemma 1.19** (Unique Reduction)**.** *Let $P|\sigma|\eta$ be a program state. Then there exists at most one measure μ such that $P|\sigma|\eta \rightsquigarrow \mu$.*

Since there is at most one measure that a program state can reduce to, we can define the kernel *step* which, given a state $P|\sigma|\eta$, returns the measure μ the state reduces to, or, if the state cannot be reduced, returns the Dirac measure $\delta(P|\sigma|\eta)$.

I.135, p.124
$$\mathit{step}(P|\sigma|\eta) \stackrel{\text{def}}{=} \begin{cases} \mu & \text{if } P|\sigma|\eta \rightsquigarrow \mu, \\ \delta(P|\sigma|\eta) & \text{otherwise.} \end{cases}$$

Since the $\rightsquigarrow$ relation does not reduce value states, the kernel *step* is invariant on the set of value states *Val*. Hence the limit $\lim_{\mathit{Val}} \mathit{step}$ exists and we use it to define the *denotation* of a program state $P|\sigma|\eta$: By restricting the measure $\mathit{step}^n(P|\sigma|\eta)$ to value states, we only consider program paths that terminate after at most n steps. The denotation of $P|\sigma|\eta$ is given by the limit of such measures, i.e., the measure after all terminating paths have been evaluated.

I.137, p.124 **Definition 1.20** (Denotation)**.** *Let $P|\sigma|\eta$ be a program state and let Val be the set of all value states. We define the denotation of $P|\sigma|\eta$ as*

$$[\![P|\sigma|\eta]\!] \stackrel{\mathit{def}}{=} (\lim_{\mathit{Val}} \mathit{step})(P|\sigma|\eta).$$

1.8 Typing the Language

Type systems characterize a set of well-behaving programs. E.g., the property of type safety (see Section 1.11.1) states that well-typed programs that are not values are not "stuck" and can be reduced using $\rightsquigarrow$ such that almost all programs in the resulting distribution are also well-typed. Requiring programs to be well-typed can prevent many kinds of implementation errors; also, it leads to more concise statements as undesired behavior is excluded. We now introduce the type system of our language which is iso-recursive in order to allow the user to construct arbitrary data types. First we will define the set of types T and the set of pure types T_0. Then we will proceed by specifying the rules of a typing relation that assigns types to programs.

Definition 1.21 (Types)**.** *We define types T and pure types T_0 by the following grammars, where $X \in \Sigma_{\mathbf{B}}$ and $n \in \mathbb{N}$:* I.152, p.125 I.153, p.126

$$T ::= \mathsf{Value}_X \mid T \times T \mid T + T \mid T \to T \mid \mathsf{Ref}\, T \mid \mu T \mid \mathsf{Tvar}\, n$$
$$T_0 ::= \mathsf{Value}_X \mid T_0 \times T_0 \mid T_0 + T_0 \mid \mu T_0 \mid \mathsf{Tvar}\, n$$

For a measurable set X, the type of basic values in that set is denoted Value_X. For types T_1 and T_2, $T_1 \times T_2$ and $T_1 + T_2$ denote the pair type and the sum type of T_1 and T_2, respectively. $T_1 \to T_2$ denotes the type of functions from T_1 to T_2, and $\mathsf{Ref}\, T$ denotes the type of references of type T. Recursive types are expressed by μT which introduces a binder in de Bruijn notation and $\mathsf{Tvar}\, n$ denotes the type variable with de Bruijn index n. Free type variables are defined analogously to free variables of terms. Using the same notation as in Section 1.6.1, we define an operator $\uparrow_k$ to lift the free type variables with de Bruijn index $\geq k$ by one and an operator $\{\cdot/_k\}$ to substitute a type into another type (see Figure 1.23). Intuitively the type μT represents the infinite type that is obtained by recursively substituting $\mathsf{Tvar}\, 0$ with μT in T. The set of pure types T_0 consists of the types that do not contain function or reference types. I.160, p.126 I.161, p.126

Using pairs, sums, and recursive types we can define other types such as the unit type, Booleans, lists, and natural numbers. E.g., we set $\mathsf{Unit} \stackrel{\text{def}}{=} \mathsf{Value}_{\mathbb{U}}$ for the unit type and $\mathsf{Bool} \stackrel{\text{def}}{=} \mathsf{Unit} + \mathsf{Unit}$ for Booleans. We then define $\mathsf{true} \stackrel{\text{def}}{=} \mathsf{inl}\,(\mathsf{value}\, \mathit{unit})$ and $\mathsf{false} \stackrel{\text{def}}{=} \mathsf{inr}\,(\mathsf{value}\, \mathit{unit})$, and introduce syntactic sugar for conditionals via $\mathsf{if}\ P_1\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \stackrel{\text{def}}{=} \mathsf{case}\ P_1\ (\lambda \uparrow_0 P_2)\ (\lambda \uparrow_0 P_3)$. We set $\mathsf{List}\, T \stackrel{\text{def}}{=} \mu(\mathsf{Unit} + (T \times \mathsf{Tvar}\, 0))$ for lists over type T, where the usual constructors for lists are defined by $\mathit{nil} \stackrel{\text{def}}{=} \mathsf{fold}\,(\mathsf{inl}\,(\mathsf{value}\, \mathit{unit}))$ and $P_1 :: P_2 \stackrel{\text{def}}{=} \mathsf{fold}\,(\mathsf{inr}\,(P_1, P_2))$. Assuming a binary representation of natural numbers (with a special treatment of 0), we define $\mathsf{Nat} \stackrel{\text{def}}{=} \mathsf{Unit} + \mathsf{List}\, \mathsf{Bool}$. We further assume the definition of a type Char of characters and a type $\mathsf{String} \stackrel{\text{def}}{=} \mathsf{List}\, \mathsf{Char}$. I.154, p.126 I.155, p.126 I.75, p.114 I.87, p.115 I.156, p.126 I.72, p.114 I.73, p.114 I.157, p.126 I.158, p.126

We now introduce the typing rules of the language. We model type environments as lists of program types. In the following, let Γ be a type

environment that is used to assign types to variables (by giving $\mathsf{var}\,n$ the n-th type in Γ, counting from 0) and let Θ be a type environment to assign types to store locations (by giving $\mathsf{loc}\,n$ the n-th type in Θ). Let $T{::}\Gamma$ denote the environment Γ with the type T prepended.

I.162, p.126 **Definition 1.22** (Program Typing)**.** *A program P has type T under the type environments Γ and Θ, iff $\Gamma|\Theta \vdash P : T$ can be inferred from the rules given in Figure 1.24. We say that a program P is of type T, iff $[]|[] \vdash P : T$.*

Most of the typing rules presented in Figure 1.24 are straightforward. The interesting rules are the following ones: Programs of the form $\mathsf{value}\,v$ have type Value_X, provided that $v \in X$ (rule VAL). The program $\mathsf{event}\,e$ has type Unit (rule EV). The type assigned to $\mathsf{eventlist}$ is $\mathsf{List\,String}$ (rule EVLIST), because it returns a list of previously raised events. The case construct expects a sum of type $T_1 + T_2$ as first argument, a function of type $T_1 \to T$ as second argument (which is applied to the first argument, if it is a T_1), and a function of type $T_2 \to T$ as third argument (which is applied to the first argument, if it is a T_2). The overall type of the construct is T (rule CASE). If P has pure type T_0', and for all v the measure $f(v)$ (where f is a submarkov kernel on pure values) contains only programs of pure type T_0, then $\mathsf{fun}(f, P)$ has type T_0 (rule FUN). If P has type $T\{\mu T/_0\}$, then $\mathsf{fold}\,P$ has type μT (rule FOLD), where $T_1\{T/_0\}$ denotes the substitution of $\mathsf{Tvar}\,0$ in T_1 with T (see Figure 1.23). Likewise, if P has type μT, then $\mathsf{unfold}\,P$ has type $T\{\mu T/_0\}$ (rule UNFOLD).

Recursive types are very expressive. To illustrate their power, consider
I.147, p.125 the following term $\omega \stackrel{\text{def}}{=} \lambda((\mathsf{unfold}\,(\mathsf{var}\,0))\,\mathsf{var}\,0)$. We can define the program
I.148, p.125 $\bot \stackrel{\text{def}}{=} \omega(\mathsf{fold}\,\omega)$ which folds the term ω and applies it to itself [108]. The program $\bot$ has the interesting property that it reduces to itself in two steps:

I.149, p.125 $$\bot|\sigma|\eta \rightsquigarrow \delta((\mathsf{unfold}\,(\mathsf{fold}\,\omega))(\mathsf{fold}\,\omega)|\sigma|\eta), \text{ and then}$$

I.150, p.125 $$(\mathsf{unfold}\,(\mathsf{fold}\,\omega))(\mathsf{fold}\,\omega)|\sigma|\eta \rightsquigarrow \delta(\bot|\sigma|\eta)$$

In particular, this means that the term $\bot$ diverges. It never reduces to
I.151, p.125 a value, and hence its denotation is the 0 distribution: $[\![\bot|\sigma|\eta]\!] = 0$. In a simply typed calculus such a self-applying term cannot be typed, but using our recursive type system $\vdash$ the programs ω and $\bot$ are typeable [108]. For this, note that ω is a function which applies $\mathsf{var}\,0$ to itself (after unfolding it), i.e., we need to give $\mathsf{var}\,0$ a function type $T' \to T$ where $T' = T' \to T$. In a simply typed calculus, this equation is not satisfiable, but we can resolve this by instantiating T' with the recursive type $\mu(\mathsf{Tvar}\,0 \to T)$. By unfolding T', i.e., by substituting T' into its body, we obtain $(\mathsf{Tvar}\,0 \to T)\{T'/_0\} = T' \to T$. Using the rules from Figure 1.24, we can infer $\Gamma|\Theta \vdash$
I.167, p.127 $\omega : (\mu(\mathsf{Tvar}\,0 \to T)) \to T$. Conversely, by folding the type again, we can give
I.168, p.127 program $\bot$ the type T, namely $\Gamma|\Theta \vdash \bot : T$. Note that this means that we can assign $\bot$ an arbitrary type T. The only restriction is that T must not

$$
\begin{array}{ll}
\uparrow_k(\mathsf{Tvar}\, n) & = \begin{cases} \mathsf{Tvar}\,(n{+}1) & \text{if } n{\geq}k \\ \mathsf{Tvar}\, n & \text{if } n{<}k \end{cases} \\
\uparrow_k(\mu T) & = \mu(\uparrow_{k+1} T) \\
\uparrow_k(\mathsf{Value}_X) & = \mathsf{Value}_X \\
\uparrow_k(T_1 \times T_2) & = \uparrow_k T_1 \times \uparrow_k T_2 \\
\uparrow_k(T_1 + T_2) & = \uparrow_k T_1 + \uparrow_k T_2 \\
\uparrow_k(T_1 \to T_2) & = \uparrow_k T_1 \to \uparrow_k T_2 \\
\uparrow_k(\mathsf{Ref}\, T) & = \mathsf{Ref}\,(\uparrow_k T)
\end{array}
$$

$$
\begin{array}{rl}
(\mathsf{Tvar}\, n)\{T/_k\} & = \begin{cases} \mathsf{Tvar}\, n{-}1 & \text{if } n{>}k \\ T & \text{if } n{=}k \\ \mathsf{Tvar}\, n & \text{if } n{<}k \end{cases} \\
(\mu T_1)\{T/_k\} & = \mu(T_1\{\uparrow_0 T/_{k+1}\}) \\
(\mathsf{Value}_X)\{T/_k\} & = \mathsf{Value}_X \\
(T_1 \times T_2)\{T/_k\} & = T_1\{T/_k\} \times T_2\{T/_k\} \\
(T_1 + T_2)\{T/_k\} & = T_1\{T/_k\} + T_2\{T/_k\} \\
(T_1 \to T_2)\{T/_k\} & = T_1\{T/_k\} \to T_2\{T/_k\} \\
(\mathsf{Ref}\, T_1)\{T/_k\} & = \mathsf{Ref}\,(T_1\{T/_k\})
\end{array}
$$

Figure 1.23: Left: The definition of the operator $\uparrow_k$ to lift free type variables. Right: The definition of the operator $\{\cdot/_k\}$ to substitute free type variables.

$$\frac{v \in X}{\Gamma|\Theta \vdash \mathsf{value}\, v : \mathsf{Value}_X}\ \textsc{Val} \qquad \frac{}{T{::}\Gamma|\Theta \vdash \mathsf{var}\, 0 : T}\ \textsc{Var0}$$

$$\frac{\Gamma|\Theta \vdash \mathsf{var}\, n : T}{T'{::}\Gamma|\Theta \vdash \mathsf{var}\,(n+1) : T}\ \textsc{Var1}$$

$$\frac{\Gamma|\Theta \vdash P : T_0' \qquad \forall v. \forall t{\leftarrow}f(v).\ []|[] \vdash t : T_0}{\Gamma|\Theta \vdash \mathsf{fun}(f, P) : T_0}\ \textsc{Fun}$$

$$\frac{\Gamma|\Theta \vdash P_1 : T_1 \qquad \Gamma|\Theta \vdash P_2 : T_2}{\Gamma|\Theta \vdash (P_1, P_2) : T_1 \times T_2}\ \textsc{Pair} \qquad \frac{T_1{::}\Gamma|\Theta \vdash P : T_2}{\Gamma|\Theta \vdash \lambda P : T_1 \to T_2}\ \textsc{Lam}$$

$$\frac{\Gamma|\Theta \vdash P_1 : T_1 \to T_2 \qquad \Gamma|\Theta \vdash P_2 : T_1}{\Gamma|\Theta \vdash P_1 P_2 : T_2}\ \textsc{App} \qquad \frac{}{\Gamma|T{::}\Theta \vdash \mathsf{loc}\, 0 : T}\ \textsc{Loc0}$$

$$\frac{\Gamma|\Theta \vdash \mathsf{loc}\, n : T}{\Gamma|T'{::}\Theta \vdash \mathsf{loc}\,(n+1) : T}\ \textsc{Loc1} \qquad \frac{\Gamma|\Theta \vdash P : T}{\Gamma|\Theta \vdash \mathsf{ref}\, P : \mathsf{Ref}\, T}\ \textsc{Ref}$$

$$\frac{\Gamma|\Theta \vdash P : \mathsf{Ref}\, T}{\Gamma|\Theta \vdash !P : T}\ \textsc{Deref} \qquad \frac{\Gamma|\Theta \vdash P_1 : \mathsf{Ref}\, T \qquad \Gamma|\Theta \vdash P_2 : T}{\Gamma|\Theta \vdash P_1 := P_2 : \mathsf{Unit}}\ \textsc{Ass}$$

$$\frac{}{\Gamma|\Theta \vdash \mathsf{event}\, e : \mathsf{Unit}}\ \textsc{Ev} \qquad \frac{}{\Gamma|\Theta \vdash \mathsf{eventlist} : \mathsf{List\,String}}\ \textsc{EvList}$$

$$\frac{\Gamma|\Theta \vdash P : T_1 \times T_2}{\Gamma|\Theta \vdash \mathsf{fst}\, P : T_1}\ \textsc{Fst} \qquad \frac{\Gamma|\Theta \vdash P : T_1 \times T_2}{\Gamma|\Theta \vdash \mathsf{snd}\, P : T_2}\ \textsc{Snd}$$

$$\frac{\Gamma|\Theta \vdash P : T\{\mu T/_0\}}{\Gamma|\Theta \vdash \mathsf{fold}\, P : \mu T}\ \textsc{Fold} \qquad \frac{\Gamma|\Theta \vdash P : \mu T}{\Gamma|\Theta \vdash \mathsf{unfold}\, P : T\{\mu T/_0\}}\ \textsc{Unfold}$$

$$\frac{\Gamma|\Theta \vdash P : T}{\Gamma|\Theta \vdash \mathsf{inl}\, P : T + T'}\ \textsc{Inl} \qquad \frac{\Gamma|\Theta \vdash P : T}{\Gamma|\Theta \vdash \mathsf{inr}\, P : T' + T}\ \textsc{Inr}$$

$$\frac{\Gamma|\Theta \vdash P_1 : T_1 + T_2 \qquad \Gamma|\Theta \vdash P_2 : T_1 \to T \qquad \Gamma|\Theta \vdash P_3 : T_2 \to T}{\Gamma|\Theta \vdash \mathsf{case}\ P_1\ P_2\ P_3 : T}\ \textsc{Case}$$

Figure 1.24: Typing rules

contain free type variables, as this would interfere with the folding/unfolding of T'. This example of a diverging term $\bot$ can be generalized to a fixed-point
I.79, p.114 combinator which allows us to define recursive functions in our language. We refer to [108] for details and for further applications of recursive types.

We now define the type of program states. Intuitively, a program state $P|\sigma|\eta$ has type T under the environments Γ and Θ if P has type T, σ contains only values, and every value in σ has the type required by Θ.

I.165, p.127 **Definition 1.25** (Type of Program State)**.** *A program state $P|\sigma|\eta$ has type T under the environments Γ and Θ, iff $\Gamma|\Theta \vDash P|\sigma|\eta : T$ can be inferred by the rule*

$$\frac{\Gamma|\Theta \vdash P : T \qquad |\Theta| = |\sigma| \qquad \sigma \subseteq V \qquad \forall n < |\Theta| \,.\, \Gamma|\Theta \vdash \sigma!n : \Theta!n}{\Gamma|\Theta \vDash P|\sigma|\eta : T}.$$

If a program state $P|\sigma|\eta$ has a type T under the environments Γ and Θ, the rules Loc0 and Loc1 in Figure 1.24 ensure that the state contains no dangling locations, i.e., for each location $\mathsf{loc}\, n$ in P and σ it holds that
I.93, p.117 $n < |\sigma|$. We call such a state *store closed*. Furthermore the rules Var0 and Var1 in Figure 1.24 ensure that P and all terms in σ have no free variables,
I.94, p.117 if the environment Γ is empty; i.e., the state is *variable closed*. We say that a state $P|\sigma|\eta$ is *fully closed*, iff it is store closed and variable closed.

I.95, p.117 **Definition 1.26** (Fully Closed)**.** *We call a program state $P|\sigma|\eta$* fully closed*, iff P and all elements of σ have no free variables, and for all locations $\mathsf{loc}\, n$ in P and σ, it holds $n < |\sigma|$.*

Lemma 1.27. *Let $P|\sigma|\eta$ be a program state, Θ an environment, and T a type. The following two properties hold:*

I.172, p.128 - Variable closed: *If $[]|\Theta \vdash P : T$, then $\mathcal{FV}(P) = \{\}$.*

I.175, p.128 - Fully closed: *If $[]|\Theta \vDash P|\sigma|\eta : T$, then $P|\sigma|\eta$ is fully closed.*

Given a program P that has type T under the environments Γ and Θ, it is easy to see that P also has type T under the weakened environments $\Gamma@\Gamma'$ and $\Theta@\Theta'$.

I.176, p.128 **Lemma 1.28** (Weakening)**.** *Let P be a program, T be a type, and Γ, Γ', Θ, and Θ' be type environments. If $\Gamma|\Theta \vdash P : T$, then also $\Gamma@\Gamma'|\Theta@\Theta' \vdash P : T$.*

1.8.1 Typing Contexts

When typing the program $C[P]$, where a program P has been inserted into a context C, the type of $C[P]$ depends on the type of program P. Assuming that P has type T' under the environments Γ' and Θ', we can try to infer a type T for $C[P]$ under the environments Γ and Θ as follows: We apply the rules from Figure 1.24 in order to infer $\Gamma|\Theta \vdash C : T$ with the addition that, whenever we need to infer some type T'' for a hole $\square$ under some environments Γ'' and Θ'', we check whether $T'' = T'$, Γ' is a prefix of Γ'' and Θ' is a prefix of Θ''. Using Lemma 1.28 we know that in this case $\Gamma''|\Theta'' \vdash P : T''$.

We can use the observation from the previous paragraph and extend the typing relation $\vdash$ in order to give types to contexts C. For this, we define the relation $\Gamma'|\Theta'|T'|\Gamma|\Theta \vdash C : T$ which is defined analogously to the relation $\vdash$ from Figure 1.24 with the addition that it assumes that all holes $\square$ have type T' under the environments Γ' and Θ': I.181, p.128

$$\frac{\Gamma = \Gamma'@\Gamma'' \qquad \Theta = \Theta'@\Theta''}{\Gamma'|\Theta'|T'|\Gamma|\Theta \vdash \square : T'} \textsc{Hole}$$

The other rules are as in Figure 1.24: They are independent of Γ', Θ' and T' and just carry them over, e.g., the rules for λ-abstractions and applications are as follows:

$$\frac{\Gamma'|\Theta'|T'|T_1::\Gamma|\Theta \vdash C : T_2}{\Gamma'|\Theta'|T'|\Gamma|\Theta \vdash \lambda C : T_1 \to T_2} \textsc{Lam}$$

$$\frac{\Gamma'|\Theta'|T'|\Gamma|\Theta \vdash C_1 : T_1 \to T_2 \qquad \Gamma'|\Theta'|T'|\Gamma|\Theta \vdash C_2 : T_1}{\Gamma'|\Theta'|T'|\Gamma|\Theta \vdash C_1 C_2 : T_2} \textsc{App}$$

Assuming a context C with $\Gamma'|\Theta'|T'|\Gamma|\Theta \vdash C : T$, the rule Hole requires that all holes in context C have the type T' under the environments Γ' and Θ'. Therefore we can insert any program P with $\Gamma'|\Theta' \vdash P : T'$ into context C with the result that $C[P]$ has type T under the environments Γ and Θ. Note that if the environments Γ and Θ are empty, this furthermore implies that the state $C[P]||[]|[]$ is fully closed. I.183, p.129

Lemma 1.29. *Let P be a program, C be a context, T and T' be types, and Γ, Γ', Θ, and Θ' be type environments. Given that $\Gamma'|\Theta' \vdash P : T'$ and $\Gamma'|\Theta'|T'|\Gamma|\Theta \vdash C : T$, it holds $\Gamma|\Theta \vdash C[P] : T$.* I.182, p.129

1.9 Embedding the Type System in HOL

In Section 1.8 we presented the typing rules of the language as they are implemented in Isabelle/HOL. The implementation allows for using the inference rules of the typing relation $\vdash$ to prove that some program P has

a certain type T. While this proof has to be done interactively, Isabelle provides automatic type inference for its own higher-order logic (HOL); i.e., it is not even possible to write an ill-typed HOL term without being rejected by Isabelle's type checker. For example, it is not possible to write the application $f\,f$ in HOL, since f cannot have type $\alpha \to \beta$ and type α simultaneously (See Section 1.4.1 for a brief review of higher-order logic). Yet, writing the application PP for some program P in our language is not rejected by Isabelle: The application PP of two (equal) programs is a valid program term, even though it is not typeable with respect to the typing relation $\vdash$.

In the following section we depict how our typing system can be embedded into HOL. Such an embedding allows us to benefit from Isabelle's automatic type checking, because it prevents us from accidentally writing ill-typed programs. Moreover, the embedding permits more crisp formulations of theorems and definitions, because we do not need to specify all the side conditions that require the used programs to be well-typed. Instead, these conditions come for free. We proceed by showing how our programs can be embedded into this logic.

1.9.1 Embedding Programs into HOL

After implementing programs as described in Definition 1.10 in HOL, all programs are of the same HOL-type. When writing down a program immediately in HOL, we have to explicitly ensure that the program is well-typed with respect to the typing relation $\vdash$, i.e., we lose the advantage of automatic type inference and type checking as supported in HOL. To leverage the power of HOL to our programs, we define the type classes $\mathsf{program\_type}$ and $\mathsf{environment}$, and a new type $\mathsf{program}$ in order to embed our notion of types, environments and programs into HOL:

Embedding Types

I.184, p.129 The class $\mathsf{program\_type}$ for a HOL type α introduces the constant $\mathsf{prog\_type}_\alpha$ of type $\mathcal{T}$, where $\mathcal{T}$ denotes the HOL type of the program types in our language. Furthermore, the class has the constraint that the type $\mathsf{prog\_type}_\alpha$ must by inhabited, i.e., there must exist a program P such that $[]|[] \vdash$ I.219, p.132 $P : \mathsf{prog\_type}_\alpha$. E.g., for the unit set we define $\mathsf{prog\_type}_{\mathbb{U}} \stackrel{\text{def}}{=} \mathsf{Unit}$, for I.228, p.132 Booleans we define $\mathsf{prog\_type}_{\mathbb{B}} \stackrel{\text{def}}{=} \mathsf{Bool}$, and for natural numbers we define I.233, p.132 $\mathsf{prog\_type}_{\mathbb{N}} \stackrel{\text{def}}{=} \mathsf{Nat}$. Given types α and β of type class $\mathsf{program\_type}$, we define I.195, p.130 $\mathsf{prog\_type}_{\alpha\,\mathsf{list}} \stackrel{\text{def}}{=} \mathsf{List}\,\mathsf{prog\_type}_\alpha$ and $\mathsf{prog\_type}_{\alpha\sim\beta} \stackrel{\text{def}}{=} \mathsf{prog\_type}_\alpha \sim \mathsf{prog\_type}_\beta$, I.193, p.130 where $\sim \in \{\times, +, \to\}$. Furthermore we introduce a (dummy) HOL type I.197, p.130 $\alpha\,\mathsf{ref}$ and define $\mathsf{prog\_type}_{\alpha\,\mathsf{ref}} \stackrel{\text{def}}{=} \mathsf{Ref}\,(\mathsf{prog\_type}_\alpha)$.

Embedding Environments

The type class environment embeds the notion of type environments. Since environments are lists of program types, the class introduces the constant env_types of type $\mathcal{T}$ list. We model two instances of environment. First we define env_nil which embeds empty environments and hence has env_types implemented as the empty list []. Second, to build non-empty environments, we define the type (α, γ)env_cons, for which env_types is implemented as $\mathsf{prog\_type}_\alpha$::$\mathsf{env\_types}_\gamma$, where α is of type class program_type and γ is of type class environment. I.185, p.129 I.187, p.129 I.188, p.130 I.189, p.130 I.190, p.130

Embedding Programs

We now define the main type of the embedding. The type (γ, α)program represents the set of all programs that have type $\mathsf{prog\_type}_\alpha$ under the variable type environment $\mathsf{env\_types}_\gamma$ and the empty store type environment. It is defined as the set $\{P \mid \mathsf{env\_types}_\gamma | [] \vdash P : \mathsf{prog\_type}_\alpha\}$. Here γ is of type class environment and α is of type class program_type. I.191, p.130

Using the HOL type (γ, α)program it is possible to define typed program term constructors. For instance we can define the constant VAR0, whose representation is the variable var 0. This variable representation only types in non-empty environments and has the first element in the environment as type. In particular, the HOL type of VAR0 is $((\alpha, \gamma)\mathsf{env\_cons}, \alpha)\mathsf{program}$. For variables representing larger de Bruijn indices, we introduce the function VAR_SUC, which given a variable representing index k returns a representation of a variable for the succeeding index $k + 1$; i.e., given a variable representation of type (γ, β)program, it returns a variable representation of type $((\alpha, \gamma)\mathsf{env\_cons}, \beta)\mathsf{program}$. Similarly we can define a constant ABSTRACT that embeds the λP construct. It expects an argument P that types in a non-empty environment, i.e., P should have the type $((\alpha, \gamma)\mathsf{env\_cons}, \beta)\mathsf{program}$, and has the return type $(\gamma, \alpha \to \beta)$program. The constant APPLY takes two programs of types $(\gamma, \alpha \to \beta)$program and (γ, α)program and represents their application of type (γ, β)program. Likewise we define a constructor PAIR which takes two arguments of type (γ, α)program and (γ, β)program and returns a program of type $(\gamma, \alpha \times \beta)$program. When applied to such programs, the destructors FST and SND again return the respective components of type (γ, α)program and (γ, β)program. Analogously for sum types and references, we define constants INL, INR, CASE, REF, DEREF and ASSIGN. Using this paradigm it is even possible to define constructs that are not present in the underlying language. For instance we can define a program NIL of type $(\gamma, \alpha\,\mathsf{list})$program which represents empty lists, and a corresponding list constructor CONS of type $(\gamma, \alpha)\mathsf{program} \to (\gamma, \alpha\,\mathsf{list})\mathsf{program} \to (\gamma, \alpha\,\mathsf{list})\mathsf{program}$. Furthermore it is possible to define typed variants of the operators $\uparrow\!\!_k$ and $\{\cdot/\!\!_k\}$. In particular, the operator LIFT represents the oper- I.238, p.132 I.239, p.132 I.240, p.133 I.241, p.133 I.242, p.133 I.243, p.133 I.244, p.133 I.255, p.134 I.256, p.134 I.262, p.134

ation $\uparrow_0$ which lifts all free variables and hence has the type $(\gamma,\beta)\mathsf{program} \to$ I.264, p.134 $((\alpha,\gamma)\mathsf{env\_cons},\beta)\mathsf{program}$; the operator SUBSTITUTE represents the operation $\{\cdot/_0\}$ and has type $((\alpha,\gamma)\mathsf{env\_cons},\beta)\mathsf{program} \to (\gamma,\alpha)\mathsf{program} \to (\gamma,\beta)\mathsf{program}$. Exploiting these newly introduced constants for writing programs, the type-checker of Isabelle will ensure that we cannot write ill-typed programs.

In the present section, we have shown how to embed programs as defined in Section 1.6 into the type system of HOL. That is, we have defined a single HOL type $\mathcal{P}$ containing all programs, then we have defined a type system $\vdash$ on $\mathcal{P}$, and finally we have defined the HOL-type $(\gamma,\alpha)\mathsf{program}$ as the type of all programs of type $\mathsf{prog\_type}_\alpha$. The question arises whether it is necessary to perform this three-step approach. Instead, one might want to directly define a HOL-type $\alpha\,\mathsf{program}$ of programs of type α in terms of smaller types. For example, the type $\alpha\times\beta\,\mathsf{program}$ might be defined as the type containing all terms (P_1,P_2) with P_1 having type $\alpha\,\mathsf{program}$ and P_2 having type $\beta\,\mathsf{program}$. Unfortunately, this approach does not work as, e.g., the type $\alpha\times\beta\,\mathsf{program}$ also has to contain program applications P_1P_2 with P_1 of type $\gamma \to \alpha\times\beta\,\mathsf{program}$ and P_2 of type $\gamma\,\mathsf{program}$ (for any type γ). Hence the definition of the HOL-type $\alpha\times\beta\,\mathsf{program}$ has to depend (i) on an infinite number of other types and (ii) on larger types. Such constructions are not supported by HOL and would need much more elaborate (and thus more complicated) type systems than HOL.

Stretching the Limitations of the Embedding

Note that the operators LIFT and SUBSTITUTE only operate on the de Bruijn I.263, p.134 index 0. We can also define more general operators LIFTk and SUBSTITUTEk I.265, p.134 which take an additional argument k, but here we reach a limitation in the capabilities of HOL: For the lifting of all variables with index $\geq k$ we would like to specify the type

$$\begin{aligned}&((\alpha_1,\ldots(\alpha_n,\gamma)\mathsf{env\_cons})\ldots\mathsf{env\_cons},\beta)\mathsf{program}\\ \to&((\alpha_1,\ldots(\alpha_{k-1},(\alpha,(\alpha_k,\ldots(\alpha_n,\gamma)\mathsf{env\_cons})\ldots\mathsf{env\_cons},\beta)\mathsf{program},\end{aligned} \tag{1.30}$$

where we insert an α of type class $\mathsf{program\_type}$ at the k-th position into the environment of the return type of LIFTk. But this means that the type of the operator LIFTk depends on the *value* of its argument k. Such dependent types are not supported by HOL. Instead we have to sacrifice type precision and specify the general type $\mathsf{nat} \to (\gamma_1,\beta)\mathsf{program} \to (\gamma_2,\beta)\mathsf{program}$ for LIFTk. The same problem occurs for SUBSTITUTEk for which we specify the general type $\mathsf{nat} \to (\gamma_1,\beta_1)\mathsf{program} \to (\gamma_2,\beta_2)\mathsf{program} \to (\gamma_2,\beta_1)\mathsf{program}$.

Because of this loss of precision, the automatic type inference of Isabelle might infer too general type constraints for statements involving these operators, in which case the statements may become unprovable. If a state-

ment only deals with concrete values of k, we can resolve this problem by adding the precise type annotations to the statement. However, for statements involving arbitrary operator arguments k, such type annotations are not possible as this would require dependent types. E.g., for any concrete value k we can prove the lemma $\textsc{LiftK}\,k\,\textsc{Var}_k = \textsc{Var}_{k+1}$, where $\textsc{Var}_k$ and $\textsc{Var}_{k+1}$ represent the variables $\mathsf{var}\,k$ and $\mathsf{var}\,(k+1)$ respectively. This is possible, because here we can annotate LiftK with the respective type as in Equation 1.30. However, the statement $\forall k.\,\textsc{LiftK}\,k\,\textsc{Var}_k = \textsc{Var}_{k+1}$ is unprovable in HOL as this would require a dependent type annotation for LiftK.

We mitigate this problem as follows: For concrete values of k, proofs of lemmas of the form $\textsc{LiftK}\,k\,\textsc{Var}_k = \textsc{Var}_{k+1}$ all follow the same design pattern. Therefore it is possible to implement a simplification procedure in Isabelle that automatically generates proofs for lemmas of this form for arbitrary k. Therefore, even though we cannot prove the general statement $\forall k.\,\textsc{LiftK}\,k\,\textsc{Var}_k = \textsc{Var}_{k+1}$, the simplification machinery of Isabelle behaves as if such a lemma had been proven.

We have implemented similar procedures for the cases where LiftK is applied to other typed programs built from constants like Abstract or Apply as introduced above. For the operator SubstituteK we have implemented similar simplification procedures as well. The result is that Isabelle can simplify statements involving the (precisely) typed operators LiftK and SubstituteK just as it can simplify corresponding statements involving the untyped operators $\Uparrow_k$ and $\{\cdot/_k\}$.

1.9.2 Embedding Values into Programs

So far we have seen how to embed our language into the type system of HOL. Conversely, it is also possible to embed objects from HOL into our language. For this we introduce another type class $\mathsf{embeddable}$ which introduces a constant $\mathsf{prog\_embedding}_\alpha$ of type $\alpha \to \mathcal{P}$, where $\mathcal{P}$ denotes the HOL type of the program terms of our language. Furthermore we require that such embeddings are well-typed with respect to $\vdash$, i.e., we require for any x of type α that $[]|[] \vdash \mathsf{prog\_embedding}_\alpha(x) : \mathsf{prog\_type}_\alpha$. Note that it is straightforward to embed the typed variant of our language back into our language given an empty environment. For this we simply define $\mathsf{prog\_embedding}_{(\mathsf{env\_nil},\alpha)\mathsf{program}}(P)$ as the program that P represents in our language and let $\mathsf{prog\_type}_{(\mathsf{env\_nil},\alpha)\mathsf{program}} \stackrel{\text{def}}{=} \mathsf{prog\_type}_\alpha$. I.199, p.130 I.207, p.131 I.206, p.131

We introduce another type class to model the embedding of HOL values into the language. The class $\mathsf{embeddable\_val}$ adds the constraint that for all x the embedding $\mathsf{prog\_embedding}_\alpha(x)$ should be a value. There are two conceptually different ways to embed HOL values into our language: I.200, p.130

First, we can use the construct $\mathsf{value}\,v$ to embed basic values $v \in \mathbf{B}$ into the language. In combination with the constructors for pairs, sums, and

recursive types, we can then also embed compound values like pairs, sums, Booleans, lists, and natural numbers. We introduce a special type class
I.201, p.130 embeddable_pure for such types that are compounds built from basic values. This class also assumes a constant $\mathsf{inv\_prog\_embedding}_\alpha$ of type $\mathcal{P} \to \alpha$ which is the inverse of $\mathsf{prog\_embedding}_\alpha$. Furthermore this class assumes that $\mathsf{prog\_embedding}_\alpha(x)$ is a pure value $\in V_0$ for any x. Also we require $\mathsf{prog\_embedding}_\alpha$ and $\mathsf{inv\_prog\_embedding}_\alpha$ to be measurable.

Second, we can use the $\mathsf{fun}(\cdot,\cdot)$ construct and even embed higher-order functions $f : \alpha_1 \to \ldots \to \alpha_n$ from HOL into our language where the types
I.209, p.131 α_i are of type class embeddable_pure. For this, let $\mathsf{kernel\_of}(f)$ be the deterministic kernel of the uncurried form of f, i.e., $\mathsf{kernel\_of}(f)$ is a kernel from $\alpha_1 \times \ldots \times \alpha_{n-1}$ to α_n. We embed f into our language using the construct
I.210, p.131 $\mathsf{prog\_embedding}(f) \stackrel{\text{def}}{=} \underbrace{\lambda \ldots \lambda}_{n-1 \text{ times}} \mathsf{fun}(\mathsf{kernel\_of}(f), (\mathsf{var}\,(n-2), \ldots, \mathsf{var}\,0))$.

1.9.3 Embedding Contexts into HOL

Similarly to the techniques from Section 1.9.1, we can also define a HOL type consisting of all well-typed contexts according to the $\vdash$ relation as
I.300, p.137 introduced in Section 1.8.1: We define the type $(\gamma', \alpha', \gamma, \alpha)\mathsf{typed\_context}$ as the set $\{C \mid \mathsf{env\_types}_{\gamma'}|[]|\mathsf{prog\_type}_{\alpha'}|\mathsf{env\_types}_\gamma|[] \vdash C : \mathsf{prog\_type}_\alpha\}$.

As we have seen in Lemma 1.29, if we insert a $(\gamma', \alpha')\mathsf{program}$ into a $(\gamma', \alpha', \gamma, \alpha)\mathsf{typed\_context}$, the result will be a $(\gamma, \alpha)\mathsf{program}$. Therefore, such a context behaves like a function of type $(\gamma', \alpha')\mathsf{program} \to (\gamma, \alpha)\mathsf{program}$. The definition of context functions captures this observation:

I.302, p.137 **Definition 1.31** (Typed Context Function)**.** *Given a function F of type $(\gamma', \alpha')\mathsf{program} \to (\gamma, \alpha)\mathsf{program}$, we call F a (typed)* context function*, iff there exists a $(\gamma', \alpha', \gamma, \alpha)\mathsf{typed\_context}$ C, such that $F(P) = C[P]$ for all P of type $(\gamma', \alpha')\mathsf{program}$.*

Context functions and their corresponding contexts are closely related, but context functions have one big advantage over their corresponding counterparts: Isabelle has a built-in unification machinery which can match functional symbols against subgoals when applying tactics. Using context functions we can take advantage of this unification implemented in Isabelle. Instead of having to construct cumbersome contexts by hand while conducting a proof, we can let Isabelle automatically infer the corresponding context functions. This allows for much shorter and more direct formalizations of proof scripts.

Since not all functions of type $(\gamma', \alpha')\mathsf{program} \to (\gamma, \alpha)\mathsf{program}$ are context functions, the unification as explained above can also infer functions that are not context functions. In order to guide the unification when applying a tactic, we have implemented a series of introductory rules. E.g.,
I.303, p.137 we have shown that the identity function is a context function; the cor-

responding context is the hole $\Box$. Also the constant function $\lambda x.\, P$ is a context function. Let F_1 and F_2 be context functions; we can show that $\lambda P.\, \textsc{Abstract}(F_1(P))$, and $\lambda P.\, \textsc{Apply}(F_1(P))(F_2(P))$ are also context functions. We have shown analogous results concerning the other constants Pair, Fst, etc., that were introduced in Section 1.9.1 as well. I.304, p.137 I.305, p.137 I.306, p.138

1.9.4 Syntactic Sugar

The HOL-embedding explained above offers automatic type checking, but it is still not very convenient to actually write programs in this language. For example, one does not want to write Abstract Var0 for the identity function. We overcome this problem by implementing parse and print translations in Isabelle; this allows us to program in an ML-style syntax. We are able to hide the de Bruijn implementation of variables and use named abstractions instead. For example the identity function above is written as $\ddot{\lambda} x.\, x\ddot{}$. We are also able to express syntactic sugar for constructs that are not present in the underlying language. E.g., we introduce the syntax of a let construct $\ddot{}\mathsf{let}\ x_1 \leftarrow P_1\ \mathsf{in}\ P\ddot{}$ as a synonym for $\ddot{}(\lambda x.\, P)P_1\ddot{}$, together with a sequenced variant $\ddot{}\mathsf{let}\ x_1 \leftarrow P_1; \ldots x_n \leftarrow P_n\ \mathsf{in}\ P\ddot{}$ which is a shorthand for $\ddot{}\mathsf{let}\ x_1 \leftarrow P_1\ \mathsf{in}\ \ldots\ \mathsf{let}\ x_n \leftarrow P_n\ \mathsf{in}\ P\ddot{}$. Furthermore we introduce pattern matching for tuples in order to write, e.g., $\ddot{\lambda}(x, y).\, x\ddot{}$, and an equality symbol $=$ that is the program embedding of the equality in HOL. The quotes in $\ddot{}P\ddot{}$ inform Isabelle that P should be parsed as a program and not as a mathematical expression. We also introduce the antiquotations $:P:$ and $\widehat{v}$. The syntax $:P:$ tells Isabelle that P is to be parsed using the normal mathematical syntax and is supposed to evaluate to a closed program, i.e., $:P:$ is of type $(\mathsf{env\_nil}, \alpha)\mathsf{program}$. The syntax $\widehat{v}$ denotes the program representing $\mathsf{prog\_embedding}(v)$, i.e., the embedding of the HOL value v into the language. Finally, to capture a common pattern in cryptographic definitions, for Boolean programs P of type $(\mathsf{env\_nil}, \mathbb{B})\mathsf{program}$ we define $\Pr[P]$ as the probability that P reduces to true.

$$\Pr[P] \stackrel{\text{def}}{=} [\![\mathsf{prog\_embedding}(P)|[]|[]]\!]\ \{(\mathsf{inl}\, P'|\sigma|\eta) \mid (P'|\sigma|\eta) \in \Omega\} \qquad \text{I.268, p.135}$$

1.10 Program Relations

Game-based proofs are conducted by transforming games, i.e., programs, such that a game and its transformation are in some sense equivalent or indistinguishable. In this section, we show how several such relations can be formalized in our framework. We start with the notion of denotational equivalence, proceed with the notion of observational equivalence, and by defining polynomial-time programs we conclude with the notion of computational indistinguishability.

1.10.1 Denotational Equivalence

Among the relations we consider, denotational equivalence constitutes the strongest such relation. Two programs are *denotationally equivalent*, if their denotations are the same for all stores and event lists.

I.321, p.140 **Definition 1.32** (Denotational Equivalence)**.** *Two programs P_1 and P_2 are called* denotationally equivalent*, iff for all stores σ and event lists η it holds $[\![P_1|\sigma|\eta]\!] = [\![P_2|\sigma|\eta]\!]$.*

This relation is too strong in many cases, since many game transformations do not preserve the denotation completely and introduce small errors. Consider for example two programs P_1 and P_2, where P_1 selects a bitstring uniformly at random from $\{0,1\}^n$ and P_2 from $\{0,1\}^n \setminus \{0^n\}$. It is clear that P_1 and P_2 are denotationally different, but their statistical difference is very small. The following definition introduces the notion of denotational equivalence up to some error.

I.322, p.140 **Definition 1.33** (Denotational Equivalence up to Error)**.** *Two programs P_1 and P_2 are called* denotationally equivalent up to error ϵ*, iff for all stores σ and event lists η it holds*

$$\max_{S \in \Sigma} \{|[\![P_1|\sigma|\eta]\!](S) - [\![P_2|\sigma|\eta]\!](S)|\} \leq \epsilon,$$

where Σ is the canonical σ-algebra over the set of program states.

Denotational equivalence is also too strong from another perspective: Since it compares programs based on the distribution over program states they compute, these states also contain store allocations that may be inaccessible after the execution. Consequently, two programs that compute the same function are denotationally different if, e.g., the first program uses references while the second one does not. Moreover the terms in the distributions have to be equal *syntactically*. Consider the values $\lambda x.\,(\lambda y.\,y)x$ and $\lambda x.\,x$. Both compute the identity function, but they are denotationally different. To overcome these problems, we introduce the notion of observational equivalence.

1.10.2 Observational Equivalence

Instead of comparing programs based on their denotation, we can also compare them based on their behavior. This yields the notion of *observational equivalence*. The idea is that when used as part of a larger program P, two observationally equivalent programs should be replaceable with each other without affecting the computation of P, since it is impossible for P to observe which program it contains. Two programs P_1 and P_2 are observationally equivalent, if their behavior is equivalent in every context C and

state $\cdot|\sigma|\eta$. Here we restrict the set of contexts to those that capture all free variables of P_1 and P_2 and that do not contain locations outside the store. More formally, we require $C[P_1]|\sigma|\eta$ and $C[P_2]|\sigma|\eta$ to be fully closed. If these states terminate with the same probability, we call the programs observationally equivalent.

Definition 1.34 (Observational Equivalence)**.** *Two programs P_1 and P_2 are* observationally equivalent*, iff for all contexts C, stores σ, and event lists η such that $C[P_1]|\sigma|\eta$ and $C[P_2]|\sigma|\eta$ are fully closed, we have that $[\![C[P_1]|\sigma|\eta]\!](\Omega) = [\![C[P_2]|\sigma|\eta]\!](\Omega)$. Here Ω is the set of all program states. We write $P_1 \approx_{obs} P_2$ to denote that programs P_1 and P_2 are observationally equivalent.* I.336, p.141

Observational equivalence is the main relation that we will use when connecting games in individual steps of a game-based proof. Such a sequence of steps allows us to deduce the observational equivalence of the initial and the final game. This is possible because the relation $\approx_{obs}$ is transitive. More precisely, $\approx_{obs}$ is an equivalence relation.

Lemma 1.35. *The relation $\approx_{obs}$ is an equivalence relation.* I.347, p.142

Proof. Reflexivity and symmetry follow directly from Definition 1.34. To prove transitivity, assume three programs P_1, P_2, and P_3 where $P_1 \approx_{obs} P_2$ and $P_2 \approx_{obs} P_3$. Furthermore let C be a context, σ a store, and η and event list such that $C[P_1]|\sigma|\eta$ and $C[P_3]|\sigma|\eta$ are fully closed. We need to show that $[\![C[P_1]|\sigma|\eta]\!](\Omega) = [\![C[P_3]|\sigma|\eta]\!](\Omega)$.

Note that we cannot use Definition 1.34 directly to prove the transitivity of $\approx_{obs}$, since $C[P_2]|\sigma|\eta$ is not necessarily fully closed. In particular, $C[P_2]$ might contain free variables and locations $\mathsf{loc}\, n$ with $n \geq |\sigma|$. Let i be the maximal de Bruijn index such that $\mathsf{var}\,(i-1)$ is free in $C[P_2]$ (or $i = 0$ if no such i exists) and let l be the maximal location index such that $\mathsf{loc}\,(l-1)$ occurs in $C[P_2]$ (or $l = 0$ if no such l exists). We consider the state $C_i[C[P_2]]|\sigma@\sigma'|\eta$, where $\sigma' = [\mathsf{value}\, unit, \ldots, \mathsf{value}\, unit]$ such that $|\sigma@\sigma'| \geq l$ and C_i is defined recursively as $C_0 \stackrel{\text{def}}{=} \Box$ and $C_{i+1} \stackrel{\text{def}}{=} (\lambda C_i)(\mathsf{value}\, unit)$. This state is fully closed: The context C_i captures all free variables of $C[P_2]$ and for all locations $\mathsf{loc}\, n$ in $C[P_2]$ it holds $n < |\sigma@\sigma'|$. Furthermore the states $C_i[C[P_1]]|\sigma@\sigma'|\eta$ and $C_i[C[P_3]]|\sigma@\sigma'|\eta$ are fully closed, too. Since $P_1 \approx_{obs} P_2$ and $P_2 \approx_{obs} P_3$, it holds

$$[\![C_i[C[P_1]]|\sigma@\sigma'|\eta]\!](\Omega) = [\![C_i[C[P_2]]|\sigma@\sigma'|\eta]\!](\Omega) = [\![C_i[C[P_3]]|\sigma@\sigma'|\eta]\!](\Omega).$$

We can reduce the state $C_i[C[P_1]]|\sigma@\sigma'|\eta$ by applying the rule Beta i times. Since $C[P_1]$ has no free variables, it holds $step^i(C_i[C[P_1]]|\sigma@\sigma'|\eta) = \delta(C[P_1]|\sigma@\sigma'|\eta)$, and hence $[\![C_i[C[P_1]]|\sigma@\sigma'|\eta]\!](\Omega) = [\![C[P_1]|\sigma@\sigma'|\eta]\!](\Omega)$.

Since extending the store of a fully closed state does not affect its probability of termination, we also have $[\![C[P_1]|\sigma@\sigma'|\eta]\!](\Omega) = [\![C[P_1]|\sigma|\eta]\!](\Omega)$.

Analogously for $C[P_3]$, it holds $[\![C_i[C[P_3]]|\sigma@\sigma'|\eta]\!](\Omega) = [\![C[P_3]|\sigma|\eta]\!](\Omega)$. We conclude as follows:

$$\begin{aligned}[\![C[P_1]|\sigma|\eta]\!](\Omega) &= [\![C_i[C[P_1]]|\sigma@\sigma'|\eta]\!](\Omega) \\ &= [\![C_i[C[P_3]]|\sigma@\sigma'|\eta]\!](\Omega) \\ &= [\![C[P_3]|\sigma|\eta]\!](\Omega)\end{aligned}$$ □

In a game-based proof, one often modifies only a small part of the game in each step. In our setting this corresponds to replacing a program P_1 in a game $C[P_1]$ with another program P_2 where C can be an arbitrary context. If $P_1 \approx_{obs} P_2$ then it also holds $C[P_1] \approx_{obs} C[P_2]$. This is an important property of observational equivalence, as it allows us to focus on the programs P_1 and P_2 rather than to deal with the possibly much larger games $C[P_1]$ and $C[P_2]$. As explained in Section 1.9.3, this process can be even be further simplified by using context functions, which Isabelle's unification machinery can automatically infer.

I.349, p.142 **Lemma 1.36** (Composability of $\approx_{obs}$)**.** *Assume programs P_1, P_2 and a context C. If $P_1 \approx_{obs} P_2$, then also $C[P_1] \approx_{obs} C[P_2]$.*

Proof. Let C' be a context, σ a store, and η an event list such that the states $C'[C[P_1]]|\sigma|\eta$ and $C'[C[P_2]]|\sigma|\eta$ are fully closed. Consider the context $C_0 \stackrel{\text{def}}{=} C'[C]$. It holds

$$\begin{aligned}[\![C'[C[P_1]]|\sigma|\eta]\!](\Omega) &= [\![C_0[P_1]|\sigma|\eta]\!](\Omega) \\ &= [\![C_0[P_2]|\sigma|\eta]\!](\Omega) && \text{(since } P_1 \approx_{obs} P_2\text{)} \\ &= [\![C'[C[P_2]]|\sigma|\eta]\!](\Omega)\end{aligned}$$ □

Cryptographic games often return a Boolean value which is used to determine whether an adversary was successful in attacking a system. Therefore we are interested in the probability $\Pr[P]$ of a game P returning true. Observational equivalence preserves this probability and hence can be used to relate the probabilities of programs P_1 and P_2 returning true.

I.340, p.141 **Lemma 1.37.** *Assume programs P_1 and P_2 without free variables and locations such that $P_1 \approx_{obs} P_2$. Then it holds $\Pr[P_1] = \Pr[P_2]$.*

Proof. The expression $\Pr[P]$ denotes the probability of true in the denotation of a program P, while observational equivalence is about the termination probability of $C[P]$ for contexts C. In order to relate $\Pr[P]$ and observational equivalence, we construct a context C for which $C[P]$ terminates if and only if P returns a value $\mathsf{inl}\, V$, i.e, the value true assuming that P returns a Boolean. We use the diverging term $\bot$ that was introduced in Section 1.8 and define the context

$$C \stackrel{\text{def}}{=} \mathsf{case}\ \square\ (\lambda \mathsf{inl}\, (\mathsf{var}\, 0))\ (\lambda \bot).$$

For values $\mathsf{inl}\, V$, the denotation $[\![C[\mathsf{inl}\, V]|\sigma|\eta]\!]$ is $\delta(\mathsf{inl}\, V|\sigma|\eta)$, while for all other values V it holds $[\![C[V]|\sigma|\eta]\!] = 0$. This means that the context C behaves like the restriction kernel $\downarrow_{\{(\mathsf{inl}\, P'|\sigma|\eta)|(P'|\sigma|\eta)\in\Omega\}}$. Using this observation, we can show that for program $P \in \{P_1, P_2\}$ it holds

$$\begin{aligned}[\![C[P]|[]|[]]\!]\,\Omega &\overset{(*)}{=} (\downarrow_{\{(\mathsf{inl}\, P'|\sigma|\eta)|(P'|\sigma|\eta)\in\Omega\}} \cdot [\![P|[]|[]]\!])\Omega \\ &= [\![P|[]|[]]\!]\,\{(\mathsf{inl}\, P'|\sigma|\eta) \mid (P'|\sigma|\eta) \in \Omega\} \\ &= \Pr[P],\end{aligned}$$

where $(*)$ uses a chaining property of the denotation which we will prove later in Theorem 1.48.

Since P_1 and P_2 contain no free variables and locations, the program states $C[P_1]|[]|[]$ and $C[P_2]|[]|[]$ are fully closed. From $P_1 \approx_{obs} P_2$ it follows that $[\![C[P_1]|[]|[]]\!]\,\Omega = [\![C[P_2]|[]|[]]\!]\,\Omega$ and hence $\Pr[P_1] = \Pr[P_2]$. □

1.10.3 Polynomial Runtime

Cryptographic proofs often only assert security if the runtime of the adversary is bounded polynomially in the security parameter. In this section, we give a formal definition of such polynomial-time programs in our language. Since it is unclear what polynomial time means for programs handling non-computational objects such as reals or arbitrary kernels, we exclude such programs from our definition of polynomial-time programs. More precisely, we only allow programs handling *unit*, Booleans (or bits), and kernels performing bit-operations. Furthermore, the notion of polynomial-time programs shall not depend on whether events have been raised or not.

Definition 1.38 ((Non)Computational/Eventless)**.** *A program is a* non-computational atom *iff it is of the form* $\mathsf{eventlist}$*,* $\mathsf{value}\, v$*, where* $v \neq unit$*, or* $\mathsf{fun}(f, P)$*, where* f *does not compute a coin-toss or one of the bit-operations* $\neg, \wedge, \vee$*. A program is called* computational*, iff it does not contain any non-computational atoms. A program is called* eventless*, iff it does not contain any programs of the form* $\mathsf{eventlist}$ *or* $\mathsf{event}\, s$*.* I.374, p.144 I.375, p.144 I.377, p.144

Since the definition of polynomial-time programs should be able to deal with oracles, we have to exclude the time that is spent for executing the oracles, i.e., the notion of polynomial-time shall not depend on the running-time of the oracles the program under consideration calls. We express this by transforming the program such that it raises a distinguished *step event* for every step it takes. If the oracles do not raise events themselves, the running time of the program is defined as the number of step events it raises. Given a program P, the step-annotated program $P!$ is constructed by replacing every sub-term t of P by $(\lambda \Uparrow_0 t)(\mathsf{event}\, step)$. I.360, p.143

In cryptographic games, one often calls adversaries by giving them the security parameter k as a first argument in unary form and then requires

them to be polynomial-time in the size of this input. Such adversaries are called *uniform*, because they use the same implementation for all security parameters. In contrast, a *non-uniform* adversary $\mathcal{A} = \{\mathsf{A}_k\}_{k\in\mathbb{N}}$ is a family of programs and uses a (possibly) different implementation A_k for each security parameter k. In cryptographic games involving non-uniform adversaries, it is not necessary to supply the security parameter as an argument, as one can call the respective implementation A_k directly, instead.

In the uniform case, if a program takes a pair (V_0, P_1) as argument, we define polynomial-time in the size $|V_0|$ of the first component. Here V_0 is a pure value, e.g., the security parameter. The argument P_1 may contain other arguments that potentially include oracles. A program is defined to be polynomial-time if the number of step events it raises while running $P!$ on (V_0, P_1) is polynomial in $|V_0|$.

I.379, p.144 **Definition 1.39** (Uniform Polynomial-time)**.** *A program P of type $(T_0 \times T) \to U$ is* (uniform) polynomial-time*, iff it is computational, T_0 is a pure type, and there is a polynomial q such that for all inputs (V_0, P_1) where V_0 is of type T_0 and P_1 of type T is eventless, it holds*

$$\forall n.\, \forall (P'|\sigma|\eta) \leftarrow step^n(P!\,(V_0, P_1)|[]|[]).\, \#_{step}(\eta) \leq q(|V_0|),$$

where $\#_{step}(\eta)$ denotes the number of step events in the event list η.

For programs of type $T_0 \to U$ that should be polynomial-time in the size of their input, where T_0 is a pure type (i.e., they do not use oracles), we define the notion of first-order polynomial-time: A program P of type
I.380, p.145 $T_0 \to U$ is *first-order polynomial time*, iff the program $\lambda x.(\Uparrow_0 P)(\mathsf{fst}\, x)$ is
I.381, p.145 polynomial-time. We call a program an *efficient algorithm*, iff it is first-order polynomial time, eventless and does not use references. A function f
I.382, p.145 is *efficiently computable* if there is an efficient algorithm P, such that P and the embedding $\widehat{f}$ of the function are observationally equivalent.

In the non-uniform case, we define polynomial-time analogously to the uniform Definition 1.39. The main difference is that the allowed runtime does not depend on the first argument given to the program. Instead, the runtime depends on the choice of the respective implementation from the program family.

I.383, p.145 **Definition 1.40** (Non-uniform Polynomial-time)**.** *A family of programs $\mathcal{P} = \{P_k\}_{k\in\mathbb{N}}$ of type $T \to U$ is* (non-uniform) polynomial-time*, iff each P_k is computational and there is a polynomial q such that for all k and inputs P where P is of type T and eventless, it holds*

$$\forall n.\, \forall (P'|\sigma|\eta) \leftarrow step^n(P_k!\, P|[]|[]).\, \#_{step}(\eta) \leq q(k),$$

where $\#_{step}(\eta)$ denotes the number of step events in the event list η.

1.10.4 Computational Indistinguishability

We finally define the notion of computational indistinguishability for families of (closed) programs. Intuitively, two families $\{P_k\}_{k\in\mathbb{N}}$ and $\{P'_k\}_{k\in\mathbb{N}}$ are computationally indistinguishable if every polynomial-time program family $\mathcal{D}$ distinguishes P_k and P'_k with at most negligible probability in k. Here a function $f : \mathbb{N} \to \mathbb{R}$ is called negligible [81], if for every $c \in \mathbb{N}$ there is an $N \in \mathbb{N}$ such that $|f(n)| \leq \frac{1}{n^c}$ for all $n \geq N$. I.353, p.142

Definition 1.41 (Computational Indistinguishability)**.** *Given two families of (closed) programs $\mathcal{P} = \{P_k\}_{k\in\mathbb{N}}$ and $\mathcal{P}' = \{P'_k\}_{k\in\mathbb{N}}$, we call $\mathcal{P}$ and $\mathcal{P}'$* computationally indistinguishable*, written $\mathcal{P} \approx_{ind} \mathcal{P}'$, iff for all non-uniform polynomial-time families $\{D_k\}_{k\in\mathbb{N}}$ it holds that* I.384, p.145

$$\left|\Pr[D_k(P_k)] - \Pr[D_k(P'_k)]\right| \text{ is negligible in } k.$$

Lemma 1.42. *The relation $\approx_{ind}$ is an equivalence relation.* I.388, p.145

Proof. Symmetry holds trivially and reflexivity follows from the fact that the zero function $\lambda k.\, 0$ is negligible. To prove transitivity, assume three program families $\mathcal{P} = \{P_k\}_{k\in\mathbb{N}}$, $\mathcal{P}' = \{P'_k\}_{k\in\mathbb{N}}$, and $\mathcal{P}'' = \{P''_k\}_{k\in\mathbb{N}}$ where $\mathcal{P} \approx_{ind} \mathcal{P}'$ and $\mathcal{P}' \approx_{ind} \mathcal{P}''$. Assuming a non-uniform polynomial-time family $\{D_k\}_{k\in\mathbb{N}}$, we obtain that $|\Pr[D_k(P_k)] - \Pr[D_k(P'_k)]|$ and $|\Pr[D_k(P'_k)] - \Pr[D_k(P''_k)]|$ are negligible in k. We calculate as follows: I.354, p.142

$$\begin{aligned}
0 &\leq \left|\Pr[D_k(P_k)] - \Pr[D_k(P''_k)]\right| \\
&= \left|\Pr[D_k(P_k)] - \Pr[D_k(P'_k)] + \Pr[D_k(P'_k)] - \Pr[D_k(P''_k)]\right| \\
&\leq \left|\Pr[D_k(P_k)] - \Pr[D_k(P'_k)]\right| + \left|\Pr[D_k(P'_k)] - \Pr[D_k(P''_k)]\right|
\end{aligned}$$

Since the sum of two negligible terms is negligible, the last line of the previous calculation is negligible in k. Therefore, the (positive) expression $|\Pr[D_k(P_k)] - \Pr[D_k(P''_k)]|$ is bounded from above by a negligible function and hence must also be negligible in k. Hence it holds $\mathcal{P} \approx_{ind} \mathcal{P}''$. □ I.355, p.142 I.356, p.142

We can show that the observational equivalence of corresponding programs of two program families implies the computational indistinguishability of these families.

Lemma 1.43. *Assume program families $\mathcal{P} = \{P_k\}_{k\in\mathbb{N}}$ and $\mathcal{P}' = \{P'_k\}_{k\in\mathbb{N}}$. Given that $P_k \approx_{obs} P'_k$ for all $k \in \mathbb{N}$, it holds $\mathcal{P} \approx_{ind} \mathcal{P}'$.* I.389, p.145

Proof. Assume a non-uniform polynomial-time family $\{D_k\}_{k\in\mathbb{N}}$. Using the composability of $\approx_{obs}$ (Lemma 1.36), the observational equivalence $P_k \approx_{obs} P'_k$ implies $D_k(P_k) \approx_{obs} D_k(P'_k)$ for all $k \in \mathbb{N}$. From Lemma 1.37 we obtain the equality $\Pr[D_k(P_k)] = \Pr[D_k(P'_k)]$ for all $k \in \mathbb{N}$, and hence $\mathcal{P} \approx_{ind} \mathcal{P}'$. □

1.11 Fundamental Properties of the Language

1.11.1 Type Safety

Given a fully closed program state of type T, we can show that the reduction relation $\rightsquigarrow$ can progress (if we are not already in a value state), and that the relation preserves the type. Namely, the states in the resulting measure are also of type T.

Theorem 1.44 (Progress and Preservation)**.** *Let $P|\sigma|\eta$ be a program state, Γ and Θ environments, and T a type. Then the following properties hold:*

I.177, p.128
- Progress: *If $[]|\Theta \vDash P|\sigma|\eta : T$ and P is not a value, then there is a measure μ over program states such that $P|\sigma|\eta \rightsquigarrow \mu$.*

I.180, p.128
- Preservation – untyped: *If $P|\sigma|\eta$ is fully closed, then for almost all $P'|\sigma'|\eta' \in step(P|\sigma|\eta)$, the state $P'|\sigma'|\eta'$ is fully closed.*

I.178, p.128
- Preservation – typed: *If $\Gamma|\Theta \vDash P|\sigma|\eta : T$ and $P|\sigma|\eta \rightsquigarrow \mu$ for some measure μ, then there exists an environment Θ' such that*

$$\forall (P'|\sigma'|\eta') \leftarrow \mu.\, \Gamma|\Theta@\Theta' \vDash P'|\sigma'|\eta' : T.$$

1.11.2 Evaluation Contexts and Redexes

The computation rules in Figure 1.18 characterize a set of reducible expressions – so-called *redexes* – whereas the congruence rules locate the redex for the current step. An alternative method to express this localization is to use so-called *evaluation contexts*. These are contexts with a single hole $\Box$ at the position where the next reduction step should be performed. The sets of redexes and of evaluation contexts are defined as follows.

I.123, p.122
I.122, p.121
Definition 1.45 (Redexes and Evaluation Contexts)**.** *The sets of redexes R and of evaluation contexts E are defined by the following grammar, where s denotes strings and f denotes submarkov kernels from V_0 to V_0:*

$$\begin{aligned}
R ::= {} & VV \mid \mathsf{fun}(f, V) \mid \mathsf{ref}\, V \mid !V \mid V := V \mid \\
& \mathsf{event}\, s \mid \mathsf{eventlist} \mid \mathsf{fst}\, V \mid \mathsf{snd}\, V \mid \mathsf{case}\ V\ V\ V \mid \mathsf{unfold}\, V \\
E ::= {} & \Box \mid \mathsf{fun}(f, E) \mid EP \mid VE \mid \mathsf{ref}\, E \mid !E \mid E := P \mid V := E \mid \\
& \mathsf{fold}\, E \mid \mathsf{unfold}\, E \mid (E, P) \mid (V, E) \mid \mathsf{fst}\, E \mid \mathsf{snd}\, E \mid \\
& \mathsf{inl}\, E \mid \mathsf{inr}\, E \mid \mathsf{case}\ E\ P\ P \mid \mathsf{case}\ V\ E\ P \mid \mathsf{case}\ V\ V\ E
\end{aligned}$$

Note that each syntactic case for evaluation contexts (except for the case $\Box$) corresponds to one of the congruence rules in Figure 1.18. For example the case EP specifies that applications can be reduced in their first component, which corresponds to rule App1; the case VE specifies that applications can be reduced in their second component, if the first

component is a value, which corresponds to the rule App2. Furthermore we can show that each term P that is not a value can be decomposed into an evaluation context E and a redex R. This leads to an alternative formulation of the reduction relation $\rightsquigarrow$ based on evaluation contexts and redexes. I.129, p.123

Lemma 1.46 (Reduction on Redexes)**.** *Let $P|\sigma|\eta$ be a program state and μ a measure over program states. The relation $P|\sigma|\eta \rightsquigarrow \mu$ holds, if and only if there is an evaluation context E, a redex R, and a measure over program states μ_R, such that* I.146, p.125

- $P = E[R]$,
- $R|\sigma|\eta \rightsquigarrow \mu_R$, *and*
- $\mu = (\lambda(P'|\sigma'|\eta').\, E[P']|\sigma'|\eta')\, \mu_R$

Lemma 1.47. *Let $E[P]|\sigma|\eta$ be a program state where E is an evaluation context and P is not a value. Then it holds* I.145, p.125

$$\mathit{step}(E[P]|\sigma|\eta) = (\lambda(P'|\sigma'|\eta').\, (E[P']|\sigma'|\eta'))\, (\mathit{step}(P|\sigma|\eta))$$

1.11.3 A Chaining Rule for Denotations

The formulation of the *step* kernel in Lemma 1.47 suggests that, when evaluating a program $E[P]$ where E is an evaluation context, first P is reduced until it is a value V', and then the reduction proceeds with $E[V']$. Considering that the program P may branch probabilistically and execute a different number of steps in the different branches, we will start evaluating $E[V']$ for some values V' after a different number of steps in different branches. In general there is no $n \in \mathbb{N}$ such that $\mathit{step}^n(E[P]|\sigma|\eta)$ is a distribution over states of the form $E[V']|\sigma'|\eta'$ for some values V'. However, we can establish the following result in the limit of n: If we compute the denotation $[\![P|\sigma|\eta]\!]$ and apply to this measure the kernel that maps $V'|\sigma'|\eta'$ to $E[V']|\sigma'|\eta'$ and then computes its denotation, the result is equal to the denotation $[\![E[P]|\sigma|\eta]\!]$. This lemma is a powerful tool for reasoning about denotational equivalence. In particular, it directly entails that if P_1 and P_2 are denotationally equivalent then $E[P_1]$ and $E[P_2]$ are denotationally equivalent as well.

Theorem 1.48 (Chaining denotations)**.** *Let E be an evaluation context, P a program, σ a store and η an event list. Then it holds* I.328, p.140

$$[\![E[P]|\sigma|\eta]\!] = (\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![P|\sigma|\eta]\!].$$

Proof. We will use Theorem 1.9 in order to prove this theorem. For this purpose let Ω be the set of all tuples (P, σ, η, b) where P is a program, σ is a store, η is an event list, and b is a Boolean. We define the sets U, D_K, V,

the functions f, g, and the kernels K, L, M from Ω to Ω as

$$
\begin{aligned}
U &\stackrel{\text{def}}{=} \{(E[P'], \sigma', \eta', \mathit{true}) \mid P' \text{ is a value}\} \subseteq \Omega \\
D_K &\stackrel{\text{def}}{=} \{(E[P'], \sigma', \eta', \mathit{true}) \mid \mathit{true}\} \subseteq \Omega \\
V &\stackrel{\text{def}}{=} \mathit{Val} \times \{\mathit{false}\} = \{(P', \sigma', \eta', \mathit{false}) \mid P' \text{ is a value}\} \subseteq \Omega \\
g &\stackrel{\text{def}}{=} \lambda(P'|\sigma'|\eta').\,(E[P'], \sigma', \eta', \mathit{true}) \\
f &\stackrel{\text{def}}{=} \lambda(P', \sigma', \eta', b).\,(P'|\sigma'|\eta') \\
K(x) &\stackrel{\text{def}}{=} \begin{cases} g(\mathit{step}(P'|\sigma'|\eta')) & \text{if } x{=}(E[P'], \sigma', \eta', \mathit{true}){\in}D_K \\ 0 & \text{if } x{\notin}D_K \end{cases} \\
L(x) &\stackrel{\text{def}}{=} \begin{cases} (\lambda s.\,(s, \mathit{false}))(\mathit{step}(P'|\sigma'|\eta')) & \text{if } x{=}(P', \sigma', \eta', b'){\in}D_K^{\complement} \cup U \\ 0 & \text{if } x{\in}D_K \setminus U \end{cases} \\
M(x) &\stackrel{\text{def}}{=} \begin{cases} K(x) & \text{if } x{\in}D_K \setminus U \\ L(x) & \text{otherwise.} \end{cases}
\end{aligned}
$$

Since *step* is invariant on *Val*, K is invariant on U and L is invariant on V. Furthermore $V \cap D_K = \varnothing$, $U \subseteq D_K$ and for all $x \in \Omega$, we have $K(x)(D_K^{\complement}) = 0$ and $L(x)(D_K) = 0$. For all $x \in D_K \setminus U$ we have $L(x) = 0$, and for all $x \in D_K^{\complement}$, we have $K(x) = 0$. Thus Condition 1.4 is satisfied, and by Theorem 1.9 we have that M is invariant on V and for all $x \in D_K$ it holds

$$\lim_n (\downarrow_V \circ M^n)\, x = (\lim_n (\downarrow_V \circ L^n)) \circ (\lim_n (\downarrow_U \circ K^n))\, x. \tag{1.49}$$

Fix some $x \in \Omega$. If $x \in D_K \setminus U$, it is of the form $x = (E[P'], \sigma', \eta', \mathit{true})$ where P' is not a value. Then we have

$$
\begin{aligned}
\mathit{step}(f(x)) &= \mathit{step}(E[P']|\sigma'|\eta') \\
&= (\lambda(P'|\sigma'|\eta').\,(E[P']|\sigma'|\eta'))\mathit{step}(P'|\sigma'|\eta') && \text{(Lemma 1.47)} \\
&= f(K(E[P'], \sigma', \eta', \mathit{true})) \\
&= f(M(E[P'], \sigma', \eta', \mathit{true})).
\end{aligned}
$$

If $x = (P', \sigma', \eta', b) \in D_K^{\complement} \cup U$, we have $\mathit{step}(f(x)) = \mathit{step}(P'|\sigma'|\eta') = f(L(x)) = f(M(x))$. Hence, for all $x \in \Omega$ we have that $\mathit{step}(f(x)) = f(M(x))$. Also, it holds for all states $P'|\sigma'|\eta'$ that $g(\mathit{step}(P'|\sigma'|\eta')) = K(g(P'|\sigma'|\eta'))$. Let $\overline{f}$ and $\overline{g}$ denote the deterministic kernels $\lambda x.\,\delta(fx)$ and

$\lambda x.\, \delta(gx)$, respectively. We calculate as follows:

$$\begin{aligned}
[\![E[P]|\sigma|\eta]\!] &= \lim_n(\downarrow_{Val} \circ\, step^n)(f(g(P|\sigma|\eta))) \\
&= \lim_n(\downarrow_{Val} \circ\, \overline{f} \circ M^n)(g(P|\sigma|\eta)) \\
&= \overline{f} \circ \lim_n(\downarrow_V \circ\, M^n)(g(P|\sigma|\eta)) \\
&\overset{(1.49)}{=} \overline{f} \circ (\lim_n(\downarrow_V \circ\, L^n)) \circ (\lim_n(\downarrow_U \circ\, K^n))\,(g(P|\sigma|\eta)) \\
&= \lim_n(\downarrow_{Val} \circ\, \overline{f} \circ L^n) \circ \lim_n(\downarrow_U \circ\, \overline{g} \circ step^n)\ (P|\sigma|\eta) \\
&= \lim_n(\downarrow_{Val} \circ\, step^n) \circ \overline{f} \circ \overline{g} \circ \lim_n(\downarrow_{Val} \circ\, step^n)\ (P|\sigma|\eta) \\
&= (\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![P|\sigma|\eta]\!] \qquad \square
\end{aligned}$$

The following lemma follows:

Lemma 1.50. $[\![(\lambda P')P|\sigma|\eta]\!] = (\lambda(V'|\sigma'|\eta').\ [\![P'\{V'/_0\}|\sigma'|\eta']\!]) \cdot [\![P|\sigma|\eta]\!]$ I.330, p.140

1.11.4 The CIU Theorem

Establishing observational equivalence of two programs P_1 and P_2 can be difficult in general. Since it is defined using an arbitrary context C it is challenging to argue about all possible interactions of C with P_1 and P_2. Following the ideas of [93], we can show that it suffices to only consider *evaluation* contexts E for all closing instantiations of P_1 and P_2's variables in order to show their observational equivalence.

The Instantiation Operation

Definition 1.51 (Instantiation Operation)**.** *Given a program P and a list of values v, we define the* instantiation *of P's variables with v, written P^v, recursively as* I.416, p.152

$$\begin{aligned}
P^{[]} &\overset{def}{=} P \\
P^{V::v} &\overset{def}{=} (P^v)\{V/_0\}.
\end{aligned}$$

This operation has several interesting properties. For example, since the list v contains only values, the instantiation V^v is also a value for all values V. Hence it follows that for programs P and values V, the program $((\lambda P)V)^v$ reduces to $(P\{V/_0\})^v$. I.418, p.152

Lemma 1.52. *Given a program P, a value V, a store σ, an event list η, and a list of values v, it holds that $((\lambda P)V)^v|\sigma|\eta \rightsquigarrow \delta((P\{V/_0\})^v|\sigma|\eta)$.* I.422, p.152

If the instantiation P^v contains no free variables, then the instantiation operation is idempotent. This also holds, if the instantiation is performed in combination with substitution:

I.423, p.152 **Lemma 1.53.** *Let P and P' be programs and let v be a list of values. If P^v contains no free variables, then it holds $(P'\{P^v/0\})^v = (P'\{P/0\})^v$.*

Given fully closed states $P_1^v|\sigma|\eta$ and $P_2^v|\sigma|\eta$, it is possible to construct a list of values v' that yields the same instantiations for P and Q, but contains only values without free variables and without locations $\geq |\sigma|$.

I.428, p.153 **Lemma 1.54.** *Assume programs P_1, P_2, a store σ, an event list η, and a list of values v, such that $P_1^v|\sigma|\eta$ and $P_2^v|\sigma|\eta$ are fully closed. Then there exists a list of values v' for which $P_1^v = P_1^{v'}$ and $P_2^v = P_2^{v'}$ such that v' contains no free variables and no locations $\mathsf{loc}\, n$ with $n \geq |\sigma|$.*

Proof. We prove this by induction over v. The case $v = []$ is trivial, so let $v = vs@[V]$ where $P_1^{vs@[V]}|\sigma|\eta$ and $P_2^{vs@[V]}|\sigma|\eta$ are fully closed. Since $P_1^{vs@[V]} = (P_1\{V/0\})^{vs}$ and $P_2^{vs@[V]} = (P_2\{V/0\})^{vs}$, we can assume by induction that $(P_1\{V/0\})^{vs} = (P_1\{V/0\})^{vs'}$ and $(P_2\{V/0\})^{vs} = (P_2\{V/0\})^{vs'}$ for some list of closed values vs' without locations $\geq |\sigma|$.

If $\mathsf{var}\, 0 \notin \mathcal{FV}(P_1) \cup \mathcal{FV}(P_2)$, then the substitution operation on P_1 and P_2 is independent on the value that it substitutes with. In particular, it
I.112, p.119 holds $P_1\{V/0\} = P_1\{\mathsf{value}\, unit/0\}$ and $P_2\{V/0\} = P_2\{\mathsf{value}\, unit/0\}$. Since $\mathsf{value}\, unit$ is closed and contains no locations, we set $v' \stackrel{\text{def}}{=} vs'@[\mathsf{value}\, unit]$. It holds $P_1^v = (P_1\{V/0\})^{vs} = (P_1\{V/0\})^{vs'} = (P_1\{\mathsf{value}\, unit/0\})^{vs'} = P_1^{v'}$ and analogously for P_2 it holds $P_2^v = P_2^{v'}$.

If $\mathsf{var}\, 0 \in \mathcal{FV}(P_1) \cup \mathcal{FV}(P_2)$, we know that V occurs in $P_1\{V/0\}$ or in $P_2\{V/0\}$. Since both terms contain no locations $\geq |\sigma|$, value V also contains no locations $\geq |\sigma|$. Likewise, because $(P_1\{V/0\})^{vs'}$ and $(P_2\{V/0\})^{vs'}$ contain no free variables, value V does not contain free variables $\geq |vs'|$. Therefore $V^{vs'}$ contains no free variables and no locations $\geq |\sigma|$. We set $v' \stackrel{\text{def}}{=} vs'@[V^{vs'}]$. Thus $P_1^v = (P_1\{V/0\})^{vs} = (P_1\{V/0\})^{vs'} \stackrel{(*)}{=} (P_1\{V^{vs'}/0\})^{vs'} = P_1^{v'}$, where $(*)$ follows from Lemma 1.53. □

Given a list of values v, we can construct a *substituting context* C_v which, when inserted a program P, instantiates the variables of P with v when reducing $C_v[P]$. We define C_v recursively as

I.417, p.152
$$C_{[]} \stackrel{\text{def}}{=} \square$$
$$C_{V::v} \stackrel{\text{def}}{=} (\lambda C_v)V.$$

When reducing $C_v[P]$, each step substitutes one de Bruijn index according to the rule Beta of the $\rightsquigarrow$ relation (Figure 1.18). Therefore, $C_v[P]$ reduces eventually to P^v. In particular, we can show that the denotation
I.424, p.152 $[\![C_v[P]|\sigma|\eta]\!]$ equals the denotation $[\![P^v|\sigma|\eta]\!]$. We can combine this result with Lemma 1.54. Given fully closed states $P_1^v|\sigma|\eta$ and $P_2^v|\sigma|\eta$, there is a corresponding list of values v' for which the states $C_{v'}[P_1]|\sigma|\eta$ and $C_{v'}[P_2]|\sigma|\eta$ are also fully closed:

Lemma 1.55. *Assume programs P_1, P_2, a store σ, an event list η, and a list of values v, such that $P_1^v|\sigma|\eta$ and $P_2^v|\sigma|\eta$ are fully closed. Then there exists a list of values v' for which $C_{v'}[P_1]|\sigma|\eta$ and $C_{v'}[P_2]|\sigma|\eta$ are fully closed, $[\![C_{v'}[P_1]|\sigma|\eta]\!] = [\![P_1^v|\sigma|\eta]\!]$, and $[\![C_{v'}[P_2]|\sigma|\eta]\!] = [\![P_2^v|\sigma|\eta]\!]$.* I.429, p.153

CIU Equivalence

We call two programs *CIU equivalent*,[3] if no evaluation context can distinguish closed instantiations of the programs.

Definition 1.56 (CIU Equivalence)**.** *Let P_1 and P_2 be programs. We call P_1 and P_2* CIU equivalent*, iff for all evaluation contexts E, all stores σ, all event lists η, and all lists of closed values v without locations $\mathsf{loc}\, n$ with $n \geq |\sigma|$, the following holds: If the program states $E[P_1^v]|\sigma|\eta$ and $E[P_2^v]|\sigma|\eta$ are fully closed, then $[\![E[P_1^v]|\sigma|\eta]\!](\Omega) = [\![E[P_2^v]|\sigma|\eta]\!](\Omega)$ where Ω is the set of all program states. We write $P_1 \approx_{ciu} P_2$ to denote that programs P_1 and P_2 are CIU equivalent.* I.461, p.157

CIU equivalence is much easier to handle than observational equivalence (Definition 1.34). Nevertheless, CIU equivalent programs are also observationally equivalent, as we will see later. To demonstrate the practicability of the CIU equivalence, assume two programs P_1 and P_2 that are denotationally equivalent for all lists of values and states that close them. It is easy to verify that P_1 and P_2 are CIU equivalent as stated by the following lemma.

Lemma 1.57. *Let P_1 and P_2 be programs. Assume that for all lists of values v, stores σ, and event lists η such that $P_1^v|\sigma|\eta$ and $P_2^v|\sigma|\eta$ are fully closed it holds $[\![P_1^v|\sigma|\eta]\!] = [\![P_2^v|\sigma|\eta]\!]$. Then it holds $P_1 \approx_{ciu} P_2$.* I.490, p.160

Proof. Let v be a list of values, σ a store, and η an event list such that $E[P_1^v]|\sigma|\eta$ and $E[P_2^v]|\sigma|\eta$ are fully closed. Using Theorem 1.48 it holds

$$[\![E[P_1^v]|\sigma|\eta]\!] = f \cdot [\![P_1^v|\sigma|\eta]\!] = f \cdot [\![P_2^v|\sigma|\eta]\!] = [\![E[P_2^v]|\sigma|\eta]\!]\,,$$

where $f \stackrel{\text{def}}{=} \lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]$. Hence P_1 and P_2 are CIU equivalent. □

The following theorem states that CIU equivalence implies observational equivalence. It plays a central role when proving the observational equivalence of programs.

Theorem 1.58 (CIU Theorem)**.** *Let P_1 and P_2 be programs. The CIU equivalence $P_1 \approx_{ciu} P_2$ implies the observational equivalence $P_1 \approx_{obs} P_2$.* I.489, p.160

[3] In [93] the acronym CIU stands for "all closed instantiations of all uses."

1.11.5 Proof of the CIU Theorem

In this section we will prove the CIU Theorem 1.58. The proof follows the ideas of [93] with some extensions to cope with the probabilistic nature of our language. [93] performs an induction over the length of the computation of a terminating program. For probabilistic programs, this length is not necessarily bounded.

We introduce the following notation: $T(P|\sigma|\eta)$ denotes the probability that the program state $P|\sigma|\eta$ terminates, i.e., we define $T(P|\sigma|\eta) \stackrel{\text{def}}{=} [\![P|\sigma|\eta]\!](Val)$, where Val is the set of all value states. By $T_n(P|\sigma|\eta)$ we denote the probability of termination after n steps, i.e., we define $T_n(P|\sigma|\eta) \stackrel{\text{def}}{=} step^n(P|\sigma|\eta)(Val)$. Moreover, we define the following asymmetric variants of observational equivalence and CIU equivalence where the left term is allowed to have a smaller probability of termination than the right term.

I.337, p.141 **Definition 1.59** (Observational Approximation)**.** *Let P_1 and P_2 be program terms. Then $P_1 \preceq_{obs} P_2$, iff for all contexts C, stores σ, and event lists η such that $C[P_1]|\sigma|\eta$ and $C[P_2]|\sigma|\eta$ are fully closed it holds $T(C[P_1]|\sigma|\eta) \leq T(C[P_2]|\sigma|\eta)$.*

I.460, p.156 **Definition 1.60** (CIU Approximation)**.** *Let P_1 and P_2 be programs. Then $P_1 \preceq_{ciu} P_2$, iff for all evaluation contexts E, all stores σ, all event lists η, and all lists of closed values v without locations $\mathsf{loc}\, n$ with $n \geq |\sigma|$, the following holds: If the program states $E[P_1^v]|\sigma|\eta$ and $E[P_2^v]|\sigma|\eta$ are fully closed, then $T(E[P_1^v]|\sigma|\eta) \leq T(E[P_2^v]|\sigma|\eta)$.*

It is easy to see that we can express observational and CIU equivalence using their respective approximate variants:

I.338, p.141 **Lemma 1.61.** *Let P_1 and P_2 be programs. Then $P_1 \approx_{obs} P_2$, iff $P_1 \preceq_{obs} P_2$ and $P_2 \preceq_{obs} P_1$.*

I.462, p.157 **Lemma 1.62.** *Let P_1 and P_2 be programs. Then $P_1 \approx_{ciu} P_2$, iff $P_1 \preceq_{ciu} P_2$ and $P_2 \preceq_{ciu} P_1$.*

Proof Outline

In order to prove the CIU Theorem 1.58, we are going to prove that $P_1 \preceq_{ciu} P_2$ implies $P_1 \preceq_{obs} P_2$. On a high level, the proof works as follows: First we generalize the notion of program terms in order to define a set of conditions we call a *CIU counterexample*. We show that such a CIU counterexample must exist, if $P_1 \preceq_{ciu} P_2$ but not $P_1 \preceq_{obs} P_2$. Then we proceed by showing that each CIU counterexample can be transformed into a smaller one with respect to a well-founded order relation. This implies that CIU counterexamples cannot exist as the well-foundedness would also require the existence of minimal CIU counterexamples.

Generalized Terms

The definition of observational equivalence considers arbitrary contexts C whereas CIU equivalence is defined using evaluation contexts where the hole $\square$ can only appear at positions that will be evaluated first. When evaluating the term $C[P]$, the program P may be subject to a sequence of substitution and lift operations before it is evaluated. In order to relate observational and CIU equivalence, we need a way to track these operations that happen at the positions of the holes in C while evaluating $C[P]$. The problem is that the term $C[P]$ contains no holes any more and thus the position information has been lost. Therefore, we generalize the notion of programs by introducing a new symbol $\varepsilon^{\dot{a}}$ which plays a similar role as a hole $\square$ in contexts, but is additionally annotated with a list $\dot{a}$ consisting of pending substitution and lift operations. For readability, generalized terms will always by marked with a dot on top. Similar to a hole $\square$ in contexts, we can replace all $\varepsilon^{\dot{a}}$ that occur in a generalized $\dot{C}$ with a program P, written $\dot{C}[P/_{\varepsilon}]$. This has the additional effect of applying the annotated operations $\dot{a}$ to P. We will now give the generalized definitions of programs, values, redexes, contexts, evaluation contexts, and stores, as well as the recursive definition of the ε-insertion operation $\dot{C}[P/_{\varepsilon}]$.

Definition 1.63 (Generalized Terms)**.** *The sets of* generalized programs $\dot{P}$*, of* generalized instantiations $\dot{a}$*, and of* generalized values $\dot{V}$ *are defined mutually inductively as follows:* I.394, p.146

- *The set of* generalized programs $\dot{P}$ *is defined inductively by the same introduction rules as programs, except that the rules refer to generalized programs instead of programs plus the following rule:*

$$\frac{\dot{a} \text{ is a generalized instantiation}}{\varepsilon^{\dot{a}} \in \dot{P}}$$

  where ε is a special symbol.
- *A* generalized instantiation $\dot{a}$ *is a list with entries of the form* $\{\dot{V}/_k\}$ *and* $\uparrow_k$ *where $k \in \mathbb{N}$ and $\dot{V}$ is a generalized value.*
- *The set of* generalized values $\dot{V}$ *is defined inductively by the same introduction rules as values, except that the rules refer to generalized programs and generalized values instead of programs and values.*

Definition 1.64 (Generalized Contexts)**.** *The set of* generalized contexts $\dot{C}$ *is defined inductively by the same introduction rules as contexts, except that the rules refer to generalized contexts instead of contexts plus the following rule:* I.395, p.147

$$\frac{\dot{a} \text{ is a generalized instantiation}}{\varepsilon^{\dot{a}} \in \dot{C}}$$

I.396, p.147 **Definition 1.65** (Generalized Evaluation Contexts)**.** *The set of* generalized evaluation contexts $\dot{E}$ *is defined inductively by the same introduction rules as the set of evaluation contexts, except that the rules refer to generalized evaluation contexts, generalized programs, and generalized values instead of evaluation contexts, programs, and values.*

I.397, p.148 **Definition 1.66** (Generalized Redexes)**.** *The set of* generalized redexes $\dot{R}$ *is defined inductively by the same introduction rules as the set of redexes, except that the rules refer to generalized values instead of values.*

Definition 1.67 (Generalized Store)**.** *A* generalized store $\dot{\sigma}$ *is a list of generalized values.*

I.403, p.149 **Definition 1.68** (ε-Insertion)**.** *Given a generalized term $\dot{C}$ and a program P, we recursively define $\dot{C}[P/\varepsilon]$ by the rules*

$$\begin{aligned}\varepsilon^{[]}[P/\varepsilon] &\stackrel{def}{=} P\\ \varepsilon^{\{\dot{V}/k\}::\dot{a}}[P/\varepsilon] &\stackrel{def}{=} \left(\varepsilon^{\dot{a}}[P/\varepsilon]\right)\{\dot{V}[P/\varepsilon]/k\}\\ \varepsilon^{\uparrow_k::\dot{a}}[P/\varepsilon] &\stackrel{def}{=} \uparrow_k(\varepsilon^{\dot{a}}[P/\varepsilon]),\end{aligned}$$

and for all other generalized terms $\dot{C}$, $\dot{C}[P/\varepsilon]$ recursively descends into the subterms (E.g., $(\dot{C}_1\dot{C}_2)[P/\varepsilon] \stackrel{def}{=} (\dot{C}_1[P/\varepsilon])(\dot{C}_2[P/\varepsilon])$). For generalized stores $\dot{\sigma}$ and event lists η, we define $\dot{\sigma}[P/\varepsilon]$ element-wise and $(\dot{C},\dot{\sigma},\eta)[P/\varepsilon]$ as
I.437, p.154 *$(\dot{C}[P/\varepsilon]\,|\,\dot{\sigma}[P/\varepsilon]\,|\,\eta)$. For generalized instantiation $\dot{a}$, we define $\dot{a}[P/\varepsilon]$ by*

$$\begin{aligned}[][P/\varepsilon] &\stackrel{def}{=} []\\ (\uparrow_k::\dot{a})[P/\varepsilon] &\stackrel{def}{=} \uparrow_k::(\dot{a}[P/\varepsilon])\\ (\{\dot{V}/k\}::\dot{a})[P/\varepsilon] &\stackrel{def}{=} \{\dot{V}[P/\varepsilon]/k\}::(\dot{a}[P/\varepsilon]).\end{aligned}$$

Furthermore we extend the definitions of the lift operator $\uparrow_k$ and the substitution operator $\{\cdot/k\}$ to generalized terms. For ε-terms, these operators simply add their respective operation to the annotated generalized instantiation:

I.405, p.149 $$\uparrow_k\varepsilon^{\dot{a}} \stackrel{\text{def}}{=} \varepsilon^{\uparrow_k::\dot{a}}$$

I.406, p.150 $$\varepsilon^{\dot{a}}\{\dot{V}/k\} \stackrel{\text{def}}{=} \varepsilon^{\{\dot{V}/k\}::\dot{a}}$$

Note that the extended definitions of these operators fulfill the following homomorphic properties with respect to ε-insertion:

Lemma 1.69. *Given a program term P, generalized terms $\dot{C}$ and $\dot{V}$, and $k \in \mathbb{N}$, it holds*

I.410, p.151 $$(\uparrow_k\dot{C})[P/\varepsilon] = \uparrow_k(\dot{C}[P/\varepsilon])$$

I.411, p.151 $$(\dot{C}\{\dot{V}/k\})[P/\varepsilon] = (\dot{C}[P/\varepsilon])\{\dot{V}[P/\varepsilon]/k\}$$

Similarly to Definition 1.51, where we introduced the instantiation operation P^v for lists of values v, we can define a *generalized instantiation operation* $\dot{P}^{\dot{a}}$ for generalized instantiations $\dot{a}$ as follows: I.407, p.150

$$\begin{aligned} \dot{P}^{[]} &\stackrel{\text{def}}{=} \dot{P} \\ \dot{P}^{\uparrow_k::\dot{a}} &\stackrel{\text{def}}{=} \uparrow_k \dot{P}^{\dot{a}} \\ \dot{P}^{\{\dot{V}/_k\}::\dot{a}} &\stackrel{\text{def}}{=} \dot{P}^{\dot{a}}\{\dot{V}/_k\} \end{aligned}$$

Instantiation Operation P^v vs. Generalized Instantiation $\dot{P}^{\dot{a}}$

There is a close connection between the instantiation operation P^v for lists of values v and the generalized instantiation operation $\dot{P}^{\dot{a}}$ for generalized instantiations $\dot{a}$. For example, we can transform a list of values v into a generalized instantiation $\dot{v}$ by mapping each V in v to the entry $\{V/_0\}$. In this case it is easy to see that $P^{\dot{v}} = P^v$ for arbitrary programs P. The converse direction is also possible, i.e., we can transform generalized instantiations into corresponding lists of values under certain conditions. This direction is more complicated, because generalized instantiations can include lift operations and handle arbitrary de Bruijn indices, for which there is no direct representation as a list of values. We will now sketch a process that can transform generalized instantiations $\dot{a}$ into lists of values v such that $P^v = P^{\dot{a}}$. I.435, p.154 I.436, p.154

One requirement for this process to work is that the generalized instantiation $\dot{a}$ must not contain ε-terms, because such terms do not exist in normal programs and hence cannot be represented as a value. We call a generalized instantiation $\dot{a}$ a *programterm instantiation*, if for all entries of $\dot{a}$ of the form $\{\dot{V}/_k\}$ it holds that $\dot{V}$ is a non-generalized program, i.e., it contains no ε-terms. Note that for programterm instantiations $\dot{a}$ it holds $\varepsilon^{\dot{a}}[P/_\varepsilon] = P^{\dot{a}}$. Also note that $\dot{a}[P/_\varepsilon]$ (see Definition 1.68) is a programterm instantiation and it holds that $\varepsilon^{\dot{a}}[P/_\varepsilon] = P^{\dot{a}[P/_\varepsilon]}$. I.430, p.153 I.434, p.154 I.447, p.155 I.449, p.155

There is another requirement for the transformation of a programterm instantiation $\dot{a}$ to a list of values v such that $P^v = P^{\dot{a}}$. Namely, we require $P^{\dot{a}}$ to have no free variables. In this case we can construct a programterm instantiation $\dot{a}_c$ that contains no free variables, i.e., for all substitution entries $\{V/_k\}$ in $\dot{a}_c$, it holds $\mathcal{FV}(V) = \{\}$. Such programterm instantiations are called *closed instantiations*. I.431, p.153

We construct $\dot{a}_c$ as follows: Note that the leftmost substitution operation $\{V/_k\}$ in $\dot{a}$ is performed last when computing $P^{\dot{a}}$. Since $P^{\dot{a}}$ has no free variables, the value V in the leftmost substitution operation must also be closed.[4] Now we employ the following trick: Analogously to Lemma 1.15 there are rules to swap the ordering of consecutive elements in the list $\dot{a}$ without changing $P^{\dot{a}}$. Therefore we can push $\{V/_k\}$ towards the right end of I.438, p.154

[4] If $\mathsf{var}\, k$ is not free in P, then V is not necessarily closed, but in this case we can simply drop the operation $\{V/_k\}$ completely, as it has no effect.

the list. During this process the value V is substituted into other elements in the list and we finally obtain a new leftmost substitution operation $\{V'/_{k'}\}$ which, for the same reason as before, must also be closed. We apply this process repeatedly until all the substitution operations have been closed. The result is the closed instantiation $\dot{a}_c$. Since the reordering of the list does not change the result of the instantiation, we can show that $P^{\dot{a}_c} = P^{\dot{a}}$.

Starting from this closed instantiation $\dot{a}_c$, we again reorder the instantiation using rules as in Lemma 1.15. This time we sort the list such that lift operations only occur at the beginning of the list and for consecutive substitution operations $\{V/_k\}$, $\{V'/_{k'}\}$ we have that $k \geq k'$. We call instantiations with such an ordering *normalized instantiations*. As before, we can show that the instantiation $\dot{a}_n$ we obtain from this reordering fulfills $P^{\dot{a}_n} = P^{\dot{a}_c}$.

We will now see how to transform closed, normalized instantiations $\dot{a}_n$ into lists of values v such that $P^v = P^{\dot{a}_n}$. Note that the list $\dot{a}_n$ is ordered such that, when computing $P^{\dot{a}_n}$, smaller de Bruijn indices are substituted first. The same happens when a non-generalized instantiation P^v is evaluated. Here we always substitute the index 0 which causes higher indices in P to be decreased by one in each step. We can exploit this decrease in order to emulate the first substitution operation $\{V/_k\}$ by using a sequence of k dummy values: Let u^k be the list consisting of the k-times repetition of $\mathsf{value}\, unit$, then the list of values $V{::}u^k$ instantiates $\mathsf{var}\, k$ with V when computing $P^{V::u^k}$. Similarly, we insert between consecutive substitution operations $\{V/_k\}$, $\{V'/_{k'}\}$ the list $u^{k-k'}$. The result is a list of values v for which $P^v = P^{\dot{a}_n}$.

We finally combine the steps that we outlined above with Lemma 1.54 in order to show that, given a fully closed state, we can transform programterm instantiations into lists of values without free variables and without locations outside the store.

Lemma 1.70. *Assume programs P_1, P_2, a store σ, an event list η, and a programterm instantiation $\dot{a}$ (i.e., elements of $\dot{a}$ contain no ε-terms), such that $P_1^{\dot{a}}|\sigma|\eta$ and $P_2^{\dot{a}}|\sigma|\eta$ are fully closed. Then there exists a list of values v for which $P_1^v = P_1^{\dot{a}}$ and $P_2^v = P_2^{\dot{a}}$ such that v contains no free variables and no locations $\mathsf{loc}\, n$ with $n \geq |\sigma|$.*

CIU Counterexamples

We will now define the notion of a *CIU counterexample*. It consists of a set of conditions that can be fulfilled, if for two programs P_1 and P_2 it holds that $P_1 \preceq_{ciu} P_2$, but not $P_1 \preceq_{obs} P_2$. The main idea is that in this case we can find a generalized program $\dot{C}$ and an $n \in \mathbb{N}$ for which the probability of termination of program $\dot{C}[P_1/_\varepsilon]$ after n steps is higher than the probability of termination of program $\dot{C}[P_2/_\varepsilon]$ after arbitrarily many steps. We use this fact to construct a contradiction as follows: We show that we can always

find a smaller CIU counterexample that uses an $n' < n$ or a $\dot{C}'$ containing less not λ-protected ε-terms. Since this ordering on CIU counterexamples is well-founded, this leads to a contradiction. As a side condition, we require all ε-terms to fulfill the following closedness condition:

Definition 1.71 ($[S/_\varepsilon]$-closed)**.** *Given a generalized program term $\dot{C}$, a generalized store $\dot{\sigma}$, and a set of program terms S, the pair $(\dot{C}, \dot{\sigma})$ is called* $[S/_\varepsilon]$-closed*, iff for every $P \in S$ and for every subterm of $\dot{C}$ and $\dot{\sigma}$ of the form $\varepsilon^{\dot{a}}$ it holds that for every entry in $\dot{a}$ of the form $\{\dot{V}/_k\}$ we have that $\dot{V}[P/_\varepsilon]$ is closed and does not contain locations $\mathsf{loc}\, n$ with $n \geq |\sigma|$.* I.464, p.157

Definition 1.72 (CIU Counterexample)**.** *Given programs P_1 and P_2, a generalized program $\dot{C}$, a generalized store $\dot{\sigma}$, an event list η, and an $n \in \mathbb{N}$, the tuple $(P_1, P_2, \dot{C}, \dot{\sigma}, \eta, n)$ is called a* CIU counterexample*, iff* I.466, p.158

- $P_1 \preceq_{ciu} P_2$,
- $T_n\big((\dot{C}, \dot{\sigma}, \eta)[P_1/_\varepsilon]\big) \;>\; T\big((\dot{C}, \dot{\sigma}, \eta)[P_2/_\varepsilon]\big)$,
- $(\dot{C}, \dot{\sigma}, \eta)[P_1/_\varepsilon]$ *and* $(\dot{C}, \dot{\sigma}, \eta)[P_2/_\varepsilon]$ *are fully closed, and*
- $(\dot{C}, \dot{\sigma})$ *is* $[\{P_1, P_2\}/_\varepsilon]$*-closed.*

We call a CIU counterexample minimal*, iff $(n, \#_\varepsilon \dot{C})$ is minimal with respect to the lexicographic order on $\mathbb{N} \times \mathbb{N}$. Here, $\#_\varepsilon \dot{C}$ denotes the number of occurrences of ε-terms in $\dot{C}$ that are not inside a λ-abstraction.* I.467, p.158 I.409, p.151

Lemma 1.73. *If there are program terms P_1 and P_2 such that $P_1 \preceq_{ciu} P_2$ but not $P_1 \preceq_{obs} P_2$, then there exists a CIU counterexample.* I.468, p.158

Proof. Since not $P_1 \preceq_{obs} P_2$, there exists a context C, a store σ and an event list η such that $C[P_1]|\sigma|\eta$ and $C[P_2]|\sigma|\eta$ are fully closed and $T(C[P_1]|\sigma|\eta) > T(C[P_2]|\sigma|\eta)$. Since $T(C[P_1]|\sigma|\eta) = \lim_n T_n(C[P_1]|\sigma|\eta)$, there exists an n such that $T_n(C[P_1]|\sigma|\eta) > T(C[P_2]|\sigma|\eta)$. Therefore $(P_1, P_2, C[\varepsilon^{[]}], \sigma, \eta, n)$ is a CIU counterexample. □

Given a CIU counterexample $(P_1, P_2, \dot{C}, \dot{\sigma}, \eta, n)$, note that $\dot{C}$ cannot be a generalized value, as in this case $\dot{C}[P_2/_\varepsilon]$ would be a value which implies $T((\dot{C}, \dot{\sigma}, \eta)[P_2/_\varepsilon]) = 1$ in contradiction to the second condition of Definition 1.72. Therefore we can use the following lemma to distinguish two cases where $\dot{C}$ is split into a generalized evaluation context $\dot{E}$ and a generalized redex $\dot{R}$ or into a generalized evaluation context $\dot{E}$ and an ε-term.

Lemma 1.74. *Any generalized program term that is not a generalized value can be written as $\dot{E}[\dot{P}]$ where $\dot{E}$ is a generalized evaluation context and $\dot{P}$ is either a generalized redex or of the form $\varepsilon^{\dot{a}}$ for some generalized instantiation $\dot{a}$.* I.414, p.151

We will show that in both cases we can find a smaller CIU counterexample according to the lexicographic order presented in Definition 1.72. First we consider the case where $\dot{C}$ can be split into a generalized evaluation context $\dot{E}$ and a generalized redex $\dot{R}$. We will show that in this case we can find a smaller CIU counterexample as follows: By reducing $(\dot{C}, \dot{\sigma}, \eta)[P_1/_\varepsilon]$ and $(\dot{C}, \dot{\sigma}, \eta)[P_2/_\varepsilon]$ one step we can obtain a generalized program term $\dot{C}'$ for which the second condition of Definition 1.72 is already fulfilled after $n-1$ steps. For this argument to work, we need to show that the step relation preserves the properties of a CIU counterexample. In particular, such a $\dot{C}'$ can only exist, if the ε-insertion of P_1 and P_2 can be factored out of the distributions resulting from reducing $(\dot{C}, \dot{\sigma}, \eta)[P_1/_\varepsilon]$ and $(\dot{C}, \dot{\sigma}, \eta)[P_2/_\varepsilon]$. This property is captured by the definition of CIU uniformity.

I.469, p.158 **Definition 1.75** (CIU Uniformity)**.** *We call a triple* $(\dot{C}, \dot{\sigma}, \eta)$ uniform*, iff there is a distribution* $\dot{\mu}$ *such that*

- *for all programs* P *we have* $step((\dot{C}, \dot{\sigma}, \eta)[P/_\varepsilon]) = \dot{\mu}[P/_\varepsilon]$*,*
- *if* $\dot{C}$ *is a generalized program and* $\dot{\sigma}$ *is a generalized store, then for almost all* $(\dot{C}', \dot{\sigma}', \eta') \in \dot{\mu}$ *it holds that* $\dot{C}'$ *is generalized program and* $\dot{\sigma}'$ *is a generalized store, and*
- *for all sets of programs* S*, it holds that, if* $(\dot{C}, \dot{\sigma})$ *is* $[S/_\varepsilon]$*-closed and for all* $P \in S$*,* $(\dot{C}, \dot{\sigma}, \eta)[P/_\varepsilon]$ *is fully closed, then for almost all* $(\dot{C}', \dot{\sigma}', \eta') \in \dot{\mu}$*, it holds that* $(\dot{C}', \dot{\sigma}')$ *is* $[S/_\varepsilon]$*-closed and* $|\dot{\sigma}'| \in \{|\dot{\sigma}|, |\dot{\sigma}|+1\}$*.*

I.482, p.159 **Lemma 1.76.** *Assume that* $\dot{E}$ *is a generalized evaluation context and* $\dot{R}$ *is a generalized redex. Then* $(\dot{E}[\dot{R}], \dot{\sigma}, \eta)$ *is uniform.*

Proof. We first show that $(\dot{R}, \dot{\sigma}, \eta)$ is uniform. By Definition 1.66 of redexes, $\dot{R}$ has one of the following forms: $\dot{V}_1\dot{V}_2$, $\mathsf{fun}(f, \dot{V})$, $\mathsf{ref}\,\dot{V}$, $!\dot{V}$, $\dot{V}_1 := \dot{V}_2$, $\mathsf{event}\,s$, $\mathsf{eventlist}$, $\mathsf{fst}\,\dot{V}$, $\mathsf{snd}\,\dot{V}$, $\mathsf{case}\ \dot{V}_1\ \dot{V}_2\ \dot{V}_3$, or $\mathsf{unfold}\,\dot{V}$. We examine the two cases of $\dot{R} = \dot{V}_1\dot{V}_2$ and $\dot{R} = \mathsf{fun}(f, \dot{V})$. The other cases are analogous and have also been verified using Isabelle.

I.471, p.158
- Considering $\dot{R} = \dot{V}_1\dot{V}_2$, we distinguish the two cases $\dot{V}_1 = \lambda\dot{C}_1$ and $\dot{V}_1 \neq \lambda\dot{C}_1$. In case $\dot{V}_1 = \lambda\dot{C}_1$, Lemma 1.69 implies that for any program P, any generalized store $\dot{\sigma}$ and any event list η, it holds that $step((\dot{V}_1\dot{V}_2, \dot{\sigma}, \eta)[P/_\varepsilon]) = \delta(\dot{C}_1\{\dot{V}_2/_0\}, \dot{\sigma}, \eta)[P/_\varepsilon]$. Furthermore, $\dot{C}_1\{\dot{V}_2/_0\}$ is a generalized program, if $\dot{C}_1$ and $\dot{V}_2$ are generalized programs. Also, assuming that $(\dot{V}_1\dot{V}_2, \dot{\sigma}, \eta)[P/_\varepsilon]$ is fully closed for all $P \in S$ and that $(\dot{V}_1\dot{V}_2, \dot{\sigma})$ is $[S/_\varepsilon]$-closed, we substitute with the closed term $\dot{V}_2[P/_\varepsilon]$ and $(\dot{C}_1\{\dot{V}_2/_0\}, \dot{\sigma})$ is $[S/_\varepsilon]$-closed, too. If $\dot{V}_1 \neq \lambda\dot{C}_1$, then the evaluation is stuck and for any program P, it holds that $step((\dot{V}_1\dot{V}_2, \dot{\sigma}, \eta)[P/_\varepsilon]) = \delta(\dot{V}_1\dot{V}_2, \dot{\sigma}, \eta)[P/_\varepsilon]$ and the other conditions in Definition 1.75 hold trivially.

- In case $\dot{R} = \mathsf{fun}(f, \dot{V})$, since $\dot{V}$ is a generalized value, so is $\dot{V}[P/_\varepsilon]$ for any program P. If $\dot{V}$ contains an ε, then (as $\dot{V}$ is a generalized value) it necessarily contains a λ-abstraction. Thus $\dot{V}[P/_\varepsilon]$ contains a λ-abstraction and is not a pure value. If $\dot{V}$ contains no ε and is not a pure value, then $\dot{V}[P/_\varepsilon] = \dot{V}$ is not a pure value. If $\dot{V}$ contains no ε and is a pure value, then $\dot{V}[P/_\varepsilon] = \dot{V}$ is a pure value. Thus either $\dot{V}[P/_\varepsilon]$ is a pure value for all programs P or for no program P. If $\dot{V}[P/_\varepsilon]$ is a pure value for no program P, we have $\mathit{step}((\mathsf{fun}(f, \dot{V}), \dot{\sigma}, \eta)[P/_\varepsilon]) = \delta(\mathsf{fun}(f, \dot{V}), \dot{\sigma}, \eta)[P/_\varepsilon]$ and the other conditions in Definition 1.75 hold trivially. If $\dot{V}[P/_\varepsilon]$ is a pure value for all programs P, we have that $\mathit{step}((\mathsf{fun}(f, \dot{V}), \dot{\sigma}, \eta)[P/_\varepsilon]) = ((\lambda x.\,(x, \dot{\sigma}, \eta))(f(\dot{V})))[P/_\varepsilon]$ (Note that in this case, $\dot{V}$ is a pure value and thus in the domain of kernel f). I.470, p.158

Hence in each case, for all programs P, it holds that $\mathit{step}((\dot{R}, \dot{\sigma}, \eta)[P/_\varepsilon]) = \dot{\mu}'[P/_\varepsilon]$ for some distribution $\dot{\mu}'$. Because $\dot{R}$ is not a generalized value and $\dot{E}$ is a generalized evaluation context, $\dot{R}[P/_\varepsilon]$ is not a value and $\dot{E}[P/_\varepsilon]$ is an evaluation context. Thus from Lemma 1.47 we have $\mathit{step}((\dot{E}[\dot{R}], \dot{\sigma}, \eta)[P/_\varepsilon]) = \mathit{step}((\dot{E}[P/_\varepsilon])[\dot{R}[P/_\varepsilon]], \dot{\sigma}[P/_\varepsilon], \eta) = \dot{\mu}[P/_\varepsilon]$ where the distribution $\dot{\mu}$ is given by $\dot{\mu} \stackrel{\text{def}}{=} (\lambda(\dot{C}, \dot{\sigma}, \eta).\,(\dot{E}[\dot{C}], \dot{\sigma}, \eta))\,\dot{\mu}'$. In order to show that $\dot{\mu}$ is $[S/_\varepsilon]$-closed we use the property that $|\dot{\sigma}'| \in \{|\dot{\sigma}|, |\dot{\sigma}| + 1\}$ for almost all stores $\dot{\sigma}' \in \dot{\mu}'$. □

Lemma 1.77. *Let $(P_1, P_2, \dot{E}[\dot{R}], \dot{\sigma}, \eta, n)$ be a CIU counterexample where $\dot{E}$ is a generalized evaluation context and $\dot{R}$ is a generalized redex. Then there exists a smaller CIU counterexample $(P_1, P_2, \dot{C}', \dot{\sigma}', \eta', n-1)$.* I.483, p.159

Proof. Let $\dot{\mathrm{FC}}$ denote the set of all $(\dot{C}', \dot{\sigma}', \eta)$ where $\dot{C}'$ is a generalized program term and $\dot{\sigma}'$ is a generalized store such that $(\dot{C}', \dot{\sigma}', \eta)[P_1/_\varepsilon]$ and $(\dot{C}', \dot{\sigma}', \eta)[P_2/_\varepsilon]$ are fully closed and $(\dot{C}', \dot{\sigma}')$ is $[\{P_1, P_2\}/_\varepsilon]$-closed. Using Lemma 1.76, we obtain a distribution $\dot{\mu}$ such that $\mathit{step}((\dot{E}[\dot{R}], \dot{\sigma}, \eta)[P/_\varepsilon]) = \dot{\mu}[P/_\varepsilon]$ for all program terms P. Since $(P_1, P_2, \dot{E}[\dot{R}], \dot{\sigma}, \eta, n)$ is a CIU counterexample, it furthermore holds for almost all $(\dot{C}', \dot{\sigma}', \eta') \in \dot{\mu}$ that $\dot{C}'$ is a generalized program term, $\dot{\sigma}'$ is a generalized store, and $(\dot{C}', \dot{\sigma}')$ is $[\{P_1, P_2\}/_\varepsilon]$-closed. Also, by Theorem 1.44 we obtain that $(\dot{C}', \dot{\sigma}', \eta')[P_1/_\varepsilon]$ and $(\dot{C}', \dot{\sigma}', \eta')[P_2/_\varepsilon]$ are fully closed for almost all $(\dot{C}', \dot{\sigma}', \eta') \in \dot{\mu}$. Therefore we obtain that

$$\text{for almost all } (\dot{C}', \dot{\sigma}', \eta') \in \dot{\mu}, \text{ it holds that } (\dot{C}', \dot{\sigma}', \eta') \in \dot{\mathrm{FC}}. \tag{1.78}$$

Since $\dot{R}$ is a generalized redex, and $\dot{E}$ is a generalized evaluation context, we have that $\dot{E}[P_1/_\varepsilon]$ and $\dot{R}[P_1/_\varepsilon]$ are an evaluation context and a redex, respectively. Thus $(\dot{E}[\dot{R}])[P_1/_\varepsilon]$ is not a value. Therefore, we have $T_0((\dot{E}[\dot{R}], \dot{\sigma}, \eta)[P_1/_\varepsilon]) = 0$ and thus $n > 0$ by Definition 1.72. Thus for

$P \in \{P_1, P_2\}$ we have

$$\begin{aligned} T_n((\dot{E}[\dot{R}], \dot{\sigma}, \eta)[P/\varepsilon]) &= \int T_{n-1}(C|\sigma|\eta)\ d(step((\dot{E}[\dot{R}], \dot{\sigma}, \eta)[P/\varepsilon]))(C|\sigma|\eta) \\ &= \int T_{n-1}((\dot{C}, \dot{\sigma}, \eta)[P/\varepsilon])\ d\dot{\mu}(\dot{C}, \dot{\sigma}, \eta) \\ &\overset{(1.78)}{=} \int_{\dot{\mathrm{FC}}} T_{n-1}((\dot{C}, \dot{\sigma}, \eta)[P/\varepsilon])\ d\dot{\mu}(\dot{C}, \dot{\sigma}, \eta) \end{aligned} \tag{1.79}$$

and by taking the limit of n on both sides we get

$$T((\dot{E}[\dot{R}], \dot{\sigma}, \eta)[P/\varepsilon]) = \int_{\dot{\mathrm{FC}}} T((\dot{C}, \dot{\sigma}, \eta)[P/\varepsilon])\ d\dot{\mu}(\dot{C}, \dot{\sigma}, \eta) \tag{1.80}$$

Assume that for all $(\dot{C}', \dot{\sigma}', \eta') \in \dot{\mathrm{FC}}$ we have $T_{n-1}((\dot{C}', \dot{\sigma}', \eta')[P_1/\varepsilon]) \leq T((\dot{C}', \dot{\sigma}', \eta')[P_2/\varepsilon])$. By (1.79) and (1.80), it holds $T_n((\dot{E}[\dot{R}], \dot{\sigma}, \eta)[P_1/\varepsilon]) \leq T((\dot{E}[\dot{R}], \dot{\sigma}, \eta)[P_2/\varepsilon])$ which contradicts the fact that $(P_1, P_2, \dot{E}[\dot{R}], \dot{\sigma}, \eta, n)$ is a CIU counterexample.

Thus there is a $(\dot{C}', \dot{\sigma}', \eta') \in \dot{\mathrm{FC}}$ such that $T_{n-1}((\dot{C}', \dot{\sigma}', \eta')[P_1/\varepsilon]) > T((\dot{C}', \dot{\sigma}', \eta')[P_2/\varepsilon])$. Hence $(P_1, P_2, \dot{C}', \dot{\sigma}', \eta', n-1)$ is a CIU counterexample. □

Now we consider the case where $\dot{C}$ can be split into a generalized evaluation context $\dot{E}$ and an ε-term $\varepsilon^{\dot{a}}$ for some generalized instantiation $\dot{a}$. We will see that in this case we can also find a smaller CIU counterexample. For this, we consider the term $\dot{D} \overset{\text{def}}{=} \dot{E}[P_1^{\dot{a}}]$ and show that $(P_1, P_2, \dot{D}, \dot{\sigma}, \eta, n)$ is also a CIU counterexample. If P_1 is a value, this is already a smaller CIU counterexample, as then $\#_\varepsilon P_1^{\dot{a}} = 0$ and hence $\#_\varepsilon \dot{D} < \#_\varepsilon \dot{C}$. If P_1 is not a value, then $\dot{D}$ can be split into a generalized evaluation context and a generalized redex and we can use Lemma 1.77 to obtain a smaller CIU counterexample.

I.485, p.159 **Lemma 1.81.** *Let $(P_1, P_2, \dot{C}, \dot{\sigma}, \eta, n)$ be a CIU counterexample where $\dot{C} = \dot{E}[\varepsilon^{\dot{a}}]$ for some generalized evaluation context $\dot{E}$ and some generalized instantiation $\dot{a}$. Then there exists a CIU counterexample $(P_1, P_2, \dot{C}', \dot{\sigma}', \eta', n')$ where $n' < n$ or $n' = n \wedge \#_\varepsilon \dot{C}' < \#_\varepsilon \dot{C}$.*

Proof. Let $X \overset{\text{def}}{=} (P_1, P_2, \dot{C}, \dot{\sigma}, \eta, n)$. Since X is a CIU counterexample, it holds that $(\dot{C}, \dot{\sigma}, \eta)[P_1/\varepsilon]$ and $(\dot{C}, \dot{\sigma}, \eta)[P_2/\varepsilon]$ are fully closed and $(\dot{C}, \dot{\sigma})$ is $[\{P_1, P_2\}/\varepsilon]$-closed. Thus, in P_1, P_2, $\dot{C}$, and $\dot{\sigma}$, no locations $\mathsf{loc}\, l$ with $l \geq |\dot{\sigma}|$ occur. Furthermore $\dot{E}$, $P_1^{\dot{a}[P_1/\varepsilon]}$, $P_2^{\dot{a}[P_2/\varepsilon]}$, and $\dot{\sigma}$ contain no free variables. Since $(\dot{C}, \dot{\sigma})$ is $[\{P_1, P_2\}/\varepsilon]$-closed, $\dot{a}[P_1/\varepsilon]$ and $\dot{a}[P_2/\varepsilon]$ substitute only with closed values. Therefore both instantiations substitute the same indices with
I.484, p.159 closed values, and hence $P_1^{\dot{a}[P_2/\varepsilon]}$ contains no free variables as well.

Thus $(P_1^{\dot{a}[P_2/\varepsilon]} \mid \dot{\sigma}[P_2/\varepsilon] \mid \eta)$ and $(P_2^{\dot{a}[P_2/\varepsilon]} \mid \dot{\sigma}[P_2/\varepsilon] \mid \eta)$ are fully closed. Since $\dot{a}[P_2/\varepsilon]$ is a programterm instantiation, we can use Lemma 1.70 to

obtain a list v of closed values without locations $\mathsf{loc}\, l$ with $l \geq |\dot{\sigma}|$ such that $P_1^{\dot{a}[P_2/\varepsilon]} = P_1^v$ and $P_2^{\dot{a}[P_2/\varepsilon]} = P_2^v$.

Thus $(\dot{E}[P_1^v], \dot{\sigma}, \eta)[P_2/\varepsilon]$ and $(\dot{E}[P_2^v], \dot{\sigma}, \eta)[P_2/\varepsilon]$ are fully closed and since $P_1 \preceq_{ciu} P_2$ we obtain $T((\dot{E}[P_1^v], \dot{\sigma}, \eta)[P_2/\varepsilon]) \leq T((\dot{E}[P_2^v], \dot{\sigma}, \eta)[P_2/\varepsilon])$ and hence

$$T(\dot{E}[P_2/\varepsilon][P_1^{\dot{a}[P_2/\varepsilon]}] \mid \dot{\sigma}[P_2/\varepsilon] \mid \eta) \;\leq\; T(\dot{E}[P_2/\varepsilon][P_2^{\dot{a}[P_2/\varepsilon]}] \mid \dot{\sigma}[P_2/\varepsilon] \mid \eta). \tag{1.82}$$

Consider the term $\dot{D} \stackrel{\text{def}}{=} \dot{E}[P_1^{\dot{a}}]$. Note that $\dot{D}[P_1/\varepsilon] = \dot{C}[P_1/\varepsilon]$ and $\dot{D}[P_2/\varepsilon] = \dot{E}[P_2/\varepsilon][P_1^{\dot{a}[P_2/\varepsilon]}]$. Then it holds

$$\begin{aligned}
&T_n((\dot{D}, \dot{\sigma}, \eta)[P_1/\varepsilon]) \\
&= T_n((\dot{C}, \dot{\sigma}, \eta)[P_1/\varepsilon]) \\
&\overset{(*)}{>} T((\dot{C}, \dot{\sigma}, \eta)[P_2/\varepsilon]) \\
&= T(\dot{E}[P_2/\varepsilon][P_2^{\dot{a}[P_2/\varepsilon]}] \mid \dot{\sigma}[P_2/\varepsilon] \mid \eta) \\
&\overset{(1.82)}{\geq} T(\dot{E}[P_2/\varepsilon][P_1^{\dot{a}[P_2/\varepsilon]}] \mid \dot{\sigma}[P_2/\varepsilon] \mid \eta) \\
&= T((\dot{D}, \dot{\sigma}, \eta)[P_2/\varepsilon])
\end{aligned}$$

where $(*)$ holds because X is a CIU counterexample.

Since $\dot{D}$ contains no new ε-terms, $(\dot{D}, \dot{\sigma})$ is $[\{P_1, P_2\}/\varepsilon]$-closed. Furthermore $(\dot{D}, \dot{\sigma}, \eta)[P_1/\varepsilon]$ and $(\dot{D}, \dot{\sigma}, \eta)[P_2/\varepsilon]$ are fully closed, and hence $Y \stackrel{\text{def}}{=} (P_1, P_2, \dot{D}, \dot{\sigma}, \eta, n)$ is a CIU counterexample. To finish the proof we consider the following two cases:

- If P_1 is a value, then $P_1^{\dot{a}}$ is a generalized value and hence $\#_\varepsilon P_1^{\dot{a}} = 0$. I.412, p.151 Therefore we have $\#_\varepsilon \dot{D} = \#_\varepsilon \dot{E}[P_1^{\dot{a}}] < \#_\varepsilon \dot{E}[\varepsilon^{\dot{a}}] = \#_\varepsilon \dot{C}$. Thus Y is a smaller counterexample than X.
- If P_1 is not a value, then $P_1 = E_1[R_1]$ where E_1 is an evaluation context and R_1 is a redex. Then $E_1^{\dot{a}}$ is a generalized evaluation context and $R_1^{\dot{a}}$ is a generalized redex. Hence, by Lemma 1.77, there is a smaller CIU counterexample $(P_1, P_2, \dot{C}', \dot{\sigma}', \eta', n-1)$. □

Lemma 1.83. *There is no minimal CIU counterexample.* I.486, p.159

Proof. For contradiction, let $X = (P_1, P_2, \dot{C}, \dot{\sigma}, \eta, n)$ be a minimal CIU counterexample. If $\dot{C}$ is a generalized value then $\dot{C}[P_2/\varepsilon]$ is a value and thus $T_n((\dot{C}, \dot{\sigma}, \eta)[P_1/\varepsilon]) \leq 1 = T((\dot{C}, \dot{\sigma}, \eta)[P_2/\varepsilon])$, which contradicts the fact that X is a CIU counterexample.

Therefore $\dot{C}$ is not a generalized value and using Lemma 1.74, $\dot{C} = \dot{E}[\dot{P}]$ where $\dot{E}$ is a generalized evaluation context and $\dot{P}$ is either a generalized redex or $\dot{P} = \varepsilon^{\dot{a}}$ for some generalized instantiation $\dot{a}$.

If $\dot{P}$ is a generalized redex, using Lemma 1.77 we obtain a smaller counterexample which contradicts the minimality of X.

Likewise, if $\dot{P} = \varepsilon^{\dot{a}}$ for some generalized instantiation $\dot{a}$, by Lemma 1.81 there also exists a smaller counterexample which contradicts the minimality of X. □

Using the previous Lemma 1.83 and Lemma 1.73 it is straightforward to prove the following theorem:

I.488, p.159 **Theorem 1.84.** *Let P_1 and P_2 be programs. If $P_1 \preceq_{ciu} P_2$, then $P_1 \preceq_{obs} P_2$.*

Proof. Assume for contradiction that $P_1 \preceq_{ciu} P_2$ but not $P_1 \preceq_{obs} P_2$, then by Lemma 1.73 there exists a CIU counterexample. Since the lexicographic order on CIU counterexamples from Definition 1.72 is well-founded, there
I.487, p.159 also exists a minimal CIU counterexample. This contradicts Lemma 1.83. □

Proving the CIU Theorem 1.58 is now straightforward. It follows directly from Theorem 1.84 and the lemmas 1.61 and 1.62.

1.12 Program Transformations

Game-based proofs consist of a sequence of games or programs where consecutive programs in this sequence are connected by some relation as observational equivalence or computational indistinguishability. In this section we will present a set of rules that can be used to transform programs into equivalent ones. The rules themselves are based on the observational equivalence of small concise programs that are matched against subterms of the game to be transformed. We prove the correctness of the rules by verifying their corresponding statements of observational equivalence. The proofs heavily rely on properties of the language that were presented in Section 1.11, in particular the chaining rule for denotations (Theorem 1.48) and the CIU Theorem 1.58.

1.12.1 Using $\approx_{obs}$ to Transform Programs

In a typical step of a game-based proof, we need to establish the observational equivalence of two programs, or games, G_1 and G_2. In order to show $G_1 \approx_{obs} G_2$, we can transform the games in an observationally equivalent manner, e.g., game G_1 into G_1', given that $G_1 \approx_{obs} G_1'$. This leaves us with the, hopefully easier, task of proving the equivalence $G_1' \approx_{obs} G_2$. We call rules of the following form *program transformations*:

$$\frac{G_1' \approx_{obs} G_2'}{G_1 \approx_{obs} G_2}$$

Program transformations are usually based on the observational equivalence of small concise programs by exploiting the composability of $\approx_{obs}$ (Lemma 1.36). For example, given the equivalence $P \approx_{obs} P'$, we can easily construct the following transformations for arbitrary contexts C and games G:

$$\frac{C[P'] \approx_{obs} G}{C[P] \approx_{obs} G} \qquad \frac{G \approx_{obs} C[P']}{G \approx_{obs} C[P]}$$

These transformations allow us to match a subterm of a game against program P and to replace it with the program P'. This process can be further simplified by using context functions (Definition 1.31) instead of contexts, as this allows us to take advantage of Isabelle's unification machinery to automatically infer the context function and to match the program P.

Since statements of observational equivalence and program transformations are so closely related, in the following we will only present the observational equivalences and omit their corresponding program transformations.

1.12.2 Transformations based on Computation Rules

A useful class of transformations deals with the evaluation of subterms of a program. For example given a program containing the redex $(\lambda P)V$, we would like to replace this redex with the program $P\{V/0\}$, i.e., the redex is replaced with the program it reduces to according to the rule BETA of the $\rightsquigarrow$ relation (Figure 1.18). This is a valid transformation and the transformed program is observationally equivalent to the original program containing the redex $(\lambda P)V$. We will now present the required steps to prove the validity of this transformation.

First, we will prove that two programs are observationally equivalent, if one reduces to the other for all instantiations of their variables. Namely, we prove the following lemma:

Lemma 1.85 (Observationally Equivalent Reduction)**.** *Given two programs P and P', assume that for all lists of values v, stores σ and event lists η it holds $P^v|\sigma|\eta \rightsquigarrow \delta(P'^v|\sigma|\eta)$. Then it holds $P \approx_{obs} P'$.* I.491, p.160

Proof. Using the CIU Theorem 1.58, it suffices to show that $P \approx_{ciu} P'$. In particular, it suffices to show that for all evaluation contexts E, lists of values v, stores σ and event lists η it holds $[\![E[P^v]|\sigma|\eta]\!] = [\![E[P'^v]|\sigma|\eta]\!]$. Given the assumption, we can infer from the definition of the denotation that $[\![P^v|\sigma|\eta]\!]$ and $[\![P'^v|\sigma|\eta]\!]$ are equal for all lists of values v, stores σ and event lists η. Using the chaining rule for denotations (Theorem 1.48), we conclude the proof as follows:

$$\begin{aligned}
[\![E[P^v]|\sigma|\eta]\!] &= (\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![P^v|\sigma|\eta]\!] \\
&= (\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![P'^v|\sigma|\eta]\!] \\
&= [\![E[P'^v]|\sigma|\eta]\!]
\end{aligned}$$

□

This lemma entails that the programs $(\lambda P)V$ and $P\{V/_0\}$ are observationally equivalent. Note that the lemma applies, because $((\lambda P)V)^v$ reduces to $(P\{V/_0\})^v$, as we have seen in Lemma 1.52.

I.493, p.160 **Lemma 1.86** (Beta Transformation)**.** *Given a program P and a value V, we have the equivalence $(\lambda P)V \approx_{obs} P\{V/_0\}$.*

This lemma implies the validity of the following transformation: Given a program containing the redex $(\lambda P)V$, we can replace the redex with $P\{V/_0\}$. The resulting program will be observationally equivalent to the original program.

Analogously, we can use Lemma 1.85 to show that, given two values V_1 and V_2, the programs $\mathsf{fst}\,(V_1, V_2)$ and V_1 are observationally equivalent. Note that the lemma applies, because for all values V and lists of values v it holds that V^v is also a value, and therefore $(\mathsf{fst}\,(V_1, V_2))^v|\sigma|\eta \rightsquigarrow \delta({V_1}^v|\sigma|\eta)$.

I.495, p.160 **Lemma 1.87.** *Given values V_1 and V_2, it holds $\mathsf{fst}\,(V_1, V_2) \approx_{obs} V_1$.*

The corresponding program transformation which replaces the redex $\mathsf{fst}\,(V_1, V_2)$ with V_1 is already quite useful, as it allows to simplify subterms of a program, but the transformation can still be improved. Consider the program $\mathsf{fst}\,(P, V)$, where V is a value and P is an arbitrary program. To evaluate this program, one first evaluates P and whenever one of its evaluation branches has evaluated to a value V' one performs the step $(\mathsf{fst}\,(V', V))|\sigma|\eta \rightsquigarrow \delta(V'|\sigma|\eta)$. This means that the programs $\mathsf{fst}\,(P, V)$ and P evaluate to the same distribution. In fact, the programs $\mathsf{fst}\,(P, V)$ and P are observationally equivalent, which gives us a more general program transformation, as we only require the second component of the pair to be a value.

To prove the observational equivalence $\mathsf{fst}\,(P, V) \approx_{obs} P$, we first prove a generalized statement which we can reuse to validate similar transformations as well. We will prove a theorem that uses evaluation contexts to lift observational equivalence from values V to arbitrary programs P.

I.492, p.160 **Theorem 1.88.** *Let E_1 and E_2 be evaluation contexts such that for all values V it holds $E_1[V] \approx_{obs} E_2[V]$. Then for all programs P it holds $E_1[P] \approx_{obs} E_2[P]$.*

Proof. Using the CIU Theorem 1.58, it suffices to show the equivalence $E_1[P] \approx_{ciu} E_2[P]$. So let E be an evaluation context, σ a store, η an event list, and v a list of values without free variables and without locations $\geq |\sigma|$, such that $E[(E_1[P])^v]|\sigma|\eta$ and $E[(E_2[P])^v]|\sigma|\eta$ are fully closed. To complete the proof, we need to show $[\![E[(E_1[P])^v]|\sigma|\eta]\!]\,(\Omega) = [\![E[(E_2[P])^v]|\sigma|\eta]\!]\,(\Omega)$. Using the chaining rule, we obtain the following two equalities:

$$[\![E[(E_1[P])^v]|\sigma|\eta]\!]\,(\Omega) = (\lambda(V|\sigma'|\eta').\ [\![E[E_1^v[V]]|\sigma'|\eta']\!]) \cdot [\![P^v|\sigma|\eta]\!](\Omega)$$
$$[\![E[(E_2[P])^v]|\sigma|\eta]\!]\,(\Omega) = (\lambda(V|\sigma'|\eta').\ [\![E[E_2^v[V]]|\sigma'|\eta']\!]) \cdot [\![P^v|\sigma|\eta]\!](\Omega)$$

The right hand sides only differ in the terms $E_1^v[V]$ and $E_2^v[V]$. Therefore, to show the equality $[\![E[(E_1[P])^v]|\sigma|\eta]\!](\Omega) = [\![E[(E_2[P])^v]|\sigma|\eta]\!](\Omega)$, it suffices to show that $E_1^v[V] \approx_{obs} E_2^v[V]$, where V is a value and $E_1^v[V]$, $E_2^v[V]$, and V contain no free variables. As before, we use the CIU Theorem and prove $E_1^v[V] \approx_{ciu} E_2^v[V]$. So let E' be an evaluation context, σ' a store, η' an event list, and v' a list of values, such that $E'[(E_1^v[V])^{v'}]|\sigma'|\eta'$ and $E'[(E_2^v[V])^{v'}]|\sigma'|\eta'$ are fully closed. Note that since $E_1^v[V]$ and $E_2^v[V]$ are closed, the instantiation with v' has no effect on them. Using Lemma 1.55 let C_v be a substituting context such that $C_v[E_1[V]]|\sigma'|\eta'$ and $C_v[E_2[V]]|\sigma'|\eta'$ are fully closed, $[\![C_v[E_1[V]]|\sigma'|\eta']\!] = [\![E_1^v[V]|\sigma'|\eta']\!]$, and $[\![C_v[E_2[V]]|\sigma'|\eta']\!] = [\![E_2^v[V]|\sigma'|\eta']\!]$. Using the chaining rule, we conclude the proof as follows:

$$
\begin{aligned}
&[\![E'[E_1^v[V]]|\sigma'|\eta']\!](\Omega)\\
&= (\lambda(V''|\sigma''|\eta'').\ [\![E'[V'']|\sigma''|\eta'']\!]) \cdot [\![E_1^v[V]|\sigma'|\eta']\!](\Omega)\\
&= (\lambda(V''|\sigma''|\eta'').\ [\![E'[V'']|\sigma''|\eta'']\!]) \cdot [\![C_v[E_1[V]]|\sigma'|\eta']\!](\Omega)\\
&= (\lambda(V''|\sigma''|\eta'').\ [\![E'[V'']|\sigma''|\eta'']\!]) \cdot [\![C_v[E_2[V]]|\sigma'|\eta']\!](\Omega)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{since } E_1[V] \approx_{obs} E_2[V])\\
&= (\lambda(V''|\sigma''|\eta'').\ [\![E'[V'']|\sigma''|\eta'']\!]) \cdot [\![E_2^v[V]|\sigma'|\eta']\!](\Omega)\\
&= [\![E'[E_2^v[V]]|\sigma'|\eta']\!](\Omega) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \square
\end{aligned}
$$

Validating the transformation that replaces $\mathsf{fst}\,(P, V)$ with P is now straightforward. The observational equivalence follows from Theorem 1.88 and Lemma 1.87 by using the evaluation contexts $E_1 = \mathsf{fst}\,(\square, V)$ and $E_2 = \square$.

Lemma 1.89. *Given a program P and a value V, it holds* I.496, p.160

$$\mathsf{fst}\,(P, V) \approx_{obs} P.$$

Analogously to Lemma 1.89, we can deduce the validity of further transformations. In particular, we can state similar observational equivalences for the operators snd and unfold and for the case construct.

Lemma 1.90. *Given a value V and a program P, it holds* I.498, p.160

$$\mathsf{snd}\,(V, P) \approx_{obs} P.$$

Lemma 1.91. *Given a program P, it holds* I.505, p.161

$$\mathsf{unfold}\,(\mathsf{fold}\,P) \approx_{obs} P.$$

Lemma 1.92. *Given a program P and values V_1 and V_2, it holds* I.501, p.161

$$\mathsf{case}\ (\mathsf{inl}\,P)\ V_1\ V_2 \approx_{obs} V_1 P.$$

Lemma 1.93. *Given a program P and values V_1 and V_2, it holds* I.503, p.161

$$\mathsf{case}\ (\mathsf{inr}\,P)\ V_1\ V_2 \approx_{obs} V_2 P.$$

1.12.3 Expression Propagation

In Lemma 1.86 we presented a transformation that is able to reduce programs according to the rule BETA of the $\rightsquigarrow$ relation (Figure 1.18). It can transform programs of the form $(\lambda P)V$ by substituting the value V into the program P, yielding program $P\{V/_0\}$. This is a very useful transformation, but it is only able to substitute values. We will now present a transformation that can, given certain assumptions, transform programs of the form $(\lambda P')P$ by substituting program P into P', yielding $P'\{P/_0\}$.

Note that we do not require P to be a value. Nevertheless, this generalization comes at a cost: The programs $(\lambda P')P$ and $P'\{P/_0\}$ are not observationally equivalent in general. For instance, program $P'\{P/_0\}$ may contain several instances of program P. If this is the case and P has probabilistic behavior, then different instances might evaluate to different values, whereas in program $(\lambda P')P$ this cannot happen, since program P is evaluated only once. A similar problem occurs, if P has side-effects which would occur only once in program $(\lambda P')P$ but multiple times in program $P'\{P/_0\}$. The definition of *propagatable* programs takes these considerations into account.

I.512, p.162 **Definition 1.94** (Propagatable Programs)**.** *We call a program P* propagatable*, iff for all lists of closed values v such that P^v is closed, it either holds that*

- *there exists a value V such that for all stores σ and event lists η such that $P^v|\sigma|\eta$ is fully closed, it holds that $V|\sigma|\eta$ is fully closed and $[\![P^v|\sigma|\eta]\!] = \delta(V|\sigma|\eta)$, or*
- *for all stores σ and event lists η such that $P^v|\sigma|\eta$ is fully closed, it holds $[\![P^v|\sigma|\eta]\!] = 0$.*

The first condition in Definition 1.94 handles the considerations we mentioned above: It requires the program P to be deterministic without showing side-effects that affect the store or the event list. The condition requires this behavior for all instantiations v of the free variables in P. Because this may include instantiations resulting in an ill-typed program P^v, the evaluation of P^v might get stuck. Therefore we cannot require that P^v always evaluates to a value V. The second condition in Definition 1.94 relaxes this requirement by allowing P^v to not terminate at all. In Lemma 1.96 below we give two examples of propagatable programs.

The relaxation to non-terminating programs P raises another issue that can lead to different termination behaviors of $(\lambda P')P$ and $P'\{P/_0\}$. For instance, if P does not occur in $P'\{P/_0\}$ at all, then $P'\{P/_0\}$ might terminate while $(\lambda P')P$ does not. Therefore in our transformation we require that $P'\{P/_0\}$ does not terminate whenever P does not terminate. A manual proof of this transformation has been prepared in [74]. We will now present its formal verification in Isabelle/HOL.

Lemma 1.95 (Expression Propagation)**.** *Let P and P' be programs such that P is propagatable. Assume that for all lists of closed values v, stores σ, and event lists η such that P^v is closed, it holds that, if $[\![P^v|\sigma|\eta]\!] = 0$ then also $[\![(P'\{P/\!_0\})^v|\sigma|\eta]\!] = 0$. Then it holds that* I.513, p.162

$$(\lambda P')P \quad \approx_{obs} \quad P'\{P/\!_0\}.$$

Proof. Using the CIU Theorem 1.58, we will prove $(\lambda P')P \approx_{ciu} P'\{P/_0\}$. So let E be an evaluation context, σ a store, η an event list, and v a list of closed values without locations $\geq |\sigma|$, such that $E[((\lambda P')P)^v]|\sigma|\eta$ and $E[(P'\{P/\!_0\})^v]|\sigma|\eta$ are fully closed. To complete the proof, we need to show $[\![E[((\lambda P')P)^v]|\sigma|\eta]\!]\,(\Omega) = [\![E[(P'\{P/\!_0\})^v]|\sigma|\eta]\!]\,(\Omega)$. Since program P is propagatable, we distinguish two cases:

Case 1: Assume there exists a value V such that for all stores σ' and event lists η' such that $P^v|\sigma'|\eta'$ is fully closed, it holds that $V|\sigma'|\eta'$ is fully closed and $[\![P^v|\sigma'|\eta']\!] = \delta(V|\sigma'|\eta') = [\![V|\sigma'|\eta']\!]$. Using Lemma 1.57 and the CIU Theorem 1.58, it follows that $P^v \approx_{obs} V$.

Since $E[((\lambda P')P)^v]|\sigma|\eta$ is fully closed, $P^v|\sigma|\eta$ is fully closed. Therefore $V|\sigma|\eta$ is also fully closed. Together with the fully closedness of $E[(P'\{P/\!_0\})^v]|\sigma|\eta$ it then follows that $(P'\{V/\!_0\})^v|\sigma|\eta$ is fully closed and Lemma 1.53 implies that $(P'\{P^v/\!_0\})^v|\sigma|\eta$ is fully closed as well.

Therefore Lemma 1.55 applies and there is a substituting context $C_{v'}$ such that the states $C_{v'}[P'\{V/\!_0\}]|\sigma|\eta$ and $C_{v'}[P'\{P^v/\!_0\}]|\sigma|\eta$ are fully closed with the property that $[\![C_{v'}[P'\{V/\!_0\}]|\sigma|\eta]\!] = [\![(P'\{V/\!_0\})^v|\sigma|\eta]\!]$ and $[\![C_{v'}[P'\{P^v/\!_0\}]|\sigma|\eta]\!] = [\![(P'\{P^v/\!_0\})^v|\sigma|\eta]\!]$. We conclude as follows:

$$
\begin{aligned}
&[\![E[((\lambda P')P)^v]|\sigma|\eta]\!]\,\Omega \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[(\lambda P')^v V']|\sigma'|\eta']\!]) \cdot [\![P^v|\sigma|\eta]\!]\big)\,\Omega && \text{(Theorem 1.48)} \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[(\lambda P')^v V']|\sigma'|\eta']\!]) \cdot [\![V|\sigma|\eta]\!]\big)\,\Omega && \text{(by assumption)} \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![(\lambda P')^v V|\sigma|\eta]\!]\big)\,\Omega && \text{(Theorem 1.48)} \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![((\lambda P')V)^v|\sigma|\eta]\!]\big)\,\Omega && (V \text{ is closed}) \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![(P'\{V/\!_0\})^v|\sigma|\eta]\!]\big)\,\Omega && \text{(Lemma 1.52)} \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![C_{v'}[P'\{V/\!_0\}]|\sigma|\eta]\!]\big)\,\Omega \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![C_{v'}[C_{P'}[V]]|\sigma|\eta]\!]\big)\,\Omega && \text{(Lemma 1.16)} \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![C_{v'}[C_{P'}[P^v]]|\sigma|\eta]\!]\big)\,\Omega && (P^v \approx_{obs} V) \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![C_{v'}[P'\{P^v/\!_0\}]|\sigma|\eta]\!]\big)\,\Omega && \text{(Lemma 1.16)} \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![(P'\{P^v/\!_0\})^v|\sigma|\eta]\!]\big)\,\Omega \\
&= \big((\lambda(V'|\sigma'|\eta').\ [\![E[V']|\sigma'|\eta']\!]) \cdot [\![(P'\{P/\!_0\})^v|\sigma|\eta]\!]\big)\,\Omega && \text{(Lemma 1.53)} \\
&= [\![E[(P'\{P/\!_0\})^v]|\sigma|\eta]\!]\,\Omega && \text{(Theorem 1.48)}
\end{aligned}
$$

Case 2: Assume it holds $[\![P^v|\sigma|\eta]\!] = 0$, then by assumption it also holds $[\![(P'\{P/\!\!0\})^v|\sigma|\eta]\!] = 0 = [\![P^v|\sigma|\eta]\!]$. We conclude the lemma as follows:

$$\begin{aligned}
&[\![E[((\lambda P')P)^v]|\sigma|\eta]\!]\,(\Omega)\\
&= (\lambda(V'|\sigma'|\eta').\ [\![E[(\lambda P')^v V']|\sigma'|\eta']\!]) \cdot [\![P^v|\sigma|\eta]\!]\\
&= (\lambda(V'|\sigma'|\eta').\ [\![E[(\lambda P')^v V']|\sigma'|\eta']\!]) \cdot [\![(P'\{P/\!\!0\})^v|\sigma|\eta]\!]\\
&= [\![E[(P'\{P/\!\!0\})^v]|\sigma|\eta]\!]\,(\Omega)
\end{aligned}$$

□

To give an example application of this transformation, consider the following program with free variable x, where the operator fst is applied to x.

$$(\lambda x'.\, P')(\mathsf{fst}\, x)$$

We would like to propagate the expression $\mathsf{fst}\, x$ into program P'. Note that Lemma 1.86 (the transformation based on rule Beta of the $\rightsquigarrow$ relation) does not apply here, because $\mathsf{fst}\, x$ is not a value. But assuming that the preconditions of our new transformation from Lemma 1.95 are fulfilled, we can propagate the expression $\mathsf{fst}\, x$ into P'. In fact, we can show that the expression $\mathsf{fst}\, x$ is propagatable: If x is instantiated with a pair (V_1, V_2), then $\mathsf{fst}\,(V_1, V_2)|\sigma|\eta$ evaluates to $\delta(V_1|\sigma|\eta)$. If x is instantiated with any value but a pair, then the expression $\mathsf{fst}\, x$ is stuck and its denotation is 0. Analogously, we can also show that the expression $\mathsf{snd}\, x$ is propagatable.

I.515, p.162 I.516, p.162 **Lemma 1.96.** *Given an index $n \in \mathbb{N}$, the expressions $\mathsf{fst}\,(\mathsf{var}\, n)$ and $\mathsf{snd}\,(\mathsf{var}\, n)$ are propagatable.*

1.12.4 Inlining let Statements

Many programs we encounter have a linear structure, i.e., they consist of a sequence of let statements, where programs P_i are evaluated and their results are bound to variables x_i:

$$\begin{array}{l}
\mathsf{let}\ x_1 \leftarrow P_1;\\
\quad\ x_2 \leftarrow P_2;\\
\quad\ \vdots\\
\quad\ x_n \leftarrow P_n\\
\mathsf{in}\ P
\end{array}$$

If such a linear program is used inside another program, the overall structure is not linear anymore. This situation is depicted in the left-hand side of Figure 1.97. We will now present a program transformation that can stepwise inline this program into the enclosing one. For this, each step moves one line from the inner program to the enclosing program until the inner program has been completely inlined. This inlining process is illustrated in

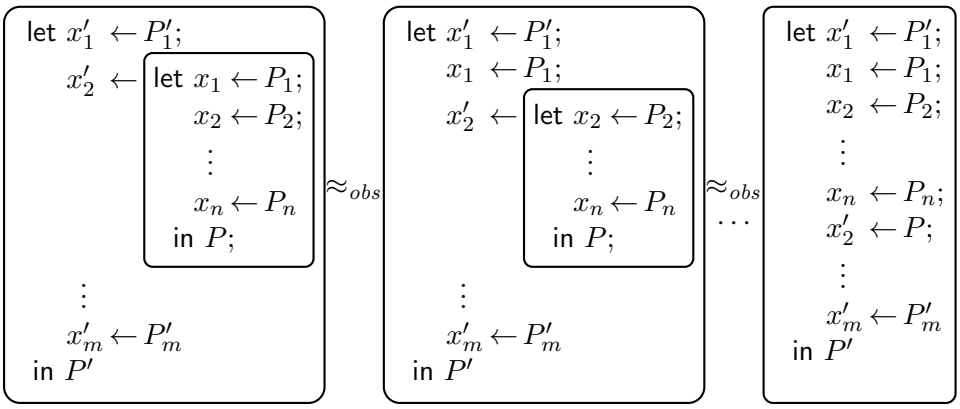


Figure 1.97: Illustration of the inlining of let statements.

Figure 1.97, where the program resulting from this inlining transformation is depicted in the right-hand side.

To prove the validity of this transformation, it is enough to consider the case where the inner program and the enclosing one only contain a single line: Using the composability of the observational equivalence (Lemma 1.36), each step in Figure 1.97 is an instance of the equivalence

$$\mathsf{let}\ x_2 \leftarrow (\mathsf{let}\ x_1 \leftarrow P_1\ \mathsf{in}\ P_2)\ \mathsf{in}\ P_3\ \approx_{obs}\ \mathsf{let}\ x_1 \leftarrow P_1;\ x_2 \leftarrow P_2\ \mathsf{in}\ P_3. \tag{1.98}$$

Note that the scope of variable x_1 differs on both sides of Equation 1.98. On the right-hand side P_3 is in the scope of x_1 whereas on the left-hand side it is not. Because the underlying language uses nameless de Bruijn indices, the transformation will have to lift the free variables of P_3 accordingly in order to adapt to the changed scope. The following lemma states the validity of Equation 1.98 in its nameless representation.

Lemma 1.99 (Inlining let Statements)**.** *Given programs P_1, P_2, and P_3,* I.506, p.161 *the following equivalence holds*

$$(\lambda P_3)\ ((\lambda P_2)\ P_1) \quad \approx_{obs} \quad (\lambda(\lambda\uparrow_1 P_3)\ P_2)\ P_1.$$

Proof. We define the evaluation contexts $E_1 \stackrel{\text{def}}{=} (\lambda P_3)\ ((\lambda P_2)\ \Box)$ and $E_2 \stackrel{\text{def}}{=} (\lambda(\lambda\uparrow_1 P_3)\ P_2)\ \Box$. Using Theorem 1.88 we prove the lemma by showing that $E_1[V] \approx_{obs} E_2[V]$ for all values V. We apply the Beta Transformation from Lemma 1.86 on both sides of the equivalence and obtain the goal $(\lambda P_3)\ (P_2\{V/_0\}) \approx_{obs} ((\lambda\uparrow_1 P_3)\ P_2)\{V/_0\}$. We have seen in Lemma 1.15 that the operators $\uparrow_k$ and $\{\cdot/_k\}$ cancel out for equal indices k, and hence it holds that $(\uparrow_1 P_3)\{V'/_1\} = P_3$ for arbitrary V'. The lemma follows from

$$((\lambda\uparrow_1 P_3)\ P_2)\{V/_0\} = (\lambda(\uparrow_1 P_3)\{\uparrow_0 V/_1\})\ (P_2\{V/_0\}) = (\lambda P_3)\ (P_2\{V/_0\}). \qquad \Box$$

1.12.5 Line Swapping

In order to apply a specific transformation, it is often necessary to regroup the subterms of a program. Therefore one often needs to reorder the evaluation sequence of a program. We will now present a transformation that can alter the order in which a program evaluates its subterms. In particular, we identify a simple condition under which it is possible to swap the evaluation order of two programs.

Swapping the evaluation order of two programs is only possible, if the programs are independent of each other, i.e., the evaluation of one program must not affect the evaluation of the other; neither through the use of variables nor through accessing the state. For our transformation we exclude the use of variables by lifting the programs accordingly, and we exclude dependencies through the state by requiring one of the programs to be completely independent of the state.

I.519, p.162 **Definition 1.100** (State Independent Programs)**.** *A program P is called* state independent*, iff for all stores σ, event lists η, and lists of closed values v without locations $\geq |\sigma|$ such that $P^v|\sigma|\eta$ is fully closed, it holds that* $[\![P^v|\sigma|\eta]\!] = (\lambda(V'|\sigma'|\eta').\,(V'|\sigma|\eta))\ \ [\![P^v|[]|[]]\!]$

In the following we assume two programs P_1 and P_2 where P_1 is state independent. We will first show how to swap the evaluation order of P_1 and P_2 in a simple program and later see how to generalize this result. We first consider the following programs $P_{1;2}$ and $P_{2;1}$:

$$P_{1;2} \stackrel{\text{def}}{=} \boxed{\begin{array}{l}\mathsf{let}\ x_1 \leftarrow P_1;\\ \quad\ x_2 \leftarrow \Uparrow_0 P_2\\ \mathsf{in}\ (x_1, x_2)\end{array}} \qquad P_{2;1} \stackrel{\text{def}}{=} \boxed{\begin{array}{l}\mathsf{let}\ x_2 \leftarrow P_2;\\ \quad\ x_1 \leftarrow \Uparrow_0 P_1\\ \mathsf{in}\ (x_1, x_2)\end{array}}$$

Note that in both $P_{1;2}$ and $P_{2;1}$, we lift the program in the second line so that it is independent of the variable bound in the first line. The following lemma states the observational equivalence of $P_{1;2}$ and $P_{2;1}$ in their corresponding nameless representations.

I.520, p.162 **Lemma 1.101.** *Given programs P_1 and P_2 where P_1 is state independent, let the program $P_{1;2} \stackrel{def}{=} (\lambda(\lambda(\mathsf{var}\,1, \mathsf{var}\,0))(\uparrow_0 P_2))P_1$ and the program $P_{2;1} \stackrel{def}{=} (\lambda(\lambda(\mathsf{var}\,0, \mathsf{var}\,1))(\Uparrow_0 P_1))P_2$. Then it holds that $P_{1;2} \approx_{obs} P_{2;1}$.*

Proof. Using the CIU Theorem 1.58, we will prove $P_{1;2} \approx_{ciu} P_{2;1}$. So let E be an evaluation context, σ a store, η an event list, and v a list of closed values without locations $\geq |\sigma|$, such that $E[P^v_{1;2}]|\sigma|\eta$ and $E[P^v_{2;1}]|\sigma|\eta$ are fully closed. To complete the proof, we need to show $[\![E[P^v_{1;2}]|\sigma|\eta]\!]\,(\Omega) = [\![E[P^v_{2;1}]|\sigma|\eta]\!]\,(\Omega)$. Using the chaining rule (Theorem 1.48), it suffices to

show that $[\![P^v_{1;2}|\sigma|\eta]\!] = [\![P^v_{2;1}|\sigma|\eta]\!]$. We calculate as follows:

$$
\begin{aligned}
&[\![P^v_{1;2}|\sigma|\eta]\!] \\
&= [\![(\lambda(\lambda(\mathsf{var}\,1, \mathsf{var}\,0))(\Uparrow_0 P_2))^v P^v_1|\sigma|\eta]\!] \\
&= (\lambda(V_1|\sigma_1|\eta_1).\ [\![((\lambda(V_1, \mathsf{var}\,0))P^v_2)|\sigma_1|\eta_1]\!]) \cdot [\![P^v_1|\sigma|\eta]\!] && \text{(Lemma 1.50)} \\
&= (\lambda(V_1|\sigma_1|\eta_1).\ ((\lambda(V_2|\sigma_2|\eta_2).\ [\![(V_1, V_2)|\sigma_2|\eta_2]\!]) \cdot [\![P^v_2|\sigma_1|\eta_1]\!])) \cdot [\![P^v_1|\sigma|\eta]\!] \\
& && \text{(Lemma 1.50)} \\
&= (\lambda(V_1|\sigma_1|\eta_1).\ ((\lambda(V_2|\sigma_2|\eta_2).\ [\![(V_1, V_2)|\sigma_2|\eta_2]\!]) \cdot [\![P^v_2|\sigma|\eta]\!])) \cdot [\![P^v_1|\sigma|\eta]\!] \\
& && (P_1 \text{ state indep.}) \\
&= (\lambda(V_2|\sigma_2|\eta_2).\ ((\lambda(V_1|\sigma_1|\eta_1).\ [\![(V_1, V_2)|\sigma_2|\eta_2]\!]) \cdot [\![P^v_1|\sigma|\eta]\!])) \cdot [\![P^v_2|\sigma|\eta]\!] \\
& && \text{(swap indep. kernels)} \\
&\overset{(*)}{=} (\lambda(V_2|\sigma_2|\eta_2).\ ((\lambda(V_1|\sigma_1|\eta_1).\ [\![(V_1, V_2)|\sigma_1|\eta_1]\!]) \cdot [\![P^v_1|\sigma_2|\eta_2]\!])) \cdot [\![P^v_2|\sigma|\eta]\!] \\
& && (P_1 \text{ state indep.}) \\
&= (\lambda(V_2|\sigma_2|\eta_2).\ [\![((\lambda(\mathsf{var}\,0, V_2))P^v_1)|\sigma_2|\eta_2]\!]) \cdot [\![P^v_2|\sigma|\eta]\!] && \text{(Lemma 1.50)} \\
&= [\![(\lambda(\lambda(\mathsf{var}\,0, \mathsf{var}\,1))(\Uparrow_0 P_1))^v P^v_2|\sigma|\eta]\!] && \text{(Lemma 1.50)} \\
&= [\![P^v_{2;1}|\sigma|\eta]\!]
\end{aligned}
$$

For equation $(*)$ to hold, we need that $P^v_1|\sigma_2|\eta_2$ is fully closed. This is the case because the store σ_2 is sampled from $[\![P^v_2|\sigma|\eta]\!]$ and hence $|\sigma_2| \geq |\sigma|$. □

The previous result shows how to swap two lines in a simple program where the program following these two lines just consists of a pair of variables (x_1, x_2). We will now see how to extend this result to arbitrary programs P. Namely, we will prove the following observational equivalence:

$$
\boxed{\begin{array}{l}\mathsf{let}\ x_1 \leftarrow P_1; \\ \quad x_2 \leftarrow \Uparrow_0 P_2 \\ \mathsf{in}\ P\end{array}} \quad \approx_{obs} \quad \boxed{\begin{array}{l}\mathsf{let}\ x_2 \leftarrow P_2; \\ \quad x_1 \leftarrow \Uparrow_0 P_1 \\ \mathsf{in}\ \updownarrow P\end{array}} \tag{1.102}
$$

The swapping of the two lines results in swapped de Bruijn indices of the corresponding variables. The variable swap operator $\updownarrow$ swaps all occurrences of the free variables $\mathsf{var}\,0$ and $\mathsf{var}\,1$ in program P in order to account for the changed indices. The operator is defined as $\updownarrow P \overset{\text{def}}{=} (\uparrow_0 P)\{\mathsf{var}\,0/_2\}$, i.e., it lifts all free variables of P and then substitutes variable $\mathsf{var}\,2$ with $\mathsf{var}\,0$. The following theorem states the validity of Equation 1.102 in its nameless representation. I.90, p.116

Theorem 1.103 (Line Swapping)**.** *Let P_1, P_2, and P be programs where P_1 is state independent. Then it holds* I.524, p.163

$$(\lambda(\lambda P)(\Uparrow_0 P_2))P_1 \quad \approx_{obs} \quad (\lambda(\lambda \updownarrow P)(\Uparrow_0 P_1))P_2.$$

Proof. We want to use Lemma 1.101 to prove the theorem, therefore we need to transform the games so that they match the form of $P_{1;2}$ and $P_{2;1}$. For this we first transform program P into an *uncurried* form $\mathsf{uncurry}\ P$ where the variables $\mathsf{var}\,0$ and $\mathsf{var}\,1$ are bound using λ-binders and the program $\mathsf{uncurry}\ P$ expects a pair of variables (x_1, x_2) as argument. The program is defined as follows:

I.521, p.162
$$\mathsf{uncurry}\ P \stackrel{\text{def}}{=} \lambda\uparrow_1\uparrow_1\Big(\big(\lambda(\lambda\uparrow_2 P)\big)\ \big(\mathsf{fst}\,(\mathsf{var}\,0)\big)\ \big(\mathsf{snd}\,(\mathsf{var}\,0)\big)\Big).$$

When program $\mathsf{uncurry}\ P$ is applied to a pair of variables (x_1, x_2), then these variables are substituted for the variables $\mathsf{var}\,1$ and $\mathsf{var}\,0$ in program P during evaluation. In particular, using lemmas 1.86, 1.89, 1.90, and elementary properties of the operators $\uparrow_k$ and $\{\cdot/_k\}$, we can prove the following equivalences:

I.522, p.163
$$(\mathsf{uncurry}\ P)(\mathsf{var}\,1, \mathsf{var}\,0)\ \approx_{obs}\ P \tag{1.104}$$

I.523, p.163
$$(\mathsf{uncurry}\ P)(\mathsf{var}\,0, \mathsf{var}\,1)\ \approx_{obs}\ \updownarrow P \tag{1.105}$$

After applying these transformations, we are left to prove the following equivalence in order to complete the proof of Theorem 1.103:

$$\begin{array}{c}(\lambda(\lambda(\mathsf{uncurry}\ P)(\mathsf{var}\,1, \mathsf{var}\,0))(\uparrow_0 P_2))P_1\\ \approx_{obs}\\ (\lambda(\lambda\,(\mathsf{uncurry}\ P)(\mathsf{var}\,0, \mathsf{var}\,1))(\uparrow_0 P_1))P_2\end{array} \tag{1.106}$$

The difference between Equation 1.106 and the transformation that is given in Lemma 1.101 is the occurrence of the program $\mathsf{uncurry}\ P$. We can use the transformation for inlining let statements from Lemma 1.99 so that Lemma 1.101 becomes applicable. The required proof steps to complete the proof are illustrated in Figure 1.107. □

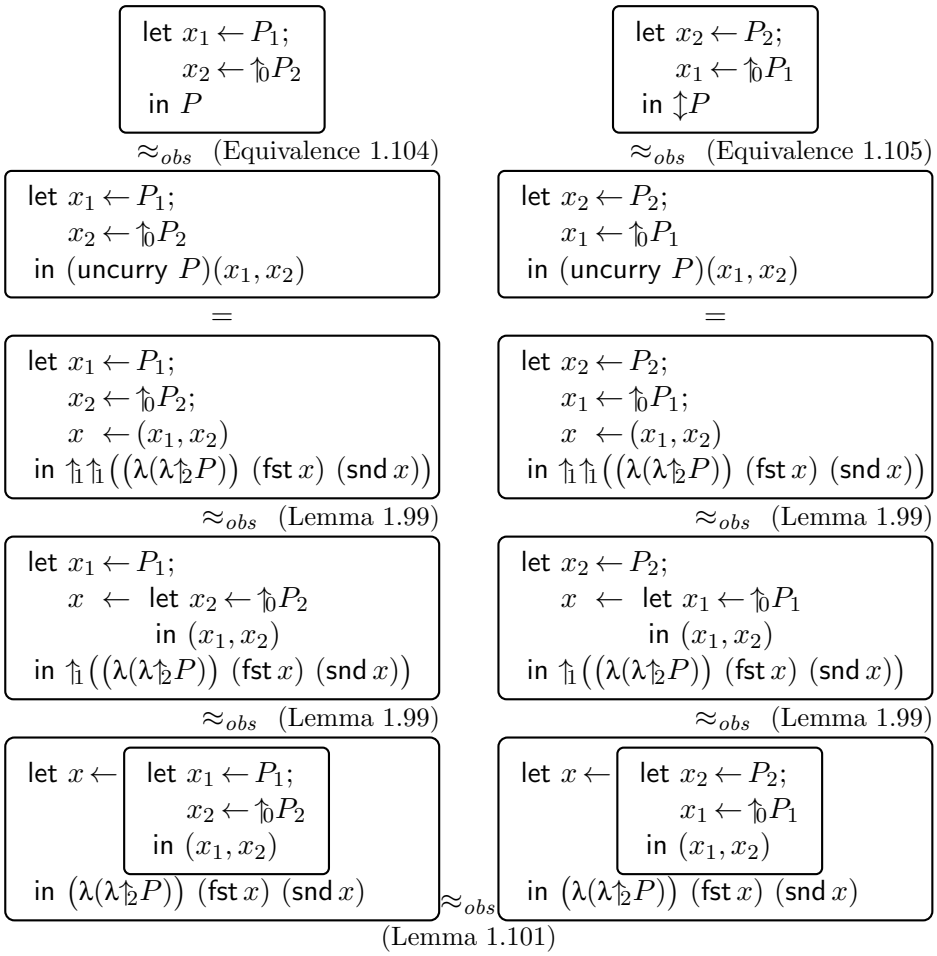


Figure 1.107: Proof steps of the line swapping Theorem 1.103 using inlining of let statements.

1.13 Sample Applications

In this section we will demonstrate the applicability of Verypto by verifying some example proofs.[5] The first example we are going to verify is that the self-composition of an injective one-way function yields another one-way function. This example makes extensive use of the expression propagation transformation we introduced in Lemma 1.95. In the second example we verify the IND-CPA security of the ElGamal encryption scheme. This proof employs several transformations including the line swapping transformation from Theorem 1.103, the inlining of let statements from Lemma 1.99, as well as specialized transformations to deal with the Diffie-Hellman assumption and properties of randomly selected group elements.

1.13.1 Composition of One-way Functions

We demonstrate the applicability of Verypto by verifying that, given a 1-1 one-way function f, the self-composition $g \stackrel{\text{def}}{=} f \circ f$ is also one-way. A function f is called *one-way*, if it is efficiently computable, but no polynomial-time program can invert the function with non-negligible probability. We define a
I.528, p.163 game $\mathsf{G}_{\mathsf{ow}}^{f,\mathsf{A}}$ which implements a corresponding challenge for an adversary A: Sampling a random value x, this game checks whether the adversary can produce a preimage for $f(x)$.

$$\mathsf{G}_{\mathsf{ow}}^{f,\mathsf{A}} \stackrel{\text{def}}{=} \boxed{\begin{array}{l}\mathsf{let}\ x \leftarrow \{0,1\}^n; \\ \quad y \leftarrow \widehat{f}\, x; \\ \quad x' \leftarrow \mathsf{A}(1^n, y) \\ \mathsf{in}\ \widehat{f}\, x = \widehat{f}\, x'\end{array}}$$

Here, $\widehat{f}$ denotes the program embedding of function f and $\{0,1\}^n$ denotes a program computing the uniform distribution over bitstrings of length n.

I.529, p.163 **Definition 1.108** (One-way Function)**.** *Let $f : \{0,1\}^* \to \{0,1\}^*$ be an efficiently computable function. f is called* one-way*, iff for any polynomial-time program A, we have that $\Pr[\mathsf{G}_{\mathsf{ow}}^{f,\mathsf{A}}]$ is negligible in n. An injective one-way function is also called a* 1-1 one-way function.

In the following let f be a 1-1 one-way function and let $\widehat{f}$ be the program embedding of f. Furthermore let $\widehat{g}$ denote the program embedding of $g \stackrel{\text{def}}{=} f \circ f$. We will show that the function g is also one-way. For this, we start from the initial game $\mathsf{G}_{\mathsf{ow}}^{g,\mathsf{A}}$ where an adversary A tries to invert the function g. Then we give a sequence of games connecting game $\mathsf{G}_{\mathsf{ow}}^{g,\mathsf{A}}$ to a final game $\mathsf{G}_{\mathsf{ow}}^{f,\mathsf{B}}$ where a suitable adversary B tries to invert the function f.

[5] Manual proofs of these examples have been prepared in the bachelor's theses [74] and [115] under my guidance. This dissertation presents their formal verification in Verypto using Isabelle/HOL.

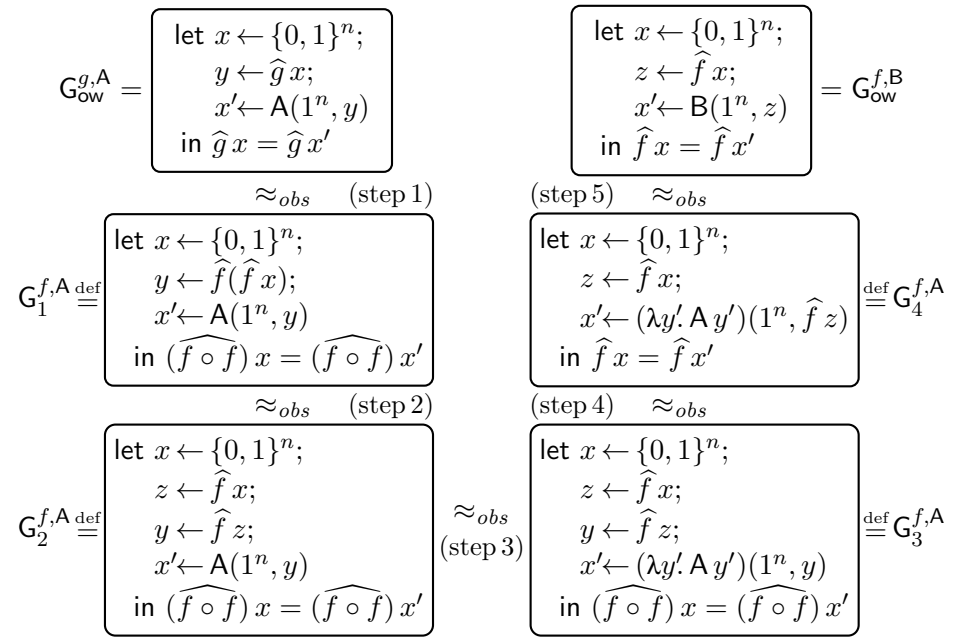


Figure 1.109: Game sequence connecting the initial game $\mathsf{G}_{\mathsf{ow}}^{g,\mathsf{A}}$ to the final game $\mathsf{G}_{\mathsf{ow}}^{f,\mathsf{B}}$.

We prove consecutive games in this sequence to be observationally equivalent and thereby reduce the one-way property of g for adversary A to the one-way property of f for adversary B. The sequence of the games is given in Figure 1.109.

We will now describe the single steps of this sequence. In step 1 of Figure 1.109 we replace the program $\widehat{g}\, x$ with two applications of $\widehat{f}$ and unfold the definition of function g. The resulting game $\mathsf{G}_1^{f,\mathsf{A}}$ is observationally equivalent to the initial game $\mathsf{G}_{\mathsf{ow}}^{g,\mathsf{A}}$. I.530, p.163

Lemma 1.110 (Step 1)**.** *For the initial game* $\mathsf{G}_{\mathsf{ow}}^{g,\mathsf{A}}$ *and the game* $\mathsf{G}_1^{f,\mathsf{A}}$ *as defined in Figure 1.109, it holds that* $\mathsf{G}_{\mathsf{ow}}^{g,\mathsf{A}} \approx_{obs} \mathsf{G}_1^{f,\mathsf{A}}$. I.537, p.164

Proof. Using the CIU Theorem 1.58, we can show $(\widehat{f \circ f})\, x \approx_{obs} \widehat{f}(\widehat{f}\, x)$: Either the variable x is instantiated with a pure value V_0, in which case both programs evaluate to the same distribution; or x is instantiated with a non-pure value V, in which case both programs are stuck and their denotations are 0. □ I.536, p.164

In step 2, we introduce the line $z \leftarrow \widehat{f}\, x$ and use variable z instead of the inner application of $\widehat{f}(\widehat{f}\, x)$. Using the expression propagation Lemma 1.95 we can show that the resulting game $\mathsf{G}_2^{f,\mathsf{A}}$ is observationally equivalent to game $\mathsf{G}_1^{f,\mathsf{A}}$. I.531, p.164

I.538, p.164 **Lemma 1.111** (Step 2)**.** *Let the games* $\mathsf{G}_1^{f,\mathsf{A}}$ *and* $\mathsf{G}_2^{f,\mathsf{A}}$ *be defined as in Figure 1.109. Then it holds that* $\mathsf{G}_1^{f,\mathsf{A}} \approx_{obs} \mathsf{G}_2^{f,\mathsf{A}}$.

Proof. The step is justified by the expression propagation Lemma 1.95: If we propagate program $\widehat{f}\,x$ in game $\mathsf{G}_2^{f,\mathsf{A}}$ and thereby substitute variable z with $\widehat{f}\,x$, we obtain game $\mathsf{G}_1^{f,\mathsf{A}}$. We still need to show that the assumptions required for Lemma 1.95 are fulfilled:

I.518, p.162
- Program $\widehat{f}\,x$ is propagatable: Either the variable x is instantiated with a non-pure value V, in which case the program is stuck and its denotation is 0; or x is instantiated with a pure value V_0, in which case the program evaluates to the Dirac measure $\delta(\widehat{f(\check{V}_0)}|\sigma|\eta)$ for all stores σ and event lists η. Here $\check{V}_0$ denotes the inverse program embedding of V_0.

- Assume $[\![(\widehat{f}\,x)^a|\sigma|\eta]\!] = 0$. We use the chaining rule (Theorem 1.48) to show that $[\![((\lambda y.\,P)(\widehat{f}(\widehat{f}\,x)))^a|\sigma|\eta]\!] = 0$ for arbitrary programs P (that is the format of the program resulting from the application of the expression propagation transformation, which matches $\mathsf{G}_1^{f,\mathsf{A}}$):

$$\begin{aligned}&[\![((\lambda y.\,P)(\widehat{f}(\widehat{f}\,x)))^a|\sigma|\eta]\!]\\ &= (\lambda(V'|\sigma'|\eta').\ [\![(\lambda y.\,P)^a(\widehat{f}\,V')|\sigma'|\eta']\!]) \cdot [\![(\widehat{f}\,x)^a|\sigma|\eta]\!] = 0\end{aligned}$$ □

I.532, p.164 Step 3 just prepares the game for the succeeding step. In game $\mathsf{G}_3^{f,\mathsf{A}}$, the adversary is wrapped with a λ-binder and applied to the corresponding variable. The resulting program $\lambda y'.\,\mathsf{A}\,y'$ is a value, which is a property we will use to prove the succeeding step.

I.539, p.164 **Lemma 1.112** (Step 3)**.** *Let the games* $\mathsf{G}_2^{f,\mathsf{A}}$ *and* $\mathsf{G}_3^{f,\mathsf{A}}$ *be defined as in Figure 1.109. Then it holds that* $\mathsf{G}_2^{f,\mathsf{A}} \approx_{obs} \mathsf{G}_3^{f,\mathsf{A}}$.

Proof. Since the pair $(1^n, y)$ is a value the lemma follows directly from Lemma 1.86, which yields $\mathsf{A}(1^n, y) \approx_{obs} (\lambda y'.\,\mathsf{A}\,y')(1^n, y)$ □

I.533, p.164 In step 4 we use the expression propagation Lemma 1.95 to propagate the program $\widehat{f}\,z$, i.e., in game $\mathsf{G}_4^{f,\mathsf{A}}$ the line $y \leftarrow \widehat{f}\,z$ has been removed and variable y has been substituted with program $\widehat{f}\,z$ instead. Furthermore, the equality test in the last line now uses the function f instead of its composition $f \circ f$; this step requires the 1-1 property of function f.

I.540, p.164 **Lemma 1.113** (Step 4)**.** *Let the games* $\mathsf{G}_3^{f,\mathsf{A}}$ *and* $\mathsf{G}_4^{f,\mathsf{A}}$ *be defined as in Figure 1.109 and assume that* f *is injective. Then it holds* $\mathsf{G}_3^{f,\mathsf{A}} \approx_{obs} \mathsf{G}_4^{f,\mathsf{A}}$.

Proof. We first show that the change in the equality test in the last line preserves the observational equivalence. For this we show that for an arbitrary function $h : \{0,1\}^* \to \{0,1\}^*$ the following equivalence holds:

$$\left(\widehat{h}\,x = \widehat{h}\,x'\right) \;\approx_{obs}\; \widehat{(\lambda x_1 x_2.\, h(x_1) = h(x_2))}\,x\,x' \qquad (1.114)$$ I.535, p.164

Equation 1.114 can be proven using the CIU Theorem 1.58: Either the variables x and x' are instantiated with pure values V_0 and V_0', in which case both programs evaluate to the Dirac measure $\delta(\widehat{h(\check{V}_0) = h(\check{V}_0')}|\sigma|\eta)$ for all stores σ and event lists η; or the variables x and x' are not both instantiated with pure values, in which case both programs are stuck and their denotations are 0. We use Equation 1.114 and the injectivity of function f to show the observational equivalence of the last lines of the games $\mathsf{G}_3^{f,\mathsf{A}}$ and $\mathsf{G}_4^{f,\mathsf{A}}$:

$$\begin{aligned}
&\left((\widehat{f \circ f})\,x = (\widehat{f \circ f})\,x'\right) \\
\approx_{obs}\;& \widehat{(\lambda x_1 x_2.\, (f \circ f)(x_1) = (f \circ f)(x_2))}\,x\,x' && \text{(Equation 1.114)} \\
=\;& \widehat{(\lambda x_1 x_2.\, f(x_1) = f(x_2))}\,x\,x' && (f \text{ is injective}) \\
\approx_{obs}\;& \left(\widehat{f}\,x = \widehat{f}\,x'\right) && \text{(Equation 1.114)}
\end{aligned}$$

As mentioned above, we can finish the proof by propagating the line $y \leftarrow \widehat{f}\,z$ using the expression propagation Lemma 1.95. We only need to show that the assumptions required for Lemma 1.95 are fulfilled:

- Program $\widehat{f}\,z$ is propagatable: This was already shown in the proof of Lemma 1.111.

- Under the assumption that $[\![(\widehat{f}\,z)^a|\sigma|\eta]\!] = 0$, we need to show that $[\![((\lambda x'.\, P)((\lambda y'.\, \mathsf{A}\,y')(1^n, \widehat{f}\,z)))^a|\sigma|\eta]\!] = 0$ for arbitrary programs P (that is the format of the program resulting from the application of the expression propagation transformation). We define the evaluation context[6] $E \stackrel{\text{def}}{=} (\lambda x'.\, P)((\lambda y'.\, \mathsf{A}\,y')(1^n, \square))$ and conclude the proof using Theorem 1.48:

$$[\![(E[\widehat{f}\,z])^a|\sigma|\eta]\!] = (\lambda(V'|\sigma'|\eta').\; [\![E^a[V']|\sigma'|\eta']\!]) \cdot [\![(\widehat{f}\,z)^a|\sigma|\eta]\!] = 0 \qquad \square$$

For step 5 we define the adversary $\mathsf{B} \stackrel{\text{def}}{=} \lambda z'.\, (\lambda y'.\, \mathsf{A}\,y')(\mathsf{fst}\,z', \widehat{f}\,(\mathsf{snd}\,z'))$. I.534, p.164
This final step of our sequence of games introduces adversary B such that the resulting game matches the challenge game from the one-way function Definition 1.108. In fact it is easy to show that the games $\mathsf{G}_4^{f,\mathsf{A}}$ and $\mathsf{G}_{\mathsf{ow}}^{f,\mathsf{B}}$ are observationally equivalent.

Lemma 1.115 (Step 5)**.** *For the final game $\mathsf{G}_{\mathsf{ow}}^{f,\mathsf{B}}$ and the game $\mathsf{G}_4^{f,\mathsf{A}}$ as defined in Figure 1.109, it holds that $\mathsf{G}_4^{f,\mathsf{A}} \approx_{obs} \mathsf{G}_{\mathsf{ow}}^{f,\mathsf{B}}$.* I.541, p.164

[6] This is why we wrapped the adversary with a λ-binder in step 3. The context $(\lambda x'.\, P)(\mathsf{A}(1^n, \square))$ is not necessarily an evaluation context, as A might not be a value.

Proof. Using the lemmas 1.86, 1.89, and 1.90, the lemma follows from

$$\mathsf{B}(1^n, z) \;=\; (\lambda z'.\ (\lambda y'.\ \mathsf{A}\ y')\ (\mathsf{fst}\ z', \widehat{f}\,(\mathsf{snd}\ z'))))\ (1^n, z) \;\approx_{obs}\; (\lambda y'.\ \mathsf{A}\ y')(1^n, \widehat{f}\ z).$$

□

We have proved all steps of the sequence of games in Figure 1.109. We can piece these steps together, which yields the following theorem:

I.542, p.164 **Theorem 1.116.** *Assuming an injective function $f : \{0,1\}^* \to \{0,1\}^*$ and an adversary A, let function $g \stackrel{def}{=} f \circ f$ and let adversary B be defined as $\lambda z'.\,(\lambda y'.\,\mathsf{A}\ y')(\mathsf{fst}\ z', \widehat{f}\,(\mathsf{snd}\ z'))$. Then it holds*

$$\Pr[\mathsf{G}_{\mathsf{ow}}^{g,\mathsf{A}}] = \Pr[\mathsf{G}_{\mathsf{ow}}^{f,\mathsf{B}}].$$

Proof. From the lemmas 1.110, 1.111, 1.112, 1.113, and 1.115 we obtain $\mathsf{G}_{\mathsf{ow}}^{g,\mathsf{A}} \approx_{obs} \mathsf{G}_{\mathsf{ow}}^{f,\mathsf{B}}$, because $\approx_{obs}$ is transitive (Lemma 1.35). The theorem then follows from Lemma 1.37. □

1.13.2 IND-CPA Security of ElGamal

In this section we will demonstrate another applicability of Verypto. We will verify that the ElGamal encryption scheme [79] has indistinguishable ciphertexts under chosen-plaintext attacks (IND-CPA). We model public-key encryption schemes $\mathcal{ES}$ as families of three programs $\{(\mathsf{Gen}_n, \mathsf{Enc}_n, \mathsf{Dec}_n)\}_{n\in\mathbb{N}}$ that are indexed by the security parameter n. Here, the key generation Gen_n computes a distribution over pairs consisting of public keys e and private keys d. The encryption algorithm Enc_n takes a public key e and a plaintext m as argument and produces a distribution over ciphertexts c. The decryption algorithm Dec_n takes a private key d and a ciphertext c as argument and produces the corresponding plaintext m. We denote the set of possible plaintexts for security parameter n by $\mathsf{dom}_n(\mathcal{ES})$.

IND-CPA Security

IND-CPA security is defined using a challenge where an adversary has to distinguish the encryptions of two plaintext messages of his choosing. More precisely, the (non-uniform) adversary is given a randomly selected public key and outputs a tuple consisting of two plaintext messages and an auxiliary value. For the challenge one of the plaintexts is encrypted and the resulting ciphertext is presented to the adversary together with the auxiliary value. The goal of the adversary is to distinguish whether the first or the second plaintext has been encrypted.

Since during the challenge the adversary is called twice, we model adversaries $\mathcal{A}$ as families of pairs of programs $\{(\mathsf{A}_n, \mathsf{B}_n)\}_{n\in\mathbb{N}}$, i.e., program A_n is given a public key and outputs a tuple consisting of two plaintext messages and an auxiliary value, whereas program B_n expects a ciphertext

and an auxiliary value and outputs a Boolean. Here the auxiliary value is used to enable communication from A_n to B_n, which models that the adversary may maintain state between its invocations. We define separate games for the two settings where the first plaintext message or where the second plaintext message is encrypted. In particular, given an encryption scheme $\mathcal{ES} = \{(\mathsf{Gen}_n, \mathsf{Enc}_n, \mathsf{Dec}_n)\}_{n\in\mathbb{N}}$ and an adversary $\mathcal{A} = \{(\mathsf{A}_n, \mathsf{B}_n)\}_{n\in\mathbb{N}}$, we define the following IND-CPA games $\mathsf{CPA1}_n^{\mathcal{ES},\mathcal{A}}$ and $\mathsf{CPA2}_n^{\mathcal{ES},\mathcal{A}}$:

$\mathsf{CPA1}_n^{\mathcal{ES},\mathcal{A}} \stackrel{\text{def}}{=}$

```
let (e, d)          ← Gen_n;
    (m1, m2, a)     ← A_n e;
    c               ← Enc_n e m1
in B_n(c, a)
```

$\mathsf{CPA2}_n^{\mathcal{ES},\mathcal{A}} \stackrel{\text{def}}{=}$

```
let (e, d)          ← Gen_n;
    (m1, m2, a)     ← A_n e;
    c               ← Enc_n e m2
in B_n(c, a)
```

I.545, p.165
I.546, p.165

In order to prevent that $\mathcal{A}$ can distinguish these two settings trivially, we need to restrict the allowed behavior of $\mathcal{A}$. In particular we require that for almost all (m_1, m_2, a) output by A_n it holds that $m_1, m_2 \in \mathsf{dom}_n(\mathcal{ES})$. Furthermore we require the programs A_n and B_n to have a polynomial runtime. Such adversaries $\mathcal{A}$ are called *IND-CPA adversaries* for encryption scheme $\mathcal{ES}$.

Definition 1.117 (IND-CPA Adversary)**.** *Given an encryption scheme $\mathcal{ES}$ and an adversary $\mathcal{A} = \{(\mathsf{A}_n, \mathsf{B}_n)\}_{n\in\mathbb{N}}$, we call $\mathcal{A}$ an* IND-CPA adversary *for $\mathcal{ES}$, iff $\{\mathsf{A}_n\}_{n\in\mathbb{N}}$ and $\{\mathsf{B}_n\}_{n\in\mathbb{N}}$ are polynomial-time and for all values V, stores σ, event lists η, $n \in \mathbb{N}$, and for almost all $t|\sigma'|\eta' \in [\![\mathsf{A}_n V|\sigma|\eta]\!]$, it holds that $t = (\widehat{m_1, m_2, a})$ for some $m_1, m_2 \in \mathsf{dom}_n(\mathcal{ES})$.*

I.547, p.165

Definition 1.118 (IND-CPA Security)**.** *An encryption scheme $\mathcal{ES}$ is* IND-CPA secure*, iff for all IND-CPA adversaries $\mathcal{A}$ for $\mathcal{ES}$ it holds that*

$$\left|\Pr\left[\mathsf{CPA1}_n^{\mathcal{ES},\mathcal{A}}\right] - \Pr\left[\mathsf{CPA2}_n^{\mathcal{ES},\mathcal{A}}\right]\right| \text{ is negligible in } n.$$

I.548, p.165

Cyclic Groups

The ElGamal encryption scheme operates on cyclic groups. Thus, for the remainder of this section we assume a family of cyclic groups $\mathcal{G} \stackrel{\text{def}}{=} \{\mathsf{G}_n\}_{n\in\mathbb{N}}$ where group G_n is of order q_n and let g_n denote a generator for G_n. Furthermore let mult_n, pow_n, and inv_n denote the multiplication, the exponentiation, and the inverse operations on G_n. Since g_n is a generator, it holds that $\mathsf{G}_n = \{\mathsf{pow}_n(\mathsf{g}_n, z) \mid z \in \{1 \dots \mathsf{q}_n\}\}$. Furthermore, since the multiplication with a group element is injective, it also holds for all $m \in \mathsf{G}_n$ that

$$\mathsf{G}_n = \{\mathsf{mult}_n(m, \mathsf{pow}_n(\mathsf{g}_n, z)) \mid z \in \{1 \dots \mathsf{q}_n\}\}. \tag{1.119}$$

I.558, p.166

Therefore, the multiplication of an element $m \in \mathsf{G}_n$ with a group element that has been selected uniformly at random behaves like a one-time

pad, i.e., the result is distributed uniformly in G_n. We will use this fact in our security proof of the ElGamal encryption scheme. In particular, we define the sampling algorithm

$$\mathsf{R}_n^m \stackrel{\text{def}}{=} \boxed{\begin{array}{l}\mathsf{let}\ z \leftarrow \{1 \ldots \mathsf{q}_n\};\\ \quad gz \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, z)\\ \mathsf{in}\ \widehat{\mathsf{mult}}_n(\widehat{m}, gz);\end{array}}$$

where, abusing notation, $\{1 \ldots \mathsf{q}_n\}$ denotes a program computing the uniform distribution over the set $\{1 \ldots \mathsf{q}_n\}$. We can prove that the denotation of R_n^m is independent of the choice of m.

I.559, p.166 **Lemma 1.120.** *The Programs $\mathsf{R}_n^{m_1}$ and $\mathsf{R}_n^{m_2}$ are denotationally equivalent for all $m_1, m_2 \in \mathsf{G}_n$.*

Proof. Let $U(S)$ denote the uniform distribution over set S. Assume a store σ and an event list η. We prove the lemma by calculating as follows:

$$\begin{aligned}
&[\![\mathsf{R}_n^{m_1}|\sigma|\eta]\!]\\
&= \left(\lambda(z|\sigma'|\eta').\ \left[\!\!\left[(\lambda gz.\, \widehat{\mathsf{mult}}_n(\widehat{m_1}, gz))\,(\widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, z))|\sigma'|\eta'\right]\!\!\right]\right) \cdot [\![\{1 \ldots \mathsf{q}_n\}|\sigma|\eta]\!]\\
&= \left(\lambda(z|\sigma'|\eta').\ \left[\!\!\left[(\lambda gz.\, \widehat{\mathsf{mult}}_n(\widehat{m_1}, gz))\,(\widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, z))|\sigma'|\eta'\right]\!\!\right]\right) \cdot\\
&\qquad\qquad (\lambda z.\, \widehat{z}|\sigma|\eta)(U\{1 \ldots \mathsf{q}_n\})\\
&= \left(\lambda z.\ \left[\!\!\left[(\lambda gz.\, \widehat{\mathsf{mult}}_n(\widehat{m_1}, gz))\,(\widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, \widehat{z}))|\sigma|\eta\right]\!\!\right]\right) \cdot U\{1 \ldots \mathsf{q}_n\}\\
&= \left(\lambda z.\, \widehat{\mathsf{mult}_n(m_1, \mathsf{pow}_n(\mathsf{g}_n, z))}|\sigma|\eta\right)(U\{1 \ldots \mathsf{q}_n\})\\
&\stackrel{(1.119)}{=} U\{(\widehat{m}|\sigma|\eta) \mid m \in \mathsf{G}_n\}\\
&= \cdots = [\![\mathsf{R}_n^{m_2}|\sigma|\eta]\!]
\end{aligned}$$

□

Decisional Diffie-Hellman Assumption

The *decisional Diffie-Hellman assumption* states the computational indistinguishability of two particular settings; namely, no polynomial-time distinguisher $\mathcal{D} = \{\mathsf{D}_n\}_{n \in \mathbb{N}}$ can distinguish the following two settings with a non-negligible probability: For security parameter n, in the first setting D_n is called with a tuple of the form $(\mathsf{g}_n^x, \mathsf{g}_n^y, \mathsf{g}_n^{xy})$, where x and y are selected uniformly at random from the set $\{1 \ldots \mathsf{q}_n\}$. In the second setting D_n is provided a tuple of random group elements, i.e., a tuple of the form $(\mathsf{g}_n^x, \mathsf{g}_n^y, \mathsf{g}_n^z)$, where x, y, and z are selected uniformly at random from the set $\{1 \ldots \mathsf{q}_n\}$. Corresponding to these settings, we define the games DDHxy_n and DDHz_n.

$\mathsf{DDHxy}_n \stackrel{\text{def}}{=}$

$$\begin{array}{l}\textsf{let } x \leftarrow \{1 \ldots \mathsf{q}_n\};\\ \quad gx \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, x);\\ \quad y \leftarrow \{1 \ldots \mathsf{q}_n\};\\ \quad gy \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, y);\\ \quad gxy \leftarrow \widehat{\mathsf{pow}}_n(gx, y)\\ \textsf{in } (gx, gy, gxy)\end{array}$$

$\mathsf{DDHz}_n \stackrel{\text{def}}{=}$

$$\begin{array}{l}\textsf{let } x \leftarrow \{1 \ldots \mathsf{q}_n\};\\ \quad gx \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, x);\\ \quad y \leftarrow \{1 \ldots \mathsf{q}_n\};\\ \quad gy \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, y);\\ \quad z \leftarrow \{1 \ldots \mathsf{q}_n\};\\ \quad gz \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, z)\\ \textsf{in } (gx, gy, gz)\end{array}$$

I.553, p.166
I.554, p.166

Definition 1.121 (DDH Assumption)**.** *We say that the* decisional Diffie-Hellman assumption *holds for $\mathcal{G}$, iff* $\{\mathsf{DDHxy}_n\}_{n\in\mathbb{N}} \approx_{ind} \{\mathsf{DDHz}_n\}_{n\in\mathbb{N}}$.

I.555, p.166

The ElGamal Encryption Scheme

The ElGamal encryption scheme $\mathcal{EG} = \{(\mathsf{Gen}_n^{\mathcal{EG}}, \mathsf{Enc}_n^{\mathcal{EG}}, \mathsf{Dec}_n^{\mathcal{EG}})\}_{n\in\mathbb{N}}$ consists of the key generation $\mathsf{Gen}_n^{\mathcal{EG}}$, the encryption algorithm $\mathsf{Enc}_n^{\mathcal{EG}}$, and the decryption algorithm $\mathsf{Dec}_n^{\mathcal{EG}}$ where $\mathsf{dom}_n(\mathcal{EG}) = \mathsf{G}_n$. The algorithms are defined as follows:

$\mathsf{Gen}_n^{\mathcal{EG}} \stackrel{\text{def}}{=}$

$$\begin{array}{l}\textsf{let } x \leftarrow \{1 \ldots \mathsf{q}_n\};\\ \quad gx \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, x)\\ \textsf{in } (gx, x)\end{array}$$

$\mathsf{Enc}_n^{\mathcal{EG}} \stackrel{\text{def}}{=}$

$$\begin{array}{l}\lambda gx.\, \lambda m.\\ \textsf{let } y \leftarrow \{1 \ldots \mathsf{q}_n\};\\ \quad gy \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, y);\\ \quad gxy \leftarrow \widehat{\mathsf{pow}}_n(gx, y);\\ \quad mgxy \leftarrow \widehat{\mathsf{mult}}_n(m, gxy)\\ \textsf{in } (gy, mgxy)\end{array}$$

$\mathsf{Dec}_n^{\mathcal{EG}} \stackrel{\text{def}}{=}$

$$\begin{array}{l}\lambda x.\, \lambda(gy, mgxy).\\ \textsf{let } gxy \leftarrow \widehat{\mathsf{pow}}_n(gy, x);\\ \quad igxy \leftarrow \widehat{\mathsf{inv}}_n\ gxy\\ \textsf{in } \widehat{\mathsf{mult}}_n(mgxy, igxy)\end{array}$$

I.549, p.165
I.550, p.165
I.551, p.165

The Sequence of Games

In the following we present a sequence of games connecting $\mathsf{CPA1}_n^{\mathcal{EG},\mathcal{A}}$ and $\mathsf{CPA2}_n^{\mathcal{EG},\mathcal{A}}$ for an IND-CPA adversary $\mathcal{A}$ which establishes the IND-CPA security of ElGamal; see Figure 1.122 for the full sequence. The proof follows the outline found in [114]. Starting from $\mathsf{CPA1}_n^{\mathcal{EG},\mathcal{A}}$, we first inline the definitions of $\mathsf{Gen}_n^{\mathcal{EG}}$ and $\mathsf{Enc}_n^{\mathcal{EG}}$ in $\mathsf{CPA1}_n^{\mathcal{EG},\mathcal{A}}$ and arrive at game $\mathsf{G1a}_n^{\mathcal{A}}$:

I.560, p.167

$\mathsf{G1a}_n^{\mathcal{A}} \stackrel{\text{def}}{=}$

$$\begin{array}{l}\textsf{let } x \leftarrow \{1 \ldots \mathsf{q}_n\};\\ \quad gx \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, x);\\ \quad (m_1, m_2, a) \leftarrow \mathsf{A}_n\ gx;\\ \quad y \leftarrow \{1 \ldots \mathsf{q}_n\};\\ \quad gy \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, y);\\ \quad gxy \leftarrow \widehat{\mathsf{pow}}_n(gx, y);\\ \quad mgxy \leftarrow \widehat{\mathsf{mult}}_n(m_1, gxy);\\ \quad c \leftarrow (gy, mgxy)\\ \textsf{in } \mathsf{B}_n(c, a)\end{array}$$

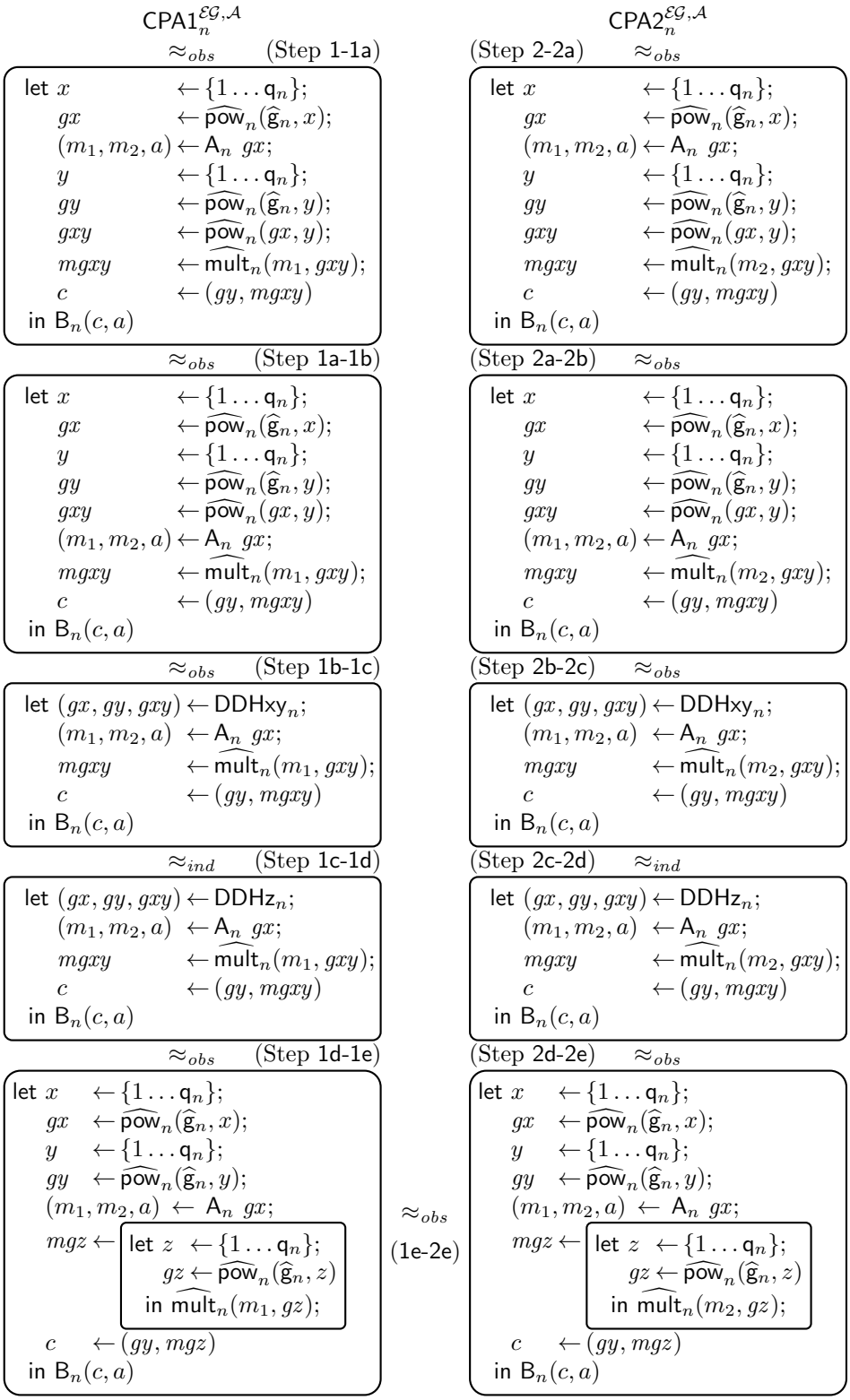


Figure 1.122: The sequence of games to prove ElGamal IND-CPA secure

Lemma 1.123 (Step 1-1a)**.** *For the games* $\mathsf{CPA1}_n^{\mathcal{EG},\mathcal{A}}$ *and* $\mathsf{G1a}_n^{\mathcal{A}}$ *as defined above the equivalence* $\mathsf{CPA1}_n^{\mathcal{EG},\mathcal{A}} \approx_{obs} \mathsf{G1a}_n^{\mathcal{A}}$ *holds.* I.562, p.167

Proof. The lemma mainly follows from the transformation to inline let statements (Lemma 1.99) and the Beta transformation (Lemma 1.86). □

For the next step we use the line swapping Theorem 1.103 in order to move the call to A_n as far down as possible, i.e., the call is moved before the multiplication operation, where the result of the call to A_n is used first. We call the resulting game $\mathsf{G1b}_n^{\mathcal{A}}$. I.564, p.167

$$\mathsf{G1b}_n^{\mathcal{A}} \stackrel{\text{def}}{=} \boxed{\begin{array}{ll} \mathsf{let}\ x & \leftarrow \{1 \dots \mathsf{q}_n\}; \\ \quad gx & \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, x); \\ \quad y & \leftarrow \{1 \dots \mathsf{q}_n\}; \\ \quad gy & \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, y); \\ \quad gxy & \leftarrow \widehat{\mathsf{pow}}_n(gx, y); \\ \quad (m_1, m_2, a) & \leftarrow \mathsf{A}_n\ gx; \\ \quad mgxy & \leftarrow \widehat{\mathsf{mult}}_n(m_1, gxy); \\ \quad c & \leftarrow (gy, mgxy) \\ \mathsf{in}\ \mathsf{B}_n(c, a) & \end{array}}$$

Lemma 1.124 (Step 1a-1b)**.** *For the games* $\mathsf{G1a}_n^{\mathcal{A}}$ *and* $\mathsf{G1b}_n^{\mathcal{A}}$ *as defined above the equivalence* $\mathsf{G1a}_n^{\mathcal{A}} \approx_{obs} \mathsf{G1b}_n^{\mathcal{A}}$ *holds.* I.566, p.167

Proof. The lemma follows from the line swapping Theorem 1.103. For this, note that all lines that are swapped with the call to A_n are state independent. □

Game $\mathsf{G1b}_n^{\mathcal{A}}$ almost fits the setup of the decisional Diffie-Hellman assumption. In fact we can easily wrap the lines starting from the call to A_n in a distinguisher $\mathcal{D}^{\mathcal{A}} \stackrel{\text{def}}{=} \{\mathsf{D}_n^{\mathcal{A}}\}_{n\in\mathbb{N}}$ in order to arrive at game $\mathsf{G1c}_n^{\mathcal{A}} \stackrel{\text{def}}{=} \mathsf{D}_n^{\mathcal{A}}(\mathsf{DDHxy}_n)$, where $\mathsf{D}_n^{\mathcal{A}}$ is defined as I.570, p.168 I.568, p.168

$$\mathsf{D}_n^{\mathcal{A}} \stackrel{\text{def}}{=} \boxed{\begin{array}{ll} \lambda(gx, gy, gxy). & \\ \mathsf{let}\ (m_1, m_2, a) & \leftarrow \mathsf{A}_n\ gx; \\ \quad mgxy & \leftarrow \widehat{\mathsf{mult}}_n(m_1, gxy); \\ \quad c & \leftarrow (gy, mgxy) \\ \mathsf{in}\ \mathsf{B}_n(c, a) & \end{array}}.$$

Lemma 1.125 (Step 1b-1c)**.** *For the games* $\mathsf{G1b}_n^{\mathcal{A}}$ *and* $\mathsf{G1c}_n^{\mathcal{A}}$ *as defined above the equivalence* $\mathsf{G1b}_n^{\mathcal{A}} \approx_{obs} \mathsf{G1c}_n^{\mathcal{A}}$ *holds.* I.572, p.168

Proof. The lemma follows from the transformation to inline let statements (Lemma 1.99) and the Beta transformation (Lemma 1.86). □

Assuming that the decisional Diffie-Hellman assumption holds for $\mathcal{G}$ and that the distinguisher $\mathcal{D}^{\mathcal{A}}$ is polynomial-time, the step to the game $\mathsf{G1d}_n^{\mathcal{A}} \stackrel{\text{def}}{=}$
I.574, p.168 $\mathsf{D}_n^{\mathcal{A}}(\mathsf{DDHz}_n)$ follows directly from Definition 1.121.

I.576, p.168 **Lemma 1.126** (Step 1c-1d)**.** *If the decisional Diffie-Hellman assumption holds for $\mathcal{G}$ and the distinguisher $\mathcal{D}^{\mathcal{A}}$ is polynomial-time, then*

$$\left|\Pr\left[\mathsf{G1c}_n^{\mathcal{A}}\right] - \Pr\left[\mathsf{G1d}_n^{\mathcal{A}}\right]\right| \text{ is negligible in } n.$$

For the next step we transform the game $\mathsf{G1d}_n^{\mathcal{A}}$ as follows: First, we unwrap the definition of the distinguisher and move the call to A_n before the sampling of z. Then we employ the inlining of let statements in order to wrap the three lines starting at the sampling of z in a new program. We
I.578, p.168 obtain game $\mathsf{G1e}_n^{\mathcal{A}}$ which is given below. Analogously, we also define the
I.579, p.169 game $\mathsf{G2e}_n^{\mathcal{A}}$ where the usage of variable m_1 has been replaced with variable m_2.

$\mathsf{G1e}_n^{\mathcal{A}} \stackrel{\text{def}}{=}$

```
let x   ←{1...q_n};
    gx  ←pow^_n(ĝ_n, x);
    y   ←{1...q_n};
    gy  ←pow^_n(ĝ_n, y);
    (m_1, m_2, a) ← A_n gx;
    mgz ← let z ←{1...q_n};
              gz ←pow^_n(ĝ_n, z)
          in mult^_n(m_1, gz);
    c   ←(gy, mgz)
 in B_n(c, a)
```

$\mathsf{G2e}_n^{\mathcal{A}} \stackrel{\text{def}}{=}$

```
let x   ←{1...q_n};
    gx  ←pow^_n(ĝ_n, x);
    y   ←{1...q_n};
    gy  ←pow^_n(ĝ_n, y);
    (m_1, m_2, a) ← A_n gx;
    mgz ← let z ←{1...q_n};
              gz ←pow^_n(ĝ_n, z)
          in mult^_n(m_2, gz);
    c   ←(gy, mgz)
 in B_n(c, a)
```

I.580, p.169 **Lemma 1.127** (Step 1d-1e)**.** *For the games $\mathsf{G1d}_n^{\mathcal{A}}$ and $\mathsf{G1e}_n^{\mathcal{A}}$ as defined above the equivalence $\mathsf{G1d}_n^{\mathcal{A}} \approx_{obs} \mathsf{G1e}_n^{\mathcal{A}}$ holds.*

Proof. The lemma follows from the Beta transformation (Lemma 1.86), the line swapping Theorem 1.103, and the transformation to inline let statements (Lemma 1.99). □

Note that the three lines that we wrapped in a new program fit the setup for Lemma 1.120. We will use this lemma in order to verify the step from game $\mathsf{G1e}_n^{\mathcal{A}}$ to $\mathsf{G2e}_n^{\mathcal{A}}$. For this, we require the fact that $\mathcal{A}$ is an IND-CPA adversary for $\mathcal{EG}$. This is because Lemma 1.120 is only applicable if the variables m_1 and m_2 are instantiated with group elements from G_n.

I.582, p.169 **Lemma 1.128** (Step 1e-2e)**.** *Let $\mathcal{A}$ be an IND-CPA adversary for $\mathcal{EG}$ and let the games $\mathsf{G1e}_n^{\mathcal{A}}$ and $\mathsf{G2e}_n^{\mathcal{A}}$ be defined as above. Then the equivalence $\mathsf{G1e}_n^{\mathcal{A}} \approx_{obs} \mathsf{G2e}_n^{\mathcal{A}}$ holds.*

Proof. Using the composability of observational equivalence (Lemma 1.36) and the CIU Theorem 1.58, it suffices to prove the following equivalence:

$$
\begin{array}{l}
\mathsf{let}\ (m_1, m_2, a) \leftarrow \mathsf{A}_n\ gx; \\
\quad mgz \leftarrow \boxed{\begin{array}{l}\mathsf{let}\ z \leftarrow \{1 \ldots \mathsf{q}_n\}; \\ \quad gz \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, z) \\ \mathsf{in}\ \widehat{\mathsf{mult}}_n(m_1, gz);\end{array}} \\
\quad c \leftarrow (gy, mgz) \\
\mathsf{in}\ \mathsf{B}_n(c, a)
\end{array}
\approx_{ciu}
\begin{array}{l}
\mathsf{let}\ (m_1, m_2, a) \leftarrow \mathsf{A}_n\ gx; \\
\quad mgz \leftarrow \boxed{\begin{array}{l}\mathsf{let}\ z \leftarrow \{1 \ldots \mathsf{q}_n\}; \\ \quad gz \leftarrow \widehat{\mathsf{pow}}_n(\widehat{\mathsf{g}}_n, z) \\ \mathsf{in}\ \widehat{\mathsf{mult}}_n(m_2, gz);\end{array}} \\
\quad c \leftarrow (gy, mgz) \\
\mathsf{in}\ \mathsf{B}_n(c, a)
\end{array}
$$

The proof works as follows: Using the chaining rule for denotations (Theorem 1.48) we separate the denotation of the call to A_n from the denotations of the games. Since $\mathcal{A}$ is an IND-CPA adversary, for almost all $t|\sigma'|\eta'$ in this denotation there are group elements $m_1, m_2 \in \mathsf{G}_n$ such that $t = (\widehat{m_1, m_2, a})$. Therefore, when reducing both games further, the inner let expressions in the games are instantiated to the sampling algorithms $\mathsf{R}_n^{m_1}$ and $\mathsf{R}_n^{m_2}$. These algorithms are denotationally equivalent (Lemma 1.120), which concludes the proof. □

So far, we have presented a sequence of games connecting the initial game $\mathsf{CPA1}_n^{\mathcal{EG},\mathcal{A}}$ to game $\mathsf{G2e}_n^{\mathcal{A}}$, which completes the first half of our sequence of steps. Analogously to the sequence of steps 1-1a-1b-1c-1d-1e, we can define corresponding games $\mathsf{G2a}_n^{\mathcal{A}}$, $\mathsf{G2b}_n^{\mathcal{A}}$, $\mathsf{G2c}_n^{\mathcal{A}}$, $\mathsf{G2d}_n^{\mathcal{A}}$ that use variable m_2 instead of variable m_1. The lemmas corresponding to the sequence of steps 2-2a-2b-2c-2d-2e then connect the game $\mathsf{CPA2}_n^{\mathcal{EG},\mathcal{A}}$ to game $\mathsf{G2e}_n^{\mathcal{A}}$. Since this second half of the sequence is performed analogously we omit the details here and proceed with the verification of the IND-CPA security of the ElGamal encryption scheme.

Theorem 1.129 (IND-CPA Security of ElGamal)**.** *Assume that the decisional Diffie-Hellman assumption holds for $\mathcal{G}$. Then the ElGamal encryption scheme $\mathcal{EG}$ is IND-CPA secure.* I.583, p.169

Proof. Let $\mathcal{A}$ be an IND-CPA adversary for $\mathcal{EG}$. Combining Lemma 1.37 and the lemmas of this section, we obtain the following equalities:

$$
\begin{aligned}
\Pr\left[\mathsf{CPA1}_n^{\mathcal{EG},\mathcal{A}}\right] &= \Pr\left[\mathsf{G1c}_n^{\mathcal{A}}\right] \\
\Pr\left[\mathsf{CPA2}_n^{\mathcal{EG},\mathcal{A}}\right] &= \Pr\left[\mathsf{G2c}_n^{\mathcal{A}}\right] \\
\Pr\left[\mathsf{G1d}_n^{\mathcal{A}}\right] &= \Pr\left[\mathsf{G2d}_n^{\mathcal{A}}\right]
\end{aligned}
$$

We calculate as follows:

$$
\begin{aligned}
0 &\leq \left|\Pr\left[\mathsf{CPA1}_n^{\mathcal{EG},\mathcal{A}}\right] - \Pr\left[\mathsf{CPA2}_n^{\mathcal{EG},\mathcal{A}}\right]\right| \\
&= \left|\Pr\left[\mathsf{G1c}_n^{\mathcal{A}}\right] - \Pr\left[\mathsf{G1d}_n^{\mathcal{A}}\right] + \Pr\left[\mathsf{G2d}_n^{\mathcal{A}}\right] - \Pr\left[\mathsf{G2c}_n^{\mathcal{A}}\right]\right| \\
&\leq \left|\Pr\left[\mathsf{G1c}_n^{\mathcal{A}}\right] - \Pr\left[\mathsf{G1d}_n^{\mathcal{A}}\right]\right| + \left|\Pr\left[\mathsf{G2c}_n^{\mathcal{A}}\right] - \Pr\left[\mathsf{G2d}_n^{\mathcal{A}}\right]\right|
\end{aligned}
$$

Since $\mathcal{A}$ is an IND-CPA adversary, it is polynomial-time. Hence the distinguisher $\mathcal{D}^{\mathcal{A}}$ is also polynomial-time.[7] Since the decisional Diffie-Hellman assumption holds for $\mathcal{G}$, Lemma 1.126 applies. Therefore we have that $\left|\Pr\left[\mathsf{G1c}_n^{\mathcal{A}}\right] - \Pr\left[\mathsf{G1d}_n^{\mathcal{A}}\right]\right|$ and, analogously, $\left|\Pr\left[\mathsf{G2c}_n^{\mathcal{A}}\right] - \Pr\left[\mathsf{G2d}_n^{\mathcal{A}}\right]\right|$ are negligible. Therefore, the (positive) expression $\left|\Pr\left[\mathsf{CPA1}_n^{\mathcal{EG},\mathcal{A}}\right] - \Pr\left[\mathsf{CPA2}_n^{\mathcal{EG},\mathcal{A}}\right]\right|$ is bounded from above by a negligible function and hence must also be negligible in n. □

[7] Here we assume that the distinguisher $\mathcal{D}^{\mathcal{A}}$ as defined above is polynomial-time, if the adversary $\mathcal{A}$ is polynomial-time. The verification of this fact is often omitted in hand-written proofs, because it is considered obvious or not an integral part of the reduction. In our formalization we also skipped its verification.

Chapter 2

EasyCrypt - Verified Security of Merkle-Damgård

2.1 Background on CertiCrypt/EasyCrypt

Independent of our development of Verypto, Barthe et al. developed the tool CertiCrypt [29, 32] which also enables the formal verification of cryptographic proofs. CertiCrypt has been implemented in the proof assistant Coq [63] and models games using a formal language called pWhile, for which game transformation are proven correct with respect to a probabilistic relational Hoare logic (pRHL). In contrast to the higher-order language of Verypto which we presented in Chapter 1, pWhile is imperative and only supports discrete measure spaces. On the one hand, this design has some limitations in terms of expressiveness compared to Verypto, e.g., with discrete probability distributions one cannot express the random sampling of infinite random tapes; also, since pWhile is not higher-order, oracles cannot be treated as first-class objects in the language. On the other hand, the pRHL for pWhile is a powerful tool to reason about randomized algorithms.

Indeed, CertiCrypt has proved itself expressive enough to handle several cryptographic constructions [29, 32, 33, 125, 34, 36, 28], which motivated the design of another framework called EasyCrypt [30] with the goal to automate the handling of CertiCrypt's game transformations in a dedicated tool. For this, EasyCrypt uses state-of-the-art verification tools, such as SMT solvers, automated theorem provers, and interactive proof assistants. EasyCrypt aims for making formally verified security accessible to cryptographers with a limited background in formal methods; it has been successfully applied to verify exact security bounds of several digital signature schemes and encryption schemes.

2.2 Contribution of this Chapter

In this chapter, we present the application of EasyCrypt to build and verify exact security proofs of the Merkle-Damgård construction [96, 70], which underlies the design of many cryptographic hash functions.[1] In its simplest formulation, Merkle-Damgård iterates a compression function $f : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ over the blocks of an input message that has been padded to a block boundary. For a fixed public initialization vector IV, the digest of a padded message with blocks $x_1, \ldots, x_\ell$ is computed as

$$f(x_\ell, f(x_{\ell-1}, \ldots f(x_1, \mathsf{IV}) \ldots)).$$

We verify in EasyCrypt that the Merkle-Damgård construction preserves the collision resistance of its underlying compression function, i.e., finding two colliding messages for the above iterated construction is at least as hard as finding two colliding inputs for the compression function f. Our proof requires that the padding function is suffix-free, i.e., the padding of a message m is not a suffix of the padding of any other message m'. This is a generalization of the seminal works of Merkle [96] and Damgård [70] who assumed that messages are padded in some specific way.

Furthermore, we verify in EasyCrypt that the Merkle-Damgård construction is indifferentiable from a random oracle when the compression function f is assumed to be ideal. The indifferentiability framework [94] provides a rigorous simulation-based definition which captures that the construction behaves like a random oracle [41] when the compression function, or some other lower-level building block, is assumed to be ideal. Indifferentiability implies a strong composability result: Glossing over technical subtleties [110], a hash function H that is indifferentiable from a random oracle can be plugged into a cryptosystem proven secure in the random oracle model for H without compromising the security of the cryptosystem. Our proof, which follows the proof of [66], applies when the padding function is prefix-free, i.e., the padding of a message m is not a prefix of the padding of any other message m'.

Our work was motivated by the increased interest in hash function security during the selection process of the new SHA-3 cryptographic hash algorithm. We discuss the applicability of our results to SHA-3 and some of its competitors in Section 3.2.

2.3 A Primer on EasyCrypt

Building a cryptographic proof in EasyCrypt is a process that can be decomposed in the following steps:

[1] This work has been published in [13] which I co-authored and also has been presented in the master's thesis [116] which was conducted under my guidance. I contributed to the elaboration of both.

- Defining a formal context, including types, constants and operators, and giving it meaning by declaring axioms and stating derived lemmas.
- Defining a number of games, each of them composed of a collection of procedures (written in the probabilistic imperative language pWhile described in Section 2.3.1) and adversaries declared as abstract procedures with access to oracles.
- Proving logical judgments that establish equivalences between games. This may be done fully automatically, with the help of hints from the user in the form of relational invariants, or interactively using basic tactics and automated strategies.
- Deriving inequalities between probabilities of events in games, either by using previously proven logical judgments or by direct computation.

In the following, we overview some key aspects of the process of building an EasyCrypt proof. In particular we introduce the input language pWhile in Section 2.3.1, define the probabilistic relational Hoare logic which is used to reason about games in Section 2.3.2, and show how to derive probability claims from it in Section 2.3.3.

2.3.1 Input Language

Probabilistic experiments are defined as programs in pWhile, a strongly-typed imperative probabilistic programming language. The grammar of pWhile commands is defined as follows:

$$
\begin{array}{rcll}
\mathcal{C} & ::= & \mathsf{skip} & \text{nop} \\
& | & \mathcal{V} \leftarrow \mathcal{E} & \text{deterministic assignment} \\
& | & \mathcal{V} \xleftarrow{\$} \mathcal{D}\mathcal{E} & \text{probabilistic assignment} \\
& | & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} & \text{conditional} \\
& | & \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} & \text{loop} \\
& | & \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E}) & \text{procedure call} \\
& | & \mathcal{C};\ \mathcal{C} & \text{sequence}
\end{array}
$$

The only non-standard feature of the language are probabilistic assignments; an assignment $x \xleftarrow{\$} d$ evaluates the expression d in the current state to a distribution μ on values, samples a value according to μ and assigns it to variable x. The key to the flexibility of EasyCrypt is that the base language of expressions and distribution expressions can be extended by the user to suit the needs of the verification task. The rich base language includes expressions over Booleans, integers, fixed-length bitstrings, lists, finite maps, and option, product, and sum types. User-defined operators can be axiomatized or defined in terms of other operators. In the following, we let $\{0,1\}^\ell$ denote the uniform distribution on bitstrings of length ℓ.

A program, i.e., a game in EasyCrypt is represented as a set of global variables together with a collection of procedures. Some of these procedures

are concrete and given a definition as a command $c \in \mathcal{C}$, while some others may be abstract and left undefined. Quantification over adversaries in cryptographic proofs is achieved by representing them as abstract procedures parametrized by a set of oracles; these oracles must be instantiated as other procedures in the program.

Commands operate on program memories, which map local and global variables to values; we let $\mathcal{M}$ denote the set of memories. The semantics of a command $c \in \mathcal{C}$ is a function $[\![c]\!] : \mathcal{M} \to \mathcal{D}(\mathcal{M})$ from program memories to subprobability distributions on program memories. Note that programs that do not terminate with probability 1 generate subprobability distributions with total probability less than 1. We refer the reader to [30] for a detailed description of the semantics of pWhile as it has been formalized in the Coq proof assistant. In what follows, we denote by $\Pr[c, m : A]$ the probability of event A with respect to the distribution $[\![c]\!]\ m$ and often omit the initial memory m when it is not relevant.

Although EasyCrypt is not tied to any particular cryptographic model, it provides good support to reason about proofs developed in the random oracle model. Random oracles [41] are functions that map values from their input domain into uniformly and independently distributed values in their output domain. In EasyCrypt a random oracle $\mathcal{O} : X \to Y$ is modeled as a stateful procedure that maps values in X into uniformly and independently distributed values in Y. The state of a random oracle can be represented as a global finite map $\boldsymbol{L}$ that is initially empty. Queries are answered consistently so that identical queries are given the same answer:

Oracle $\mathcal{O}(x)$:
  if $x \notin \mathsf{dom}(\boldsymbol{L})$ then $\boldsymbol{L}[x] \xleftarrow{\$} Y$
  return $\boldsymbol{L}[x]$

2.3.2 Probabilistic Relational Hoare Logic

The foundation of EasyCrypt is a probabilistic Relational Hoare Logic (pRHL), whose judgments are quadruples of the form

$$\vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi$$

where c_1, c_2 are programs and Ψ, Φ are first-order relational formulae. Relational formulae are defined by the grammar

$$\Psi, \Phi ::= e \mid \neg\Phi \mid \Psi \wedge \Phi \mid \Psi \vee \Phi \mid \Psi \Rightarrow \Phi \mid \forall x.\, \Phi \mid \exists x.\, \Phi$$

where e stands for a Boolean expression over logical variables and program variables tagged with either $\langle 1 \rangle$ or $\langle 2 \rangle$ to denote their interpretation in the left or right-hand side program; the only restriction is that logical variables must not occur free. The special keyword res denotes the return value of

a procedure and can be used in the place of a program variable. We write $e\langle i\rangle$ for the expression e in which all program variables are tagged with $\langle i\rangle$. A relational formula is interpreted as a relation on program memories. For example, the formula $x\langle 1\rangle + 1 \leq y\langle 2\rangle$ is interpreted as the relation

$$R = \{(m_1, m_2) \mid m_1(x) + 1 \leq m_2(y)\}.$$

The validity of a pRHL judgment is defined in terms of a lifting operator $\mathcal{L} : 2^{\mathcal{M}\times\mathcal{M}} \to 2^{\mathcal{D}(\mathcal{M})\times\mathcal{D}(\mathcal{M})}$. Concretely,

$$\begin{aligned}\models c_1 \sim c_2 : \Psi \Rightarrow \Phi \stackrel{\text{def}}{=}\\ \forall m_1, m_2.\ m_1\ \Psi\ m_2 \Rightarrow (\llbracket c_1 \rrbracket\ m_1)\ \mathcal{L}(\Phi)\ (\llbracket c_2 \rrbracket\ m_2).\end{aligned}$$

Formally, let μ_1, $\mu_2 \in \mathcal{D}(\mathcal{M})$ be distributions on memories. The lifting $\mu_1\, \mathcal{L}(R)\, \mu_2$ of a relation $R \subseteq \mathcal{M} \times \mathcal{M}$ to μ_1 and μ_2 is defined by the clause

$$\exists \mu \in \mathcal{D}(\mathcal{M}\times\mathcal{M}).\ \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2 \wedge \mathsf{supp}(\mu) \subseteq R$$

where $\pi_1(\mu)$ (resp. $\pi_2(\mu)$) denotes the projection of μ on its first (resp. second) component and $\mathsf{supp}(\mu)$ is the support of μ as a subprobability measure – if μ is discrete, this is just the set of pairs with positive probability.

Figure 2.1 shows some selected rules that can be used to derive valid pRHL judgments. There are two kinds of rules: two-sided rules, which require that the related programs have the same syntactic form, and one-sided rules, which do not impose this requirement. One-sided rules are symmetric in nature and admit a *left* and a *right* variant annotated with $\langle 1\rangle$ and $\langle 2\rangle$ respectively. We briefly comment on some rules. The two-sided rule Rnd for random assignments requires the distributions from where values are sampled be uniform on some set X; to apply the rule one must exhibit a function $f : X \to X$ that may depend on the state and is bijective if the precondition holds. The one-sided rule Rnd$\langle 1\rangle$ for random assignments simply requires that the post-condition is established for all possible outcomes; in effect, this rule treats a random assignment as a non-deterministic assignment.

Similarly to Hoare logic, the rules for while loops require to exhibit an appropriate relational invariant Φ. The two-sided rule While applies when the loops execute in lockstep and thus requires proving that the guards are equivalent. The one-sided rule While$\langle 1\rangle$ further requires exhibiting a decreasing variant v and a lower bound m. The premises ensure that the loop is absolutely terminating, which is crucial for the soundness of the rule.

The relational Hoare logic also allows capturing the well-known cryptographic argument *"x is uniformly distributed and independent of the adversary's view"*, which is certainly one of the most difficult to formalize. This argument is formalized in EasyCrypt by requiring that re-sampling x preserves the semantics of the program. Suppose we want to prove that in a program c, a variable x that is used in an oracle $\mathcal{O}$ is uniformly distributed and independent of the view of an adversary $\mathcal{A}^{\mathcal{O}}$. Let $\mathcal{O}'$ be the

$$\frac{\vdash c_1 \sim c_2 : \Phi \Longrightarrow \Phi' \qquad \vdash c_1' \sim c_2' : \Phi' \Longrightarrow \Phi''}{\vdash c_1; c_1' \sim c_2; c_2' : \Phi \Longrightarrow \Phi''}\ \textsc{Seq}$$

$$\frac{\Psi \Rightarrow \mathsf{bijective}(f) \qquad \Psi \Rightarrow \forall v \in X.\ \Phi\{v, f(v)/x\langle 1\rangle, y\langle 2\rangle\}}{\vdash x \xleftarrow{\$} X \sim y \xleftarrow{\$} X : \Psi \Longrightarrow \Phi}\ \textsc{Rnd}$$

$$\frac{\Psi \Rightarrow \forall v \in \mathsf{supp}(d).\ \Phi\{v/x\langle 1\rangle\}}{\vdash x \xleftarrow{\$} d \sim \mathsf{skip} : \Psi \Longrightarrow \Phi}\ \textsc{Rnd}\langle 1\rangle$$

$$\frac{}{\vdash x \leftarrow e \sim \mathsf{skip} : \Phi\{e\langle 1\rangle/x\langle 1\rangle\} \Longrightarrow \Phi}\ \textsc{Asn}\langle 1\rangle$$

$$\frac{}{\vdash \mathsf{skip} \sim x \leftarrow e : \Phi\{e\langle 2\rangle/x\langle 2\rangle\} \Longrightarrow \Phi}\ \textsc{Asn}\langle 2\rangle$$

$$\frac{\vdash c_1 \sim c_2 : \Psi \wedge e\langle 1\rangle \Longrightarrow \Phi \qquad \vdash c_1' \sim c_2 : \Psi \wedge \neg e\langle 1\rangle \Longrightarrow \Phi}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \sim c_2 : \Psi \Longrightarrow \Phi}\ \textsc{Cond}\langle 1\rangle$$

$$\frac{\vdash c_1 \sim c_2 : \Phi \wedge b_1\langle 1\rangle \Longrightarrow \Phi \qquad \Phi \Rightarrow b_1\langle 1\rangle = b_2\langle 2\rangle}{\vdash \mathsf{while}\ b_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ b_2\ \mathsf{do}\ c_2 : \Phi \Longrightarrow \Phi \wedge \neg b_1\langle 1\rangle}\ \textsc{While}$$

$$\frac{\vdash c_1 \sim \mathsf{skip} : \Phi \wedge (b_1 \wedge v = n)\langle 1\rangle \Longrightarrow \Phi \wedge v\langle 1\rangle < n \qquad \Phi \wedge v\langle 1\rangle \leq m \Rightarrow \neg b\langle 1\rangle}{\vdash \mathsf{while}\ b_1\ \mathsf{do}\ c_1 \sim \mathsf{skip} : \Phi \Longrightarrow \Phi \wedge \neg b_1\langle 1\rangle}\ \textsc{While}\langle 1\rangle$$

$$\frac{\Psi \Rightarrow \Psi' \qquad \vdash c_1 \sim c_2 : \Psi' \Longrightarrow \Phi' \qquad \Phi' \Rightarrow \Phi}{\vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi}\ \textsc{Sub}$$

$$\frac{\vdash c_1 \sim c_2 : \Psi \wedge \Psi' \Longrightarrow \Phi \qquad \vdash c_1 \sim c_2 : \Psi \wedge \neg\Psi' \Longrightarrow \Phi}{\vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi}\ \textsc{Case}$$

Figure 2.1: Selected pRHL rules

same as $\mathcal{O}$ except that it re-samples x when needed. We identify a condition used that holds whenever $\mathcal{A}$ obtained some information about x (and thus, re-sampling would not preserve the semantics). We then prove that the conditional statement $c' \stackrel{\text{def}}{=} \mathsf{if}\ \neg\mathsf{used}\ \mathsf{then}\ x \stackrel{\$}{\leftarrow} X$ can *swap* with calls to $\mathcal{O}$ and $\mathcal{O}'$, i.e.,

$$\vdash c'; y \leftarrow \mathcal{O}(\vec{e}) \sim y \leftarrow \mathcal{O}'(\vec{e}); c' : \Phi \Longrightarrow \Phi$$

where Φ implies equality over all global variables. From this, we can conclude that c' can also swap with calls to $\mathcal{A}^{\mathcal{O}}$ and $\mathcal{A}^{\mathcal{O}'}$, and hence that the semantics of the program c is preserved when $\mathcal{O}$ is replaced by $\mathcal{O}'$ [33]. The advantage of using such kind of reasoning is that it is generally much easier to reason about a game where x is sampled *lazily*, since its distribution is locally known.

We conclude with some observations on the mechanization of reasoning in pRHL. EasyCrypt provides several variants of two-sided and one-sided rules of pRHL in the form of tactics that can be applied in a goal-oriented fashion to prove the validity of judgments. For instance, instead of applying rule $\textsc{Rnd}\langle 1\rangle$, one can use the following combination with the $\textsc{Seq}$ rule which is more easily applicable:

$$\frac{\vdash c_1 \sim c_2 : \Psi \Longrightarrow \forall v \in \mathsf{supp}(d).\ \Phi\,\{v/x\langle 1\rangle\}}{\vdash c_1;\ x \stackrel{\$}{\leftarrow} d \sim c_2 : \Psi \Longrightarrow \Phi}$$

The application of a tactic may generate additional verification subgoals, and logical side conditions that are checked using SMT solvers, automated theorem provers and, as a last recourse, interactive proof assistants. Depending on their nature, application of the tactics can be fully automated or require user input. For instance, applying the tactics that mechanize the rules for while loops, requires the user to provide an adequate invariant. In the case of the two-sided rule, a new subgoal is generated to prove the correctness of the user-provided invariant, whereas the equivalence of the loop guards is checked automatically as a logical side-condition.

In addition to tactics that mechanize basic rules of pRHL, EasyCrypt implements automated strategies that combine the application of a weakest precondition transformer wp with heuristics to apply basic tactics. The wp transformer operates on deterministic loop-free programs. These strategies can often be used to deal automatically with large fragments of proofs, letting the user focus on the parts that require ingenuity.

2.3.3 Reasoning about Probabilities

Since cryptographic results are stated as inequalities on probabilities rather than pRHL judgments, it is important to derive probability claims from

pRHL judgments. This can be done mechanically by applying rules in the style of

$$\frac{m_1 \,\Psi\, m_2 \qquad \vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi \qquad \Phi \Rightarrow A\langle 1\rangle \Rightarrow B\langle 2\rangle}{\Pr\left[c_1, m_1 : A\right] \leq \Pr\left[c_2, m_2 : B\right]}\,.$$

Game-based proofs often argue that two programs c_1 and c_2 behave identically unless a failure event F is triggered. This is used to conclude that the difference in probability of any event between the two programs is bounded by the probability of F in one of them. Although a syntactic characterization of this lemma is often used (in which failure is represented by a Boolean flag), it can be conveniently expressed and implemented in EasyCrypt in a more general form using pRHL.

Lemma 2.2 (Fundamental Lemma)**.** *Let c_1 and c_2 be two terminating commands and A, B, F events such that*

$$\vdash c_1 \sim c_2 : \Psi \Longrightarrow F\langle 1\rangle \Leftrightarrow F\langle 2\rangle \wedge (\neg F\langle 1\rangle \Rightarrow\ A\langle 1\rangle \Leftrightarrow B\langle 2\rangle).$$

Then, if the initial memories of both games satisfy Ψ,

$$\left|\Pr\left[c_1 : A\right] - \Pr\left[c_2 : B\right]\right| \leq \Pr\left[\mathsf{c_1} : F\right] = \Pr\left[\mathsf{c_2} : F\right].$$

In most applications of the above lemma, the failure event F can only be triggered in oracle queries made by an adversary. When the adversary can only make a known bounded number of queries, the following lemma, which is implemented in EasyCrypt, provides means to bound the probability of failure. We describe its hypotheses informally, but note that most of them can be captured by pRHL judgments.

Lemma 2.3 (Failure event lemma)**.** *Consider a program $c_1; c_2$, an integer expression i, an event F, and $u \in \mathbb{R}$. Assume the following:*

- *Free variables in F and i are only modified by c_1 or oracles in some set O;*
- *After executing c_1, F does not hold and $0 \leq i$;*
- *Oracles $\mathcal{O} \in O$ do not decrease i and strictly increase i when F is triggered;*
- *For every oracle $\mathcal{O}$ in O, $\neg F \Rightarrow \Pr\left[\mathcal{O} : F\right] \leq u$*

Then, $\Pr\left[c_1; c_2 : F \wedge i \leq q\right] \leq q \cdot u$.

Finally, EasyCrypt implements a simple mechanism to directly compute bounds for the probability of an event in a program. This mechanism can establish, for instance, that the probability that a value uniformly chosen from a set X equals an independent expression is exactly $1/\left|X\right|$, or that the probability that this value belongs to an independent list of length n is at most $n/\left|X\right|$.

2.4 The Merkle-Damgård Construction

Merkle-Damgård is a method for building a variable input-length (VIL) hash function from a fixed input-length (FIL) compression function. In its simplest form, the digest of a message is computed by first padding it to a block boundary and then iterating a compression function f over the resulting blocks starting from an initial chaining value IV. To be more precise, a padding function pad converts a message of arbitrary length into a list of bitstrings of block size k, while a compression function f maps a pair of bitstrings of length k and n to a bitstring of length n:

$$\begin{aligned} \mathsf{pad} &: \{0,1\}^* \to (\{0,1\}^k)^* \\ f &: \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n \end{aligned}$$

Definition 2.4 (Merkle-Damgård)**.** *Let f be a compression function and pad a padding function as above, and let $\mathsf{IV} \in \{0,1\}^n$ be a public value, known as the* initialization vector*. The* Merkle-Damgård hash function MD *is defined as*

$$\begin{array}{lcl} \mathsf{MD} & : & \{0,1\}^* \to \{0,1\}^n \\ \mathsf{MD}(m) & \stackrel{def}{=} & f^*(\mathsf{pad}(m), \mathsf{IV}) \end{array}$$

where $f^* : (\{0,1\}^k)^* \times \{0,1\}^n \to \{0,1\}^n$ is defined recursively as

$$\begin{array}{lcl} f^*(\mathsf{nil}, y) & \stackrel{def}{=} & y \\ f^*(x{::}xs, y) & \stackrel{def}{=} & f^*(xs, f(x,y)). \end{array}$$

The security properties of the Merkle-Damgård construction greatly depend on an adequate choice of the used padding function to thwart certain types of attacks. In the remainder, we consider prefix- and suffix-free padding functions.

Definition 2.5 (Prefix- and suffix-free padding)**.** *A padding function* pad *is* prefix-free *(resp.* suffix-free*) iff for any distinct messages m, m', there is no xs such that $\mathsf{pad}(m') = \mathsf{pad}(m) \parallel xs$ (resp. $\mathsf{pad}(m') = xs \parallel \mathsf{pad}(m)$), where $\parallel$ denotes the concatenation of bitstrings.*

Security properties of hash functions are stated as claims about the difficulty of an attacker in achieving certain goals. Given a hash function H, collision resistance states that it is hard to find distinct messages m_1, m_2 with the same digest $H(m_1) = H(m_2)$. Preimage resistance states that given a digest h, it is hard to find a message m such that $H(m) = h$. Second preimage resistance states that given a message m_1, it is hard to find a message $m_2 \neq m_1$ such that $H(m_1) = H(m_2)$. Finally, resistance to length-extension attacks states that it is hard to compute $H(m_1 \parallel m_2)$ from $H(m_1)$. The precise formulation of these notions and their relationship is addressed in detail in [111].

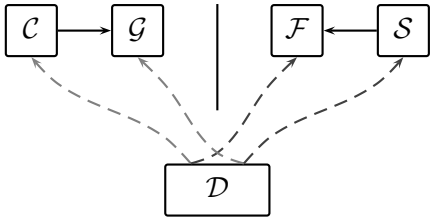
An established method for proving the security of domain extenders, as MD above, is to show that they are property-preserving: For instance, the seminal works of Merkle [96] and Damgård [70] show that if the compression function f is collision resistant, then the hash function MD with some specific padding function is also collision resistant. Property preservation also applies for other notions; a representative panorama of property preservation for collision resistance, preimage and second preimage resistance appears in [10]. In Section 2.5 we use EasyCrypt to reduce the collision resistance of MD with a suffix-free padding rule to the collision resistance of the underlying compression function.

An alternative method for proving the security of domain extenders is to show that they preserve ideal functionalities, i.e., that when applied to ideal functionalities they yield an ideal functionality. The notion of indifferentiability of [94] provides an appropriate framework.

Definition 2.6 (Indifferentiability)**.** *A procedure $\mathcal{C}$ with oracle access to an ideal primitive $\mathcal{G}$ is $(t_{\mathcal{S}}, q, \epsilon)$-indifferentiable from $\mathcal{F}$ iff there exists a simulator $\mathcal{S}$ with oracle access to $\mathcal{F}$ and executing within time $t_{\mathcal{S}}$, such that for any distinguisher $\mathcal{D}$ that makes at most q oracle queries, the following inequality holds:*

$$\left|\Pr\left[b \leftarrow \mathcal{D}^{\mathcal{C},\mathcal{G}}() : b\right] - \Pr\left[b \leftarrow \mathcal{D}^{\mathcal{F},\mathcal{S}}() : b\right]\right| \leq \epsilon$$

Intuitively this means that the goal of $\mathcal{D}$ is to distinguish between the following two scenarios: In the *real scenario* the distinguisher is given access to $\mathcal{C}^{\mathcal{G}}$ and $\mathcal{G}$, and in the *ideal scenario* it is given access to $\mathcal{F}$ and $\mathcal{S}^{\mathcal{F}}$ as illustrated below. If the probability that it succeeds in differentiating between the real scenario and the ideal scenario is small, we say that $\mathcal{C}$ is indifferentiable from $\mathcal{F}$.



In the setting considered here, $\mathcal{C}$ represents the Merkle-Damgård construction, $\mathcal{G}$ represents the compression function and $\mathcal{F}$ represents an idealized hash function, i.e., a random oracle. Thus, the role of $\mathcal{S}$ is to simulate the behavior of the compression function, namely, it should behave towards $\mathcal{F}$ as $\mathcal{G}$ behaves towards the Merkle-Damgård construction. In Section 2.6, we use EasyCrypt to define such a simulator $\mathcal{S}$ and verify the indifferentiability of MD from a VIL random oracle when the compression function $\mathcal{G}$ is modeled as a FIL random oracle. See Section 2.3.1 for a precise definition of random oracles.

We conclude this section with two observations about proofs of indifferentiability and property preservation. First, indifferentiability from a random oracle provides weaker guarantees than initially anticipated (see [57] and [110] respectively for discussions on the random oracle model and on the notion of indifferentiability), but nevertheless remains a useful heuristics in the design of hash functions. Second, the two methods are complementary. On the one hand, indifferentiability from a VIL random oracle entails resistance against collision, preimage, second preimage, and length-extension attacks. Thus, preservation of ideal functionalities apparently yields stronger guarantees than property preservation. On the other hand, however, property preservation is typically established under weaker hypotheses and exact security bounds derived from indifferentiability proofs generally deliver looser bounds than direct proofs based on property preservation.

2.5 Collision Resistance

We verify that finding collisions for MD with a suffix-free padding is at least as hard as finding collisions for its compression function f, where a collision for f consists of two input pairs (x_1, y_1), (x_2, y_2) satisfying the predicate

$$\mathsf{coll}((x_1, y_1), (x_2, y_2)) \stackrel{\text{def}}{=} (x_1, y_1) \neq (x_2, y_2) \wedge f(x_1, y_1) = f(x_2, y_2).$$

Theorem 2.7. *Let* MD *be a Merkle-Damgård hash function with compression function* f *and a suffix-free padding* pad*. For any algorithm* $\mathcal{A}$ *finding collisions for* MD *of at most length* ℓ *with probability* p*, there exists an algorithm* $\mathcal{B}$ *that finds collisions for* f *with probability at least* p *and with a time overhead of* $O(\ell\, t_f)$*, where* t_f *is a bound on the time needed for one evaluation of* f*.*

Consider the experiment $\mathsf{CR}^{\mathsf{MD}}$ below, in which an adversary $\mathcal{A}$ performs a collision attack against MD:

Game $\mathsf{CR}^{\mathsf{MD}}$:	**Oracle** $\mathsf{F}(m)$:
$(m_1, m_2) \leftarrow \mathcal{A}()$;	$xs \leftarrow \mathsf{pad}(m)$; $y \leftarrow \mathsf{IV}$;
$h_1 \leftarrow \mathsf{F}(m_1)$;	while $xs \neq \mathsf{nil}$ do
$h_2 \leftarrow \mathsf{F}(m_2)$;	$y \leftarrow f(\mathsf{hd}(xs), y)$;
return $(m_1 \neq m_2 \wedge h_1 = h_2)$	$xs \leftarrow \mathsf{tl}(xs)$;
	return y

Using $\mathcal{A}$, we construct a collision-finder $\mathcal{B}$ for the compression function f. For this, algorithm $\mathcal{B}$ obtains from $\mathcal{A}$ a pair of messages m_1, m_2, pads them, and iterates the compression function over the first blocks of the longer padded message until the remaining suffix is the same length as the other padded message. It then performs the remaining iterations needed to compute $\mathsf{MD}(m_1)$ and $\mathsf{MD}(m_2)$ in parallel. If m_1 and m_2 form a collision for

MD, a collision for f must occur during one of these iterations. Algorithm $\mathcal{B}$ stops as soon as it detects one such collision, returning the colliding inputs as a result:

Game CR^f :	**Adversary** $\mathcal{B}()$:

$(xy_1, xy_2) \leftarrow \mathcal{B}();$
return $\mathsf{coll}(xy_1, xy_2)$
	$(m_1, m_2) \leftarrow \mathcal{A}();$
	$xs_1 \leftarrow \mathsf{pad}(m_1);\ y_1 \leftarrow \mathsf{IV};$
	$xs_2 \leftarrow \mathsf{pad}(m_2);\ y_2 \leftarrow \mathsf{IV};$
	while $\lvert xs_1\rvert > \lvert xs_2\rvert$ do
	  $y_1 \leftarrow f(\mathsf{hd}(xs_1), y_1);\ xs_1 \leftarrow \mathsf{tl}(xs_1);$
	while $\lvert xs_1\rvert < \lvert xs_2\rvert$ do
	  $y_2 \leftarrow f(\mathsf{hd}(xs_2), y_2);\ xs_2 \leftarrow \mathsf{tl}(xs_2);$
	while $\neg\mathsf{coll}((\mathsf{hd}(xs_1), y_1), (\mathsf{hd}(xs_2), y_2)) \wedge xs_1 \neq \mathsf{nil}$ do
	  $y_1 \leftarrow f(\mathsf{hd}(xs_1), y_1);\ xs_1 \leftarrow \mathsf{tl}(xs_1);$
	  $y_2 \leftarrow f(\mathsf{hd}(xs_2), y_2);\ xs_2 \leftarrow \mathsf{tl}(xs_2);$
	return $((\mathsf{hd}(xs_1), y_1), (\mathsf{hd}(xs_2), y_2))$

We prove in EasyCrypt that the algorithm $\mathcal{B}$ depicted above finds collisions for f in the experiment CR^f with at least the same probability as $\mathcal{A}$ finds collisions for MD in $\mathsf{CR}^{\mathsf{MD}}$:

$$\Pr\left[\mathsf{CR}^{\mathsf{MD}} : \mathsf{res}\right] \leq \Pr\left[\mathsf{CR}^f : \mathsf{res}\right] \tag{2.8}$$

Recall here that res is a keyword that stands for the value returned by the main procedure of the games. In order to show (2.8) it suffices to prove the relational judgment:

$$\vdash \mathsf{CR}^{\mathsf{MD}} \sim \mathsf{CR}^f : \mathit{true} \Longrightarrow \mathsf{res}\langle 1\rangle \Rightarrow \mathsf{res}\langle 2\rangle \tag{2.9}$$

Proving this judgment involves non-trivial relational reasoning because equivalent computations in the related games are not performed in lockstep. Namely, in game $\mathsf{CR}^{\mathsf{MD}}$ the messages m_1, m_2 are hashed sequentially, while algorithm $\mathcal{B}$ hashes their suffixes in parallel. We begin by inlining the call to $\mathcal{B}$ in CR^f and show that the relational post-condition

$$(m_1, m_2)\langle 1\rangle = (m_1, m_2)\langle 2\rangle \ \wedge (h_1 = \mathsf{MD}(m_1) \wedge h_2 = \mathsf{MD}(m_2))\langle 1\rangle$$

holds after the two calls to F in $\mathsf{CR}^{\mathsf{MD}}$ and the call to $\mathcal{A}$ in CR^f. To show this, we prove that oracle F correctly implements function MD using the one-sided rule for loops – the needed invariant is simply $f^*(xs, y) = \mathsf{MD}(m)$. At this point, note that if $m_1 = m_2$ or $\mathsf{MD}(m_1) \neq \mathsf{MD}(m_2)$, judgment (2.9) holds trivially (we only have to check that $\mathcal{B}$ terminates). We are left with the case where $m_1 \neq m_2$ and $\mathsf{MD}(m_1) = \mathsf{MD}(m_2)$. Assume w.l.o.g. that $|\mathsf{pad}(m_2)| \leq |\mathsf{pad}(m_1)|$, in which case $\mathcal{B}$ never enters its second loop and the following invariant holds for the first:

$$\begin{array}{l} f^*(xs_1, y_1) = \mathsf{MD}(m_1) \wedge f^*(xs_2, y_2) = \mathsf{MD}(m_2) \ \wedge \\ m_1 \neq m_2 \wedge |xs_2| \leq |xs_1| \wedge xs_2 = \mathsf{pad}(m_2) \ \wedge \\ \exists xs'.\ xs' \parallel xs_1 = \mathsf{pad}(m_1) \end{array} \tag{2.10}$$

We prove that if the messages m_1, m_2 output by $\mathcal{A}$ collide, the last loop necessarily exits because a collision is found. This can be shown by means of the following loop invariant:

$$f^*(xs_1, y_1) = \mathsf{MD}(m_1) \wedge f^*(xs_2, y_2) = \mathsf{MD}(m_2) \wedge$$
$$|xs_2| = |xs_1| \wedge$$
$$(xs_1 = xs_2 \Rightarrow y_1 \neq y_2)$$

Note that (2.10) and the negation of the guard of the first loop imply that the above invariant holds initially. In particular, the last implication holds because if xs_1 and xs_2 were equal, there would exist a prefix xs' such that $xs' \parallel \mathsf{pad}(m_2) = \mathsf{pad}(m_1)$, contradicting the fact that pad is suffix-free. Finally, observe that the last loop can exit either because a collision for f is found or because $xs_1 = \mathsf{nil}$. In this latter case, it also holds that $xs_2 = \mathsf{nil}$ and therefore $y_1 = \mathsf{MD}(m_1) = \mathsf{MD}(m_2) = y_2$. However, from the last implication in the invariant we also have $y_1 \neq y_2$, which leads to a contradiction and renders this case trivial.

2.6 Indifferentiability

We verify the indifferentiability of the MD construction from a VIL random oracle in $\{0,1\}^* \to \{0,1\}^n$ when its compression function f is modeled as a FIL random oracle in $\{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ and its padding function is prefix-free. Our proof is based on [66].

Theorem 2.11 (Indifferentiability of MD)**.** *The Merkle-Damgård construction* MD *with an ideal compression function* f*, prefix-free padding* pad*, and initialization vector* IV *is* $(t_{\mathcal{S}}, q_{\mathcal{D}}, \epsilon)$*-indifferentiable from a variable input-length random oracle* $F : \{0,1\}^* \to \{0,1\}^n$*, where*

$$\epsilon = \frac{3\ell^2 q_D^2}{2^n} \qquad t_{\mathcal{S}} = O(\ell\, q_D^2)$$

and ℓ *is an upper bound on the block-length of* $\mathsf{pad}(m)$ *for any message* m *appearing in a query of the distinguisher.*

On the one hand, in the real scenario, a distinguisher $\mathcal{D}$ has access to an oracle F_q which implements the Merkle-Damgård construction MD and to a random oracle $f_q : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ which models the compression function. On the other hand, in the ideal scenario, $\mathcal{D}$ has access to a random oracle $F_q : \{0,1\}^* \to \{0,1\}^n$ and f_q is simulated. See Figure 2.12 for a formulation of these two scenarios as games G_{real} and $\mathsf{G}_{\text{ideal}}$.

To prevent $\mathcal{D}$ from making more than q oracle queries, we enforce a bound $q = \ell\, q_{\mathcal{D}}$ on the counter $\mathbf{q}_f$, that counts the number of evaluations of the compression function in game G_{real}. Note that this is more permissive than the proof of [66], as it allows the distinguisher to trade queries to F_q

Game G_{real} :
$\mathbf{q}_f \leftarrow 0$;
$\boldsymbol{T} \leftarrow \emptyset$;
$b \leftarrow \mathcal{D}^{F_q, f_q}()$;
return b

Oracle $f(x, y)$:
if $(x, y) \notin \mathsf{dom}(\boldsymbol{T})$ then
  $z \xleftarrow{\$} \{0,1\}^n$;
  $\boldsymbol{T}[x, y] \leftarrow z$
return $\boldsymbol{T}[x, y]$

Oracle $F_q(m)$:
$xs \leftarrow \mathsf{pad}(m)$;
$y \leftarrow \mathsf{IV}$;
if $\mathbf{q}_f + |xs| \leq q$ then
  $\mathbf{q}_f \leftarrow \mathbf{q}_f + |xs|$;
  while $xs \neq \mathsf{nil}$ do
    $y \leftarrow f(\mathsf{hd}(xs), y)$;
    $xs \leftarrow \mathsf{tl}(xs)$
return y

Oracle $f_q(x, y)$:
if $\mathbf{q}_f + 1 \leq q$ then
  $\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$
  $z \leftarrow f(x, y)$;
else $z \leftarrow \mathsf{IV}$
return z

Game $\mathsf{G}_{\text{ideal}}$:
$\mathbf{q}_f \leftarrow 0$;
$\boldsymbol{R}, \boldsymbol{T'} \leftarrow \emptyset$;
$b \leftarrow \mathcal{D}^{F_q, f_q}()$;
return b

Oracle $F(m)$:
if $m \notin \mathsf{dom}(\boldsymbol{R})$ then
  $z \xleftarrow{\$} \{0,1\}^n$;
  $\boldsymbol{R}[m] \leftarrow z$
return $\boldsymbol{R}[m]$

Oracle $F_q(m)$:
$xs \leftarrow \mathsf{pad}(m)$;
$y \leftarrow \mathsf{IV}$;
if $\mathbf{q}_f + |xs| \leq q$ then
  $\mathbf{q}_f \leftarrow \mathbf{q}_f + |xs|$;
  $z \leftarrow F(m)$
else $z \leftarrow \mathsf{IV}$
return z

Oracle $f_q(x, y)$:
if $\mathbf{q}_f + 1 \leq q$ then
  if $(x, y) \notin \mathsf{dom}(\boldsymbol{T'})$ then
    $xs \leftarrow \mathsf{findseq}((x, y), \boldsymbol{T'})$
    if $xs \neq \perp$ then
      $\boldsymbol{T'}[x, y] \leftarrow F(\mathsf{pad}^{-1}(xs \| x))$
    else
      $\boldsymbol{T'}[x, y] \xleftarrow{\$} \{0,1\}^n$
  $z \leftarrow \boldsymbol{T'}[x, y]$; $\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$
else $z \leftarrow \mathsf{IV}$
return z

Figure 2.12: The games G_{real} and $\mathsf{G}_{\text{ideal}}$

for queries to f_q. Indeed, if $\mathcal{D}$ makes n_f queries to f_q and n_F queries to F_q, we require

$$\mathbf{q}_f \leq n_f + \ell\, n_F \leq \ell\,(n_f + n_F) \leq \ell\, q_{\mathcal{D}} = q.$$

We show that the simulator f_q in $\mathsf{G}_{\text{ideal}}$ behaves consistently with a random oracle. Whenever the distinguisher makes a query (x, y) to oracle f_q, the simulator looks among all previous queries for a sequence that could be the chain of inputs to the compression function used to compute the hash of some message m, for which x is the last block of $\mathsf{pad}(m)$. We call such a sequence a *complete chain*, and we define it formally below. When such a sequence is found, the simulator queries the random oracle F for the hash of message m and forwards the answer to the distinguisher. Otherwise, the simulator answers with a uniformly distributed random value. Figure 2.13 shows how this simulator would react to a sequence of fresh queries

$$y_2 \leftarrow f_q(x_1, \mathsf{IV});\ y_3 \leftarrow f_q(x_2, y_2);\ y_4 \leftarrow f_q(x_3, y_3);\ y_5 \leftarrow f_q(x_4, y_4)$$

where $x_1 \| x_2 \| x_3 = \mathsf{pad}(m)$ for some message m. Note that since pad is prefix-free, no message can have the padding x_1, $x_1 \| x_2$, or $x_1 \| x_2 \| x_3 \| x_4$. Therefore only the third query completes a chain and hence is answered by forwarding $\mathsf{pad}^{-1}(x_1 \| x_2 \| x_3) = m$ to F in order to maintain consistency with

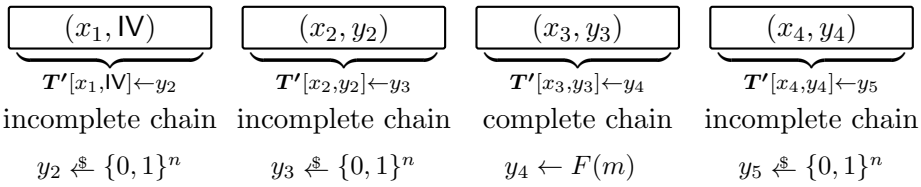


Figure 2.13: The simulator queries the random oracle F for complete chains.

the real scenario. The other queries are just answered with random values. The same holds for any other query that extends this chain further.

Definition 2.14 (Complete chain)**.** *Let $T : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a map. A complete chain in T is a sequence $(x_1, y_1) \ldots (x_i, y_i)$ such that $y_1 = \mathsf{IV}$ and*

1. $\forall j = 1 \ldots i-1.\ (x_j, y_j) \in \mathsf{dom}(T) \wedge T[x_j, y_j] = y_{j+1}$
2. *$x_1 \parallel \ldots \parallel x_i$ is in the domain of* pad^{-1}

The function $\mathsf{findseq}((x,y), \boldsymbol{T'})$ used by the simulator searches in $\boldsymbol{T'}$ for a complete chain of the form $(x_1, y_1) \ldots (x_i, y_i)(x, y)$ and returns $x_1 \| \ldots \| x_i$, or $\perp$ to indicate that no such chain exists.

In our proofs, we needed to derive several auxiliary lemmas to help the SMT solvers and automated provers to check the validity of arising logical side-conditions; e.g., if a finite map T is injective and does not map any entry to the value IV, every complete chain is determined by its last element – that is, for any given (x, y), the value of $\mathsf{findseq}((x,y), \boldsymbol{T'})$ is uniquely determined. All of these lemmas have been mechanically verified based solely on the axiomatization and definitions of elementary operations. In many cases, EasyCrypt is able to verify the validity of these lemmas automatically. The more involved lemmas have been manually verified in the Coq proof assistant.

The proof proceeds by stepwise transforming the game G_{real} into the game $\mathsf{G}_{\text{ideal}}$, upper-bounding the probability that the outcomes of consecutive games differ. By summing up over these probabilities, we obtain a concrete bound for the advantage of the distinguisher in telling apart the initial and final games. Specifically, we prove in EasyCrypt:

$$|\Pr[\mathsf{G}_{\text{real}} : b] - \Pr[\mathsf{G}_{\text{ideal}} : b]| \leq \frac{3q^2}{2^n} \tag{2.15}$$

We begin by considering the game $\mathsf{G}_{\text{real}'}$ as defined in Figure 2.16. We introduce events $\mathbf{bad}_1$, $\mathbf{bad}_2$, and $\mathbf{bad}_3$ that will be needed and explained later. First, we introduce a copy of oracle f, which we call $f_{\mathbf{bad}}$. Both use the same map $\boldsymbol{T}$ to store previously answered queries, the difference is that $f_{\mathbf{bad}}$ may trigger events $\mathbf{bad}_1$ and $\mathbf{bad}_2$. We also introduce the lists

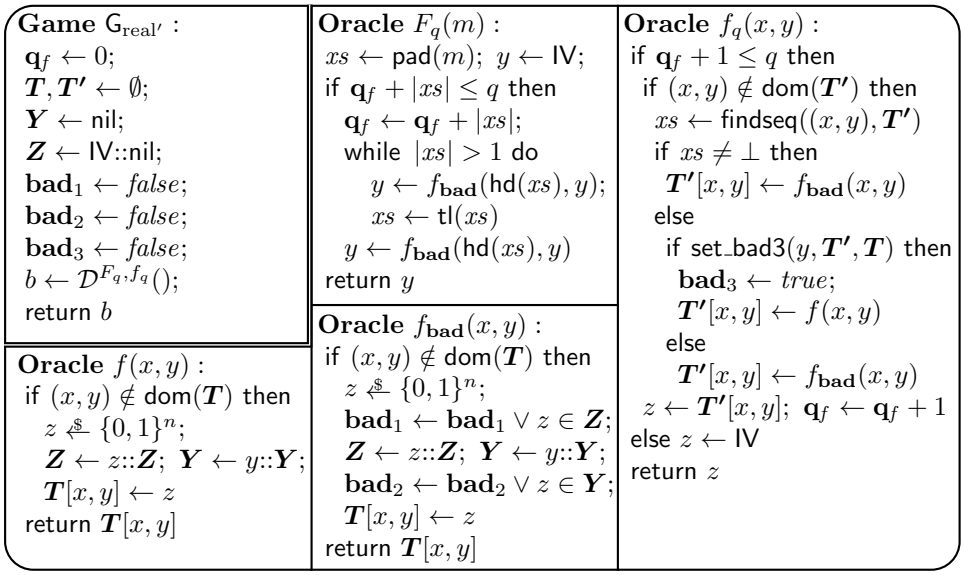


Figure 2.16: The game $\mathsf{G}_{\mathrm{real}'}$

$\boldsymbol{Y}$ and $\boldsymbol{Z}$ that allow us to appropriately detect when these events occur. In addition, we modify the simulator f_q to maintain a map $\boldsymbol{T'}$ of queries known to the distinguisher. Observe that $\boldsymbol{T'} \subseteq \boldsymbol{T}$, because queries to F_q result in entries being added only to $\boldsymbol{T}$, whereas queries to f_q result in the same entries being added to both $\boldsymbol{T}$ and $\boldsymbol{T'}$. Additionally, the simulator f_q behaves in two different ways depending on whether $\mathsf{findseq}((x, y), \boldsymbol{T'}) \neq \perp$. If this condition holds, there is a complete chain in map $\boldsymbol{T'}$ ending in (x, y). In this case, in game $\mathsf{G}_{\mathrm{ideal}}$ the simulator must call oracle F to maintain consistency with the random oracle; otherwise, the simulator can just sample a fresh random value. In this game, oracle f_q returns the same answer in both cases, but sets $\mathbf{bad}_{\{1,2,3\}}$ accordingly. Lastly, we also unroll the last iteration of the loop in F_q.

Note that the introduction of the additional map $\boldsymbol{T'}$ and the failure events $\mathbf{bad}_{\{1,2,3\}}$ does not change the observable behavior of the game. Therefore,

$$\Pr\left[\mathsf{G}_{\mathrm{real}} : b\right] = \Pr\left[\mathsf{G}_{\mathrm{real}'} : b\right].$$

In game $\mathsf{G}_{\mathrm{realF}}$, defined in Fig. 2.17, we introduce a VIL random oracle F and replace every call $f_{\mathbf{bad}}(x, y)$ in game $\mathsf{G}_{\mathrm{real}'}$ where (x, y) ends a complete chain in $\boldsymbol{T}$ with a call to $F(m, y)$ where m is the unpadded message of the chain; i.e., in oracle f_q we call F if findseq is successful and in oracle F_q we call F instead of the last call to $f_{\mathbf{bad}}$. We also introduce the map $\boldsymbol{I} : \mathbb{N} \to \{0,1\}^n \times \mathbb{B}$ which enumerates all sampled chaining values and includes a *tainted* flag to keep track of values known to the distinguisher.

We introduce an indirection in map $\boldsymbol{T}$ and $\boldsymbol{T'}$ through the use of map $\boldsymbol{I}$. This allows us to keep track of the order in which queries were made and to know which answers we could re-sample without introducing inconsistencies in the view of the distinguisher.

The failure events that were introduced in the last step capture certain dependencies on previous queries that the distinguisher might exploit to tell games $\mathsf{G}_{\text{real}'}$ and $\mathsf{G}_{\text{realF}}$ apart. We prove in EasyCrypt that games $\mathsf{G}_{\text{real}'}$ and $\mathsf{G}_{\text{realF}}$ behave equivalently provided that these failure events do not occur.

1. $\mathbf{bad}_1$ is triggered whenever oracle $f_{\mathbf{bad}}$ samples a random value that is either IV or has already been sampled for a distinct query before. The role of this event is twofold: on the one hand, if IV is sampled as a random value, then there could exist a complete chain in $\boldsymbol{T}$ that is a suffix of another complete chain in $\boldsymbol{T}$ as illustrated in the first example of Figure 2.18 (here $\boldsymbol{T}[x_2, y_2] = \mathsf{IV}$). The problem is that oracle F_q in the game G_{real} will generate the same values for the two messages corresponding to these two chains, while F_q in the game $\mathsf{G}_{\text{ideal}}$ most likely will not. On the other hand, if a value is sampled that has been sampled for another query before, then there could exist two complete chains in $\boldsymbol{T}$ that collide at some point and are identical from that point on as illustrated in the second example of Figure 2.18. Again the two corresponding messages would yield the same answer in G_{real} but most likely not in $\mathsf{G}_{\text{ideal}}$ on queries to F_q. By requiring that event $\mathbf{bad}_1$ does not occur, we guarantee that in game $\mathsf{G}_{\text{real}'}$ the map $\boldsymbol{T}$ is injective and does not map any value to IV.

2. $\mathbf{bad}_2$ is triggered whenever oracle $f_{\mathbf{bad}}$ samples a random value that has already been used as a chaining value in a previous query. This means that this query may be part of a chain for which the distinguisher has already queried later points (x, y). This newly introduced connection might, for instance, cause (x, y) to become the end of a complete chain. Since $\boldsymbol{T}[x, y]$ has already been sampled, the result for querying the corresponding message in G_{real} has already been fixed, whereas querying this message in $\mathsf{G}_{\text{ideal}}$ will most likely produce a different result. The event also captures that no fixed-points (i.e., entries of the form $\boldsymbol{T}[x, y] = y$) should be sampled. In this case a message could repeat the block x arbitrarily often without changing the corresponding hash value in G_{real}.

3. We use $\mathbf{bad}_3$ to ensure that the distinguisher constructs chains only in the right order; i.e., the distinguisher should not be able to query points in a chain without having queried all previous points of the chain. Formally, $\mathbf{bad}_3$ is triggered whenever a chaining value y in a query has already been sampled as a random value and hence is in the range of $\boldsymbol{T}$ for some previous point (x', y'), but (x', y') does not appear

Game $\mathsf{G}_{\mathrm{realF}}$:
$\mathbf{q}_f \leftarrow 0$;
$\mathbf{q}'_f \leftarrow 1$;
$\boldsymbol{T}, \boldsymbol{T'}, \boldsymbol{T'_i}, \boldsymbol{R}, \boldsymbol{I} \leftarrow \emptyset$;
$\boldsymbol{I}[0] \leftarrow (\mathsf{IV}, \mathit{false})$;
$\boldsymbol{Y} \leftarrow \mathsf{nil}$;
$\boldsymbol{Z} \leftarrow \mathsf{IV}::\mathsf{nil}$;
$\mathbf{bad}_1 \leftarrow \mathit{false}$;
$\mathbf{bad}_2 \leftarrow \mathit{false}$;
$\mathbf{bad}_3 \leftarrow \mathit{false}$;
$b \leftarrow \mathcal{D}^{F_q, f_q}()$;
return b

Oracle $F(m, y)$:
if $m \notin \mathsf{dom}(\boldsymbol{R})$ then
$z \xleftarrow{\$} \{0,1\}^n$;
$\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
$\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$;
$\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
$\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
$\boldsymbol{R}[m] \leftarrow (z, \mathbf{q}'_f)$
$\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathit{false})$
$\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$
return $\boldsymbol{R}[m]$

Oracle $F_q(m)$:
$xs \leftarrow \mathsf{pad}(m)$; $y \leftarrow \mathsf{IV}$;
$i \leftarrow 0$;
if $\mathbf{q}_f + |xs| \leq q$ then
$\mathbf{q}_f \leftarrow \mathbf{q}_f + |xs|$;
while $|xs| > 1 \wedge$
$(\mathsf{hd}(xs), y) \in \mathsf{dom}(\boldsymbol{T'})$ do
$i \leftarrow \boldsymbol{T'_i}[\mathsf{hd}(xs), y]$;
$y \leftarrow \boldsymbol{T'}[\mathsf{hd}(xs), y]$;
$xs \leftarrow \mathsf{tl}(xs)$;
while $|xs| > 1 \wedge$
$(\mathsf{hd}(xs), i) \in \mathsf{dom}(\boldsymbol{T})$ do
$i \leftarrow \boldsymbol{T}[\mathsf{hd}(xs), i]$;
$y \leftarrow \mathsf{fst}(\boldsymbol{I}[i])$;
$xs \leftarrow \mathsf{tl}(xs)$;
while $|xs| > 1$ do
$z \xleftarrow{\$} \{0,1\}^n$;
$\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
$\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
$\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
$\boldsymbol{T}[\mathsf{hd}(xs), i] \leftarrow \mathbf{q}'_f$;
$\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathit{true})$;
$i \leftarrow \mathbf{q}'_f$;
$y \leftarrow z$;
$\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$;
$xs \leftarrow \mathsf{tl}(xs)$
$y \leftarrow \mathsf{fst}(F(m, y))$
return y

Oracle $f_q(x, y)$:
if $\mathbf{q}_f + 1 \leq q$ then
if $(x, y) \notin \mathsf{dom}(\boldsymbol{T'})$ then
$xs \leftarrow \mathsf{findseq}((x, y), \boldsymbol{T'})$
if $xs \neq \perp$ then
$m \leftarrow \mathsf{pad}^{-1}(xs \parallel x)$;
$(z, j) \leftarrow F(m, y)$;
$\boldsymbol{T'}[x, y] \leftarrow z$;
$\boldsymbol{T'_i}[x, y] \leftarrow j$;
else
$\mathit{found}, \mathit{found\_b3} \leftarrow \mathit{false}$;
$j, k' \leftarrow 0$;
while $k' < \mathbf{q}'_f$ do
if $\mathsf{snd}(\boldsymbol{I}[k'])$ then
if $\mathsf{fst}(\boldsymbol{I}[k']) = y$ then
$\mathit{found\_b3} \leftarrow \mathit{true}$;
else if $\neg \mathit{found} \wedge$
$\mathsf{fst}(\boldsymbol{I}[k']) = y \wedge$
$(x, k') \in \mathsf{dom}(\boldsymbol{T}) \wedge$
$\mathsf{snd}(\boldsymbol{I}[\boldsymbol{T}[x, k']])$ then
$\mathit{found} \leftarrow \mathit{true}$;
$j \leftarrow \boldsymbol{T}[x, k']$;
$k' \leftarrow k' + 1$;
if found then
$z \leftarrow \mathsf{fst}(\boldsymbol{I}[j])$;
$\boldsymbol{I}[j] \leftarrow (z, \mathit{false})$;
$\boldsymbol{T'}[x, y] \leftarrow z$;
$\boldsymbol{T'_i}[x, y] \leftarrow j$;
else
if $\mathit{found\_b3}$ then
$\mathbf{bad}_3 \leftarrow \mathit{true}$;
$z \xleftarrow{\$} \{0,1\}^n$;
$\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathit{false})$;
$\boldsymbol{T'}[x, y] \leftarrow z$;
$\boldsymbol{T'_i}[x, y] \leftarrow \mathbf{q}'_f$;
$\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$;
else
$z \xleftarrow{\$} \{0,1\}^n$;
$\mathbf{bad}_1 \leftarrow \mathbf{bad}_1 \vee z \in \boldsymbol{Z}$;
$\boldsymbol{Z} \leftarrow z::\boldsymbol{Z}$; $\boldsymbol{Y} \leftarrow y::\boldsymbol{Y}$;
$\mathbf{bad}_2 \leftarrow \mathbf{bad}_2 \vee z \in \boldsymbol{Y}$;
$\boldsymbol{I}[\mathbf{q}'_f] \leftarrow (z, \mathit{false})$;
$\boldsymbol{T'}[x, y] \leftarrow z$;
$\boldsymbol{T'_i}[x, y] \leftarrow \mathbf{q}'_f$;
$\mathbf{q}'_f \leftarrow \mathbf{q}'_f + 1$
$z \leftarrow \boldsymbol{T'}[x, y]$; $\mathbf{q}_f \leftarrow \mathbf{q}_f + 1$
else $z \leftarrow \mathsf{IV}$
return z

Figure 2.17: The game $\mathsf{G}_{\mathrm{realF}}$

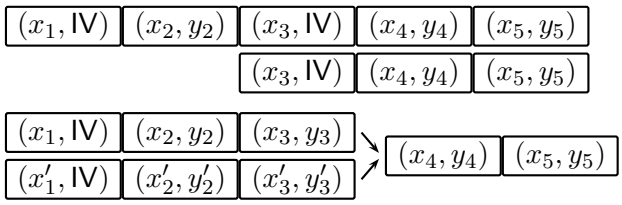


Figure 2.18: Two examples illustrating the necessity of event $\mathbf{bad}_1$

in the domain of $\boldsymbol{T'}$ and (x', y') is not the last element of a complete chain in $\boldsymbol{T}$. Intuitively, this means that y was never returned to the distinguisher by f_q or F_q and hence the distinguisher managed to guess a random value.

In order to relate games $\mathsf{G}_{\mathrm{real}'}$ and $\mathsf{G}_{\mathrm{realF}}$ in case that $\mathsf{findseq}((x, y), \boldsymbol{T'})$ in f_q succeeds in both games, we need to show that the call $f_{\mathbf{bad}}(x, y)$ in $\mathsf{G}_{\mathrm{real}'}$ and the call $F(m, y)$ in $\mathsf{G}_{\mathrm{realF}}$ behave similarly. For this we show that the following invariant is preserved in both games: For all complete chains c in the map $\boldsymbol{T}$ of game $\mathsf{G}_{\mathrm{real}'}$ with $\mathsf{last}(c) \in \mathsf{dom}(T)$, it holds that c's associated message is in $\mathsf{dom}(\boldsymbol{R})$ of game $\mathsf{G}_{\mathrm{realF}}$ and, vice versa, every message in $\mathsf{dom}(\boldsymbol{R})$ of game $\mathsf{G}_{\mathrm{realF}}$ has a corresponding complete chain c in the map $\boldsymbol{T}$ of game $\mathsf{G}_{\mathrm{real}'}$ with $\mathsf{last}(c) \in \mathsf{dom}(T)$. This invariant allows EasyCrypt to prove this case by inferring that $(x, y) \in \mathsf{dom}(\boldsymbol{T})$ in game $\mathsf{G}_{\mathrm{real}'}$ if and only if $m \in \mathsf{dom}(\boldsymbol{R})$ in game $\mathsf{G}_{\mathrm{realF}}$.

Proving that the aforementioned invariant is preserved in the games requires several other invariants. Most of them merely relate the representation of maps in both games; we omit these technical details. The essential invariant is that the distinguisher queries f_q for points in a chain only if it has already queried the preceding part of the chain. This is important as it implies that each chain will be completed by a query for its last element, in which case findseq will detect this query and the corresponding message will be added to $\boldsymbol{R}$. In game $\mathsf{G}_{\mathrm{real}'}$, the predicate set_bad3 enforces this ordering by triggering event $\mathbf{bad}_3$. The probability of this event is negligible, because it means that y was never output by f_q or F_q and hence is not known to the distinguisher. In game $\mathsf{G}_{\mathrm{realF}}$, we use the map $\boldsymbol{I}$ to iterate over all chaining values in order to check for the ordering mentioned above.

In oracle F_q of game $\mathsf{G}_{\mathrm{realF}}$, the computation of the Merkle-Damgård construction is split into three stages due to the different usage of the maps $\boldsymbol{T'}$, $\boldsymbol{T'_i}$, and $\boldsymbol{T}$. The first loop computes the construction for values that were already queried by the distinguisher and are therefore in $\mathsf{dom}(\boldsymbol{T'})$. The restriction that the distinguisher may only query chains in order implies that such values occur only in the prefix of a chain. The second loop handles values that were already used before by oracle F_q, and the third loop samples fresh chaining values. Relating the final call to $f_{\mathbf{bad}}$ in game $\mathsf{G}_{\mathrm{real}'}$ and the

final call to F in game $\mathsf{G}_{\text{realF}}$ is similar to this case in oracle f_q. Overall, we are able to prove in EasyCrypt that the advantage in differentiating between games $\mathsf{G}_{\text{real}'}$ and $\mathsf{G}_{\text{realF}}$ is upper bounded by the probability of any of the events $\mathbf{bad}_1$, $\mathbf{bad}_2$, $\mathbf{bad}_3$ occurring in game $\mathsf{G}_{\text{realF}}$:

$$|\Pr[\mathsf{G}_{\text{real}'} : b] - \Pr[\mathsf{G}_{\text{realF}} : b]| \;\leq\; \Pr[\mathsf{G}_{\text{realF}} : \mathbf{bad}_1 \vee \mathbf{bad}_2 \vee \mathbf{bad}_3]$$

To finish the proof, one has to relate $\Pr[\mathsf{G}_{\text{realF}} : b]$ with $\Pr[\mathsf{G}_{\text{ideal}} : b]$ and bound the probability of the failure events in game $\mathsf{G}_{\text{realF}}$. Focusing on the probability of $\mathbf{bad}_{1,2}$, note that event $\mathbf{bad}_1$ (resp. $\mathbf{bad}_2$) is set when a freshly sampled value z is in the list $\boldsymbol{Z}$ (resp. $\boldsymbol{Y}$). Since the size of both lists is bounded by q, this occurs with probability $\leq q\,2^{-n}$ for each query.

Note that oracles F_q, F, and f_q in game $\mathsf{G}_{\text{realF}}$ use the same code to detect the failure events $\mathbf{bad}_1$ and $\mathbf{bad}_2$ when sampling a fresh value z. By wrapping this code in a new oracle, Lemma 2.3 becomes applicable using the parameters $u = q\,2^{-n}$ and $i = |\boldsymbol{Z}| - 1$ (resp. $|\boldsymbol{Y}|$). This yields

$$\Pr[\mathsf{G}_{\text{realF}} : \mathbf{bad}_1] \leq \frac{q^2}{2^n} \qquad \Pr[\mathsf{G}_{\text{realF}} : \mathbf{bad}_2] \leq \frac{q^2}{2^n}.$$

It remains to bound the probability of $\mathbf{bad}_3$ and to relate the game $\Pr[\mathsf{G}_{\text{realF}} : b]$ with $\Pr[\mathsf{G}_{\text{ideal}} : b]$. On the one hand, note that in game $\mathsf{G}_{\text{realF}}$ chaining values are sampled eagerly, i.e., for a query m, oracle F_q samples chaining values z that are independent of the distinguisher's view (their associated flag is set to *true*). These values might later on become known to the distinguisher if it recomputes the Merkle-Damgård construction for m using oracle f_q (this case is identified by setting $\mathit{found} = \mathit{true}$). On the other hand, in game $\mathsf{G}_{\text{ideal}}$ chaining values are sampled lazily, i.e., not before they are queried by the distinguisher. Hence game $\mathsf{G}_{\text{realF}}$ must be transformed so that chaining values are sampled lazily (as in game $\mathsf{G}_{\text{ideal}}$). The same kind of argument can be used for $\mathbf{bad}_3$. This event is set whenever the distinguisher makes a query (x, y) to f_q with y coinciding with a value uniformly and independently distributed w.r.t. its view.

To prepare for this transition from eagerly to lazily sampled chaining values, game $\mathsf{G}_{\text{realF}}$ is transformed into game $\mathsf{G}_{\text{idealEager}}$ (see Figure 2.19). The body of this game contains a loop that re-samples all chaining values that are unknown to the adversary, i.e., the values for which the second component in the range of map $\boldsymbol{I}$ is set to *true*. Furthermore, game $\mathsf{G}_{\text{idealEager}}$ drops the failure events $\mathbf{bad}_{\{1,2,3\}}$, but introduces a new failure event $\mathbf{bad}_4$. It holds that if $\mathbf{bad}_3$ is triggered in game $\mathsf{G}_{\text{realF}}$, then in $\mathsf{G}_{\text{idealEager}}$ $\mathbf{bad}_4$ is set to *true* or there exists an i such that $\boldsymbol{I}[i] = (v, \mathit{true})$ with $v \in \boldsymbol{Y}$:

$$\begin{aligned} \Pr[\mathsf{G}_{\text{realF}} : b] \;&=\; \Pr[\mathsf{G}_{\text{idealEager}} : b] \\ \Pr[\mathsf{G}_{\text{realF}} : \mathbf{bad}_3] \;&\leq\; \Pr[\mathsf{G}_{\text{idealEager}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists] \end{aligned}$$

where $\mathrm{I}_\exists = \exists i.\ 0 \leq i \leq \mathbf{q}'_f \wedge \mathsf{snd}(\boldsymbol{I}[i]) \wedge \mathsf{fst}(\boldsymbol{I}[i]) \in \boldsymbol{Y}$.

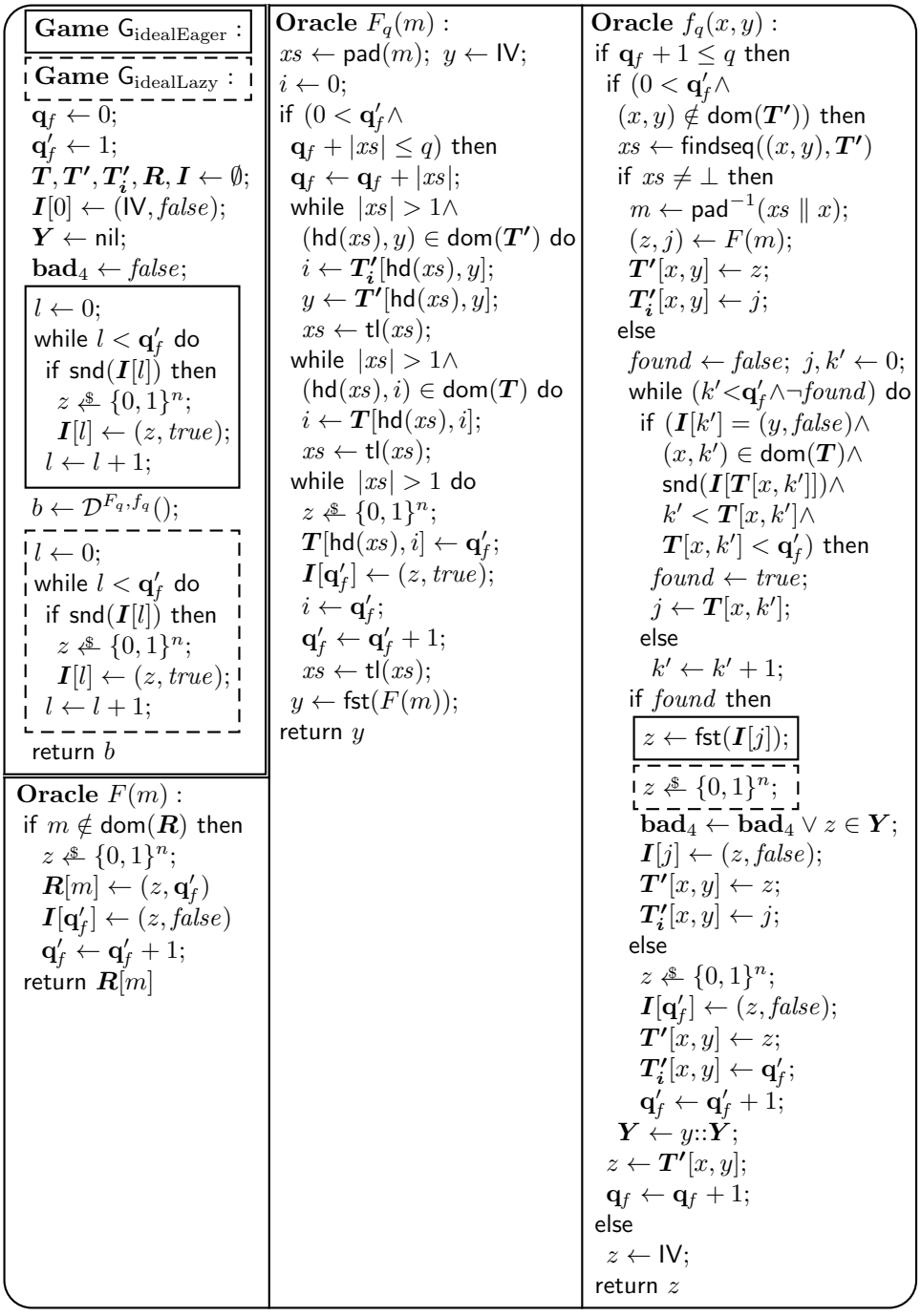


Figure 2.19: The games $\mathsf{G}_{\mathrm{idealEager}}$ (including code in solid frames) and $\mathsf{G}_{\mathrm{idealLazy}}$ (including code in dashed frames)

In game $\mathsf{G}_{\text{idealLazy}}$ (see Figure 2.19), the loop that was introduced in the last game is swapped with the call to the distinguisher and oracle f_q samples the chaining values lazily (the branch *found* re-samples the value of z). As explained in Section 2.3.2, in order to prove the equivalence with the previous game one needs to show that the loop that resamples the values unknown to the adversary can swap with calls to oracles F_q and f_q in games $\mathsf{G}_{\text{idealEager}}$ and $\mathsf{G}_{\text{idealLazy}}$. This yields

$$\begin{aligned} \Pr\left[\mathsf{G}_{\text{idealEager}} : b\right] &= \Pr\left[\mathsf{G}_{\text{idealLazy}} : b\right] \\ \Pr\left[\mathsf{G}_{\text{idealEager}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists\right] &= \Pr\left[\mathsf{G}_{\text{idealLazy}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists\right]. \end{aligned}$$

It is easy to see that games $\mathsf{G}_{\text{idealLazy}}$ and $\mathsf{G}_{\text{ideal}}$ are equivalent w.r.t. b; the global variable $\mathbf{q}_f$ and the maps $\boldsymbol{R}$ and $\boldsymbol{T'}$ are equivalent in both games. The other variables in game $\mathsf{G}_{\text{idealLazy}}$ and its loops do not influence the behavior of its oracles. Namely, it holds

$$\Pr\left[\mathsf{G}_{\text{idealLazy}} : b\right] = \Pr\left[\mathsf{G}_{\text{ideal}} : b\right].$$

It remains to bound the probability of $\mathbf{bad}_4 \vee \mathrm{I}_\exists$ in game $\mathsf{G}_{\text{idealLazy}}$. To do this, the while loop in the code of the game is modified by replacing the instruction $z \xleftarrow{\$} \{0,1\}^n$ with

$$z \xleftarrow{\$} \{0,1\}^n; \mathbf{bad}_4 \leftarrow \mathbf{bad}_4 \vee z \in \boldsymbol{Y}.$$

This leads to a game $\mathsf{G}_{\text{idealLazy}'}$, for which it holds

$$\Pr\left[\mathsf{G}_{\text{idealLazy}} : \mathbf{bad}_4 \vee \mathrm{I}_\exists\right] \leq \Pr\left[\mathsf{G}_{\text{idealLazy}'} : \mathbf{bad}_4\right].$$

Using the same technique as for $\mathbf{bad}_{1,2}$ to bound the probability of $\mathbf{bad}_4$ in game $\mathsf{G}_{\text{idealLazy}'}$ (taking $u = q\,2^{-n}$ and $i = |\boldsymbol{Y}|$) gives

$$\Pr\left[\mathsf{G}_{\text{idealLazy}'} : \mathbf{bad}_4\right] \leq \frac{q^2}{2^n}.$$

Putting the (in-)equalities proved above together proves Inequality (2.15) which completes the proof of Theorem 2.11.

Chapter 3

Discussion

3.1 Related Work on Verification

There are two fundamentally different approaches when analyzing cryptographic protocols: The computational and the symbolic model. Our work resides in the computational model, which deals with the bit-level representation of cryptographic primitives including probabilistic behavior and complexity-theoretic conditions such as polynomial-time adversaries and error probabilities. In contrast to the computational model, the symbolic model uses a term algebra to abstract away from such lower level details of cryptographic primitives and instead employs a deduction system on these terms to model the capabilities of an adversary [73, 75, 97]. These so-called Dolev-Yao models constitute a drastic simplification w.r.t. the computational model. On the one hand, this has lead to a line of work to automate the analysis of such models of cryptographic protocols [99, 95, 86, 90, 106, 119, 1, 47, 38, 11]; see [50] for a survey. On the other hand, the guarantees provided by these approaches are somewhat limited, since they only hold with respect to the simplifications of the symbolic model.

There has been a substantial amount of work on carrying these results over to the computational model for passive and active adversaries, providing abstractions for several cryptographic primitives including encryption, signatures, hash functions, as well as sophisticated objects such as zero-knowledge proofs [2, 88, 24, 89, 98, 21, 68, 58, 118, 25, 15, 19]; see [67] for a survey. Even though such computational soundness proofs can provide security guarantees on the computational level, this process usually restricts the usage of the employed primitives and requires them to satisfy stronger properties than direct proofs in the computational model.

Another approach to provide computational security guarantees is to abstract cryptographic protocols using ideal functionalities for, e.g., secure channels, commitment schemes, zero-knowledge proofs, or multi-party function evaluation [107, 55, 56, 59, 60, 17, 24, 117, 18]. Here, one shows that

each attack on a real protocol can be simulated against ideal functionalities in an indistinguishable manner. Since this kind of abstraction also preserves properties such as liveness, integrity, non-interference, and key secrecy [23, 16, 20, 22], proofs of these properties for ideal functionalities carry over to the real protocol. Nevertheless, the computational security guarantees that one obtains through this process still rely on intricate hand-written simulation proofs.

Our work pursues the more recent approach of verifying security properties directly in the computational model, where the game-playing technique [42, 114] has paved the ground for formal verification techniques. Halevi noticed this possibility and called for the development of a formal verification tool in [83].

The work that comes closest to our Verypto framework is CertiCrypt [29, 32, 124], which is a framework for reasoning about game-based cryptographic proofs in the Coq proof assistant and – like its progeny EasyCrypt [30] – uses the relational Hoare logic on the pWhile language to reason about games. Compared to the language implemented in Verypto, pWhile is not higher-order and only supports discrete measure spaces. Despite these limitations in expressiveness, CertiCrypt has been used to verify several cryptographic results such as the IND-CPA security of the ElGamal encryption scheme [29, 32], the PRP/PRF switching lemma [32, 33], the unforgeability of Full Domain Hash (FDH) signatures [125], zero-knowledge proofs based on Σ-protocols [34], the IND-CCA Security of OAEP [31], semantic security of the Boneh-Franklin identity-based encryption scheme [36], and a construction for indifferentiable hashing into elliptic curves [28]. In addition to our work on the Merkle-Damgård construction, EasyCrypt has been used to verify the IND-CPA security of the (hashed) ElGamal encryption scheme [30, 26], the IND-CCA security of the Cramer-Shoup encryption scheme [30], and the IND-CCA security of a novel variation of OAEP [37]. Other extensions of CertiCrypt have been developed to reason about differential privacy [35] and zero-knowledge proofs of knowledge [4].

CryptoVerif [48] is another tool that supports game-based cryptographic proofs. It has been used to assert the unforgeability of the FDH signature scheme [52], the security of the Kerberos protocol [51], and the OEKE key-exchange protocol [49]. CryptoVerif differs substantially from our work in that it aims at generating the sequence of games itself, based on a collection of manually proven transformations, while we aim at checking whether a sequence of games provided by the user is computationally sound. These approaches should thus be considered complementary instead of competing.

A non-relational probabilistic Hoare logic to reason about game-based proofs is presented in [65]. While this logic is used to infer the security of the ElGamal encryption, its formalism is not powerful enough to fully express the security properties of interest, e.g., notions such as negligible probability or polynomial-time adversaries cannot be expressed.

In [85], Impagliazzo and Kapron prove the soundness of a logic for computational indistinguishability. They use it to prove properties of pseudorandom generators, but cannot handle adaptive adversaries. The same holds for [126], where Zhang uses a probabilistic polynomial-time calculus to reason about computational indistinguishability.

In another approach, Nowak establishes the security of ElGamal in [103] and gives proofs for the Blum-Blum-Shub pseudorandom generator and the Goldwasser-Micali scheme in [104]. For this, he models adversaries immediately as Coq functions and hence is only capable of offering limited support for proof automation because there is no special syntax for writing games that could be conveniently used to mechanize syntactic transformations. Moreover, the formalism ignores complexity-theoretic issues such as defining a polynomial-time adversary. This issue is addressed in [105] by using the calculus of Zhang [126], but without providing tool support. In [3], Affeldt et al. give a game-based proof of the PRP/PRF switching lemma in Coq. Their formalism is custom-made for this example which impedes a general application of their framework.

In [69], Courant et al. develop a specialized Hoare logic to infer IND-CPA security in the random oracle model and implement it in a tool. Then they use a syntactic criterion to further deduce the IND-CCA security of encryption schemes.

The Computational Indistinguishability Logic (CIL) of [27] uses a system of oracles to model cryptographic games with adaptive adversaries and supports reasoning based on simulations and reductions. CIL has been used to prove unforgeability for the Probabilistic Signature Scheme (PSS) as well as FDH, and the IND-CCA security of OAEP. A Coq formalization of CIL appears in [64] and an extension of CIL has been used to manually prove the indifferentiability of hash designs [71].

3.2 SHA-3 and Related Work on Hash Security

Despite their widespread use, the formal verification of hash functions has received little attention. To our best knowledge, Toma and Borrione [120] were the first to use theorem provers to formally verify properties of SHA-1, but their focus is on functional properties, rather than security properties. Mironov and Zhang [100] use a SAT solver to construct collisions for MD4 and MD5. The first machine-checked proof of security for a hash design appears in [28], where the authors use the CertiCrypt framework to verify that the construction from Brier et al. [54] yields a hash function indifferentiable from a random oracle into ordinary elliptic curves. In [71] Daubignard et al. develop an extension of CIL to permute dependencies between oracles in a game, and apply their method to prove hash designs indifferentiable from random oracles. Their method is not implemented, although the underlying

framework has been machine-checked [64].

In response to the discovery of differential collision attacks and weaknesses for several hash algorithms including MD5 [122] and SHA-1 [121, 92], the U.S. National Institute of Standards and Technology (NIST) launched the so-called SHA-3 competition in November 2007 with the objective of selecting a new cryptographic hash function to augment the set specified by the U.S. Federal Information Processing Standard (FIPS) 180-4, which includes the SHA-1 and SHA-2 algorithms. After receiving 64 entries, NIST selected 51 candidates for the first round, further narrowed down the list to just 14 candidates for the second round, and announced 5 finalists in December 2010: BLAKE [12], Grøstl [80], JH [123], Keccak [43], and Skein [76]. Out of these Keccak was announced as the winner of the competition in October 2012 and is henceforth the new SHA-3 cryptographic hash algorithm.

Due to the growing interest in cryptographic hash functions during the course of the competition, the security of all SHA-3 finalists, and of many second round candidates, has been thoroughly scrutinized. Two survey articles summarize known results [7, 9]. Even though the algorithmic descriptions of the finalists and their exact security bounds fit in one page (see [9]), the corresponding security proofs are technically involved and need to be cautiously adapted to account for the specificities of each function. As a consequence, it is difficult to assess the validity of security claims for individual candidates and machine checking their proofs is an appealing perspective, which also motivated our work on the Merkle-Damgård construction in Chapter 2. Therefore, in the remainder of this section we discuss the applicability of the proofs presented in Sections 2.5 and 2.6 to the SHA-3 finalists.

To avoid inheriting structural weaknesses in the original Merkle-Damgård construction, many hash functions employ instead slight variants of it. One well-known variant is the wide-pipe design, which uses an internal state larger than the final output [91, 66]. Also the five SHA-3 finalists are based on the iterated design that underlies the Merkle-Damgård construction, but incorporate some variations such as round-dependent tweaks, counters, final transformations, and chopping. We observe that, in a more or less contrived way, all the finalists can be considered as variants of the *generalized Merkle-Damgård* construction which, as a final transformation, uses a separate compression function to process the last message block and chops off a number of least significant bits before producing the final output.

The compression functions of the finalists are either block-cipher based (BLAKE, Skein) or permutation-based (Grøstl, JH, Keccak). Moreover, all finalists use suffix-free padding rules, while the padding rules of BLAKE and Skein are additionally prefix-free [9].

Our formalization models compression functions as functions of two arguments: a message block and a chaining value. This represents a deviation with respect to the compression functions of BLAKE and Skein. The compression function of BLAKE additionally takes a counter and a random

salt value, whereas the compression function of Skein builds on a *tweakable* block cipher and takes as additional input a round-specific tweak. The additional arguments of the compression functions of BLAKE and Skein can be formalized as an integral part of the padding rule to match the modeling of compression functions that we use in our results about the MD hash function: The padding function can compute the appropriate round-specific values and append them to the message blocks. However, all finalists except BLAKE use chopping or a final transformation, which are formalized neither in our proof of collision resistance nor in our proof of indifferentiability. This rules out a direct application of our results, with the exception of BLAKE, for which the Theorem 2.7 on collision resistance does apply.

NIST requirements for the SHA-3 competition included collision resistance, preimage resistance and second preimage resistance. All the candidates selected as finalists satisfy these properties and (in most cases) even achieve optimal bounds for them when the underlying block-ciphers or permutations used to build their compression functions are assumed to be ideal [9]. Although the original NIST requirements did not include the property of indifferentiability from a random oracle, this notion has also been considered in the literature and is achieved by all five finalists [5, 62, 6, 45, 44, 40]. Generalizations of the sponge construction that underlies Keccak and JH have been shown to be indifferentiable [8] as well as sponges restricted to single round extraction phases [61], which includes the standardized variants of SHA-3.

These indifferentiability proofs hold in an idealized model for some of the building blocks of the hash function – the ideal-cipher model for block-cipher based hash functions, or the ideal-permutation model for permutation-based hash functions. Indifferentiability seems to be an excellent target for security proofs because it ensures that the high-level design of the hash function has no structural weaknesses, but also because it implies bounds for all of the classical properties enumerated above. Unfortunately, the assumption that some underlying primitive is ideal is at best unrealistic and at worst plainly wrong. Proofs of indifferentiability should be taken only as an indication for the security and as a palliative for the lack of proofs in the standard model.

Compared to our result of Theorem 2.11, which assumes that the compression function is ideal, the indifferentiability of all the finalists has been proved in an ideal model for lower building blocks. We point out that assuming ideality of a lower building block is weaker than assuming ideality of the entire compression function and thus these results are stronger. Indeed, assuming ideality of the entire compression function seems to be inappropriate for all the finalists:

- The compression functions of Keccak and JH are trivially non-random, as collisions and preimages can be found in only one query to the underlying permutation [46, 9].

- Finding fixed-points for the compression function of Grøstl is trivial [80].
- The compression function of BLAKE has been shown to exhibit non-random behavior [5, 62].
- Non-randomness has been shown for reduced-round versions of Threefish, the block-cipher underlying Skein [87].

The only two finalists that use a prefix-free padding rule, and for which our indifferentiability proof can apply, are BLAKE and Skein. However, our proof of indifferentiability of prefix-free Merkle-Damgård relies on the assumption that the underlying compression function behaves like an ideal primitive. Thus, it cannot be applied to BLAKE, as this assumption has been invalidated; for Skein, the assumption that its compression function is ideal is seriously weakened by the attacks on Threefish mentioned above.

3.3 Conclusion

In Chapter 1 we conceptually designed Verypto and presented its implementation in Isabelle/HOL. For this, we developed a probabilistic higher-order functional language with recursive types, references, and events that is able to express the constructs that typically occur in cryptographic proofs. Its semantics is expressive but yet simple enough to be understandable without a strong background in the theory of programming languages. Using a collection of verified game transformations, Verypto is capable of verifying game-based cryptographic security proofs in a machine-assisted manner, which we demonstrated in two example proofs. Namely, we showed that the self-composition of a 1-1 one-way function is also one-way, and we verified the IND-CPA security of the ElGamal encryption scheme.

The prevailing method for building hash functions is to iterate a compression function on a pre-processed input message. In Chapter 2 we have considered the Merkle-Damgård construction, which pioneered this design, and verified that the resulting hash function preserves collision resistance and is indifferentiable from a random oracle. Our work demonstrates that state-of-the-art verification tools can be used for proving the security of hash designs. Furthermore, our work constitutes a non-trivial result about the Merkle-Damgård construction and is a good starting point for formalizing more general security proofs that apply to a wider range of hash functions, including the new SHA-3 algorithm Keccak. Indeed, indifferentiability proofs based on weaker assumptions that apply to SHA-3 as in [8, 61] are not significantly different from the proof we have formalized and use essentially the same techniques. We see no impediment to formalizing them in EasyCrypt.

Appendix I

Formalization of **Verypto** in Isabelle/HOL

I.1 Probability theory

(1) class measurable_space =
  fixes Σ :: "'a set set"
  assumes sigma [simp]: "sigma_algebra Σ"

Section 1.5.2 page 9

(2) typedef 'a measurableset = "Σ::'a::measurable_space set set"

(3) class discrete_measurable_space = measurable_space +
  assumes univ: "Σ = UNIV"

(4) class countable_measurable_space = discrete_measurable_space + countable

instantiation bool :: measurable_space
(5) definition sigma_bool_def[simp]: "(Σ::bool set set) == UNIV"

instantiation bool :: countable_measurable_space

instantiation nat :: measurable_space
(6) definition sigma_nat_def[simp]: "(Σ::nat set set) == UNIV"

instantiation nat :: countable_measurable_space

instantiation unit :: measurable_space
(7) definition sigma_unit_def[simp]: "(Σ::unit set set) == UNIV"

instantiation unit :: countable_measurable_space

instantiation real :: measurable_space
(8) definition sigma_real_def: "(Σ::real set set) == 𝔹"

Section 1.5.2 page 9

instantiation "*" :: (measurable_space, measurable_space) measurable_space
(9) definition sigma_prod_def:
  "(Σ::(('a::measurable_space)×('b::measurable_space)) set set)
  == sigma {a×b | a b. a∈Σ ∧ b∈Σ}"

Section 1.5.2 page 9

instantiation "+" :: (measurable_space, measurable_space) measurable_space

(10) definition sigma_sum_def:
"(Σ::((’a::measurable_space)+(’b::measurable_space)) set set)
== sigma {a<+>b | a b. a∈Σ ∧ b∈Σ}"

instantiation "list" :: (measurable_space) measurable_space
(11) definition sigma_list_def:
"(Σ::((’a::measurable_space)list) set set)
== sigma {listset sl | sl::’a set list. set sl ⊆ Σ}"

instantiation "fun" :: (type,measurable_space) measurable_space
Section 1.5.2 page 9 **(12) definition** sigma_fun_def:
"(Σ::(’a⇒(’b::measurable_space)) set set)
== sigma {{f. f x : B} | B x. B : Σ}"

(13) definition
subprobability_space:: "(’a set set * (’a set ⇒ real)) ⇒ bool" **where**
"subprobability_space M ≡ measure_space M ∧ probability M UNIV ≤ 1"

Section 1.5.2 page 9 **(14) definition**
"is_sub_pr_measure M ==
subprobability_space (Σ::(’a::measurable_space) set set, M)"

Section 1.5.2 page 10 **(15) definition**
"is_measurable (f::(’a::measurable_space)⇒(’b::measurable_space)) ==
f ∈ measurable Σ Σ"

(16) lemma fst_measurable: "is_measurable fst"

(17) lemma snd_measurable: "is_measurable snd"

(18) typedef ’a subprobability =
"{μ::(’a::measurable_space) set ⇒ real. is_sub_pr_measure μ ∧ (∀E∈(-Σ).μ(E)=0)}"

(19) definition
"probability_of (μ::’a::measurable_space subprobability) M
== glb {Rep_subprobability μ A | A. A:Σ & A⊇M}"

(20) definition
"mk_subprobability f == Abs_subprobability (λE.(if E∈Σ then f E else 0))"

instantiation subprobability :: (measurable_space)zero
(21) definition "0 = mk_subprobability (λE. 0)"

Section 1.5.2 page 10 **(22) definition**
"project_measure f (μ::_ subprobability) ==
mk_subprobability (λA∈Σ. probability_of μ(f-‘A))"

(23) lemma probability_of_mono:
assumes "A⊆B"
shows "probability_of μ A ≤ probability_of μ B"

instantiation subprobability :: (measurable_space)order
Section 1.5.2 page 10 **(24) definition**
le_subprobability_def: "μ ≤ ν == probability_of μ ≤ probability_of ν"
(25) definition
less_subprobability_def: "(μ::’a subprobability) < ν ≡ μ ≤ ν ∧ ¬ μ ≥ ν"

Section 1.5.2 page 10 **(26) definition**
SPall_def: "SPall μ P == (∃S∈Σ. probability_of μ (-S) = 0 ∧ (∀x∈S. P x))"

syntax (xsymbols)
"_SPall" :: "pttrn => (’a::measurable_space subprobability) => bool => bool"

("(3∀ _←_./ _)")

(27) lemma SPall_True[simp]: "∀x←μ. True"

(28) lemma probability_of_project_measure:
assumes "S∈Σ" **and** "is_measurable f"
shows "probability_of (project_measure f A) S = probability_of A (f -‘ S)"

(29) lemma project_measure_comp:
assumes "is_measurable f"
assumes "is_measurable g"
shows "project_measure f (project_measure g μ) = project_measure (f o g) μ"

(30) lemma SPall_simps[simp]:
"!!A P Q. (ALL x<-A. P x | Q) = ((ALL x<-A. P x) | Q)"
"!!A P Q. (ALL x<-A. P | Q x) = (P | (ALL x<-A. Q x))"
"!!A P Q. (ALL x<-A. P --> Q x) = (P --> (ALL x<-A. Q x))"
"!!A P Q. (ALL x<-A. P x --> Q) = ((~(ALL x<-A. ~(P x))) --> Q)"
"!!P. (ALL x<-0. P x) = True"
"!!A P f. ⟦is_measurable f; ALL x<-project_measure f A. P x⟧ ⟹
(ALL x<-A. P (f x))"
"!!A P f. ⟦is_measurable f; is_measurable P⟧ ⟹
(ALL x<-project_measure f A. P x) = (ALL x<-A. P (f x))"
"!!A P f. ⟦is_measurable f; ∀S∈Σ. f ‘ S ∈ Σ⟧ ⟹
(ALL x<-project_measure f A. P x) = (ALL x<-A. P (f x))"

(31) typedef ('a,'b) submarkov_kernel = "{K. ∀B∈Σ. is_measurable (λw::'a::measurable_space. probability_of (K w) (B::'b::measurable_space set))}" Section 1.5.2 page 10

(32) definition
"apply_kernel == Rep_submarkov_kernel"

(33) definition
"kernel_prob_of K x == probability_of (apply_kernel K x)"

(34) definition
"mk_kernel == Abs_submarkov_kernel"

(35) definition
"mk_kernel2 (f::('a::measurable_space,'b::measurable_space) kernelT) =
mk_kernel (λx. mk_subprobability (f x))"

instantiation submarkov_kernel :: (measurable_space,measurable_space)order
(36) definition le_submarkov_kernel_def: Section 1.5.2 page 10
"K ≤ L == apply_kernel K ≤ apply_kernel L"
(37) definition less_submarkov_kernel_def:
"(K::('a,'b)submarkov_kernel) < L ≡ K ≤ L ∧ ¬ K ≥ L"

(38) definition Section 1.5.2 page 10
"unitkernel == mk_kernel2 (λx E. if x:E then 1 else 0)"

(39) definition
"constant_dist_kernel μ == mk_kernel (λx. μ)"

(40) definition
"constant_kernel c == constant_dist_kernel (apply_kernel unitkernel c)"

(41) definition Section 1.5.2 page 10
"restriction_kernel A ==
mk_kernel (λx. if x:A then apply_kernel unitkernel x else 0)"

(42) definition Section 1.5.2 page 10
integral2:: "(('a::measurable_space) ⇒ real) ⇒ ('a subprobability) ⇒ real"
("∫ _ ∂∂_") **where**

```
"∫ f ∂∂ μ == ∫ f ∂(Σ,probability_of μ)"
```

Section 1.5.2 page 10

(43) definition
```
"lift_kernel K == λμ. mk_subprobability (λA. ∫ (λw. kernel_prob_of K w A) ∂∂ μ)"
```

Section 1.5.2 page 10

(44) definition
```
compose_kernel :: "('b::measurable_space,'c::measurable_space) submarkov_kernel
⇒ ('a::measurable_space,'b) submarkov_kernel ⇒ ('a,'c) submarkov_kernel"
(infixl "oo" 55)
where
"K oo L == mk_kernel ((lift_kernel K) ∘ (apply_kernel L))"
```

(45) lemma mono_compose_kernel1:
```
assumes "K1 ≤ K2"
shows "compose_kernel K1 L ≤ compose_kernel K2 L"
```

(46) lemma mono_compose_kernel2:
```
assumes "L1 ≤ L2"
shows "compose_kernel K L1 ≤ compose_kernel K L2"
```

(47) lemma compose_kernel_assoc:
```
"K oo (L oo M) == (K oo L) oo M"
```

(48) definition
```
"uniform_distribution == λM.
if finite M ∧ M≠{} then (λE. real_of_nat(card(E∩M)) / real_of_nat(card M))
else (λx. 0)"
```

(49) lemma SPall_unitkernel[intro]:
```
assumes xS: "{x}∈Σ"
and Px: "P x"
shows "∀y←(apply_kernel unitkernel x). P y"
```

(50) lemma compose_kernel_unitkernel_left[simp]:
```
"unitkernel oo K = K"
```

(51) lemma compose_kernel_unitkernel_right[simp]:

```
  shows "K oo unitkernel = K"
```

Section 1.5.2 page 10

(52) lemma probability_of_restriction_kernel:
```
assumes "A∈Σ"
assumes "S∈Σ"
shows "probability_of (lift_kernel (restriction_kernel S) m) A
     = probability_of m (A∩S)"
```

(53) lemma Inl_measurable: "is_measurable Inl"

(54) lemma Inr_measurable: "is_measurable Inr"

Section 1.5.2 page 10

(55) definition "deterministic_kernel f ==
```
mk_kernel2 (λx E. if f x : E then 1 else 0)"
```

(56) lemma deterministic_kernel_prob:
```
"is_measurable f ⟹
apply_kernel (deterministic_kernel f) x = apply_kernel unitkernel (f x)"
```

(57) lemma lift_deterministic_kernel:
```
fixes f :: "('a::measurable_space)⇒('b::singleton_measurable_space)"
assumes measurablef: "is_measurable f"
shows "lift_kernel (deterministic_kernel f) = project_measure f"
```

(58) lemma project_unitkernel:
```
assumes "is_measurable f"
```

```
  shows "project_measure f (apply_kernel unitkernel x) =
         apply_kernel unitkernel (f x)"
```

(59) lemma kernel_lift_compose_comm:
```
  fixes K::"('b::measurable_space,'c::measurable_space)submarkov_kernel"
  and L::"('a::measurable_space,'b)submarkov_kernel"
  shows "lift_kernel (K oo L) = (lift_kernel K) o (lift_kernel L)"
```

(60) lemma apply_compose_kernel:
```
  "apply_kernel (A oo B) x = lift_kernel A (apply_kernel B x)"
```

(61) definition "kernel_limit V K == mk_kernel2 (λps m. lim (λn. kernel_prob_of ((restriction_kernel V)oo(foldr compose_kernel (replicate n K) unitkernel)) ps m))" — Definition 1.3 page 11

(62) theorem double_kernel_limit: — Theorem 1.9 page 13
```
  fixes K :: "('a::singleton_measurable_space,'a)submarkov_kernel"
    and L :: "('a,'a)submarkov_kernel"
    and U::"'a set" and V::"'a set" and DK::"'a set"
  assumes "U:Σ" "V:Σ" "DK:Σ"
  and "U⊆DK"
  and "V∩DK={}"
  and "∀u∈U. apply_kernel K u = apply_kernel unitkernel u"
  and "∀v∈V. apply_kernel L v = apply_kernel unitkernel v"
  and "∀x. kernel_prob_of K x (-DK) = 0"
  and "∀x. kernel_prob_of L x DK = 0"
  and "∀x∈(DK-U). apply_kernel L x = 0"
  and "∀x∈(-DK). apply_kernel K x = 0"
  defines "M == mk_kernel (λx. if x:DK-U then apply_kernel K x else apply_kernel L x)"
  shows "∀x∈DK. apply_kernel (kernel_limit V M) x
       = apply_kernel (kernel_limit V L oo kernel_limit U K) x"
```

I.2 Program Terms

I.2.1 Basic Values, Program Terms, and (Pure) Values

(63) datatype basicvalue = — Section 1.6 page 14
```
  bvBool bool
| bvUnit
| bvReal real
| bvChar char
| bvTape "nat ⇒ bool"
```

(64) types eventT = string

(65) types bitstring = "bool list"

(66) datatype pureterm =
```
  ptValue basicvalue
| ptPair pureterm pureterm
| ptInl pureterm
| ptInr pureterm
| ptFold pureterm
```

(67) datatype programterm = — Definition 1.10 page 14
```
   Var nat
|  Value basicvalue
|  Function "(pureterm, pureterm) submarkov_kernel" programterm
|  PairP programterm programterm
|  Abstraction programterm
|  Application programterm programterm
|  Location nat
|  Ref programterm
|  Deref programterm
```

```
| Assign programterm programterm
| Event eventT
| EventList
| Fst programterm
| Snd programterm
| Fold programterm
| Unfold programterm
| InlP programterm
| InrP programterm
| CaseP programterm programterm programterm
```

Definition 1.10 page 14

```
(68) inductive_set purevalues :: "programterm set" where
  "Value v : purevalues"
| "⟦a : purevalues; b : purevalues⟧ ⟹ PairP a b : purevalues"
| "a : purevalues ⟹ InlP a : purevalues"
| "a : purevalues ⟹ InrP a : purevalues"
| "a : purevalues ⟹ Fold a : purevalues"
```

Definition 1.10 page 14

```
(69) inductive_set "values" :: "programterm set" where
  values_Value: "Value v : values"
| values_PairP: "p1 : values ∧ p2 : values ⟹ PairP p1 p2 : values"
| values_Abstraction: "Abstraction p : values"
| values_Location: "Location n : values"
| values_Var: "Var n : values"
| values_Fold: "p : values ⟹ Fold p : values"
| values_InlP: "p : values ⟹ InlP p : values"
| values_InrP: "p : values ⟹ InrP p : values"
```

```
(70) types store = "programterm list"
```

```
(71) types state = "store × eventT list"
```

Section 1.8 page 23

```
(72) definition "Nil'P = Fold (InlP (Value bvUnit))"
```

Section 1.8 page 23

```
(73) definition "List'P x xs = Fold (InrP (PairP x xs))"
```

Section 1.8 page 23

```
(74) definition "value_unit = Value bvUnit"
(75) definition "value_true = InlP value_unit"
(76) definition "value_false = InrP value_unit"
```

Section 1.8 page 23

```
(77) definition LetP :: "programterm ⇒ programterm ⇒ programterm" where
  "LetP p1 p2 == Application p2 p1"
```

```
(78) definition "omegaf = Abstraction( Fold(Abstraction (Application
  (Var(Suc 0)) (Abstraction (Application
  (Application (Unfold (Var(Suc 0))) (Var(Suc 0))) (Var 0))))))"
```

Section 1.8 page 26

```
(79) definition "Fix = Abstraction (Application
  (Unfold (Application omegaf (Var 0))) (Application omegaf (Var 0)))"
```

```
(80) definition SequenceP :: "programterm ⇒ programterm ⇒ programterm" where
  "SequenceP p1 p2 ==
  Application (Application (Abstraction (Abstraction (Var 0))) p1) p2"
```

```
(81) definition
  "uncurry p ==
  Application (Abstraction (Abstraction (Application (Application
  (Var (Suc 0)) (Fst (Var 0))) (Snd (Var 0))))) p"
```

I.2.2 Function Definitions

(82) function pureterm_to_term :: "pureterm ⇒ programterm" **where**
```
  "pureterm_to_term (ptValue v)  = Value v"
| "pureterm_to_term (ptPair x y) =
     PairP (pureterm_to_term x) (pureterm_to_term y)"
| "pureterm_to_term (ptInl x)    = InlP (pureterm_to_term x)"
| "pureterm_to_term (ptInr x)    = InrP (pureterm_to_term x)"
| "pureterm_to_term (ptFold x)   = Fold (pureterm_to_term x)"
```

(83) fun term_to_pureterm :: "programterm ⇒ pureterm" **where**
```
  "term_to_pureterm (Value v)   = ptValue v"
| "term_to_pureterm (PairP x y) =
     ptPair (term_to_pureterm x) (term_to_pureterm y)"
| "term_to_pureterm (InlP x)    = ptInl (term_to_pureterm x)"
| "term_to_pureterm (InrP x)    = ptInr (term_to_pureterm x)"
| "term_to_pureterm (Fold x)    = ptFold (term_to_pureterm x)"
```

(84) fun is_value :: "programterm ⇒ bool" **where**
```
  "is_value (Value _)       = True"
| "is_value (PairP p1 p2)   = ((is_value p1) ∧ (is_value p2))"
| "is_value (Abstraction _) = True"
| "is_value (Location _)    = True"
| "is_value (Var _)         = True"
| "is_value (Fold p)        = is_value p"
| "is_value (InlP p)        = is_value p"
| "is_value (InrP p)        = is_value p"
| "is_value _               = False"
```

(85) definition "are_values l = (set l ⊆ values)"

(86) function lift_vars :: "nat ⇒ programterm ⇒ programterm" **where** Section 1.6.1 page 17
```
  "lift_vars k (Var i)           = (if i < k then Var i else Var (Suc i))"
| "lift_vars k (Abstraction p)   = Abstraction (lift_vars (Suc k) p)"
| "lift_vars k (Application p q) = Application (lift_vars k p) (lift_vars k q)"
| "lift_vars k (Value v)         = Value v"
| "lift_vars k (Function f p)    = Function f (lift_vars k p)"
| "lift_vars k (PairP p1 p2)     = PairP (lift_vars k p1) (lift_vars k p2)"
| "lift_vars k (Location n)      = Location n"
| "lift_vars k (Ref p)           = Ref (lift_vars k p)"
| "lift_vars k (Deref p)         = Deref (lift_vars k p)"
| "lift_vars k (Assign p1 p2)    = Assign (lift_vars k p1) (lift_vars k p2)"
| "lift_vars k (Event e)         = Event e"
| "lift_vars k EventList         = EventList"
| "lift_vars k (Fst p)           = Fst (lift_vars k p)"
| "lift_vars k (Snd p)           = Snd (lift_vars k p)"
| "lift_vars k (CaseP p1 p2 p3)  =
    CaseP (lift_vars k p1) (lift_vars k p2) (lift_vars k p3)"
| "lift_vars k (InlP p)          = InlP (lift_vars k p)"
| "lift_vars k (InrP p)          = InrP (lift_vars k p)"
| "lift_vars k (Fold p)          = Fold (lift_vars k p)"
| "lift_vars k (Unfold p)        = Unfold (lift_vars k p)"
```

(87) definition IfThenElse' :: Section 1.8 page 23
```
  "programterm ⇒ programterm ⇒ programterm ⇒ programterm" where
  "IfThenElse' p1 p2 p3 == CaseP p1 (Abstraction (lift_vars 0 p2))
                                    (Abstraction (lift_vars 0 p3))"
```

(88) function substitute' :: "[nat, programterm, programterm] ⇒ programterm" **where** Section 1.6.1 page 19
```
  "substitute' k (Var i) p           =
    (if k < i then Var (i - 1) else if i=k then p else Var i)"
| "substitute' k (Abstraction p1) p  =
    Abstraction (substitute' (Suc k) p1 (lift_vars 0 p))"
| "substitute' k (Application s t) p =
    Application (substitute' k s p) (substitute' k t p)"
```

```
| "substitute' k (Value v) p         = Value v"
| "substitute' k (Function f p1) p   = Function f (substitute' k p1 p)"
| "substitute' k (PairP p1 p2) p     =
    PairP (substitute' k p1 p) (substitute' k p2 p)"
| "substitute' k (Location n) p      = Location n"
| "substitute' k (Ref p1) p          = Ref (substitute' k p1 p)"
| "substitute' k (Deref p1) p        = Deref (substitute' k p1 p)"
| "substitute' k (Assign p1 p2) p    =
    Assign (substitute' k p1 p) (substitute' k p2 p)"
| "substitute' k (Event e) p         = Event e"
| "substitute' k EventList p         = EventList"
| "substitute' k (Fst p1) p          = Fst (substitute' k p1 p)"
| "substitute' k (Snd p1) p          = Snd (substitute' k p1 p)"
| "substitute' k (CaseP p1 p2 p3) p  =
    CaseP (substitute' k p1 p) (substitute' k p2 p) (substitute' k p3 p)"
| "substitute' k (InlP p1) p         = InlP (substitute' k p1 p)"
| "substitute' k (InrP p1) p         = InrP (substitute' k p1 p)"
| "substitute' k (Fold p1) p         = Fold (substitute' k p1 p)"
| "substitute' k (Unfold p1) p       = Unfold (substitute' k p1 p)"
```

(89) function `swap_vars :: "[nat, nat, programterm] ⇒ programterm"` **where**

```
  "swap_vars n m (Var i)           =
    (if i=n then Var m else if i=m then Var n else Var i)"
| "swap_vars n m (Abstraction p1)  = Abstraction (swap_vars (Suc n) (Suc m) p1)"
| "swap_vars n m (Application s t) = Application (swap_vars n m s) (swap_vars n m t)"
| "swap_vars n m (Value v)         = Value v"
| "swap_vars n m (Function f p1)   = Function f (swap_vars n m p1)"
| "swap_vars n m (PairP p1 p2)     = PairP (swap_vars n m p1) (swap_vars n m p2)"
| "swap_vars n m (Location l)      = Location l"
| "swap_vars n m (Ref p1)          = Ref (swap_vars n m p1)"
| "swap_vars n m (Deref p1)        = Deref (swap_vars n m p1)"
| "swap_vars n m (Assign p1 p2)    = Assign (swap_vars n m p1) (swap_vars n m p2)"
| "swap_vars n m (Event e)         = Event e"
| "swap_vars n m EventList         = EventList"
| "swap_vars n m (Fst p1)          = Fst (swap_vars n m p1)"
| "swap_vars n m (Snd p1)          = Snd (swap_vars n m p1)"
| "swap_vars n m (CaseP p1 p2 p3)  =
    CaseP (swap_vars n m p1) (swap_vars n m p2) (swap_vars n m p3)"
| "swap_vars n m (InlP p1)         = InlP (swap_vars n m p1)"
| "swap_vars n m (InrP p1)         = InrP (swap_vars n m p1)"
| "swap_vars n m (Fold p1)         = Fold (swap_vars n m p1)"
| "swap_vars n m (Unfold p1)       = Unfold (swap_vars n m p1)"
```

Section 1.12.5 page 65

(90) lemma `swap_vars_def2: "swap_vars n (Suc n) P = substitute' (Suc (Suc n)) (lift_vars n P) (Var n)"`

(91) function `locations_of :: "programterm ⇒ nat set"` **where**

```
  "locations_of (Location n)        = {n}"
| "locations_of (Var _)             = {}"
| "locations_of (Value _)           = {}"
| "locations_of (Event _)           = {}"
| "locations_of (EventList)         = {}"
| "locations_of (Function _ p)      = locations_of p"
| "locations_of (Abstraction p)     = locations_of p"
| "locations_of (Ref p)             = locations_of p"
| "locations_of (Deref p)           = locations_of p"
| "locations_of (Fst p)             = locations_of p"
| "locations_of (Snd p)             = locations_of p"
| "locations_of (InlP p)            = locations_of p"
| "locations_of (InrP p)            = locations_of p"
| "locations_of (Fold p)            = locations_of p"
| "locations_of (Unfold p)          = locations_of p"
| "locations_of (PairP p1 p2)       = (locations_of p1) ∪ (locations_of p2)"
| "locations_of (Application p1 p2) = (locations_of p1) ∪ (locations_of p2)"
| "locations_of (Assign p1 p2)      = (locations_of p1) ∪ (locations_of p2)"
```

```
| "locations_of (CaseP p1 p2 p3)    =
    (locations_of p1) ∪ (locations_of p2) ∪ (locations_of p3)"
```

(92) function freevars :: "programterm ⇒ nat set" **where** Section 1.6.1 page 17

```
  "freevars (Var i)             = {i}"
| "freevars (Abstraction s)     = {n - 1 | n. n : freevars s ∧ n > 0}"
| "freevars (Value v)           = {}"
| "freevars (Location n)        = {}"
| "freevars (Event e)           = {}"
| "freevars EventList           = {}"
| "freevars (Ref p)             = freevars p"
| "freevars (Function f p)      = freevars p"
| "freevars (Deref p)           = freevars p"
| "freevars (Fst p)             = freevars p"
| "freevars (Snd p)             = freevars p"
| "freevars (InlP p)            = freevars p"
| "freevars (InrP p)            = freevars p"
| "freevars (Fold p)            = freevars p"
| "freevars (Unfold p)          = freevars p"
| "freevars (Application p1 p2) = freevars p1 ∪ freevars p2"
| "freevars (PairP p1 p2)       = freevars p1 ∪ freevars p2"
| "freevars (Assign p1 p2)      = freevars p1 ∪ freevars p2"
| "freevars (CaseP p1 p2 p3)    = freevars p1 ∪ freevars p2 ∪ freevars p3"
```

(93) definition storageclosed :: "programterm × state ⇒ bool" **where** Section 1.8 page 26

```
  "storageclosed == λ(p,σ,η). (∀n∈(locations_of p). n<length σ) ∧
                              (∀p'∈set σ. ∀n∈(locations_of p'). n<length σ)"
```

(94) definition varclosed :: "programterm × state ⇒ bool" **where** Section 1.8 page 26

```
  "varclosed == λ(p,σ,η). freevars p = {} ∧ (∀p'∈set σ. freevars p' = {})"
```

(95) definition Definition 1.26 page 26

```
  "fullyclosed pse == storageclosed pse ∧ varclosed pse"
```

(96) function event_embedding :: "eventT ⇒ programterm" **where**

```
  "event_embedding []     = Nil'P"
| "event_embedding (c#cs) = List'P (Value (bvChar c)) (event_embedding cs)"
```

(97) function event_list_embedding :: "eventT list ⇒ programterm" **where**

```
  "event_list_embedding []     = Nil'P"
| "event_list_embedding (e#es) =
    List'P (event_embedding e) (event_list_embedding es)"
```

I.2.3 Sigma Algebras

instantiation basicvalue :: measurable_space
(98) definition sigma_basicvalue_def: "Σ::basicvalue set set

```
  == sigma (
  {{bvUnit}}∪
  (image (image bvBool) Σ) ∪
  (image (image bvReal) Σ) ∪
  (image (image bvChar) Σ) ∪
  (image (image bvTape) Σ))"
```

(99) inductive_set programterm_rects :: "programterm set set" **where** Section 1.6 page 15

```
"{Var n} ∈ programterm_rects"
|"V ∈ Σ ⟹ Value'V ∈ programterm_rects"
|"⟦A ∈ programterm_rects⟧ ⟹
  {Function f a|a f. a∈A ∧ f∈F} ∈ programterm_rects"
|"⟦A ∈ programterm_rects; B ∈ programterm_rects⟧ ⟹
  {PairP a b|a b. a∈A∧b∈B} ∈ programterm_rects"
|"A ∈ programterm_rects ⟹
  {Abstraction a|a. a∈A} ∈ programterm_rects"
|"⟦A ∈ programterm_rects; B ∈ programterm_rects⟧ ⟹
  {Application a b|a b. a∈A ∧ b∈B} ∈ programterm_rects"
```

```
|"{Location n} ∈ programterm_rects"
|"A ∈ programterm_rects ⟹
  {Ref a|a. a∈A} ∈ programterm_rects"
|"A ∈ programterm_rects ⟹
  {Deref a|a. a∈A} ∈ programterm_rects"
|"⟦A ∈ programterm_rects; B ∈ programterm_rects⟧ ⟹
  {Assign a b|a b. a∈A ∧ b∈B} ∈ programterm_rects"
|"{Event ev} ∈ programterm_rects"
|"{EventList} ∈ programterm_rects"
|"A ∈ programterm_rects ⟹
  {Fst a|a. a∈A} ∈ programterm_rects"
|"A ∈ programterm_rects ⟹
  {Snd a|a. a∈A} ∈ programterm_rects"
|"A ∈ programterm_rects ⟹
  {Fold a|a. a∈A} ∈ programterm_rects"
|"A ∈ programterm_rects ⟹
  {Unfold a|a. a∈A} ∈ programterm_rects"
|"A ∈ programterm_rects ⟹
  {InlP a|a. a∈A} ∈ programterm_rects"
|"A ∈ programterm_rects ⟹
  {InrP a|a. a∈A} ∈ programterm_rects"
|"⟦A ∈ programterm_rects; B ∈ programterm_rects; C ∈ programterm_rects⟧ ⟹
  {CaseP a b c |a b c. a∈A ∧ b∈B ∧ c∈C} ∈ programterm_rects"
```

instantiation programterm :: measurable_space
(100) definition "Σ == sigma programterm_rects"

instantiation pureterm :: measurable_space
(101) definition "Σ == (vimage pureterm_to_term) ‘ Σ"

I.2.4 Lemmas

(102) lemma is_value_substitute:
  assumes "is_value p"
  assumes "is_value q"
  shows "is_value (substitute’ k p q)"

Lemma 1.15 page 19 **(103) lemma** lift_vars_commute:
  assumes "n≤m"
  shows "lift_vars n (lift_vars m p) = lift_vars (Suc m) (lift_vars n p)"

Lemma 1.15 page 19 **(104) lemma** substitute_lift_vars_id [simp]:
  "substitute’ n (lift_vars n p) q = p"

Lemma 1.15 page 19 **(105) lemma** substitute_lift_vars_le:
  assumes "m≤n"
  shows "lift_vars m (substitute’ n p a) =
        substitute’ (Suc n) (lift_vars m p) (lift_vars m a)"

Lemma 1.15 page 19 **(106) lemma** substitute_lift_vars_ge:
  assumes "m≥n"
  shows "lift_vars m (substitute’ n p a) =
        substitute’ n (lift_vars (Suc m) p) (lift_vars m a)"

Lemma 1.15 page 19 **(107) lemma** substitute’_commute_le:
  assumes "m≤n"
  shows "substitute’ m (substitute’ (Suc n) p q) a =
  substitute’ n (substitute’ m p (lift_vars n a)) (substitute’ m q a)"

Lemma 1.15 page 19 **(108) lemma** substitute’_commute_ge:
  assumes "m≥n"
  shows "substitute’ m (substitute’ n p q) a =
  substitute’ n (substitute’ (Suc m) p (lift_vars n a)) (substitute’ m q a)"

(109) lemma lift_not_freevars:
  assumes "∀n∈freevars p. n<L"
  shows "lift_vars L p = p"

(110) lemma lift_closed:
  assumes "freevars p = {}"
  shows "lift_vars n p = p"

(111) lemma substitute_closed:
  assumes "freevars p = {}"
  shows "substitute' n p q = p"

(112) lemma substitute_not_freevars:
  assumes "k∉freevars p"
  shows "substitute' k p q1 = substitute' k p q2"

Section 1.11.4
page 44

(113) lemma locations_of_substitute':
  "locations_of (substitute' k P a) ∪ locations_of a =
   locations_of P ∪ locations_of a"

I.3 Contexts and Redexes

I.3.1 Definitions

(114) datatype "context" =
  CHole
| CVar nat
| CValue basicvalue
| CLocation nat
| CEvent eventT
| CEventList
| CFunction "(pureterm, pureterm) submarkov_kernel" "context"
| CPairP "context" "context"
| CAbstraction "context"
| CApplication "context" "context"
| CRef "context"
| CDeref "context"
| CAssign "context" "context"
| CFst "context"
| CSnd "context"
| CFold "context"
| CUnfold "context"
| CCase "context" "context" "context"
| CInl "context"
| CInr "context"

Definition 1.11
page 15

(115) function programterm_to_context :: "programterm ⇒ context" **where**
  "programterm_to_context (Var n)   = CVar n"
| "programterm_to_context (Value v)   = CValue v"
| "programterm_to_context (Location l)   = CLocation l"
| "programterm_to_context (Event e)   = CEvent e"
| "programterm_to_context (EventList)   = CEventList"
| "programterm_to_context (Function f v)   =
    CFunction f (programterm_to_context v)"
| "programterm_to_context (PairP p1 p2)   =
    CPairP (programterm_to_context p1) (programterm_to_context p2)"
| "programterm_to_context (Abstraction p)   =
    CAbstraction (programterm_to_context p)"
| "programterm_to_context (Application p1 p2) =
    CApplication (programterm_to_context p1) (programterm_to_context p2)"
| "programterm_to_context (Ref p)   = CRef (programterm_to_context p)"
| "programterm_to_context (Deref p)   = CDeref (programterm_to_context p)"

```
| "programterm_to_context (Assign p1 p2)      =
    CAssign (programterm_to_context p1) (programterm_to_context p2)"
| "programterm_to_context (Fst p)            = CFst (programterm_to_context p)"
| "programterm_to_context (Snd p)            = CSnd (programterm_to_context p)"
| "programterm_to_context (Fold p)           =
    CFold (programterm_to_context p)"
| "programterm_to_context (Unfold p)         =
    CUnfold (programterm_to_context p)"
| "programterm_to_context (CaseP p1 p2 p3)   =
    CCase (programterm_to_context p1) (programterm_to_context p2)
                                      (programterm_to_context p3)"
| "programterm_to_context (InlP p)           = CInl (programterm_to_context p)"
| "programterm_to_context (InrP p)           = CInr (programterm_to_context p)"
```

Section 1.6 page 15

(116) function `applycontext :: "context ⇒ programterm ⇒ programterm"` **where**

```
  "applycontext CHole p               = p"
| "applycontext (CFunction f c1) p    = Function f (applycontext c1 p)"
| "applycontext (CPairP c1 c2) p      = PairP (applycontext c1 p) (applycontext c2 p)"
| "applycontext (CAbstraction c) p    = Abstraction (applycontext c p)"
| "applycontext (CApplication c1 c2) p =
    Application (applycontext c1 p) (applycontext c2 p)"
| "applycontext (CRef c) p            = Ref (applycontext c p)"
| "applycontext (CDeref c) p          = Deref (applycontext c p)"
| "applycontext (CAssign c1 c2) p     = Assign (applycontext c1 p) (applycontext c2 p)"
| "applycontext (CFst c) p            = Fst (applycontext c p)"
| "applycontext (CSnd c) p            = Snd (applycontext c p)"
| "applycontext (CVar v) p            = Var v"
| "applycontext (CValue v) p          = Value v"
| "applycontext (CLocation l) p       = Location l"
| "applycontext (CEvent e) p          = Event e"
| "applycontext CEventList p          = EventList"
| "applycontext (CFold c) p           = Fold (applycontext c p)"
| "applycontext (CUnfold c) p         = Unfold (applycontext c p)"
| "applycontext (CInl c) p            = InlP (applycontext c p)"
| "applycontext (CInr c) p            = InrP (applycontext c p)"
| "applycontext (CCase c1 c2 c3) p    =
    CaseP (applycontext c1 p) (applycontext c2 p) (applycontext c3 p)"
```

(117) function `concatcontext :: "context ⇒ context ⇒ context"` **where**

```
  "concatcontext CHole c'               = c'"
| "concatcontext (CFunction f c1) c'    = CFunction f (concatcontext c1 c')"
| "concatcontext (CPairP c1 c2) c'      =
    CPairP (concatcontext c1 c') (concatcontext c2 c')"
| "concatcontext (CAbstraction c) c'    = CAbstraction (concatcontext c c')"
| "concatcontext (CApplication c1 c2) c' =
    CApplication (concatcontext c1 c') (concatcontext c2 c')"
| "concatcontext (CRef c) c'            = CRef (concatcontext c c')"
| "concatcontext (CDeref c) c'          = CDeref (concatcontext c c')"
| "concatcontext (CAssign c1 c2) c'     =
    CAssign (concatcontext c1 c') (concatcontext c2 c')"
| "concatcontext (CFst c) c'            = CFst (concatcontext c c')"
| "concatcontext (CSnd c) c'            = CSnd (concatcontext c c')"
| "concatcontext (CVar v) c'            = CVar v"
| "concatcontext (CValue v) c'          = CValue v"
| "concatcontext (CLocation l) c'       = CLocation l"
| "concatcontext (CEvent e) c'          = CEvent e"
| "concatcontext CEventList c'          = CEventList"
| "concatcontext (CFold c) c'           = CFold (concatcontext c c')"
| "concatcontext (CUnfold c) c'         = CUnfold (concatcontext c c')"
| "concatcontext (CInl c) c'            = CInl (concatcontext c c')"
| "concatcontext (CInr c) c'            = CInr (concatcontext c c')"
| "concatcontext (CCase c1 c2 c3) c'    =
    CCase (concatcontext c1 c') (concatcontext c2 c') (concatcontext c3 c')"
```

(118) lemma `concatcontext_apply:`

```
  "applycontext C1 (applycontext C2 p) = applycontext (concatcontext C1 C2) p"
```

(119) function `lift_vars_context :: "nat ⇒ context ⇒ context"` **where** Section 1.6.1 page 19

```
  "lift_vars_context k (CHole)           = CHole"
| "lift_vars_context k (CVar i)          = (if i < k then CVar i else CVar (Suc i))"
| "lift_vars_context k (CAbstraction s)  = CAbstraction (lift_vars_context (Suc k) s)"
| "lift_vars_context k (CApplication s t) =
    CApplication (lift_vars_context k s) (lift_vars_context k t)"
| "lift_vars_context k (CValue v)        = CValue v"
| "lift_vars_context k (CFunction f p)   = CFunction f (lift_vars_context k p)"
| "lift_vars_context k (CPairP p1 p2)    =
    CPairP (lift_vars_context k p1) (lift_vars_context k p2)"
| "lift_vars_context k (CLocation n)     = CLocation n"
| "lift_vars_context k (CRef p1)         = CRef (lift_vars_context k p1)"
| "lift_vars_context k (CDeref p1)       = CDeref (lift_vars_context k p1)"
| "lift_vars_context k (CAssign p1 p2)   =
    CAssign (lift_vars_context k p1) (lift_vars_context k p2)"
| "lift_vars_context k (CEvent e)        = CEvent e"
| "lift_vars_context k CEventList        = CEventList"
| "lift_vars_context k (CFst p1)         = CFst (lift_vars_context k p1)"
| "lift_vars_context k (CSnd p1)         = CSnd (lift_vars_context k p1)"
| "lift_vars_context k (CFold p1)        = CFold (lift_vars_context k p1)"
| "lift_vars_context k (CUnfold p1)      = CUnfold (lift_vars_context k p1)"
| "lift_vars_context k (CCase p1 p2 p3)  =
    CCase (lift_vars_context k p1) (lift_vars_context k p2) (lift_vars_context k p3)"
| "lift_vars_context k (CInl p1)         = CInl (lift_vars_context k p1)"
| "lift_vars_context k (CInr p1)         = CInr (lift_vars_context k p1)"
```

(120) function `substitute'_context :: "[nat, context, context] ⇒ context"` **where** Section 1.6.1 page 19

```
  "substitute'_context k CHole c             = CHole"
| "substitute'_context k (CVar i) c          =
    (if k < i then CVar (i - 1) else if i=k then c else CVar i)"
| "substitute'_context k (CAbstraction c1) c  =
    CAbstraction (substitute'_context (Suc k) c1 (lift_vars_context 0 c))"
| "substitute'_context k (CApplication s t) c =
    CApplication (substitute'_context k s c) (substitute'_context k t c)"
| "substitute'_context k (CValue v) c         = CValue v"
| "substitute'_context k (CFunction f c1) c   =
    CFunction f (substitute'_context k c1 c)"
| "substitute'_context k (CPairP c1 c2) c     =
    CPairP (substitute'_context k c1 c) (substitute'_context k c2 c)"
| "substitute'_context k (CLocation n) c      = CLocation n"
| "substitute'_context k (CRef c1) c          = CRef (substitute'_context k c1 c)"
| "substitute'_context k (CDeref c1) c        = CDeref (substitute'_context k c1 c)"
| "substitute'_context k (CAssign c1 c2) c    =
    CAssign (substitute'_context k c1 c) (substitute'_context k c2 c)"
| "substitute'_context k (CEvent e) c         = CEvent e"
| "substitute'_context k CEventList c         = CEventList"
| "substitute'_context k (CFst c1) c          = CFst (substitute'_context k c1 c)"
| "substitute'_context k (CSnd c1) c          = CSnd (substitute'_context k c1 c)"
| "substitute'_context k (CCase c1 c2 c3) c   = CCase (substitute'_context k c1 c)
    (substitute'_context k c2 c) (substitute'_context k c3 c)"
| "substitute'_context k (CInl c1) c          = CInl (substitute'_context k c1 c)"
| "substitute'_context k (CInr c1) c          = CInr (substitute'_context k c1 c)"
| "substitute'_context k (CFold c1) c         = CFold (substitute'_context k c1 c)"
| "substitute'_context k (CUnfold c1) c       = CUnfold (substitute'_context k c1 c)"
```

(121) fun `closingcontext :: "nat ⇒ context"` **where**

```
  "closingcontext 0 = CHole" |
  "closingcontext (Suc n) =
    CApplication (CAbstraction (closingcontext n)) (programterm_to_context value_unit)"
```

(122) inductive_set `evaluationcontext :: "context set"` **where** Definition 1.45 page 40

```
"CHole ∈ evaluationcontext"
|"⟦E : evaluationcontext⟧ ⟹ (CFunction f E) ∈ evaluationcontext"
|"⟦E : evaluationcontext; q = programterm_to_context p⟧ ⟹
```

```
    (CPairP E q) ∈ evaluationcontext"
|"⟦E : evaluationcontext; is_value v; cv = programterm_to_context v⟧ ⟹
    (CPairP cv E) ∈ evaluationcontext"
|"⟦E : evaluationcontext; q = programterm_to_context p⟧ ⟹
    (CApplication E q) ∈ evaluationcontext"
|"⟦E : evaluationcontext; is_value v; cv = programterm_to_context v⟧ ⟹
    (CApplication cv E) ∈ evaluationcontext"
|"⟦E : evaluationcontext⟧ ⟹ (CRef E) ∈ evaluationcontext"
|"⟦E : evaluationcontext⟧ ⟹ (CDeref E) ∈ evaluationcontext"
|"⟦E : evaluationcontext; q = programterm_to_context p⟧ ⟹
    (CAssign E q) ∈ evaluationcontext"
|"⟦E : evaluationcontext; is_value v; cv = programterm_to_context v⟧ ⟹
    (CAssign cv E) ∈ evaluationcontext"
|"⟦E : evaluationcontext⟧ ⟹ (CFst E) ∈ evaluationcontext"
|"⟦E : evaluationcontext⟧ ⟹ (CSnd E) ∈ evaluationcontext"
|"⟦E : evaluationcontext⟧ ⟹ (CFold E) ∈ evaluationcontext"
|"⟦E : evaluationcontext⟧ ⟹ (CUnfold E) ∈ evaluationcontext"
|"⟦E : evaluationcontext; q1 = programterm_to_context p1;
    q2 = programterm_to_context p2⟧ ⟹ (CCase E q1 q2) ∈ evaluationcontext"
|"⟦E : evaluationcontext; is_value v; cv = programterm_to_context v;
    q = programterm_to_context p⟧ ⟹ (CCase cv E q) ∈ evaluationcontext"
|"⟦E : evaluationcontext; is_value v1; is_value v2;
    cv1 = programterm_to_context v1;
    cv2 = programterm_to_context v2⟧ ⟹ (CCase cv1 cv2 E) ∈ evaluationcontext"
|"⟦E : evaluationcontext⟧ ⟹ (CInl E) ∈ evaluationcontext"
|"⟦E : evaluationcontext⟧ ⟹ (CInr E) ∈ evaluationcontext"
```

Definition 1.45 page 40

```
(123) inductive_set redexes :: "programterm set" where
  "⟦is_value v⟧                              ⟹ Function f v ∈ redexes"
| "⟦is_value v1; is_value v2⟧                ⟹ Application v1 v2 ∈ redexes"
| "⟦is_value v⟧                              ⟹ Ref v : redexes"
| "⟦is_value v⟧                              ⟹ Deref v : redexes"
| "⟦is_value v1; is_value v2⟧                ⟹ Assign v1 v2 : redexes"
| "Event e : redexes"
| "EventList : redexes"
| "⟦is_value v⟧                              ⟹ Fst v : redexes"
| "⟦is_value v⟧                              ⟹ Snd v : redexes"
| "⟦is_value v⟧                              ⟹ Unfold v : redexes"
| "⟦is_value v1; is_value v2; is_value v3⟧ ⟹ CaseP v1 v2 v3 : redexes"
```

```
(124) inductive_set contextfuns :: "(programterm ⇒ programterm) set" where
"(λx. x) ∈ contextfuns"
|"(λt. u) ∈ contextfuns"
|"⟦C : contextfuns⟧ ⟹ (λt.  Function f (C t)) ∈ contextfuns"
|"⟦C : contextfuns; C' : contextfuns⟧ ⟹ (λt.  PairP (C t) (C' t)) ∈ contextfuns"
|"⟦C : contextfuns⟧ ⟹ (λt.  Abstraction (C t)) ∈ contextfuns"
|"⟦C : contextfuns; C' : contextfuns⟧ ⟹ (λt.  Application (C t) (C' t)) ∈ contextfuns"
|"⟦C : contextfuns⟧ ⟹ (λt.  Ref (C t)) ∈ contextfuns"
|"⟦C : contextfuns⟧ ⟹ (λt.  Deref (C t)) ∈ contextfuns"
|"⟦C : contextfuns; C' : contextfuns⟧ ⟹ (λt.  Assign (C t) (C' t)) ∈ contextfuns"
|"⟦C : contextfuns⟧ ⟹ (λt.  Fst (C t)) ∈ contextfuns"
|"⟦C : contextfuns⟧ ⟹ (λt.  Snd (C t)) ∈ contextfuns"
|"⟦C : contextfuns⟧ ⟹ (λt.  Fold (C t)) ∈ contextfuns"
|"⟦C : contextfuns⟧ ⟹ (λt.  Unfold (C t)) ∈ contextfuns"
|"⟦C : contextfuns; C' : contextfuns; C'' : contextfuns⟧ ⟹
    (λt. CaseP (C t) (C' t) (C'' t)) ∈ contextfuns"
|"⟦C : contextfuns⟧ ⟹ (λt.  InlP (C t)) ∈ contextfuns"
|"⟦C : contextfuns⟧ ⟹ (λt.  InrP (C t)) ∈ contextfuns"
```

I.3.2 Lemmas

```
(125) lemma concat_evaluationcontext:
  assumes "C1 : evaluationcontext" and "C2 : evaluationcontext"
```

```
  shows "concatcontext C1 C2 : evaluationcontext"
```

(126) lemma applycontext_substitute_CHole: Lemma 1.16 page 19

```
  assumes "freevars P = {}"
  shows "applycontext (substitute'_context k (programterm_to_context Q) CHole) P
       = substitute' k Q P"
```

(127) lemma freevars_closingcontext:

```
  assumes "∀x∈freevars p. x < n"
  shows "freevars (applycontext (closingcontext n) p) = {}"
```

(128) lemma ex_closingconfiguration:

```
  assumes "are_values s"
  assumes "∀p'∈set s. freevars p' = {}"
  shows "∃n l. fullyclosed (applycontext (closingcontext n) p,
                            s@(replicate l value_unit), e)"
```

(129) lemma evaluationcontext_redex_split: Section 1.11.2 page 41

```
  "¬ is_value p ⟹ (∃E∈evaluationcontext. ∃R∈redexes. p=applycontext E R)"
```

(130) lemma evaluationcontext_redexes_split_unique':

```
  assumes conceq:"applycontext E1 R1 = applycontext E2 R2"
  assumes E1eval:"E1∈evaluationcontext"
  assumes E2eval:"E2∈evaluationcontext"
  assumes R1red: "R1∈redexes"
  assumes R2red: "R2∈redexes"
  shows "E1 = E2" and "R1 = R2"
```

(131) lemma fullyclosed_eval_ctxD:

```
  assumes "E ∈ evaluationcontext"
  assumes "fullyclosed (applycontext E P, σ, η)"
  shows "fullyclosed (P, σ, η)"
```

(132) lemma fullyclosed_eval_ctx:

```
  assumes "E ∈ evaluationcontext"
  assumes "fullyclosed (applycontext E P, σ, η)"
  assumes "fullyclosed (Q, σ, η)"
  shows "fullyclosed (applycontext E Q, σ, η)"
```

(133) lemma range_applycontext_eq_contextfuns:

```
  "range applycontext = contextfuns"
```

I.4 Semantics

I.4.1 The Reduction Relation and the Denotation

(134) inductive steps_to :: "(programterm × state) ⇒ (programterm × state) subprobability ⇒ bool" (**infix** "⇝" 200) Definition 1.17 page 20

```
  where
  steps_to_Function:
 "(Function f (pureterm_to_term v), state) ⇝ project_measure (λres. (pureterm_to_term
res, state)) (apply_kernel f v)"
| steps_to_Functiond: "⟦(p, state) ⇝ μ⟧ ⟹
  (Function f p, state) ⇝ project_measure (λ(p', s). (Function f p', s)) μ"
| steps_to_Beta: "⟦is_value v⟧ ⟹
  (Application (Abstraction p) v, state) ⇝ apply_kernel unitkernel (substitute' 0 p v,
state)"
| steps_to_Applicationdl: "⟦(p1, state) ⇝ μ⟧ ⟹
  (Application p1 p2, state) ⇝ project_measure (λ(p1', s). (Application p1' p2, s)) μ"
| steps_to_Applicationdr: "⟦is_value v; (p, state) ⇝ μ⟧ ⟹
  (Application v p, state) ⇝ project_measure (λ(p', s). (Application v p', s)) μ"
| steps_to_Ref: "⟦is_value v⟧ ⟹
  (Ref v, store, evs) ⇝ apply_kernel unitkernel (Location (length store), store@[v], evs)"
| steps_to_Refd: "⟦(p, state) ⇝ μ⟧ ⟹
```

```
  (Ref p, state) ⇝ project_measure (λ(p', s). (Ref p', s)) μ"
| steps_to_Deref: "⟦l < length store⟧ ⟹
  (Deref (Location l), store, evs) ⇝ apply_kernel unitkernel (store!l, store, evs)"
| steps_to_Derefd: "⟦(p, state) ⇝ μ⟧ ⟹
  (Deref p, state) ⇝ project_measure (λ(p',s). (Deref p', s)) μ"
| steps_to_Assign: "⟦is_value v; l < length store⟧ ⟹
  (Assign (Location l) v, store, evs) ⇝ apply_kernel unitkernel (value_unit, store[l:=v],
evs)"
| steps_to_Assigndl: "⟦(p1, state) ⇝ μ⟧ ⟹
  (Assign p1 p2, state) ⇝ project_measure (λ(p1', s). (Assign p1' p2, s)) μ"
| steps_to_Assigndr: "⟦is_value v; (p, state) ⇝ μ⟧ ⟹
  (Assign v p, state) ⇝ project_measure (λ(p', s). (Assign v p', s)) μ"
| steps_to_Event:
 "(Event e, store, evs) ⇝ apply_kernel unitkernel (value_unit, store, evs@[e])"
| steps_to_Eventlist:
 "(EventList, store, evs) ⇝ apply_kernel unitkernel (event_list_embedding evs, store,
evs)"
| steps_to_Fst: "⟦is_value v1; is_value v2⟧ ⟹
  (Fst (PairP v1 v2), state) ⇝ apply_kernel unitkernel (v1, state)"
| steps_to_Fstd: "⟦(p, state) ⇝ μ⟧ ⟹
  (Fst p, state) ⇝ project_measure (λ(p',s). (Fst p', s)) μ"
| steps_to_Snd: "⟦is_value v1; is_value v2⟧ ⟹
  (Snd (PairP v1 v2), state) ⇝ apply_kernel unitkernel (v2, state)"
| steps_to_Sndd: "⟦(p, state) ⇝ μ⟧ ⟹
  (Snd p, state) ⇝ project_measure (λ(p',s). (Snd p', s)) μ"
| steps_to_Pairdl: "⟦(p1, state) ⇝ μ⟧ ⟹
  (PairP p1 p2, state) ⇝ project_measure (λ(p1', s). (PairP p1' p2, s)) μ"
| steps_to_Pairdr: "⟦is_value p1; (p2, state) ⇝ μ⟧ ⟹
  (PairP p1 p2, state) ⇝ project_measure (λ(p2', s). (PairP p1 p2', s)) μ"
| steps_to_Unfold: "⟦is_value v⟧ ⟹
  (Unfold (Fold v), state) ⇝ apply_kernel unitkernel (v, state)"
| steps_to_Foldd: "⟦(p, state) ⇝ μ⟧ ⟹
  (Fold p, state) ⇝ project_measure (λ(p', s). (Fold p', s)) μ"
| steps_to_Unfoldd: "⟦(p, state) ⇝ μ⟧ ⟹
  (Unfold p, state) ⇝ project_measure (λ(p', s). (Unfold p', s)) μ"
| steps_to_Casedl: "⟦(p1, state) ⇝ μ⟧ ⟹
  (CaseP p1 p2 p3, state) ⇝ project_measure (λ(p1', s). (CaseP p1' p2 p3, s)) μ"
| steps_to_Casedm: "⟦is_value v; (p2, state) ⇝ μ⟧ ⟹
  (CaseP v p2 p3, state) ⇝ project_measure (λ(p2', s). (CaseP v p2' p3, s)) μ"
| steps_to_Casedr: "⟦is_value v; is_value vl; (p3, state) ⇝ μ⟧ ⟹
  (CaseP v vl p3, state) ⇝ project_measure (λ(p3', s). (CaseP v vl p3', s)) μ"
| steps_to_CaseInl: "⟦is_value v; is_value vl; is_value vr⟧ ⟹
  (CaseP (InlP v) vl vr, state) ⇝ apply_kernel unitkernel (Application vl v, state)"
| steps_to_CaseInr: "⟦is_value v; is_value vl; is_value vr⟧ ⟹
  (CaseP (InrP v) vl vr, state) ⇝ apply_kernel unitkernel (Application vr v, state)"
| steps_to_Inld: "⟦(p, state) ⇝ μ⟧ ⟹
  (InlP p, state) ⇝ project_measure (λ(p',s). (InlP p', s)) μ"
| steps_to_Inrd: "⟦(p, state) ⇝ μ⟧ ⟹
  (InrP p, state) ⇝ project_measure (λ(p',s). (InrP p', s)) μ"
```

Section 1.7, page 22

(135) definition

```
  "step == mk_kernel (λps. if (∃ps'. ps⇝ps')
                           then (THE ps'. ps⇝ps') else (apply_kernel unitkernel ps))"
```

(136) fun nsteps :: "nat ⇒ (programterm × state, programterm × state) submarkov_kernel"
where

```
  "nsteps 0 = unitkernel"
| "nsteps (Suc n) = step oo (nsteps n)"
```

Definition 1.20, page 22

(137) definition

```
  "denotation = kernel_limit (values×UNIV) step"
```

(138) lemma denotation_def2:

```
  "denotation = mk_kernel2 (λps m. lim (λn.
    kernel_prob_of ((restriction_kernel (values×UNIV)) oo (nsteps n)) ps m))"
```

I.4.2 Lemmas

(139) lemma steps_to_no_value_NEW:
assumes "(p,s) $\rightsquigarrow$ μ"
shows "p $\notin$ values"

(140) lemma step_stuck_on_values_NEW:
assumes "is_value v"
shows "apply_kernel step (v,s) = apply_kernel unitkernel (v,s)"

(141) lemma denotation_value:
assumes "is_value v"
shows "apply_kernel denotation (v,σ) = apply_kernel unitkernel (v,σ)"

(142) lemma denotation_contains_values:
"$\forall$ v$\leftarrow$apply_kernel denotation pse. (fst v) $\in$ values"

(143) lemma steps_to_unique: Lemma 1.19 page 22
assumes "ps $\rightsquigarrow$ μ1"
and "ps $\rightsquigarrow$ μ2"
shows "μ1 = μ2"

(144) lemma steps_to_step:
"ps$\rightsquigarrow\mu$ $\Longrightarrow$ apply_kernel step ps = μ"

(145) lemma step_evalctx': Lemma 1.47 page 41
assumes "E : evaluationcontext"
and "P $\notin$ values"
shows "project_measure(λ(P',st').(applycontext E P',st'))(apply_kernel step (P,sta))
= apply_kernel step (applycontext E P, sta)"

(146) lemma step_eq_evalctx_step: Lemma 1.46 page 41
"(p,s) $\rightsquigarrow$ μ =
($\exists$E$\in$evaluationcontext. $\exists$r$\in$redexes. $\exists\mu$r.
p = applycontext E r $\wedge$
(r,s) $\rightsquigarrow$ μr $\wedge$
μ = project_measure (λ(r',s'). (applycontext E r', s')) μr)"

(147) definition "omega = Abstraction (Application (Unfold (Var 0)) (Var 0))" Section 1.8 page 24

(148) definition "diverge = Application omega (Fold omega)" Section 1.8 page 24

(149) lemma diverge_step1: Section 1.8 page 24
"apply_kernel step (diverge,s,e)
= apply_kernel unitkernel (Application (Unfold (Fold omega)) (Fold omega), s, e)"

(150) lemma diverge_step2: Section 1.8 page 24
"apply_kernel (step oo step) (diverge,s,e)
= apply_kernel unitkernel (diverge,s,e)"

(151) lemma diverge_denotation: Section 1.8 page 24
"apply_kernel denotation (diverge,s,e) = 0"

I.5 Typing the Language

I.5.1 Definitions

(152) datatype programtype = Definition 1.21 page 23
ValueT "basicvalue measurableset"
| PairT programtype programtype
| SumT programtype programtype
| FunT programtype programtype
| RefT programtype

```
| TypeVarT nat
| MuT programtype
```

Definition 1.21 page 23

```
(153) inductive_set puretypes :: "programtype set" where
  "ValueT T ∈ puretypes"
| "⟦a ∈ puretypes; b ∈ puretypes⟧ ⟹ PairT a b ∈ puretypes"
| "⟦a ∈ puretypes; b ∈ puretypes⟧ ⟹ SumT a b ∈ puretypes"
| "a ∈ puretypes ⟹ MuT a ∈ puretypes"
| "TypeVarT n ∈ puretypes"
```

Section 1.8 page 23

```
(154) definition "UnitT = ValueT (Abs_measurableset {bvUnit})"
```

Section 1.8 page 23

```
(155) definition "Bool'T = SumT UnitT UnitT"
```

Section 1.8 page 23

```
(156) definition "List'T T = MuT (SumT UnitT (PairT T (TypeVarT 0)))"
```

Section 1.8 page 23

```
(157) definition "Nat'T = SumT UnitT (List'T Bool'T)"
```

Section 1.8 page 23

```
(158) definition "CharT = ValueT (Abs_measurableset (range bvChar))"
```

Section 1.8 page 23

```
(159) types typeenvironment = "programtype list"
```

Section 1.8 page 23

```
(160) function mu_lift_vars :: "nat ⇒ programtype ⇒ programtype" where
  "mu_lift_vars k (TypeVarT i)  = (if i < k then TypeVarT i else TypeVarT (Suc i))"
| "mu_lift_vars k (MuT t)       = MuT (mu_lift_vars (Suc k) t)"

| "mu_lift_vars k (ValueT X)    = ValueT X"
| "mu_lift_vars k (FunT t1 t2)  = FunT (mu_lift_vars k t1) (mu_lift_vars k t2)"
| "mu_lift_vars k (PairT t1 t2) = PairT (mu_lift_vars k t1) (mu_lift_vars k t2)"
| "mu_lift_vars k (SumT t1 t2)  = SumT (mu_lift_vars k t1) (mu_lift_vars k t2)"
| "mu_lift_vars k (RefT t)      = RefT (mu_lift_vars k t)"
```

Section 1.8 page 23

```
(161) function mu_substitute' :: "[nat,programtype,programtype] ⇒ programtype"
  where
  "mu_substitute' k (TypeVarT i) p  =
    (if k < i then TypeVarT (i - 1) else if i=k then p else TypeVarT i)"
| "mu_substitute' k (MuT t) p       =
    MuT (mu_substitute' (Suc k) t (mu_lift_vars 0 p))"
| "mu_substitute' k (ValueT X) p    = ValueT X"
| "mu_substitute' k (FunT t1 t2) p  =
    FunT (mu_substitute' k t1 p) (mu_substitute' k t2 p)"
| "mu_substitute' k (PairT t1 t2) p =
    PairT (mu_substitute' k t1 p) (mu_substitute' k t2 p)"
| "mu_substitute' k (SumT t1 t2) p  =
    SumT (mu_substitute' k t1 p) (mu_substitute' k t2 p)"
| "mu_substitute' k (RefT t) p      = RefT (mu_substitute' k t p)"
```

Definition 1.22 page 24

```
(162) inductive program_typing ::
  "[typeenvironment, typeenvironment, programterm, programtype] ⇒ bool" ("_|_⊢_:_")
  where
  program_typing_Var0: "(T#Γ)|Θ ⊢ Var 0 : T"
| program_typing_Varn: "Γ|Θ ⊢ Var n : T ⟹
    (U#Γ)|Θ ⊢ Var (Suc n) : T"
| program_typing_Value: "x ∈ Rep_measurableset T ⟹
    Γ|Θ ⊢ Value x : ValueT T"
| program_typing_Function: "⟦T : puretypes; U : puretypes; Γ|Θ ⊢ p : U;
    (∀x. ∀y←apply_kernel f x. ([]|[]⊢pureterm_to_term y:T))⟧ ⟹
    Γ|Θ ⊢ Function f p : T"
| program_typing_Pair: "⟦Γ|Θ ⊢ p1 : T1; Γ|Θ ⊢ p2 : T2⟧ ⟹
    Γ|Θ ⊢ PairP p1 p2 : (PairT T1 T2)"
| program_typing_Abstraction: "(T1#Γ)|Θ ⊢ p : T2 ⟹
    Γ|Θ ⊢ Abstraction p : (FunT T1 T2)"
| program_typing_Application: "⟦Γ|Θ ⊢ p1 : (FunT T1 T2); Γ|Θ ⊢ p2 : T1⟧ ⟹
```

```
    Γ|Θ ⊢ Application p1 p2 : T2"
| program_typing_Location0: "Γ|T#Θ ⊢ Location 0 : RefT T"
| program_typing_Locationn: "Γ|Θ ⊢ Location l : RefT T ⟹
    Γ|U#Θ ⊢ Location (Suc l) : RefT T"
| program_typing_Ref: "Γ|Θ ⊢ p : T ⟹
    Γ|Θ ⊢ Ref p : (RefT T)"
| program_typing_Deref: "Γ|Θ ⊢ p : (RefT T) ⟹
    Γ|Θ ⊢ Deref p : T"
| program_typing_Assign: "⟦Γ|Θ ⊢ p1 : (RefT T); Γ|Θ ⊢ p2 : T⟧ ⟹
    Γ|Θ ⊢ Assign p1 p2 : UnitT"
| program_typing_Event: "Γ|Θ ⊢ Event e : UnitT"
| program_typing_EventList: "Γ|Θ ⊢ EventList : List'T (List'T CharT)"
| program_typing_Fst: "Γ|Θ ⊢ p : (PairT T1 T2) ⟹
    Γ|Θ ⊢ Fst p : T1"
| program_typing_Snd: "Γ|Θ ⊢ p : (PairT T1 T2) ⟹
    Γ|Θ ⊢ Snd p : T2"
| program_typing_Fold: "Γ|Θ ⊢ p : (mu_substitute' 0 T (MuT T)) ⟹
    Γ|Θ ⊢ Fold p : MuT T"
| program_typing_Unfold:"⟦Γ|Θ ⊢ p : MuT T; (mu_substitute' 0 T (MuT T)) = T'⟧ ⟹
    Γ|Θ ⊢ Unfold p : T'"
| program_typing_Inl: "Γ|Θ ⊢ p : T ⟹
    Γ|Θ ⊢ InlP p : SumT T U"
| program_typing_Inr: "Γ|Θ ⊢ p : U ⟹
    Γ|Θ ⊢ InrP p : SumT T U"
| program_typing_Case:
    "⟦Γ|Θ ⊢ p1 : SumT T U; Γ|Θ ⊢ p2 : (FunT T V); Γ|Θ ⊢ p3 : (FunT U V)⟧ ⟹
    Γ|Θ ⊢ CaseP p1 p2 p3 : V"
```

(163) definition `of_type :: "programterm ⇒ programtype ⇒ bool" where`
`"of_type == program_typing [] []"`

(164) definition `"welltyped_programs == {P. ∃T. of_type P T}"`

(165) definition `configuration_typing ::`
`"[typeenvironment, typeenvironment, programterm×state, programtype] ⇒ bool"`
`("_|_⊨_:_") where`
`"configuration_typing ≡`
`λΓ Θ (p,s,e) T. ((length Θ = length s) ∧`
`(are_values s) ∧`
`(∀n<length Θ. (Γ|Θ⊢(s!n):(Θ!n))) ∧`
`(Γ|Θ⊢p:T))"`

Definition 1.25
page 26

I.5.2 Lemmas

(166) inductive `max_typevar :: "nat ⇒ programtype ⇒ bool" where`
`"l<k ⟹ max_typevar k (TypeVarT l)"`
`| "max_typevar (Suc k) p1 ⟹ max_typevar k (MuT p1)"`

```
| "max_typevar k (ValueT X)"
| "⟦max_typevar k p1; max_typevar k p2⟧ ⟹ max_typevar k (PairT p1 p2)"
| "⟦max_typevar k p1; max_typevar k p2⟧ ⟹ max_typevar k (SumT p1 p2)"
| "⟦max_typevar k p1; max_typevar k p2⟧ ⟹ max_typevar k (FunT p1 p2)"
| "⟦max_typevar k p1⟧ ⟹ max_typevar k (RefT p1)"
```

(167) lemma `program_typing_omega:`
assumes `"max_typevar 0 T"`
shows `"Γ|Θ ⊢ omega : FunT (MuT(FunT(TypeVarT 0)T)) T"`

Section 1.8
page 24

(168) lemma `program_typing_diverge:`
assumes `"max_typevar 0 T"`
shows `"Γ|Θ ⊢ diverge: T"`

Section 1.8
page 24

(169) lemma `liftn_typing:`

```
  assumes "(Γ@Γ')|Θ⊢ p : T"
  and "length Γ = n"
  shows "(Γ@t#Γ')|Θ⊢ lift_vars n p : T"
```

(170) lemma substitute_typing:
```
  "⟦(Γ@[a]@Γ')|Θ⊢p:T; (Γ@Γ')|Θ⊢q:a⟧ ⟹ (Γ@Γ')|Θ⊢(substitute' (length Γ) p q):T"
```

(171) lemma freevars_in_gamma: "Γ|Θ⊢ P : T ⟹ ∀n∈freevars P. n<length Γ"

Lemma 1.27 page 26 **(172) lemma** empty_typing_program_closed: "[]|Θ⊢ P : T ⟹ freevars P = {}"

(173) lemma locations_of_in_theta: "Γ|Θ⊢ P : T ⟹ ∀n∈locations_of P. n < length Θ"

(174) lemma empty_typing_fullyclosed:
```
  "[]|[]⊢P:T ⟹ fullyclosed (P,[],e)"
```

Lemma 1.27 page 26 **(175) lemma** configuration_typing_fullyclosed: "[]|Θ⊨pse:T ⟹ fullyclosed pse"

Lemma 1.28 page 26 **(176) lemma** weakening:
```
  assumes "Γ|Θ⊢ p : T"
  shows "(Γ@Γ')|(Θ@Θ')⊢ p : T"
```

I.5.3 Progress and Preservation

Theorem 1.44 page 40 **(177) theorem** progress:
```
  assumes "[]|Θ⊨ ps : T"
  and "¬ is_value (fst ps)"
  shows "∃μ. ps⇝μ"
```

Theorem 1.44 page 40 **(178) theorem** preservation:
```
  assumes "Γ|Θ⊨ ps : T"
  and "ps ⇝ μ"
  shows "∃Θ'. (ALL ps'<-μ. (Γ|Θ@Θ'⊨ ps' : T))"
```

(179) lemma fullyclosed_preservation:
```
  assumes "fullyclosed pse"
  assumes "pse ⇝ μ"
  shows "∀vse←μ. fullyclosed vse ∧ length (fst(snd pse)) ≤ length (fst(snd vse))"
```

Theorem 1.44 page 40 **(180) lemma** fullyclosed_preservation_step:
```
  assumes "fullyclosed pse"
  assumes "μ = apply_kernel step pse"
  shows "∀vse←μ. fullyclosed vse"
```

I.5.4 Typing Contexts

Section 1.8.1 page 27 **(181) inductive** context_typing :: "[typeenvironment, typeenvironment, programtype,
```
  typeenvironment, typeenvironment, context, programtype] ⇒ bool" where
  context_typing_Hole: "context_typing Γ' Θ' T' (Γ'@Γ) (Θ'@Θ) CHole T'"
| context_typing_Var0: "context_typing Γ' Θ' T' (T#Γ) Θ (CVar 0) T"
| context_typing_Varn: "context_typing Γ' Θ' T' Γ Θ (CVar n) T ⟹
    context_typing Γ' Θ' T' (U#Γ) Θ (CVar (Suc n)) T"
| context_typing_Value: "x ∈ Rep_measurableset T ⟹
    context_typing Γ' Θ' T' Γ Θ (CValue x) (ValueT T)"
| context_typing_Function: "⟦T : puretypes; U : puretypes;
    context_typing Γ' Θ' T' Γ Θ p U;
    (∀x. ∀y←apply_kernel f x. ([]|[]⊢pureterm_to_term y:T))⟧ ⟹
    context_typing Γ' Θ' T' Γ Θ (CFunction f p) T"
| context_typing_Pair: "⟦context_typing Γ' Θ' T' Γ Θ p1 T1;
    context_typing Γ' Θ' T' Γ Θ p2 T2⟧ ⟹
    context_typing Γ' Θ' T' Γ Θ (CPairP p1 p2) (PairT T1 T2)"
| context_typing_Abstraction: "⟦context_typing Γ' Θ' T' (T1#Γ) Θ p T2⟧ ⟹
```

```
    context_typing Γ' Θ' T' Γ Θ (CAbstraction p) (FunT T1 T2)"
| context_typing_Application: "⟦context_typing Γ' Θ' T' Γ Θ p1 (FunT T1 T2);
    context_typing Γ' Θ' T' Γ Θ p2 T1⟧ ⟹
    context_typing Γ' Θ' T' Γ Θ (CApplication p1 p2) T2"
| context_typing_Location0: "context_typing Γ' Θ' T' Γ (T#Θ) (CLocation 0) (RefT T)"
| context_typing_Locationn: "context_typing Γ' Θ' T' Γ Θ (CLocation l) (RefT T) ⟹
    context_typing Γ' Θ' T' Γ (U#Θ) (CLocation (Suc l)) (RefT T)"
| context_typing_Ref: "⟦context_typing Γ' Θ' T' Γ Θ p T⟧ ⟹
    context_typing Γ' Θ' T' Γ Θ (CRef p) (RefT T)"
| context_typing_Deref: "⟦context_typing Γ' Θ' T' Γ Θ p (RefT T)⟧ ⟹
    context_typing Γ' Θ' T' Γ Θ (CDeref p) T"
| context_typing_Assign: "⟦context_typing Γ' Θ' T' Γ Θ p1 (RefT T);
    context_typing Γ' Θ' T' Γ Θ p2 T⟧ ⟹
    context_typing Γ' Θ' T' Γ Θ (CAssign p1 p2) UnitT"
| context_typing_Event: "context_typing Γ' Θ' T' Γ Θ (CEvent e) UnitT"
| context_typing_EventList:
    "context_typing Γ' Θ' T' Γ Θ CEventList (List'T (List'T CharT))"
| context_typing_Fst: "⟦context_typing Γ' Θ' T' Γ Θ p (PairT T1 T2)⟧ ⟹
    context_typing Γ' Θ' T' Γ Θ (CFst p) T1"
| context_typing_Snd: "⟦context_typing Γ' Θ' T' Γ Θ p (PairT T1 T2)⟧ ⟹
    context_typing Γ' Θ' T' Γ Θ (CSnd p) T2"
| context_typing_Fold: "context_typing Γ' Θ' T' Γ Θ p (mu_substitute' 0 T (MuT T)) ⟹
    context_typing Γ' Θ' T' Γ Θ (CFold p) (MuT T)"
| context_typing_Unfold: "⟦context_typing Γ' Θ' T' Γ Θ p (MuT T);
    (mu_substitute' 0 T (MuT T)) = T''⟧ ⟹
    context_typing Γ' Θ' T' Γ Θ (CUnfold p) T''"
| context_typing_Inl: "context_typing Γ' Θ' T' Γ Θ p T ⟹
    context_typing Γ' Θ' T' Γ Θ (CInl p) (SumT T U)"
| context_typing_Inr: "context_typing Γ' Θ' T' Γ Θ p U ⟹
    context_typing Γ' Θ' T' Γ Θ (CInr p) (SumT T U)"
| context_typing_Case: "⟦context_typing Γ' Θ' T' Γ Θ p1 (SumT T U);
    context_typing Γ' Θ' T' Γ Θ p2 (FunT T V);
    context_typing Γ' Θ' T' Γ Θ p3 (FunT U V)⟧ ⟹
    context_typing Γ' Θ' T' Γ Θ (CCase p1 p2 p3) V"
```

(182) lemma `typing_applycontext:` — Lemma 1.29, page 27
assumes `"Γ|Θ⊢P:T"`
and `"context_typing Γ Θ T Γ' Θ' C T'"`
shows `"Γ'|Θ'⊢applycontext C P : T'"`

(183) lemma `applycontext_fullyclosed:` — Section 1.8.1, page 27
assumes `"Γ|Θ⊢P:T"`
and `"context_typing Γ Θ T [] [] C T'"`
shows `"fullyclosed (applycontext C P, [], [])"`

I.6 Embedding the Type System in HOL

I.6.1 Embedding Types, Environments, and Programs

(184) class `program_type =` — Section 1.9.1, page 28
fixes `prog_type :: "'a itself ⇒ programtype"`
assumes `inhabited: "∃p. ([]|[]⊢p:(prog_type TYPE('a)))"`
assumes `closed_type: "max_typevar 0 (prog_type TYPE('a))"`

(185) class `environment =` — Section 1.9.1, page 29
fixes `env_types :: "'a itself ⇒ programtype list"`

(186) class `empty_environment = environment +`
assumes `empty_environment: "env_types TYPE('a) == []"`

(187) typedef `env_nil = "{()}"` — Section 1.9.1, page 29
instantiation `env_nil :: empty_environment`

Section 1.9.1 page 29
(188) definition env_types_nil:
"env_types (T::env_nil itself) == []"

Section 1.9.1 page 29
(189) typedef ('a,'b) env_cons = "{()}"
instantiation env_cons :: (program_type,environment)environment
Section 1.9.1 page 29
(190) definition env_types_cons:
"env_types (_::('a::program_type,'b::environment) env_cons itself)
== prog_type(TYPE('a)) # env_types(TYPE('b))"

Section 1.9.1 page 29
(191) typedef (program) ('e,'t) "⟦⟧" =
"{p. env_types(TYPE('e::environment))|[]⊢p:(prog_type(TYPE('t::program_type)))}"

syntax (xsymbols)
"_program_type" :: "[types,type,type] ⇒ type" ("(⟦(_ ... _) /⊢ (_)⟧)")
"_program_type" :: "[type,type] ⇒ type" ("(⟦(... _) /⊢ (_)⟧)")
"_program_type_nil" :: "[types,type] ⇒ type" ("(⟦_/⊢ (_)⟧)")
"_program_type_nil" :: "[type] ⇒ type" ("(⟦/⊢ (_)⟧)")
"_program_type_nil" :: "[type] ⇒ type" ("(⟦(_)⟧)")

instantiation "fun" :: (program_type,program_type)program_type
(192) definition prog_type_fun:
"prog_type == λ(_::('a::program_type⇒'b::program_type) itself).
FunT (prog_type TYPE('a)) (prog_type TYPE('b))"

instantiation "*" :: (program_type,program_type)program_type
Section 1.9.1 page 28
(193) definition prog_type_prod:
"prog_type == λ(_::('a::program_type × 'b::program_type) itself).
PairT (prog_type TYPE('a)) (prog_type TYPE('b))"

instantiation "+" :: (program_type,program_type)program_type
(194) definition prog_type_sum:
"prog_type == λ(_::('a::program_type + 'b::program_type) itself).
SumT (prog_type TYPE('a)) (prog_type TYPE('b))"

instantiation "list" :: (program_type)program_type
Section 1.9.1 page 28
(195) definition prog_type_list:
"prog_type == λ(_::('a::program_type) list itself).
List'T (prog_type TYPE('a))"

(196) datatype 'a reference = Reference 'a

instantiation "reference" :: (program_type)program_type
Section 1.9.1 page 28
(197) definition prog_type_ref: "prog_type ==
λ(_::('a::program_type) reference itself). RefT (prog_type TYPE('a))"

instantiation "⟦⟧" :: (environment,program_type)measurable_space
(198) definition "Σ == (vimage Rep_program) ' Σ"

I.6.2 Embedding HOL Objects into the Language

Section 1.9.2 page 31
(199) class embeddable = program_type + default +
fixes prog_embedding :: "'a ⇒ programterm"
assumes prog_embedding_welltyped: "[]|[]⊢(prog_embedding x):(prog_type TYPE('a))"

Section 1.9.2 page 31
(200) class embeddable_val = embeddable +
assumes prog_embedding_value: "prog_embedding x : values"
fixes kernel_of :: "'a ⇒ (pureterm,pureterm)submarkov_kernel"

Section 1.9.2 page 32
(201) class embeddable_pure = embeddable_val + measurable_space +
fixes inv_prog_embedding :: "programterm ⇒ 'a"
assumes pure_prog_embedding: "prog_embedding x : purevalues"

```
  and has_pure_type: "prog_type TYPE('a) : puretypes"
  and inv_prog_embedding: "!!x. inv_prog_embedding (prog_embedding x) = x"
  and kernel_of_pure:
    "kernel_of v = constant_kernel (term_to_pureterm (prog_embedding v))"
  and prog_embedding_measurable: "prog_embedding : measurable Σ Σ"
  and inv_prog_embedding_measurable: "inv_prog_embedding : measurable Σ Σ"
  assumes pure_prog_embedding_surjective:
    "[]|[]⊢p:(prog_type TYPE('a)) ⟹ p : values ⟹ (∃x. p = prog_embedding x)"

(202) fun split_fun_type :: "programtype ⇒ (programtype list × programtype)"
where
  "split_fun_type (FunT a b) = (a#(fst (split_fun_type b)),snd (split_fun_type b))"
| "split_fun_type a = ([],a)"

(203) class embeddable_kernel = embeddable_val +
  assumes kernel_of:
  "∀v←apply_kernel(deterministic_kernel pureterm_to_term oo (kernel_of f))x.
       []|[]⊢v:(snd (split_fun_type (prog_type TYPE('a))))"
  assumes pk_args_puretype:
  "set (fst (split_fun_type (prog_type TYPE('a)))) ⊆ puretypes"
  assumes pk_out_puretype:
  "snd (split_fun_type (prog_type TYPE('a))) ∈ puretypes"

(204) fun pk_invoke :: "programtype list ⇒ programterm ⇒ programterm" where
  "pk_invoke [] X = X"
| "pk_invoke (a#args) X = Abstraction (pk_invoke args X)"

(205) fun pk_construct_arg :: "programtype list ⇒ programterm" where
  "pk_construct_arg [] = value_unit"
| "pk_construct_arg (a#args) = PairP (Var (length args)) (pk_construct_arg args)"

subclass (in embeddable_pure) embeddable_kernel

instantiation "⟦⟧"::(empty_environment,program_type)embeddable

(206) definition prog_type_program:                                    Section 1.9.2
  "prog_type == λ(_::⟦...'a::empty_environment⊢'b⟧ itself). prog_type TYPE('b)"   page 31
(207) definition prog_embedding_program: "prog_embedding == Rep_program"
                                                                       Section 1.9.2
instantiation "fun" :: (embeddable_pure,embeddable_kernel)embeddable_kernel   page 31
(208) definition default_fun: "default == λx. default"
(209) definition kernel_of_fun: "kernel_of f =                         Section 1.9.2
  mk_kernel (λxy::pureterm. case xy of (ptPair x y) ⇒                  page 32
  apply_kernel (kernel_of (f (inv_prog_embedding (pureterm_to_term x)))) y
  | _ ⇒ 0)"
(210) definition prog_embedding_fun: "prog_embedding (f::'a⇒'b) ==      Section 1.9.2
  let args = (fst (split_fun_type (prog_type TYPE('a⇒'b)))) in         page 32
  pk_invoke args (Function (kernel_of f) (pk_construct_arg args))"

instantiation "*" :: (embeddable,embeddable) embeddable
(211) definition "default == (default,default)"
(212) definition "prog_embedding == λ(x,y). PairP (prog_embedding x) (prog_embedding y)"

instantiation "*" :: (embeddable_val,embeddable_val) embeddable_val
(213) definition "kernel_of (v::'a*'b) =
  constant_kernel (term_to_pureterm (prog_embedding v))"

instantiation "*" :: (embeddable_pure,embeddable_pure)embeddable_pure
(214) definition "inv_prog_embedding ==
  inv_default prog_embedding default :: programterm ⇒ ('a*'b)"

instantiation "+" :: (embeddable,embeddable) embeddable
(215) definition "default == Inl default"
```

(216) definition "prog_embedding == λx. case x of Inl x ⇒ InlP (prog_embedding x) | Inr x ⇒ InrP (prog_embedding x)"

instantiation "+" :: (embeddable_val,embeddable_val) embeddable_val
(217) definition "kernel_of (v::'a+'b) =
  constant_kernel (term_to_pureterm (prog_embedding v))"

instantiation "+" :: (embeddable_pure,embeddable_pure)embeddable_pure
(218) definition "inv_prog_embedding ==
  inv_default prog_embedding default :: programterm ⇒ ('a+'b)"

instantiation unit :: embeddable_pure
Section 1.9.1 page 28
(219) definition prog_type_unit: "prog_type == λ(_::unit itself). UnitT"
(220) definition "prog_embedding == λx::unit. Value bvUnit"
(221) definition "inv_prog_embedding ==
  inv_default prog_embedding default :: programterm ⇒ unit"
(222) definition "kernel_of (v::unit) =
  constant_kernel (term_to_pureterm (prog_embedding v))"

(223) function list_to_programterm ::
  "('a::embeddable) list ⇒ programterm"
where
  "list_to_programterm [] = Nil'P"
| "list_to_programterm (x#xs) = List'P (prog_embedding x) (list_to_programterm xs)"

instantiation list :: (embeddable)embeddable
(224) definition default_list: "default == []"
(225) definition prog_embedding_list: "prog_embedding == list_to_programterm"

instantiation list :: (embeddable_val)embeddable_val
(226) definition "kernel_of (v::'a list) =
  constant_kernel (term_to_pureterm (prog_embedding v))"

instantiation list :: (embeddable_pure)embeddable_pure
(227) definition inv_prog_embedding_list: "inv_prog_embedding ==
  (inv_default prog_embedding default) :: programterm ⇒ 'a list"

instantiation bool :: embeddable_pure
Section 1.9.1 page 28
(228) definition prog_type_bool: "prog_type == λ(_::bool itself). Bool'T"
(229) definition default_bool: "default == False"
(230) definition prog_embedding_bool: "prog_embedding ==
  λb. if b then value_true else value_false"
(231) definition inv_prog_embedding_bool: "inv_prog_embedding ==
  inv_default prog_embedding default :: programterm ⇒ bool"
(232) definition "kernel_of (v::bool) =
  constant_kernel (term_to_pureterm (prog_embedding v))"

instantiation nat :: embeddable_pure
Section 1.9.1 page 28
(233) definition prog_type_nat: "prog_type == λ(_::nat itself). Nat'T"
(234) definition "default == 0 :: nat"
(235) definition "prog_embedding == λx. prog_embedding (nat_to_bitstring2 x)"
(236) definition "inv_prog_embedding ==
  λx. bitstring_to_nat2 (inv_prog_embedding x)"
(237) definition "kernel_of (v::nat) =
  constant_kernel (term_to_pureterm (prog_embedding v))"

I.6.3 Representations of Programs in HOL

Section 1.9.1 page 29
(238) definition
  "VAR0 == Abs_program (Var 0) :: ⟦'a::program_type...'e::environment⊢'a⟧"

Section 1.9.1 page 29
(239) definition

```
  "VAR_SUC (b::⟦...'e::environment⊢'a::program_type⟧) ==
  (case Rep_program b of Var n ⇒ Abs_program (Var (Suc n))
                       | _ ⇒ Abs_program (SOME p. []|[]⊢p:(prog_type(TYPE('a)))))
  :: ⟦'b::program_type...'e⊢'a⟧"
```

(240) definition Section 1.9.1 page 29

```
  "ABSTRACT (name::string) (p::⟦'a::program_type...'e::environment⊢'b::program_type⟧)
  ==
  (Abs_program (Abstraction (Rep_program p))) :: ⟦...'e⊢'a⇒'b⟧"
```

(241) definition Section 1.9.1 page 29

```
  "APPLY (p1::⟦...'e::environment⊢'t1::program_type⇒'t2::program_type⟧)
         (p2::⟦...'e⊢'t1::program_type⟧) ==
  (Abs_program (Application (Rep_program p1) (Rep_program p2))) :: ⟦...'e⊢'t2⟧"
```

(242) definition Section 1.9.1 page 29

```
  "PAIR (p1::⟦...'e::environment⊢'t1::program_type⟧)
        (p2::⟦...'e::environment⊢'t2::program_type⟧) ==
  (Abs_program (PairP (Rep_program p1) (Rep_program p2))) :: ⟦...'e⊢'t1×'t2⟧"
```

(243) definition Section 1.9.1 page 29

```
  "FST (p::⟦...'e::environment⊢('a::program_type)×('b::program_type)⟧) ==
  (Abs_program (Fst (Rep_program p))) :: ⟦...'e⊢'a⟧"
```

(244) definition Section 1.9.1 page 29

```
  "SND (p::⟦...'e::environment⊢('a::program_type)×('b::program_type)⟧) ==
  (Abs_program (Snd (Rep_program p))) :: ⟦...'e⊢'b⟧"
```

(245) definition

```
  "CASE (p1::⟦...'e::environment⊢'t1::program_type + 't2::program_type⟧)
        (p2::⟦...'e::environment⊢'t1 ⇒ 't::program_type⟧)
        (p3::⟦...'e::environment⊢'t2 ⇒ 't⟧) ==
  (Abs_program (CaseP (Rep_program p1) (Rep_program p2) (Rep_program p3))) :: ⟦...'e⊢'t⟧"
```

(246) definition

```
  "INL (p::⟦...'e::environment⊢'a::program_type⟧) ==
  (Abs_program (InlP (Rep_program p))) :: ⟦...'e⊢'a+'b::program_type⟧"
```

(247) definition

```
  "INR (p::⟦...'e::environment⊢'b::program_type⟧) ==
  (Abs_program (InrP (Rep_program p))) :: ⟦...'e⊢'a::program_type+'b⟧"
```

(248) definition

```
  "VALUE (v::'a::embeddable) ==
  (Abs_program (prog_embedding v)) :: ⟦...'e::environment⊢'a⟧"
```

(249) definition

```
"kernel2puretermkernel f ==
  deterministic_kernel (term_to_pureterm o prog_embedding) oo
  f oo deterministic_kernel (inv_prog_embedding o pureterm_to_term)"
```

(250) definition

```
  "FUNCTION (f::('a,'b)submarkov_kernel)
            (p::⟦...'e::environment⊢'a::embeddable_pure⟧) ==
  (Abs_program (Function (kernel2puretermkernel f) (Rep_program p)))
  :: ⟦...'e⊢'b::embeddable_pure⟧"
```

(251) definition

```
  "REF (p::⟦...'e::environment⊢'t::program_type⟧) ==
  (Abs_program (Ref (Rep_program p))) :: ⟦...'e⊢'t reference⟧"
```

(252) definition

```
  "DEREF (p::⟦...'e::environment⊢('t::program_type) reference⟧) ==
```

```
  (Abs_program (Deref (Rep_program p))) :: ⟦...'e⊢'t⟧"
```

(253) definition
```
  "ASSIGN (p1::⟦...'e::environment⊢('t::program_type) reference⟧) (p2::⟦...'e⊢'t⟧) ==
  (Abs_program (Assign (Rep_program p1) (Rep_program p2))) :: ⟦...'e⊢unit⟧"
```

(254) definition
```
  "EVENT (e::eventT) == (Abs_program (Event e)) :: ⟦...'e::environment⊢unit⟧"
```

Section 1.9.1 page 29 **(255) definition**
```
  "NIL == Abs_program Nil'P :: ⟦...'e::environment⊢'t::program_type list⟧"
```

Section 1.9.1 page 29 **(256) definition**
```
  "CONS (p::⟦...'e⊢'t⟧) (ps::⟦...'e::environment⊢('t::program_type) list⟧) ==
  Abs_program (List'P (Rep_program p) (Rep_program ps)) :: ⟦...'e⊢'t list⟧"
```

(257) definition
```
  "IFTHENELSE (c::⟦...'e::environment⊢bool⟧)
              (p1::⟦...'e::environment⊢'t::program_type⟧)
              (p2::⟦...'e::environment⊢'t⟧) ==
  (Abs_program (IfThenElse' (Rep_program c) (Rep_program p1) (Rep_program p2)))
  :: ⟦...'e⊢'t⟧"
```

(258) definition
```
  "FIX == Abs_program Fix ::
  ⟦...'e::environment⊢(('a::program_type ⇒ 'b::program_type)⇒('a ⇒ 'b))⇒('a ⇒ 'b)⟧"
```

(259) definition
```
  "LET (p1::⟦...'e1::environment⊢('t1::program_type)⟧)
       (p2::⟦...'e1::environment⊢'t1 ⇒ ('t2::program_type)⟧) ==
  (Abs_program (LetP (Rep_program p1) (Rep_program p2))) :: ⟦...'e1::environment⊢'t2⟧"
```

(260) definition
```
  "SEQUENCE (p1::⟦...'e::environment⊢'t1::program_type⟧)
            (p2::⟦...'e::environment⊢'t2::program_type⟧) ==
  (Abs_program (SequenceP (Rep_program p1) (Rep_program p2)))
  :: ⟦...'e::environment⊢'t2⟧"
```

(261) definition
```
  "UNCURRY
  (p::⟦...'e::environment⊢('t1::program_type ⇒ 't2::program_type ⇒ 't::program_type)⟧)
  == Abs_program (uncurry (Rep_program p))
  :: ⟦...'e::environment⊢('t1::program_type × 't2::program_type) ⇒ 't::program_type⟧"
```

Section 1.9.1 page 29 **(262) definition**
```
  "LIFT (p::⟦...'e::environment⊢('t::program_type)⟧) ==
  (Abs_program (lift_vars 0 (Rep_program p)))
  :: ⟦'a::program_type... ('e::environment)⊢'t⟧"
```

Section 1.9.1 page 30 **(263) definition**
```
  "LIFTk n (p::⟦...'e1::environment⊢'t⟧) ==
  (Abs_program (lift_vars n (Rep_program p)))
  :: ⟦...'e2::environment⊢('t::program_type)⟧"
```

Section 1.9.1 page 30 **(264) definition**
```
  "SUBSTITUTE (p::⟦'a ... 'e⊢('t::program_type)⟧) (q::⟦...'e⊢('a::program_type)⟧)  ==
  (Abs_program (substitute' 0 (Rep_program p) (Rep_program q)))
  :: ⟦... ('e::environment)⊢'t⟧"
```

Section 1.9.1 page 30 **(265) definition**
```
  "SUBSTITUTEk n (p::⟦...'e1::environment⊢'t1::program_type⟧)
                 (q::⟦...'e2::environment⊢'t2::program_type⟧) ==
```

```
  (Abs_program (substitute' n (Rep_program p) (Rep_program q)))
  :: ⟦...'e2::environment⊢'t1::program_type⟧"
```

(266) definition
```
  "SWAPn n ((p::⟦...'e1::environment⊢'t::program_type⟧)) ==
  (Abs_program (swap_vars n (Suc n) (Rep_program p))) :: ⟦...'e2::environment⊢'t⟧"
```

(267) definition
```
  "WEAKEN (p::⟦...env_nil⊢('t::program_type)⟧) ==
  (Abs_program (Rep_program p)) :: ⟦... ('e::environment)⊢'t⟧"
```

(268) definition
```
  "prog_probability (P::⟦bool⟧) ==
  kernel_prob_of denotation (prog_embedding P,[],[]) (InlP`UNIV × UNIV)"
```

Section 1.9.4
page 33

Program Representation Rules

(269) lemma Rep_program_type:
```
  "(env_types TYPE('e))|[] ⊢ (Rep_program (p::⟦...'e::environment⊢'t::program_type⟧))
                           : (prog_type TYPE('t))"
```

(270) lemma Rep_program_rule_VAR0: "Rep_program VAR0 == Var 0"

(271) lemma Rep_program_rule_VAR_SUC:
```
  assumes "Rep_program B == Var n"
  shows "Rep_program (VAR_SUC B::⟦'a::program_type...'e::environment⊢'t::program_type⟧)

  == Var (Suc n)"
```

(272) lemma Rep_program_rule_ABSTRACT:
```
  fixes p :: "⟦'in::program_type...'e::environment⊢'out::program_type⟧"
  shows "Rep_program (ABSTRACT n p) == Abstraction (Rep_program p)"
```

(273) lemma Rep_program_rule_APPLY:
```
  fixes p1 :: "⟦...'e::environment⊢'in::program_type⇒'out::program_type⟧"
  and   p2 :: "⟦...'e⊢'in⟧"
  shows "Rep_program (APPLY p1 p2) == Application (Rep_program p1) (Rep_program p2)"
```

(274) lemma Rep_program_rule_PAIR:
```
  fixes p1 :: "⟦...'e::environment⊢'t1::program_type⟧"
  and   p2 :: "⟦...'e⊢'t2::program_type⟧"
  shows "Rep_program (PAIR p1 p2) == PairP (Rep_program p1) (Rep_program p2)"
```

(275) lemma Rep_program_rule_FST:
```
  fixes p :: "⟦...'e::environment⊢('a::program_type)×('b::program_type)⟧"
  shows "Rep_program (FST p) == Fst (Rep_program p)"
```

(276) lemma Rep_program_rule_SND:
```
  fixes p :: "⟦...'e::environment⊢('a::program_type)×('b::program_type)⟧"
  shows "Rep_program (SND p) == Snd (Rep_program p)"
```

(277) lemma Rep_program_rule_CASE:
```
  fixes p1 :: "⟦...'e::environment⊢'t1::program_type + 't2::program_type⟧"
  and   p2 :: "⟦...'e⊢'t1 ⇒ 't::program_type⟧"
  and   p3 :: "⟦...'e⊢'t2 ⇒ 't::program_type⟧"
  shows "Rep_program (CASE p1 p2 p3) ==
         CaseP (Rep_program p1) (Rep_program p2) (Rep_program p3)"
```

(278) lemma Rep_program_rule_INL:
```
  fixes p :: "⟦...'e::environment⊢'a::program_type⟧"
  shows "Rep_program ((INL p)::⟦...'e⊢('a)+('b::program_type)⟧) ==
         InlP (Rep_program p)"
```

```
(279) lemma Rep_program_rule_INR:
  fixes p :: "⟦...'e::environment⊢'b::program_type⟧"
  shows "Rep_program ((INR p)::⟦...'e⊢('a::program_type)+('b)⟧) ==
         InrP (Rep_program p)"

(280) lemma Rep_program_rule_VALUE:
  fixes v :: "'a::embeddable"
  shows "Rep_program ((VALUE v)::⟦...'e::environment⊢_⟧) == prog_embedding v"

(281) lemma Rep_program_VALUE:
  fixes v :: "⟦'a::program_type⟧"
  shows "Rep_program ((VALUE v)::⟦...'e::environment⊢_⟧) = Rep_program v"

(282) lemma Rep_program_rule_FUNCTION:
  fixes f :: "('a::embeddable_pure,'b::embeddable_pure)submarkov_kernel"
  and   p :: "⟦...'e::environment⊢'a::embeddable_pure⟧"
  shows "Rep_program (FUNCTION f p) ==
         Function (kernel2puretermkernel f) (Rep_program p)"

(283) lemma Rep_program_rule_REF:
  fixes p :: "⟦...'e::environment⊢'t::program_type⟧"
  shows "Rep_program (REF p) == Ref (Rep_program p)"

(284) lemma Rep_program_rule_DEREF:
  fixes p :: "⟦...'e::environment⊢('t::program_type) reference⟧"
  shows "Rep_program (DEREF p) == Deref (Rep_program p)"

(285) lemma Rep_program_rule_ASSIGN:
  fixes p1 :: "⟦...'e::environment⊢('t::program_type) reference⟧"
  and   p2 :: "⟦...'e⊢'t⟧"
  shows "Rep_program (ASSIGN p1 p2) == Assign (Rep_program p1) (Rep_program p2)"

(286) lemma Rep_program_rule_EVENT:
  shows "Rep_program ((EVENT ev) :: ⟦...'e::environment⊢unit⟧) == Event ev"

(287) lemma Rep_program_rule_NIL:
  shows "Rep_program (NIL::⟦...'e::environment⊢'t::program_type list⟧) == Nil'P"

(288) lemma Rep_program_rule_CONS:
  fixes p1 :: "⟦...'e::environment⊢'t1::program_type⟧"
  and   p2 :: "⟦...'e⊢'t1 list⟧"
  shows "Rep_program (CONS p1 p2) == List'P (Rep_program p1) (Rep_program p2)"

(289) lemma Rep_program_rule_IFTHENELSE:
  fixes c  :: "⟦...'e::environment⊢bool⟧"
  and   p1 :: "⟦...'e⊢'t::program_type⟧"
  and   p2 :: "⟦...'e⊢'t⟧"
  shows "Rep_program (IFTHENELSE c p1 p2) ==
         IfThenElse' (Rep_program c) (Rep_program p1) (Rep_program p2)"

(290) lemma Rep_program_rule_FIX:
  shows "Rep_program (FIX::
  ⟦...'e::environment⊢(('a::program_type ⇒ 'b::program_type)⇒('a ⇒ 'b))⇒('a ⇒ 'b)⟧)
== Fix"

(291) lemma Rep_program_rule_LET:
  fixes p1 :: "⟦...'e::environment⊢'in::program_type⟧"
  and   p2 :: "⟦...'e⊢'in⇒'out::program_type⟧"
  shows "Rep_program (LET p1 p2) == LetP (Rep_program p1) (Rep_program p2)"

(292) lemma Rep_program_rule_SEQUENCE:
  fixes p1 :: "⟦...'e::environment⊢'t1::program_type⟧"
  and   p2 :: "⟦...'e⊢'t2::program_type⟧"
  shows "Rep_program (SEQUENCE p1 p2) == SequenceP (Rep_program p1) (Rep_program p2)"
```

(293) lemma Rep_program_rule_UNCURRY:
fixes p ::
"⟦...'e::environment⊢('t1::program_type⇒'t2::program_type⇒'t::program_type)⟧"
shows "Rep_program (UNCURRY p) == uncurry (Rep_program p)"

(294) lemma Rep_program_rule_LIFT:
fixes p :: "⟦...'e::environment⊢'t::program_type⟧"
shows "Rep_program (LIFT p::⟦'a::program_type... 'e⊢'t⟧) ==
lift_vars 0 (Rep_program p)"

(295) lemma Rep_program_rule_SUBSTITUTE:
fixes p :: "⟦'a::program_type...'e::environment⊢'t::program_type⟧"
and q :: "⟦...'e::environment⊢'a⟧"
shows "Rep_program (SUBSTITUTE p q) == substitute' 0 (Rep_program p) (Rep_program q)"

(296) lemma Rep_program_rule_SWAP:
fixes p :: "⟦'a::program_type,'b::program_type...'e::environment⊢'t::program_type⟧"
shows "Rep_program (SWAP p::⟦'b,'a... 'e⊢'t⟧) == swap_vars 0 1 (Rep_program p)"

(297) lemma Rep_program_rule_WEAKEN:
fixes p :: "⟦...env_nil⊢('t::program_type)⟧"
shows "Rep_program (WEAKEN p::⟦... ('e::environment)⊢'t⟧) == (Rep_program p)"

(298) lemma Rep_program_rule_LIFT_WEAKEN:
"Rep_program (LIFT (WEAKEN P::⟦... 'e::environment ⊢ 't::program_type⟧)) ==
Rep_program P"

(299) lemma progress_typed:
assumes "¬ is_value (Rep_program (P::⟦'a::program_type⟧))"
shows "∃μ. (Rep_program P,[],[]) ⇝ μ"

I.6.4 Typed Contexts and Context Functions

(300) typedef ('ein,'tin,'eout,'tout) "typed_context" = Section 1.9.3 page 32
"{c. context_typing (env_types(TYPE('ein::environment))) []
(prog_type TYPE('tin::program_type)) (env_types(TYPE('eout::environment))) []
c (prog_type(TYPE('tout::program_type)))}"

(301) definition applycontext_typed ::
"('ein::environment,'tin::program_type,
'eout::environment,'tout::program_type) typed_context
⇒ ⟦...'ein::environment⊢'tin::program_type⟧
⇒ ⟦...'eout::environment⊢'tout::program_type⟧" **where**
"applycontext_typed C P =
Abs_program (applycontext (Rep_typed_context C) (Rep_program P))"

(302) definition is_contextfun_typed :: Definition 1.31 page 32
"(⟦... 'ein::environment ⊢ 'tin::program_type⟧
⇒ ⟦... 'eout::environment ⊢ 'tout::program_type⟧) ⇒ bool" **where**
"is_contextfun_typed F = (∃C. applycontext_typed C = F)"

(303) lemma is_contextfun_typed_id: Section 1.9.3 page 32
"is_contextfun_typed (λx. x::⟦...'e::environment⊢'t::program_type⟧)"

(304) lemma is_contextfun_typed_const: Section 1.9.3 page 33
"is_contextfun_typed (λx. u)"

(305) lemma is_contextfun_typed_ABSTRACT: Section 1.9.3 page 33
assumes "is_contextfun_typed F"
shows "is_contextfun_typed (λx. ABSTRACT n (F x))"

Section 1.9.3 page 33

(306) lemma is_contextfun_typed_APPLY:
  assumes "is_contextfun_typed F1"
  assumes "is_contextfun_typed F2"
  shows "is_contextfun_typed (λx. APPLY (F1 x) (F2 x))"

(307) lemma is_contextfun_typed_PAIR:
  assumes "is_contextfun_typed F1"
  assumes "is_contextfun_typed F2"
  shows "is_contextfun_typed (λx. PAIR (F1 x) (F2 x))"

(308) lemma is_contextfun_typed_FST:
  assumes "is_contextfun_typed F"
  shows "is_contextfun_typed (λx. FST (F x))"

(309) lemma is_contextfun_typed_SND:
  assumes "is_contextfun_typed F"
  shows "is_contextfun_typed (λx. SND (F x))"

(310) lemma is_contextfun_typed_CASE:
  assumes "is_contextfun_typed F1"
  assumes "is_contextfun_typed F2"
  assumes "is_contextfun_typed F3"
  shows "is_contextfun_typed (λx. CASE (F1 x) (F2 x) (F3 x))"

(311) lemma is_contextfun_typed_INL:
  assumes "is_contextfun_typed F"
  shows "is_contextfun_typed ((λx. INL (F x))::
    ⟦...'ein⊢'tin⟧ ⇒ ⟦...'eout⊢('t1out+'t2out::program_type)⟧)"

(312) lemma is_contextfun_typed_INR:
  assumes "is_contextfun_typed F"
  shows "is_contextfun_typed ((λx. INR (F x))::
    ⟦...'ein⊢'tin⟧ ⇒ ⟦...'eout⊢('t1out::program_type+'t2out)⟧)"

(313) lemma is_contextfun_typed_FUNCTION:
  assumes "is_contextfun_typed F"
  assumes "∀x. ∀y←apply_kernel (kernel2puretermkernel f) x.
          []|[]⊢pureterm_to_term y:(prog_type TYPE('fout))"
  shows "is_contextfun_typed (λx. FUNCTION f (F x))"

(314) lemma is_contextfun_typed_REF:
  assumes "is_contextfun_typed F"
  shows "is_contextfun_typed (λx. REF (F x))"

(315) lemma is_contextfun_typed_DEREF:
  assumes "is_contextfun_typed F"
  shows "is_contextfun_typed (λx. DEREF (F x))"

(316) lemma is_contextfun_typed_ASSIGN:
  assumes "is_contextfun_typed F1"
  assumes "is_contextfun_typed F2"
  shows "is_contextfun_typed (λx. ASSIGN (F1 x) (F2 x))"

(317) lemma is_contextfun_typed_CONS:
  assumes "is_contextfun_typed F1"
  and "is_contextfun_typed F2"
  shows "is_contextfun_typed (λx. CONS (F1 x) (F2 x))"

(318) lemma is_contextfun_typed_LET:
  assumes "is_contextfun_typed F1"
  assumes "is_contextfun_typed F2"
  shows "is_contextfun_typed (λx. LET (F1 x) (F2 x))"

(319) lemma is_contextfun_typed_UNCURRY:
assumes "is_contextfun_typed F"
shows "is_contextfun_typed (λx. UNCURRY (F x))"

(320) lemma is_contextfun_typed_SEQUENCE:
assumes "is_contextfun_typed F1"
assumes "is_contextfun_typed F2"
shows "is_contextfun_typed (λx. SEQUENCE (F1 x) (F2 x))"

I.6.5 Syntax for Typed Programs

syntax (xsymbols)
"_pretty_program" :: "pretty_program ⇒ ⟦...'e⊢'a⟧" ("¨_¨")
"_program_pretty" :: "(⟦...env_nil⊢'a⟧) ⇒ pretty_program" (":_:")

"" :: "pretty_program ⇒ pretty_program" ("'(_')")
"_NamedVAR" :: "id ⇒ pretty_program" ("_")
"_VALUE" :: "'a ⇒ pretty_program" ("^_")
"_FUNCTION" :: "[('a, 'b) submarkov_kernel, pretty_program] ⇒ pretty_program" ("[_]
_")
"_CONS" :: "[pretty_program,pretty_program] ⇒ pretty_program" ("_ :: _")
"_NIL" :: "pretty_program" ("nil")
"_ABSTRACT" :: "[pttrn,pretty_program] ⇒ pretty_program" ("λ_. (_)")
"_APPLY" :: "[pretty_program,pretty_program] ⇒ pretty_program" ("_ _")
"_LOCATION" :: "nat ⇒ pretty_program" ("loc _")
"_REF" :: "[pretty_program] ⇒ pretty_program" ("ref _")
"_DEREF" :: "[pretty_program] ⇒ pretty_program" ("! _")
"_ASSIGN" :: "[pretty_program,pretty_program] ⇒ pretty_program" ("_ := _")
"_FIX" :: "[pretty_program] ⇒ pretty_program" ("fix _")
"_IFTHENELSE" :: "[pretty_program,pretty_program,pretty_program] ⇒ pretty_program" ("if
(_) then (_) else (_)")
"_EVENT" :: "eventT ⇒ pretty_program" ("event _")
"_EVENTLIST" :: "pretty_program" ("eventlist")
"_FST" :: "[pretty_program] ⇒ pretty_program" ("#1 _")
"_SND" :: "[pretty_program] ⇒ pretty_program" ("#2 _")
"_SPLITLIST" :: "[pretty_program,pretty_program,pretty_program] ⇒ pretty_program" ("case
(_) of (_) (_)")
"_SEQUENCE" :: "[pretty_program,pretty_program] ⇒ pretty_program" ("_; _")
"_LIFT" :: "[pretty_program] ⇒ pretty_program" ("⌈_⌉")

"_programbind":: "[pttrn, pretty_program] => programletbind" ("(2_ ←/ _)")
"_programsetbind" :: "[pttrn, pretty_program] => programletbind" ("(2_ ←$/ _)")
"" :: "programletbind => programletbinds" ("_")
"_programbinds" :: "[programletbind, programletbinds] => programletbinds" ("_;/ _")
"_programLet" :: "[programletbinds, pretty_program] => pretty_program" ("(let (_)/ in
(_))")

"_programtuple" :: "pretty_program => programtuple_args => pretty_program" ("(1'(_,/
_'))")
"_programtuple_arg" :: "pretty_program => programtuple_args" ("_")
"_programtuple_args" :: "pretty_program=> programtuple_args => programtuple_args" ("_,/
_")

"_prog_probability" :: "pretty_program ⇒ real" ("Pr[(_)]")
"_prog_letprobability" :: "pretty_program ⇒ programletbinds ⇒ real" ("Pr[(_) : (_)]")
"_OPEQ" :: "pretty_program ⇒ pretty_program ⇒ pretty_program" (**infixl** "=")

I.7 Program Relations

I.7.1 Denotational Equivalence

Definitions

Definition 1.32 page 34

(321) definition
```
"denotationally_equivalent p p' ==
∀σ η. apply_kernel denotation (p,σ,η) = apply_kernel denotation (p',σ,η)"
```

Definition 1.33 page 34

(322) definition
```
"denotationally_equivalent_upto ε A B ==
∀σ η. ∀pset∈Σ. |kernel_prob_of denotation (A,σ,η) pset
                - kernel_prob_of denotation (B,σ,η) pset| ≤ ε"
```

(323) definition applycontext_kernel_def_raw:
```
"applycontext_kernel C = deterministic_kernel (λ(P,σ,η). (applycontext C P, σ, η))"
```

(324) lemma applycontext_kernel_def:
```
"apply_kernel (applycontext_kernel C) x =
 apply_kernel unitkernel ((λ(P,σ,η). (applycontext C P, σ, η)) x)"
```

Lemmas

(325) lemma denotationally_equivalent_upto_refl:
assumes "ε ≥ 0"
shows "denotationally_equivalent_upto ε A A"

(326) lemma denotationally_equivalent_upto_sym:
assumes "denotationally_equivalent_upto ε A B"
shows "denotationally_equivalent_upto ε B A"

(327) lemma denotationally_equivalent_upto_trans:
assumes "denotationally_equivalent_upto ε1 A B"
assumes "denotationally_equivalent_upto ε2 B C"
shows "denotationally_equivalent_upto (ε1+ε2) A C"

Theorem 1.48 page 41

(328) theorem denot_equiv_eval_ctx':
fixes E P σ η
assumes Eeval: "E : evaluationcontext"
shows "apply_kernel denotation (applycontext E P, σ, η) =
lift_kernel(mk_kernel(λ(V',σ',η'). apply_kernel denotation(applycontext E V',σ',η')))
           (apply_kernel denotation (P,σ,η))"

(329) theorem denot_equiv_eval_ctx:
fixes E P σ η
assumes "E : evaluationcontext"
shows "apply_kernel denotation (applycontext E P, σ, η) =
lift_kernel denotation (lift_kernel (applycontext_kernel E)
                        (apply_kernel denotation (P,σ,η)))"

Lemma 1.50 page 43

(330) lemma chaining_denotation_beta:
shows "(apply_kernel denotation (Application (Abstraction P') P,se))
     = lift_kernel denotation (lift_kernel
       (deterministic_kernel (λ(v,se). (substitute' 0 P' v,se)))
       (apply_kernel denotation (P,se)))"

(331) lemma chaining_denotation_beta':
```
"apply_kernel denotation (Application (Abstraction P') P,σ,η) =
lift_kernel(mk_kernel(λ(V',σ',η'). apply_kernel denotation(substitute' 0 P' V',σ',η')))
            (apply_kernel denotation (P,σ,η))"
```

(332) lemma denot_equiv_stepsto_unitkernel:
assumes Punit: "pse ⇝ apply_kernel unitkernel pse'"
shows "apply_kernel denotation pse = apply_kernel denotation pse'"

(333) lemma termination_equivalent_closingcontext:
assumes "freevars p = {}"
shows "kernel_prob_of denotation (applycontext (closingcontext n) p, se) UNIV
= kernel_prob_of denotation (p, se) UNIV"

(334) lemma termination_equivalent_extend_store:
assumes "storageclosed (p, s, e)"
shows "kernel_prob_of denotation (p, s @ replicate l value_unit, e) UNIV
= kernel_prob_of denotation (p, s, e) UNIV"

(335) lemma termination_equivalent_closingconfiguration:
assumes fc: "fullyclosed (p, s, e)"
shows "kernel_prob_of denotation (applycontext (closingcontext n) p,
s@(replicate l value_unit), e) UNIV
= kernel_prob_of denotation (p, s, e) UNIV"

I.7.2 Observational Equivalence

Definitions

(336) definition "observationally_equivalent_untyped P Q ==
∀C s e. are_values s ∧
fullyclosed (applycontext C P, s, e) ∧ fullyclosed (applycontext C Q, s, e)
⟶ kernel_prob_of denotation (applycontext C P, s, e) UNIV
= kernel_prob_of denotation (applycontext C Q, s, e) UNIV"

Definition 1.34
page 35

(337) definition
"observationally_approximated_untyped P Q ==
∀C s e. are_values s ∧
fullyclosed (applycontext C P, s, e) ∧ fullyclosed (applycontext C Q, s, e)
⟶ kernel_prob_of denotation (applycontext C P, s, e) UNIV
≤ kernel_prob_of denotation (applycontext C Q, s, e) UNIV"

Definition 1.59
page 46

(338) lemma observationally_equivalent_untyped_def2:
"observationally_equivalent_untyped P Q ==
observationally_approximated_untyped P Q ∧ observationally_approximated_untyped Q P"

Lemma 1.61
page 46

(339) definition "observationally_equivalent
(P::⟦...'e::environment⊢'t::program_type⟧) (Q::⟦...'e⊢'t⟧) =
observationally_equivalent_untyped (Rep_program P) (Rep_program Q)"

(340) lemma obseq_imp_probeq:
assumes "observationally_equivalent P Q"
shows "prog_probability P = prog_probability Q"

Lemma 1.37
page 36

All Variants are Equivalence Relations

(341) lemma observationally_equivalent_untyped_refl[simp]:
"observationally_equivalent_untyped P P"

(342) lemma observationally_equivalent_refl[simp]:
"observationally_equivalent P P"

(343) lemma observationally_equivalent_untyped_sym:
"observationally_equivalent_untyped P Q = observationally_equivalent_untyped Q P"

(344) lemma observationally_equivalent_sym:
```
"observationally_equivalent P Q = observationally_equivalent Q P"
```

(345) lemma observationally_equivalent_untyped_trans:
```
assumes "observationally_equivalent_untyped P Q"
and "observationally_equivalent_untyped Q R"
shows "observationally_equivalent_untyped P R"
```

(346) lemma observationally_equivalent_trans:
```
assumes "observationally_equivalent P Q"
and "observationally_equivalent Q R"
shows "observationally_equivalent P R"
```

Lemma 1.35 page 35

(347) lemma observationally_equivalent_untyped_equiv:
```
"equiv UNIV (split observationally_equivalent_untyped)"
```

(348) lemma observationally_equivalent_equiv:
```
"equiv UNIV (split observationally_equivalent)"
```

Composability

Lemma 1.36 page 36

(349) lemma obseq_untyped_applycontext:
```
assumes eqAB: "observationally_equivalent_untyped A B"
shows "observationally_equivalent_untyped (applycontext C A) (applycontext C B)"
```

(350) lemma obseq_untyped_contextfuns:
```
assumes eqAB: "observationally_equivalent_untyped A B"
and conF: "F : contextfuns"
shows "observationally_equivalent_untyped (F A) (F B)"
```

(351) lemma obseq_contextfun_typed:
```
assumes eqAB: "observationally_equivalent A B"
and conF: "is_contextfun_typed F"
shows "observationally_equivalent (F A) (F B)"
```

I.7.3 Polynomial Runtime

Asymptotics

(352) definition
```
"polynomially_bounded (p::nat⇒nat) == ∃(a::nat) (b::nat). ∀n. p n ≤ (n^a)+b"
```

Section 1.10.4 page 39

(353) definition
```
"negligible (f::nat⇒real) ==
(∀(c::nat).∃(N::nat).∀n≥N. abs (f n) < 1/(real(n^c)))"
```

Section 1.10.4 page 39

(354) lemma negligible_zero: "negligible (λn.0)"

Section 1.10.4 page 39

(355) lemma negligible_add:
```
"⟦negligible f; negligible g⟧ ⟹ negligible (λn.(f n + g n))"
```

Section 1.10.4 page 39

(356) lemma negligible_pos_le:
```
assumes "negligible f"
assumes "!!n. g n ≤ f n"
assumes "!!n. 0 ≤ g n"
shows "negligible g"
```

(357) lemma negligible_mul:
```
assumes neglf: "negligible f"
assumes neglg: "negligible g"
shows "negligible (λn.(f n * g n))"
```

Definitions

(358) definition `"stepname == ''step''"`

(359) definition `eventstep :: "programterm ⇒ programterm" where`
```
  "eventstep p = Application (Abstraction (lift_vars 0 p)) (Event stepname)"
```

(360) function `annotate_eventsteps :: "programterm ⇒ programterm" where`
```
  "annotate_eventsteps (Function fun p)    =
    eventstep(Function fun (annotate_eventsteps p))"
| "annotate_eventsteps (PairP p1 p2)       =
    eventstep(PairP (annotate_eventsteps p1) (annotate_eventsteps p2))"
| "annotate_eventsteps (Abstraction p)     =
    eventstep(Abstraction (annotate_eventsteps p))"
| "annotate_eventsteps (Application p1 p2) =
    eventstep(Application (annotate_eventsteps p1) (annotate_eventsteps p2))"
| "annotate_eventsteps (Ref p)             = eventstep(Ref (annotate_eventsteps p))"
| "annotate_eventsteps (Deref p)           = eventstep(Deref (annotate_eventsteps p))"
| "annotate_eventsteps (Assign p1 p2)      =
    eventstep(Assign (annotate_eventsteps p1) (annotate_eventsteps p2))"
| "annotate_eventsteps (Fst p)             = eventstep(Fst (annotate_eventsteps p))"
| "annotate_eventsteps (Snd p)             = eventstep(Snd (annotate_eventsteps p))"
| "annotate_eventsteps (Var n)             = eventstep(Var n)"
| "annotate_eventsteps (Value b)           = eventstep(Value b)"
| "annotate_eventsteps (Location l)        = eventstep(Location l)"
| "annotate_eventsteps (Event e)           = eventstep(Event e)"
| "annotate_eventsteps (EventList)         = eventstep(EventList)"
| "annotate_eventsteps (Fold p)            = eventstep(Fold (annotate_eventsteps p))"
| "annotate_eventsteps (Unfold p)          = eventstep(Unfold (annotate_eventsteps p))"
| "annotate_eventsteps (CaseP p1 p2 p3)    = eventstep(CaseP (annotate_eventsteps p1)
    (annotate_eventsteps p2) (annotate_eventsteps p3))"
| "annotate_eventsteps (InlP p)            = eventstep(InlP (annotate_eventsteps p))"
| "annotate_eventsteps (InrP p)            = eventstep(InrP (annotate_eventsteps p))"
```

Section 1.10.3 page 37

(361) lemma `annotate_type_preservation:`
```
  assumes "Γ|Θ⊢p:T"
  shows "Γ|Θ⊢ (annotate_eventsteps p):T"
```

(362) definition
```
  "ANNOTATE_EVENTSTEPS (p::⟦... 'e::environment⊢'t::program_type⟧)
  ==
  (Abs_program (annotate_eventsteps (Rep_program p))
  ::
  ⟦...'e⊢'t⟧)"
```

(363) lemma `Rep_program_rule_ANNOTATE_EVENTSTEPS:`
```
  fixes p :: "⟦...'e::environment⊢'t::program_type⟧"
  shows "Rep_program (ANNOTATE_EVENTSTEPS p) == annotate_eventsteps (Rep_program p)"
  (is "?DEF == ?def")
```

(364) definition `count_eventsteps :: "eventT list ⇒ nat" where`
```
"count_eventsteps l = length (filter (λe. e=stepname) l)"
```

(365) definition `max_eventsteps ::"(programterm×state)measureT ⇒ nat ⇒ bool" where`
```
"max_eventsteps M n = (M {(p,s,e). count_eventsteps e ≤ n} = 1)"
```

(366) fun `unary_parameter :: "nat ⇒ ⟦...'e::environment⊢bitstring⟧" where`
```
  "unary_parameter 0       = NIL"
| "unary_parameter (Suc n) = CONS (VALUE True) (unary_parameter n)"
```

(367) fun `programterm_size :: "programterm ⇒ nat" where`
```
  "programterm_size (Value v) = 1"
| "programterm_size (PairP p1 p2) = programterm_size p1 + programterm_size p2"
| "programterm_size (InlP p) = programterm_size p"
```

```
| "programterm_size (InrP p) = programterm_size p"
| "programterm_size (Fold p) = programterm_size p"
```

(368) function subterms :: "programterm ⇒ programterm set" **where**
```
  "subterms (Var n)             = {Var n}"
| "subterms (Value v)           = {Value v}"
| "subterms (Function f p)      = {Function f p} ∪ subterms p"
| "subterms (PairP p1 p2)       = {PairP p1 p2} ∪ subterms p1 ∪ subterms p2"
| "subterms (Abstraction p)     = {Abstraction p} ∪ subterms p"
| "subterms (Application p1 p2) = {Application p1 p2} ∪ subterms p1 ∪ subterms p2"
| "subterms (Location i)        = {Location i}"
| "subterms (Ref p)             = {Ref p} ∪ subterms p"
| "subterms (Deref p)           = {Deref p} ∪ subterms p"
| "subterms (Assign p1 p2)      = {Assign p1 p2} ∪ subterms p1 ∪ subterms p2"
| "subterms (Event e)           = {Event e}"
| "subterms (EventList)         = {EventList}"
| "subterms (Fst p)             = {Fst p} ∪ subterms p"
| "subterms (Snd p)             = {Snd p} ∪ subterms p"
| "subterms (CaseP p1 p2 p3)    =
    {CaseP p1 p2 p3} ∪ subterms p1 ∪ subterms p2 ∪ subterms p3"
| "subterms (InlP p)            = {InlP p} ∪ subterms p"
| "subterms (InrP p)            = {InrP p} ∪ subterms p"
| "subterms (Fold p)            = {Fold p} ∪ subterms p"
| "subterms (Unfold p)          = {Unfold p} ∪ subterms p"
```

(369) definition subterm_filter :: "programterm set ⇒ programterm set" **where**
```
  "subterm_filter M = {p. subterms p ∩ M = {}}"
```

(370) definition
```
"not_kernel == deterministic_kernel (λb. ¬ b)"
```

(371) definition
```
"and_kernel == deterministic_kernel (λ(b1,b2). b1 ∧ b2)"
```

(372) definition
```
"or_kernel == deterministic_kernel (λ(b1,b2). b1 ∨ b2)"
```

(373) definition
```
"cointoss == mk_kernel2 (λu::unit. uniform_distribution {True, False})"
```

Definition 1.38 page 37 **(374) definition**
```
  "non_computational_atoms ==
  {Value x| x. x ∉ {bvBool True, bvBool False, bvUnit}} ∪ {EventList} ∪
  {Function f t| f t. f ∉ {(kernel2puretermkernel not_kernel),
                           (kernel2puretermkernel and_kernel),
                           (kernel2puretermkernel or_kernel),
                           (kernel2puretermkernel cointoss)}}"
```

Definition 1.38 page 37 **(375) definition**
```
  "is_computational_program p ==
   Rep_program p : subterm_filter non_computational_atoms"
```

(376) definition
```
  "storeless_programs =
   subterm_filter {p | p p1 p2. p=Ref p1 ∨ p=Deref p1 ∨ p=Assign p1 p2}"
```

Definition 1.38 page 37 **(377) definition**
```
  "eventless_programs = subterm_filter {p | p e. p=EventList ∨ p=Event e}"
```

(378) definition
```
  "stateless_programs = storeless_programs ∩ eventless_programs"
```

Definition 1.39 page 38 **(379) definition** polynomial_time ::
```
  "(⟦...env_nil⊢('a::embeddable_pure) × ('b::program_type) ⇒ ('c::program_type)⟧)
  ⇒ bool" where
  "polynomial_time prog ==
  (is_computational_program prog) ∧
```

```
  (∃p. (polynomially_bounded p) ∧
  (∀(pt::'a) (B::⟦...env_nil⊢'b⟧). Rep_program B : eventless_programs ⟶
   (∀n. max_eventsteps (kernel_prob_of (nsteps n)
        (Rep_program (¨:ANNOTATE_EVENTSTEPS prog:(^pt, :B:)¨::⟦...env_nil⊢_⟧),[],[]))

        (p (programterm_size (prog_embedding pt))))))"
```

(380) definition `first_order_polynomial_time ::`
`"(⟦...env_nil⊢('a::embeddable_pure) ⇒ ('c::program_type)⟧) ⇒ bool"` **where**
`"first_order_polynomial_time prog ==`
`polynomial_time (¨λxu. :prog: (#1 xu)¨ :: ⟦...env_nil⊢'a × unit ⇒ 'c⟧)"`

Section 1.10.3 page 38

(381) definition
`"efficient_algorithm f ==`
`(first_order_polynomial_time f) ∧ (Rep_program f : stateless_programs)"`

Section 1.10.3 page 38

(382) definition
`"efficiently_computable f == ∃f'.`
`(efficient_algorithm f') ∧`
`(observationally_equivalent f' (VALUE f))"`

Section 1.10.3 page 38

(383) definition `nonuniform_polynomial_time ::`
`"(nat ⇒ ⟦...env_nil⊢('b::program_type) ⇒ ('c::program_type)⟧) ⇒ bool"` **where**
`"nonuniform_polynomial_time prog ==`
`(∀k. is_computational_program (prog k)) ∧`
`(∃p. (polynomially_bounded p) ∧`
`(∀k (B::⟦...env_nil⊢'b⟧). Rep_program B : eventless_programs ⟶`
`(∀n. max_eventsteps (kernel_prob_of (nsteps n)`
`(Rep_program (¨:ANNOTATE_EVENTSTEPS (prog k): :B:¨::⟦...env_nil⊢_⟧),[],[]))`
`(p k))))"`

Definition 1.40 page 38

I.7.4 Computational Indistinguishability

(384) definition `"computationally_indistinguishable P Q ==`
`∀D. nonuniform_polynomial_time D ⟶ negligible`
`(λk. |Pr[:D k: (:P k:)] - Pr[:D k: (:Q k:)]|)"`

Definition 1.41 page 39

(385) lemma `computationally_indistinguishable_refl:`
`"computationally_indistinguishable P P"`

(386) lemma `computationally_indistinguishable_sym:`
assumes `"computationally_indistinguishable P Q"`
shows `"computationally_indistinguishable Q P"`

(387) lemma `computationally_indistinguishable_trans:`
assumes `"computationally_indistinguishable P Q"`
assumes `"computationally_indistinguishable Q R"`
shows `"computationally_indistinguishable P R"`

(388) lemma `computationally_indistinguishable_equiv:`
`"equiv UNIV (split computationally_indistinguishable)"`

Lemma 1.42 page 39

(389) lemma `obseq_imp_comp_indist:`
assumes `"!!n. observationally_equivalent (P n) (Q n)"`
shows `"computationally_indistinguishable P Q"`

Lemma 1.43 page 39

I.8 The CIU Theorem

I.8.1 Generalized Program Terms

Definitions

(390) datatype `'a gp_epsinstruction =`

```
  GPESubst nat 'a
| GPELift nat
```

(391) datatype generalized_programtermT =

```
  GPEpsilon "(generalized_programtermT gp_epsinstruction) list"
| GPHole
| GPVar nat
| GPValue basicvalue
| GPLocation nat
| GPEvent eventT
| GPEventList
| GPFunction "(pureterm, pureterm) submarkov_kernel" "generalized_programtermT"
| GPPairP "generalized_programtermT" "generalized_programtermT"
| GPAbstraction "generalized_programtermT"
| GPApplication "generalized_programtermT" "generalized_programtermT"
| GPRef "generalized_programtermT"
| GPDeref "generalized_programtermT"
| GPAssign "generalized_programtermT" "generalized_programtermT"
| GPFst "generalized_programtermT"
| GPSnd "generalized_programtermT"
| GPFold "generalized_programtermT"
| GPUnfold "generalized_programtermT"
| GPCase "generalized_programtermT" "generalized_programtermT" "generalized_programtermT"
| GPInl "generalized_programtermT"
| GPInr "generalized_programtermT"
```

(392) types generalized_instantiationT = "(generalized_programtermT gp_epsinstruction) list"

(393) types generalized_configurationT =

```
  "generalized_programtermT × programterm × generalized_programtermT list × eventT list"
```

Definition 1.63 page 47

(394) inductive_set

```
  generalized_programterm :: "generalized_programtermT set"
  and generalized_value :: "generalized_programtermT set"
  and generalized_instantiation :: "generalized_instantiationT set" where
  — generalized programterms
  gp_GPEpsilon: "p : generalized_instantiation ⟹ GPEpsilon p : generalized_programterm"
| gp_GPVar: "GPVar n : generalized_programterm"
| gp_GPValue: "GPValue basicvalue : generalized_programterm"
| gp_GPLocation: "GPLocation n : generalized_programterm"
| gp_GPEvent: "GPEvent eventT : generalized_programterm"
| gp_GPEventList: "GPEventList : generalized_programterm"
| gp_GPFunction: "p : generalized_programterm ⟹
    GPFunction f p : generalized_programterm"
| gp_GPPairP: "⟦p1 : generalized_programterm; p2 : generalized_programterm⟧ ⟹
    GPPairP p1 p2 : generalized_programterm"
| gp_GPAbstraction: "⟦p : generalized_programterm⟧ ⟹
    GPAbstraction p : generalized_programterm"
| gp_GPApplication: "⟦p1 : generalized_programterm; p2 : generalized_programterm⟧ ⟹
    GPApplication p1 p2 : generalized_programterm"
| gp_GPRef: "⟦p : generalized_programterm⟧ ⟹ GPRef p : generalized_programterm"
| gp_GPDeref: "⟦p : generalized_programterm⟧ ⟹ GPDeref p : generalized_programterm"
| gp_GPAssign: "⟦p1 : generalized_programterm; p2 : generalized_programterm⟧ ⟹
    GPAssign p1 p2 : generalized_programterm"
| gp_GPFst: "⟦p : generalized_programterm⟧ ⟹ GPFst p : generalized_programterm"
| gp_GPSnd: "⟦p : generalized_programterm⟧ ⟹ GPSnd p : generalized_programterm"
| gp_GPFold: "p : generalized_programterm ⟹ GPFold p : generalized_programterm"
| gp_GPUnfold: "p : generalized_programterm ⟹ GPUnfold p : generalized_programterm"
| gp_GPCase: "⟦p1 : generalized_programterm; p2 : generalized_programterm;
    p3 : generalized_programterm⟧ ⟹ GPCase p1 p2 p3 : generalized_programterm"
| gp_GPInl: "p : generalized_programterm ⟹ GPInl p : generalized_programterm"
| gp_GPInr: "p : generalized_programterm ⟹ GPInr p : generalized_programterm"
  — generalized values
| gv_GPValue: "GPValue v : generalized_value"
```

```
| gv_GPPairP: "⟦v1 : generalized_value; v2 : generalized_value⟧ ⟹
    GPPairP v1 v2 : generalized_value"
| gv_GPAbstraction: "p : generalized_programterm ⟹
    GPAbstraction p : generalized_value"
| gv_GPLocation: "GPLocation n : generalized_value"
| gv_GPVar: "GPVar n : generalized_value"
| gv_GPFold: "v : generalized_value ⟹ GPFold v : generalized_value"
| gv_GPInl: "v : generalized_value ⟹ GPInl v : generalized_value"
| gv_GPInr: "v : generalized_value ⟹ GPInr v : generalized_value"
  — generalized instantiations
| ga_nil: "[] : generalized_instantiation"
| ga_subst: "⟦v : generalized_value; p : generalized_instantiation⟧ ⟹
    (GPESubst n v)#p : generalized_instantiation"
| ga_lift: "p : generalized_instantiation ⟹ (GPELift n)#p : generalized_instantiation"
```

(395) inductive_set `gp_context :: "generalized_programtermT set"` Definition 1.64 page 47

```
  where
  gc_GPHole: "GPHole : gp_context"
| gc_GPEpsilon: "p : generalized_instantiation ⟹ GPEpsilon p : gp_context"
| gc_GPVar: "GPVar n : gp_context"
| gc_GPValue: "GPValue basicvalue : gp_context"
| gc_GPLocation: "GPLocation n : gp_context"
| gc_GPEvent: "GPEvent eventT : gp_context"
| gc_GPEventList: "GPEventList : gp_context"
| gc_GPFunction: "c : gp_context ⟹ GPFunction f c : gp_context"
| gc_GPPairP: "⟦c1 : gp_context; c2 : gp_context⟧ ⟹ GPPairP c1 c2 : gp_context"
| gc_GPAbstraction: "⟦c : gp_context⟧ ⟹ GPAbstraction c : gp_context"
| gc_GPApplication: "⟦c1 : gp_context; c2 : gp_context⟧ ⟹
    GPApplication c1 c2 : gp_context"
| gc_GPRef: "⟦c : gp_context⟧ ⟹ GPRef c : gp_context"
| gc_GPDeref: "⟦c : gp_context⟧ ⟹ GPDeref c : gp_context"
| gc_GPAssign: "⟦c1 : gp_context; c2 : gp_context⟧ ⟹ GPAssign c1 c2 : gp_context"
| gc_GPFst: "⟦c : gp_context⟧ ⟹ GPFst c : gp_context"
| gc_GPSnd: "⟦c : gp_context⟧ ⟹ GPSnd c : gp_context"
| gc_GPFold: "c : gp_context ⟹ GPFold c : gp_context"
| gc_GPUnfold: "c : gp_context ⟹ GPUnfold c : gp_context"
| gc_GPCase: "⟦c1 : gp_context; c2 : gp_context; c3 : gp_context⟧ ⟹
    GPCase c1 c2 c3 : gp_context"
| gc_GPInl: "c : gp_context ⟹ GPInl c : gp_context"
| gc_GPInr: "c : gp_context ⟹ GPInr c : gp_context"
```

(396) inductive_set `generalized_evalcontext :: "generalized_programtermT set"` Definition 1.65 page 48

```
  where
  ge_GPHole: "GPHole ∈ generalized_evalcontext"
| ge_GPFunction: "⟦e ∈ generalized_evalcontext⟧ ⟹
    (GPFunction f e) ∈ generalized_evalcontext"
| ge_GPPairPl: "⟦e ∈ generalized_evalcontext; p : generalized_programterm⟧ ⟹
    (GPPairP e p) ∈ generalized_evalcontext"
| ge_GPPairPr: "⟦e ∈ generalized_evalcontext; v : generalized_value⟧ ⟹
    (GPPairP v e) ∈ generalized_evalcontext"
| ge_GPApplicationl: "⟦e ∈ generalized_evalcontext; p : generalized_programterm⟧ ⟹
    (GPApplication e p) ∈ generalized_evalcontext"
| ge_GPApplicationr: "⟦e ∈ generalized_evalcontext; v : generalized_value⟧ ⟹
    (GPApplication v e) ∈ generalized_evalcontext"
| ge_GPRef: "⟦e ∈ generalized_evalcontext⟧ ⟹ (GPRef e) ∈ generalized_evalcontext"
| ge_GPDeref: "⟦e ∈ generalized_evalcontext⟧ ⟹ (GPDeref e) ∈ generalized_evalcontext"
| ge_GPAssignl: "⟦e ∈ generalized_evalcontext; p : generalized_programterm⟧ ⟹
    (GPAssign e p) ∈ generalized_evalcontext"
| ge_GPAssignr: "⟦e ∈ generalized_evalcontext; v : generalized_value⟧ ⟹
    (GPAssign v e) ∈ generalized_evalcontext"
| ge_GPFst: "⟦e ∈ generalized_evalcontext⟧ ⟹ (GPFst e) ∈ generalized_evalcontext"
| ge_GPSnd: "⟦e ∈ generalized_evalcontext⟧ ⟹ (GPSnd e) ∈ generalized_evalcontext"
| ge_GPFold: "e : generalized_evalcontext ⟹ (GPFold e) : generalized_evalcontext"
| ge_GPUnfold: "e : generalized_evalcontext ⟹ (GPUnfold e) : generalized_evalcontext"
| ge_GPCasel: "⟦e ∈ generalized_evalcontext; p1 : generalized_programterm;
```

```
    p2 : generalized_programterm⟧ ⟹ (GPCase e p1 p2) ∈ generalized_evalcontext"
| ge_GPCasem: "⟦e ∈ generalized_evalcontext; v : generalized_value;
    p : generalized_programterm⟧ ⟹ (GPCase v e p) ∈ generalized_evalcontext"
| ge_GPCaser: "⟦e ∈ generalized_evalcontext; v1 : generalized_value;
    v2 : generalized_value⟧ ⟹ GPCase v1 v2 e ∈ generalized_evalcontext"
| ge_GPInl: "e : generalized_evalcontext ⟹ (GPInl e) : generalized_evalcontext"
| ge_GPInr: "e : generalized_evalcontext ⟹ (GPInr e) : generalized_evalcontext"
```

Definition 1.66 page 48

```
(397) inductive_set generalized_redex :: "generalized_programtermT set"
  where
  gr_GPFunction: "⟦v : generalized_value⟧ ⟹ GPFunction f v ∈ generalized_redex"
| gr_GPApplication: "⟦v1 : generalized_value; v2 : generalized_value⟧ ⟹
    GPApplication v1 v2 ∈ generalized_redex"
| gr_GPRef: "⟦v : generalized_value⟧ ⟹ GPRef v : generalized_redex"
| gr_GPDeref: "⟦v : generalized_value⟧ ⟹ GPDeref v : generalized_redex"
| gr_GPAssign: "⟦v1 : generalized_value; v2 : generalized_value⟧ ⟹
    GPAssign v1 v2 : generalized_redex"
| gr_GPEvent: "GPEvent e : generalized_redex"
| gr_GPEventList: "GPEventList : generalized_redex"
| gr_GPFst: "⟦v : generalized_value⟧ ⟹ GPFst v : generalized_redex"
| gr_GPSnd: "⟦v : generalized_value⟧ ⟹ GPSnd v : generalized_redex"
| gr_GPUnfold: "v : generalized_value ⟹ GPUnfold v : generalized_redex"
| gr_GPCase: "⟦v : generalized_value; p1 : generalized_value; p2 : generalized_value⟧
    ⟹ GPCase v p1 p2 : generalized_redex"

(398) inductive_set gp_epsilon :: "generalized_programtermT set"
  where
  "a : generalized_instantiation ⟹ GPEpsilon a : gp_epsilon"
```

Sigma Algebras

```
(399) inductive_set gp_rects :: "generalized_programtermT set set"
  and gpa_rects :: "generalized_instantiationT set set"
  where
  — gpa-rects
  gpa_rects_nil: "{[]} : gpa_rects"
| gpa_rects_scons: "⟦P : gp_rects; A : gpa_rects⟧ ⟹
    {(GPESubst n p)#a | p a n. p:P ∧ a:A ∧ n:N} : gpa_rects"
| gpa_rects_lcons: "⟦A : gpa_rects⟧ ⟹ {(GPELift n)#a | a n. a:A ∧ n:N} : gpa_rects"
  — gp-rects
| gp_rects_Epsilon: "A : gpa_rects ⟹ GPEpsilon ‘ A : gp_rects"
| gp_rects_Hole: "{GPHole} ∈ gp_rects"
| gp_rects_Var: "{GPVar n} ∈ gp_rects"
| gp_rects_Value: "V ∈ Σ ⟹ GPValue‘V ∈ gp_rects"
| gp_rects_Function: "⟦A ∈ gp_rects⟧ ⟹ {GPFunction f a|a f. a∈A ∧ f∈F} ∈ gp_rects"
| gp_rects_PairP: "⟦A ∈ gp_rects; B ∈ gp_rects⟧ ⟹
    {GPPairP a b|a b. a∈A∧b∈B} ∈ gp_rects"
| gp_rects_Abstraction: "A ∈ gp_rects ⟹ {GPAbstraction a|a. a∈A} ∈ gp_rects"
| gp_rects_Application: "⟦A ∈ gp_rects; B ∈ gp_rects⟧ ⟹
    {GPApplication a b|a b. a∈A ∧ b∈B} ∈ gp_rects"
| gp_rects_Location: "{GPLocation n} ∈ gp_rects"
| gp_rects_Ref: "A ∈ gp_rects ⟹ {GPRef a|a. a∈A} ∈ gp_rects"
| gp_rects_Deref: "A ∈ gp_rects ⟹ {GPDeref a|a. a∈A} ∈ gp_rects"
| gp_rects_Assign: "⟦A ∈ gp_rects; B ∈ gp_rects⟧ ⟹
    {GPAssign a b|a b. a∈A ∧ b∈B} ∈ gp_rects"
| gp_rects_Event: "{GPEvent ev} ∈ gp_rects"
| gp_rects_EventList: "{GPEventList} ∈ gp_rects"
| gp_rects_Fst: "A ∈ gp_rects ⟹ {GPFst a|a. a∈A} ∈ gp_rects"
| gp_rects_Snd: "A ∈ gp_rects ⟹ {GPSnd a|a. a∈A} ∈ gp_rects"
| gp_rects_Fold: "A ∈ gp_rects ⟹ {GPFold a|a. a∈A} ∈ gp_rects"
| gp_rects_Unfold: "A ∈ gp_rects ⟹ {GPUnfold a|a. a∈A} ∈ gp_rects"
| gp_rects_CaseP: "⟦A ∈ gp_rects; B ∈ gp_rects; C ∈ gp_rects⟧ ⟹
    {GPCase a b c |a b c. a∈A ∧ b∈B ∧ c∈C} ∈ gp_rects"
```

```
| gp_rects_InlP: "A ∈ gp_rects ⟹ {GPInl a|a. a∈A} ∈ gp_rects"
| gp_rects_InrP: "A ∈ gp_rects ⟹ {GPInr a|a. a∈A} ∈ gp_rects"

instantiation "generalized_programtermT" :: measurable_space
(400) definition "Σ = sigma gp_rects"

(401) inductive_set gp_epsinstr_rects ::
  "('a::measurable_space) gp_epsinstruction set set" where
  gp_epsinstr_rects_subst: "⟦A∈Σ⟧⟹{GPESubst n a|n a. n∈N∧a∈A}∈gp_epsinstr_rects"
| gp_epsinstr_rects_lift: "{GPELift n|n. n∈N} ∈ gp_epsinstr_rects"

instantiation "gp_epsinstruction" :: (measurable_space)measurable_space
(402) definition "Σ = sigma gp_epsinstr_rects"
```

Function Definitions

```
(403) function insert_programterm :: "[generalized_programtermT, context] ⇒ context"
  where
  "insert_programterm (GPEpsilon []) p    = p"
| "insert_programterm (GPEpsilon ((GPESubst k q)#l)) p =
    substitute'_context k (insert_programterm (GPEpsilon l) p) (insert_programterm q p)"
| "insert_programterm (GPEpsilon ((GPELift k)#l)) p =
    lift_vars_context k (insert_programterm (GPEpsilon l) p)"
| "insert_programterm GPHole p            = CHole"
| "insert_programterm (GPFunction f c1) p = CFunction f (insert_programterm c1 p)"
| "insert_programterm (GPPairP c1 c2) p   =
    CPairP (insert_programterm c1 p) (insert_programterm c2 p)"
| "insert_programterm (GPAbstraction c) p = CAbstraction (insert_programterm c p)"
| "insert_programterm (GPApplication c1 c2) p =
    CApplication (insert_programterm c1 p) (insert_programterm c2 p)"
| "insert_programterm (GPRef c) p         = CRef (insert_programterm c p)"
| "insert_programterm (GPDeref c) p       = CDeref (insert_programterm c p)"
| "insert_programterm (GPAssign c1 c2) p  =
    CAssign (insert_programterm c1 p) (insert_programterm c2 p)"
| "insert_programterm (GPFst c) p         = CFst (insert_programterm c p)"
| "insert_programterm (GPSnd c) p         = CSnd (insert_programterm c p)"
| "insert_programterm (GPVar v) p         = CVar v"
| "insert_programterm (GPValue v) p       = CValue v"
| "insert_programterm (GPLocation l) p    = CLocation l"
| "insert_programterm (GPEvent e) p       = CEvent e"
| "insert_programterm GPEventList p       = CEventList"
| "insert_programterm (GPFold c) p        = CFold (insert_programterm c p)"
| "insert_programterm (GPUnfold c) p      = CUnfold (insert_programterm c p)"
| "insert_programterm (GPCase c1 c2 c3) p =
    CCase (insert_programterm c1 p) (insert_programterm c2 p) (insert_programterm c3 p)"
| "insert_programterm (GPInl c) p         = CInl (insert_programterm c p)"
| "insert_programterm (GPInr c) p         = CInr (insert_programterm c p)"
```

Definition 1.68 page 48

```
(404) definition insert_programterm' ::
  "[generalized_programtermT, programterm] ⇒ programterm" where
  "insert_programterm' P A =
   context_to_programterm (insert_programterm P (programterm_to_context A))"

(405) function gp_lift_vars ::
  "nat ⇒ generalized_programtermT ⇒ generalized_programtermT" where
  "gp_lift_vars k (GPEpsilon l)    = GPEpsilon ((GPELift k)#l)"
| "gp_lift_vars k (GPVar i)        = (if i < k then GPVar i else GPVar (Suc i))"
| "gp_lift_vars k (GPHole)         = GPHole"
| "gp_lift_vars k (GPAbstraction p)= GPAbstraction (gp_lift_vars (Suc k) p)"
| "gp_lift_vars k (GPApplication p q) =
    GPApplication (gp_lift_vars k p) (gp_lift_vars k q)"
| "gp_lift_vars k (GPValue v)      = GPValue v"
| "gp_lift_vars k (GPFunction f p) = GPFunction f (gp_lift_vars k p)"
```

Section 1.11.5 page 48

```
| "gp_lift_vars k (GPPairP p1 p2)  = GPPairP (gp_lift_vars k p1) (gp_lift_vars k p2)"
| "gp_lift_vars k (GPLocation n)   = GPLocation n"
| "gp_lift_vars k (GPRef p)        = GPRef (gp_lift_vars k p)"
| "gp_lift_vars k (GPDeref p)      = GPDeref (gp_lift_vars k p)"
| "gp_lift_vars k (GPAssign p1 p2) = GPAssign (gp_lift_vars k p1) (gp_lift_vars k p2)"
| "gp_lift_vars k (GPEvent e)      = GPEvent e"
| "gp_lift_vars k GPEventList      = GPEventList"
| "gp_lift_vars k (GPFst p)        = GPFst (gp_lift_vars k p)"
| "gp_lift_vars k (GPSnd p)        = GPSnd (gp_lift_vars k p)"
| "gp_lift_vars k (GPFold p)       = GPFold (gp_lift_vars k p)"
| "gp_lift_vars k (GPUnfold p)     = GPUnfold (gp_lift_vars k p)"
| "gp_lift_vars k (GPCase p1 p2 p3)=
    GPCase (gp_lift_vars k p1) (gp_lift_vars k p2) (gp_lift_vars k p3)"
| "gp_lift_vars k (GPInl p)        = GPInl (gp_lift_vars k p)"
| "gp_lift_vars k (GPInr p)        = GPInr (gp_lift_vars k p)"
```

Section 1.11.5 page 48

(406) function gp_substitute' ::
"[nat,generalized_programtermT,generalized_programtermT] ⇒ generalized_programtermT"
where

```
  "gp_substitute' k (GPEpsilon l) p     = GPEpsilon ((GPESubst k p)#l)"
| "gp_substitute' k (GPVar i) p         =
    (if k < i then GPVar (i - 1) else if i=k then p else GPVar i)"
| "gp_substitute' k (GPAbstraction p1) p =
    GPAbstraction (gp_substitute' (Suc k) p1 (gp_lift_vars 0 p))"
| "gp_substitute' k GPHole p            = GPHole"
| "gp_substitute' k (GPApplication s t) p =
    GPApplication (gp_substitute' k s p) (gp_substitute' k t p)"
| "gp_substitute' k (GPValue v) p       = GPValue v"
| "gp_substitute' k (GPFunction f p1) p = GPFunction f (gp_substitute' k p1 p)"
| "gp_substitute' k (GPPairP p1 p2) p   =
    GPPairP (gp_substitute' k p1 p) (gp_substitute' k p2 p)"
| "gp_substitute' k (GPLocation n) p    = GPLocation n"
| "gp_substitute' k (GPRef p1) p        = GPRef (gp_substitute' k p1 p)"
| "gp_substitute' k (GPDeref p1) p      = GPDeref (gp_substitute' k p1 p)"
| "gp_substitute' k (GPAssign p1 p2) p  =
    GPAssign (gp_substitute' k p1 p) (gp_substitute' k p2 p)"
| "gp_substitute' k (GPEvent e) p       = GPEvent e"
| "gp_substitute' k GPEventList p       = GPEventList"
| "gp_substitute' k (GPFst p1) p        = GPFst (gp_substitute' k p1 p)"
| "gp_substitute' k (GPSnd p1) p        = GPSnd (gp_substitute' k p1 p)"
| "gp_substitute' k (GPFold p1) p       = GPFold (gp_substitute' k p1 p)"
| "gp_substitute' k (GPUnfold p1) p     = GPUnfold (gp_substitute' k p1 p)"
| "gp_substitute' k (GPCase p1 p2 p3) p =
    GPCase (gp_substitute' k p1 p) (gp_substitute' k p2 p) (gp_substitute' k p3 p)"
| "gp_substitute' k (GPInl p1) p        = GPInl (gp_substitute' k p1 p)"
| "gp_substitute' k (GPInr p1) p        = GPInr (gp_substitute' k p1 p)"
```

Section 1.11.5 page 49

(407) function apply_instantiation ::
"[generalized_programtermT,generalized_instantiationT]⇒generalized_programtermT" **where**

```
  "apply_instantiation A [] = A"
| "apply_instantiation A (GPESubst k P#a) = gp_substitute' k (apply_instantiation A a)
P"
| "apply_instantiation A (GPELift k#a) = gp_lift_vars k (apply_instantiation A a)"
```

(408) function gp_concatcontext ::
"[generalized_programtermT,generalized_programtermT]⇒generalized_programtermT" **where**

```
  "gp_concatcontext GPHole p            = p"
| "gp_concatcontext (GPEpsilon a) p     = GPEpsilon a"
| "gp_concatcontext (GPFunction f c1) p = GPFunction f (gp_concatcontext c1 p)"
| "gp_concatcontext (GPPairP c1 c2) p   =
    GPPairP (gp_concatcontext c1 p) (gp_concatcontext c2 p)"
| "gp_concatcontext (GPAbstraction c) p = GPAbstraction (gp_concatcontext c p)"
| "gp_concatcontext (GPApplication c1 c2) p =
    GPApplication (gp_concatcontext c1 p) (gp_concatcontext c2 p)"
```

```
| "gp_concatcontext (GPRef c) p         = GPRef (gp_concatcontext c p)"
| "gp_concatcontext (GPDeref c) p       = GPDeref (gp_concatcontext c p)"
| "gp_concatcontext (GPAssign c1 c2) p  =
    GPAssign (gp_concatcontext c1 p) (gp_concatcontext c2 p)"
| "gp_concatcontext (GPFst c) p         = GPFst (gp_concatcontext c p)"
| "gp_concatcontext (GPSnd c) p         = GPSnd (gp_concatcontext c p)"
| "gp_concatcontext (GPVar v) p         = GPVar v"
| "gp_concatcontext (GPValue v) p       = GPValue v"
| "gp_concatcontext (GPLocation l) p    = GPLocation l"
| "gp_concatcontext (GPEvent e) p       = GPEvent e"
| "gp_concatcontext GPEventList p       = GPEventList"
| "gp_concatcontext (GPFold c) p        = GPFold (gp_concatcontext c p)"
| "gp_concatcontext (GPUnfold c) p      = GPUnfold (gp_concatcontext c p)"
| "gp_concatcontext (GPCase c1 c2 c3) p =
    GPCase (gp_concatcontext c1 p) (gp_concatcontext c2 p) (gp_concatcontext c3 p)"
| "gp_concatcontext (GPInl c) p         = GPInl (gp_concatcontext c p)"
| "gp_concatcontext (GPInr c) p         = GPInr (gp_concatcontext c p)"
```

(409) function `unprotected_epsilons :: "generalized_programtermT ⇒ nat"` Definition 1.72 page 51

where

```
  "unprotected_epsilons (GPEpsilon a)    = 1"
| "unprotected_epsilons GPHole           = 0"
| "unprotected_epsilons (GPVar n)        = 0"
| "unprotected_epsilons (GPValue v)      = 0"
| "unprotected_epsilons (GPLocation l)   = 0"
| "unprotected_epsilons (GPEvent e)      = 0"
| "unprotected_epsilons GPEventList      = 0"
| "unprotected_epsilons (GPFunction f p) = unprotected_epsilons p"
| "unprotected_epsilons (GPPairP p q)    =
    unprotected_epsilons p + unprotected_epsilons q"
| "unprotected_epsilons (GPAbstraction P)= 0" — We do not descend into abstractions
| "unprotected_epsilons (GPApplication p q) =
    unprotected_epsilons p + unprotected_epsilons q"
| "unprotected_epsilons (GPRef p)        = unprotected_epsilons p"
| "unprotected_epsilons (GPDeref p)      = unprotected_epsilons p"
| "unprotected_epsilons (GPAssign p q)   =
    unprotected_epsilons p + unprotected_epsilons q"
| "unprotected_epsilons (GPFst p)        = unprotected_epsilons p"
| "unprotected_epsilons (GPSnd p)        = unprotected_epsilons p"
| "unprotected_epsilons (GPFold p)       = unprotected_epsilons p"
| "unprotected_epsilons (GPUnfold p)     = unprotected_epsilons p"
| "unprotected_epsilons (GPCase p1 p2 p3)=
    unprotected_epsilons p1 + unprotected_epsilons p2 + unprotected_epsilons p3"
| "unprotected_epsilons (GPInl p)        = unprotected_epsilons p"
| "unprotected_epsilons (GPInr p)        = unprotected_epsilons p"
```

(410) lemma `gp_lift_vars_insert_programterm:` Lemma 1.69 page 48

```
  "lift_vars_context k (insert_programterm p a) =
   insert_programterm (gp_lift_vars k p) a"
```

(411) lemma `gp_substitute'_insert_programterm:` Lemma 1.69 page 48

```
  "substitute'_context k (insert_programterm p a) (insert_programterm q a) =
   insert_programterm (gp_substitute' k p q) a"
```

(412) lemma `unprotected_epsilons_value:` Section 1.11.5 page 55

```
  assumes "V : generalized_value"
  shows "unprotected_epsilons V = 0"
```

(413) lemma `unprotected_epsilons_decrease:`

```
  assumes v: "unprotected_epsilons V = 0"
  and e: "E : generalized_evalcontext"
  shows "unprotected_epsilons (gp_concatcontext E (GPEpsilon p)) >
         unprotected_epsilons (gp_concatcontext E V)"
```

(414) lemma `gp_evaluationcontext_redex_split:` Lemma 1.74 page 51

```
  assumes "p : generalized_programterm"
  assumes "p ∉ generalized_value"
  shows "∃ E∈generalized_evalcontext.
         ∃ R∈(generalized_redex ∪ gp_epsilon).
         p = gp_concatcontext E R"
```

(415) lemma gp_evaluationcontext_redex_split_unique':

```
  assumes conceq:"gp_concatcontext E1 R1 = gp_concatcontext E2 R2"
  assumes E1eval:"E1∈generalized_evalcontext"
  assumes E2eval:"E2∈generalized_evalcontext"
  assumes R1red: "R1∈generalized_redex∪gp_epsilon"
  assumes R2red: "R2∈generalized_redex∪gp_epsilon"
  shows "E1 = E2" and "R1 = R2"
```

I.8.2 The Instantiation Operator

Definition 1.51, page 43

(416) definition

```
  "substituteLr n == (λp vl. foldr (λv p. substitute' n p v) vl p)"
```

Section 1.11.4, page 44

(417) function substituting_context :: "programterm list ⇒ context" **where**

```
  "substituting_context []     = CHole"
| "substituting_context (v#vs) =
    CApplication (CAbstraction (substituting_context vs)) (programterm_to_context v)"
```

Section 1.11.4, page 43

(418) lemma substituteLr_preserves_value:

```
  assumes "is_value v"
  assumes "are_values vl"
  shows "is_value (substituteLr 0 v vl)"
```

(419) lemma lift_substituteLr_le:

```
  assumes "m≤n"
  shows "lift_vars m (substituteLr n p vl)
       = substituteLr (Suc n) (lift_vars m p) (map (lift_vars m) vl)"
```

(420) lemma substituteLr_commute:

```
  shows "substituteLr k p (vl@[v])
       = substitute' k (substituteLr (Suc k) p (map (lift_vars k) vl))
                       (substituteLr k v vl)"
```

(421) lemma substituteLr_commute_ge:

```
  assumes "m≥n"
  assumes "freevars q = {}"
  assumes "∀x∈set a. freevars x = {}"
  shows "substitute' m (substituteLr n p a) q
       = substituteLr n (substitute' (m + length a) p q) a"
```

Lemma 1.52, page 43

(422) lemma substituteLr_beta:

```
  assumes "is_value v"
  assumes "are_values vl"
  shows "(substituteLr 0 (Application (Abstraction p) v) vl,state)
       ⇝ apply_kernel unitkernel (substituteLr 0 p (vl@[v]), state)"
```

Lemma 1.53, page 44

(423) lemma substituteLr_substitute_closed:

```
  assumes "freevars (substituteLr l P a) = {}"
  shows "substituteLr l (substitute' u Q (substituteLr l P a)) a
       = substituteLr l (substitute' u Q P) a"
```

Section 1.11.4, page 44

(424) lemma denotation_substituting_context_substituteLr:

```
  assumes "set as ⊆ values"
  shows "apply_kernel denotation (applycontext (substituting_context as) P, σ, η)
       = apply_kernel denotation (substituteLr 0 P as, σ, η)"
```

(425) lemma freevars_substituting_context:

```
  assumes "∀v∈set a. freevars v ={}"
```

```
  assumes "∀x∈freevars p. x < length a"
  shows "freevars (applycontext (substituting_context a) p) = {}"
```

(426) lemma fullyclosed_substituting_context:
```
  assumes "∀v∈set a. freevars v ={}"
  assumes "∀v∈set a. (∀n∈(locations_of v). n<length s)"
  assumes "∀x∈freevars p. x < length a"
  assumes "∀p'∈set s. freevars p' = {}"
  assumes "storageclosed (p, s, e)"
  shows "fullyclosed (applycontext (substituting_context a) p, s, e)"
```

(427) lemma freevars_substituteLr:
```
  assumes "∀n∈freevars p. n<length a"
  assumes "∀v∈set a. freevars v ={}"
  shows "freevars (substituteLr 0 p a) = {}"
```

(428) lemma ex_closed_substitutionlist: Lemma 1.54 page 44
```
  assumes "fullyclosed (substituteLr 0 P a, σ, η)"
  assumes "fullyclosed (substituteLr 0 Q a, σ, η)"
  shows "∃vl. substituteLr 0 P vl = substituteLr 0 P a
           ∧ substituteLr 0 Q vl = substituteLr 0 Q a
           ∧ length vl = length a
           ∧ (∀v∈set vl. freevars v ={} ∧ (∀l∈locations_of v. l<length σ))
           ∧ (set a ⊆ values ⟶ set vl ⊆ values)"
```

(429) lemma ex_closed_substituting_context: Lemma 1.55 page 45
```
  assumes aval: "set a ⊆ values"
  assumes fcPa: "fullyclosed (substituteLr 0 P a, σ, η)"
  assumes fcQa: "fullyclosed (substituteLr 0 Q a, σ, η)"
  shows "∃vl. set vl ⊆ values
           ∧ substituteLr 0 P a = substituteLr 0 P vl
           ∧ substituteLr 0 Q a = substituteLr 0 Q vl
           ∧ length a = length vl
           ∧ (∀v∈set vl. freevars v ={}
             ∧ (∀l∈locations_of v. l<length σ))
           ∧ fullyclosed (applycontext (substituting_context vl) P, σ, η)
           ∧ fullyclosed (applycontext (substituting_context vl) Q, σ, η)
           ∧ apply_kernel denotation (substituteLr 0 P a, σ, η)
            =apply_kernel denotation (applycontext (substituting_context vl) P,σ,η)
           ∧ apply_kernel denotation (substituteLr 0 Q a, σ, η)
            =apply_kernel denotation (applycontext (substituting_context vl) Q,σ,η)"
```

Generalized Instantiations and Substitution Lists

(430) function programterm_instantiation :: Section 1.11.5 page 49
```
  "generalized_instantiationT ⇒ bool" where
  "programterm_instantiation [] = True"
| "programterm_instantiation (GPELift k # a) = programterm_instantiation a"
| "programterm_instantiation (GPESubst k P # a) =
    ((∃p. P=programterm_to_gp p) ∧ programterm_instantiation a)"
```

(431) function closed_instantiation :: "generalized_instantiationT ⇒ bool" **where** Section 1.11.5 page 49
```
  "closed_instantiation [] = True"
| "closed_instantiation (GPELift k # a) = closed_instantiation a"
| "closed_instantiation (GPESubst k P # a) =
     ((∃p. freevars p = {} ∧ P=programterm_to_gp p) ∧ closed_instantiation a)"
```

(432) function normalized_instantiation :: "generalized_instantiationT ⇒ bool" **where**
```
  "normalized_instantiation [] = True"
| "normalized_instantiation (GPELift k # a) = normalized_instantiation a"
| "normalized_instantiation [GPESubst k p] = True"
| "normalized_instantiation (GPESubst l p # GPELift k # a) = False"
| "normalized_instantiation (GPESubst l p # GPESubst k q # a) =
     (if l<k then False else normalized_instantiation (GPESubst k q # a))"
```

Section 1.11.5 page 50

(433) lemma normalized_instantiation_def2:

```
  "normalized_instantiation a =
   (∀a1 l p k q aa. (a ≠ a1 @ GPESubst l p # GPELift k # aa)
                  ∧ (a ≠ a1 @ GPESubst l p # GPESubst (l + Suc k) q # aa))"
```

Section 1.11.5 page 49

(434) lemma insert_programterm_instantiation_apply_instantiation:

```
  assumes "a ∈ generalized_instantiation"
  assumes "programterm_instantiation a"
  shows "programterm_to_gp (insert_programterm' (GPEpsilon a) A)
       = apply_instantiation (programterm_to_gp A) a"
```

Section 1.11.5 page 49

(435) definition list_to_gp_instantiation ::

```
  "programterm list ⇒ generalized_instantiationT" where
  "list_to_gp_instantiation l = map (λv. GPESubst 0 (programterm_to_gp v)) l"
```

Section 1.11.5 page 49

(436) lemma list_to_gp_instantiation_substituteLr:

```
  assumes "set l ⊆ values"
  shows "apply_instantiation (programterm_to_gp p) (list_to_gp_instantiation l)
       = programterm_to_gp (substituteLr 0 p l)"
```

Definition 1.68 page 48

(437) function map_insert_programterm' ::

```
  "programterm ⇒ generalized_instantiationT ⇒ generalized_instantiationT" where
  "map_insert_programterm' A []                  = []"
| "map_insert_programterm' A (GPELift k # a)    =
    (GPELift k # map_insert_programterm' A a)"
| "map_insert_programterm' A (GPESubst k p # a) =
    (GPESubst k (programterm_to_gp (insert_programterm' p A)) # map_insert_programterm'
A a)"
```

Section 1.11.5 page 49

(438) function substitute_instantiation ::

```
  "[nat,generalized_instantiationT,generalized_programtermT]⇒generalized_instantiationT"
where
  "substitute_instantiation m [] A = [GPESubst m A]"
| "substitute_instantiation m (GPESubst n Q # a) A =
  (if n≤m then GPESubst n (gp_substitute' m Q A) # substitute_instantiation (Suc m) a
A
      else GPESubst (n- 1)(gp_substitute' m Q A) # substitute_instantiation  m a A)"
| "substitute_instantiation m (GPELift n # a) A =
          (if n=m then a else
           if n<m then GPELift  n     # substitute_instantiation (m- 1) a A
                  else GPELift (n- 1) # substitute_instantiation  m     a A)"

(439) function open_prefix_length :: "generalized_instantiationT ⇒ nat" where
  "open_prefix_length [] = 0"
| "open_prefix_length (i # a) = (if closed_instantiation (i # a) then 0
                                 else Suc (open_prefix_length a))"

(440) lemma open_prefix_length_closed_instantiation:
  assumes "closed_instantiation a"
  shows "open_prefix_length a = 0"

(441) lemma open_prefix_length_substitute_instantiation_less:
  assumes p_cl: "freevars p = {}"
  assumes "a ∈ generalized_instantiation"
  shows "open_prefix_length (substitute_instantiation k a (programterm_to_gp p))
      < Suc (open_prefix_length a)"
```

Section 1.11.5 page 50

(442) function close_instantiation ::

```
  "[programterm,programterm,generalized_instantiationT]⇒generalized_instantiationT"
  where
  "close_instantiation A B []                  = []"
| "close_instantiation A B (GPELift k # a)    = close_instantiation A B a"
| "close_instantiation A B (GPESubst k P # a) =
     (if closed_instantiation (GPESubst k P # a)
       ∨ (GPESubst k P # a) ∉ generalized_instantiation
```

```
     ∨ ¬ programterm_instantiation (GPESubst k P # a)
     ∨ ¬ freevars (insert_programterm' (GPEpsilon (GPESubst k P # a)) A) = {}
     ∨ ¬ freevars (insert_programterm' (GPEpsilon (GPESubst k P # a)) B) = {}
    then (GPESubst k P # a) else
    if k ∈ freevars (insert_programterm' (GPEpsilon a) A)
     ∨ k ∈ freevars (insert_programterm' (GPEpsilon a) B)
    then close_instantiation A B (substitute_instantiation k a P)
    else close_instantiation A B a)"
```

(443) function `insert_epsinstruction :: "generalized_instantiationT ⇒ generalized_programtermT gp_epsinstruction ⇒ generalized_instantiationT" where` Section 1.11.5 page 50

```
  "insert_epsinstruction [] eps = [eps]"
| "insert_epsinstruction a (GPELift k) = GPELift k # a"
| "insert_epsinstruction (GPELift m # a) (GPESubst n p) =
       (if m<n then GPELift m # (insert_epsinstruction a (GPESubst (n - 1) p))
   else if m>n then GPELift (m - 1) # (insert_epsinstruction a (GPESubst n p))
   else a)"
| "insert_epsinstruction (GPESubst m q # a) (GPESubst n p) =
       (if n≥m then GPESubst n p # GPESubst m q # a
        else GPESubst (m - 1) q # insert_epsinstruction a (GPESubst n p))"
```

(444) definition Section 1.11.5 page 50

```
  "normalize_instantiation a = foldr (λeps acc. insert_epsinstruction acc eps) a []"
```

(445) function `normalized_instantiation_to_list :: "generalized_instantiationT ⇒ programterm list" where` Section 1.11.5 page 50

```
  "normalized_instantiation_to_list [] = []"
| "normalized_instantiation_to_list (GPELift k # a) =
     normalized_instantiation_to_list a"
| "normalized_instantiation_to_list [GPESubst k P] =
    (THE p. P=programterm_to_gp p) # replicate k value_unit "
| "normalized_instantiation_to_list (GPESubst k P # GPESubst l Q # a) =
    (THE p. P=programterm_to_gp p)
   # replicate (k-l) value_unit
   @ normalized_instantiation_to_list (GPESubst l Q # a)"
| "normalized_instantiation_to_list (GPESubst k P # GPELift l # a) = undefined"
```

(446) lemma `map_insert_programterm'_generalized_instantiation:`
assumes `aass: "a ∈ generalized_instantiation"`
shows `"map_insert_programterm' A a ∈ generalized_instantiation"`

(447) lemma `programterm_instantiation_map_insert_programterm':` Section 1.11.5 page 49

```
  "programterm_instantiation (map_insert_programterm' A a)"
```

(448) lemma `insert_programterm'_map_insert_programterm':`
assumes `aass: "a ∈ generalized_instantiation"`
shows `"insert_programterm' (GPEpsilon a) A`
`= insert_programterm' (GPEpsilon (map_insert_programterm' A a)) A"`

(449) lemma `apply_instantiation_map_insert_programterm':` Section 1.11.5 page 49
assumes `aass: "a ∈ generalized_instantiation"`
shows `"programterm_to_gp (insert_programterm' (GPEpsilon a) A)`
`=apply_instantiation (programterm_to_gp A) (map_insert_programterm' A a)"`

(450) lemma `insert_programterm'_substitute_instantiation:`
assumes `fv_p: "freevars p = {}"`
assumes `valp: "p ∈ values"`
assumes `"a ∈ generalized_instantiation"`
assumes `"programterm_instantiation a"`
shows `"insert_programterm'(GPEpsilon(GPESubst k (programterm_to_gp p) # a))A`
`= insert_programterm'(GPEpsilon(substitute_instantiation k a (programterm_to_gp p)))A"`

(451) lemma `insert_programterm'_close_instantiation:` Section 1.11.5 page 50
assumes `"a∈generalized_instantiation"`

```
  assumes "programterm_instantiation a"
  assumes "freevars (insert_programterm' (GPEpsilon a) A) = {}"
  assumes "freevars (insert_programterm' (GPEpsilon a) B) = {}"
  shows "insert_programterm' (GPEpsilon a) A
       = insert_programterm' (GPEpsilon (close_instantiation A B a)) A"
  and   "insert_programterm' (GPEpsilon a) B
       = insert_programterm' (GPEpsilon (close_instantiation A B a)) B"
```

(452) lemma close_instantiation_is_instantiation:

```
  assumes "a∈generalized_instantiation"
  assumes "programterm_instantiation a"
  assumes "freevars (insert_programterm' (GPEpsilon a) A) = {}"
  shows "close_instantiation A B a ∈ generalized_instantiation"
```

Section 1.11.5 page 50 **(453) lemma** close_instantiation_is_closed:

```
  assumes "a∈generalized_instantiation"
  assumes "programterm_instantiation a"
  assumes "freevars (insert_programterm' (GPEpsilon a) A) = {}"
  assumes "freevars (insert_programterm' (GPEpsilon a) B) = {}"
  shows "closed_instantiation (close_instantiation A B a)"
```

(454) lemma apply_instantiation_insert_epsinstruction:

```
  assumes "closed_instantiation (eps#a)"
  shows "apply_instantiation (programterm_to_gp p) (insert_epsinstruction a eps)
       = apply_instantiation (programterm_to_gp p) (eps#a)"
```

Section 1.11.5 page 50 **(455) lemma** apply_instantiation_normalize_instantiation:

```
  assumes "closed_instantiation a"
  shows "apply_instantiation (programterm_to_gp p) (normalize_instantiation a)
       = apply_instantiation (programterm_to_gp p) a"
```

(456) lemma normalized_instantiation_insert_epsinstruction:

```
  assumes "normalized_instantiation a"
  shows "normalized_instantiation (insert_epsinstruction a eps)"
```

Section 1.11.5 page 50 **(457) lemma** normalized_instantiation_normalize_instantiation:

```
  "normalized_instantiation (normalize_instantiation a)"
```

Section 1.11.5 page 50 **(458) lemma** instantiation_to_list:

```
  assumes "a∈generalized_instantiation"
  assumes "normalized_instantiation a"
  assumes "closed_instantiation a"
  assumes "freevars (THE p. (apply_instantiation (programterm_to_gp P) a)
                          = programterm_to_gp p) = {}"
  shows "apply_instantiation (programterm_to_gp P) a
       = programterm_to_gp (substituteLr 0 P (normalized_instantiation_to_list a))"
```

Lemma 1.70 page 50 **(459) lemma** generalized_instantiation_to_closed_list:

```
  assumes a_ass: "a∈generalized_instantiation"
  assumes a_pass: "programterm_instantiation a"
  assumes Afc: "fullyclosed (insert_programterm' (GPEpsilon a) A, σ, η)"
  assumes Bfc: "fullyclosed (insert_programterm' (GPEpsilon a) B, σ, η)"
  shows "∃vl∈{l | l. are_values l ∧
                  (∀v∈set l. freevars v = {} ∧ (∀l∈locations_of v. l<length σ))}.
         insert_programterm' (GPEpsilon a) A = substituteLr 0 A vl ∧
         insert_programterm' (GPEpsilon a) B = substituteLr 0 B vl"
```

I.8.3 Definition of CIU Relations

Definition 1.60 page 46 **(460) definition**

```
  "ciu_approx A B ==
  ∀E∈evaluationcontext.
  ∀σ∈{l | l. are_values l}.
  ∀vl∈{l | l. are_values l
```

```
           ∧ (∀v∈set l. freevars v = {} ∧ (∀l∈locations_of v. l<length σ))}.
  ∀η::eventT list.
  let EA = (applycontext E (substituteLr 0 A vl), σ, η) in
  let EB = (applycontext E (substituteLr 0 B vl), σ, η) in
  fullyclosed EA ∧ fullyclosed EB ⟶
  kernel_prob_of denotation EA UNIV ≤ kernel_prob_of denotation EB UNIV
  "
```

(461) definition Definition 1.56 page 45

```
  "ciu_equiv A B ==
  ∀E∈evaluationcontext.
  ∀σ∈{l | l. are_values l}.
  ∀vl∈{l | l. are_values l
           ∧ (∀v∈set l. freevars v = {} ∧ (∀l∈locations_of v. l<length σ))}.
  ∀η::eventT list.
  let EA = (applycontext E (substituteLr 0 A vl), σ, η) in
  let EB = (applycontext E (substituteLr 0 B vl), σ, η) in
  fullyclosed EA ∧ fullyclosed EB ⟶
  kernel_prob_of denotation EA UNIV = kernel_prob_of denotation EB UNIV"
```

(462) lemma ciu_equiv_def2: Lemma 1.62 page 46

```
  "ciu_equiv A B == ciu_approx A B ∧ ciu_approx B A"
```

I.8.4 CIU Counterexamples

(463) inductive epsclosed ::

```
  "(programterm set) ⇒ nat ⇒ (generalized_programtermT set)" where
  "⟦epsclosed S n q;
    ∀p∈S. freevars (insert_programterm' q p)={}
         ∧ (∀l∈locations_of (insert_programterm' q p). l<n);
    epsclosed S n (GPEpsilon a)⟧
  ⟹ epsclosed S n (GPEpsilon (GPESubst k q # a))"
| "epsclosed S n (GPEpsilon [])"
| "epsclosed S n (GPEpsilon a) ⟹ epsclosed S n (GPEpsilon (GPELift k # a))"
| "epsclosed S n (GPVar k)"
| "epsclosed S n (GPValue basicvalue)"
| "epsclosed S n (GPLocation l)"
| "epsclosed S n (GPEvent e)"
| "epsclosed S n GPEventList"
| "epsclosed S n P ⟹ epsclosed S n (GPFunction f P)"
| "⟦epsclosed S n P1; epsclosed S n P2⟧ ⟹ epsclosed S n (GPPairP P1 P2)"
| "epsclosed S n P ⟹ epsclosed S n (GPAbstraction P)"
| "⟦epsclosed S n P1; epsclosed S n P2⟧ ⟹ epsclosed S n (GPApplication P1 P2)"
| "epsclosed S n P ⟹ epsclosed S n (GPRef P)"
| "epsclosed S n P ⟹ epsclosed S n (GPDeref P)"
| "⟦epsclosed S n P1; epsclosed S n P2⟧ ⟹ epsclosed S n (GPAssign P1 P2)"
| "epsclosed S n P ⟹ epsclosed S n (GPFst P)"
| "epsclosed S n P ⟹ epsclosed S n (GPSnd P)"
| "epsclosed S n P ⟹ epsclosed S n (GPFold P)"
| "epsclosed S n P ⟹ epsclosed S n (GPUnfold P)"
| "⟦epsclosed S n P1; epsclosed S n P2; epsclosed S n P3⟧ ⟹
    epsclosed S n (GPCase P1 P2 P3)"
| "epsclosed S n P ⟹ epsclosed S n (GPInl P)"
| "epsclosed S n P ⟹ epsclosed S n (GPInr P)"
```

(464) definition epsclosed_state :: Definition 1.71 page 51

```
  "[programterm set,generalized_programtermT,generalized_programtermT list] ⇒ bool"
  where "epsclosed_state S C σ =
  (epsclosed S (length σ) C ∧ (∀p∈set σ. epsclosed S (length σ) p))"
```

(465) types ciu_counterexampleT =

```
  "programterm × programterm × generalized_programtermT
 × generalized_programtermT list × char list list × nat"
```

Definition 1.72, page 51

(466) definition "ciu_counterexample::ciu_counterexampleT set ==
λ(A, B, C, σ, η, n).
let CA = insert_programterm' C A in
let CB = insert_programterm' C B in
let σA = map (λp. insert_programterm' p A) σ in
let σB = map (λp. insert_programterm' p B) σ in
(C : generalized_programterm ∧
set σ ⊆ generalized_value ∧
(* set σ ⊆ generalized_programterm ∧ *)
ciu_approx A B ∧
kernel_prob_of ((restriction_kernel (values×UNIV)) oo nsteps n) (CA,σA,η) UNIV
> kernel_prob_of denotation (CB,σB,η) UNIV ∧
fullyclosed (CA,σA,η) ∧
fullyclosed (CB,σB,η) ∧
epsclosed_state {A,B} C σ)"

Definition 1.72, page 51

(467) definition "minimal_ciu_counterexample == λ(A,B,C,σ,η,n).
(ciu_counterexample (A,B,C,σ,η,n) ∧ (∀(A',B',C',σ',η',n')∈ciu_counterexample.
(n < n' ∨ (n = n' ∧ unprotected_epsilons C ≤ unprotected_epsilons C'))))"

Lemma 1.73, page 51

(468) lemma ciu_ex:
assumes ciu: "ciu_approx A B"
and nobs: "¬ observationally_approximated_untyped A B"
shows "∃X. ciu_counterexample X"

I.8.5 Uniformity

Definition 1.75, page 52

(469) definition
"ciu_uniform pse =
(∃μ. (∀X. lift_kernel step (apply_kernel (gp_insert_kernel X) pse) =
lift_kernel (gp_insert_kernel X) μ) ∧
(fst pse ∈ generalized_programterm ∧
set(fst(snd(pse)))⊆ generalized_value
⟶ (∀pse'←μ. fst pse' ∈ generalized_programterm ∧
set(fst(snd(pse')))⊆ generalized_value)) ∧
(∀X. epsclosed_state X (fst pse) (fst(snd(pse))) ∧
(∀x∈X. fullyclosed(insert_programterm' (fst pse) x,
map (λp. insert_programterm' p x) (fst(snd(pse))),
snd(snd(pse))))
⟶ (∀pse'←μ. epsclosed_state X (fst pse') (fst(snd(pse'))) ∧
length(fst(snd(pse'))) ∈ {length(fst(snd(pse))),Suc(length(fst(snd(pse))))}))"

Section 1.11.5, page 53

(470) lemma function_uniform:
assumes v: "v : generalized_value"
shows "ciu_uniform (GPFunction f v,σ,η)"

Section 1.11.5, page 52

(471) lemma application_uniform:
assumes v: "v : generalized_value"
and w: "w : generalized_value"
shows "ciu_uniform (GPApplication w v,σ,η)"

(472) lemma ref_uniform:
assumes v: "v : generalized_value"
shows "ciu_uniform (GPRef v,σ,η)"

(473) lemma deref_uniform:
assumes w: "w : generalized_value"
shows "ciu_uniform (GPDeref w, σ, η)"

(474) lemma assign_uniform:
assumes v: "v : generalized_value"
assumes w: "w : generalized_value"
shows "ciu_uniform (GPAssign w v,σ,η)"

(475) lemma event_uniform: "ciu_uniform (GPEvent e, σ, η)"

(476) lemma eventlist_uniform: "ciu_uniform (GPEventList, σ, η)"

(477) lemma fst_uniform:
assumes z: "z : generalized_value"
shows "ciu_uniform (GPFst z,σ,η)"

(478) lemma snd_uniform:
assumes z: "z : generalized_value"
shows "ciu_uniform (GPSnd z,σ,η)"

(479) lemma unfold_uniform:
assumes z: "z : generalized_value"
shows "ciu_uniform (GPUnfold z,σ,η)"

(480) lemma case_uniform:
assumes v: "v : generalized_value"
assumes w: "w : generalized_value"
assumes u: "u : generalized_value"
shows "ciu_uniform (GPCase v w u,σ,η)"

(481) lemma redex_uniform:
assumes "R : generalized_redex"
shows "ciu_uniform (R,σ,η)"

(482) lemma evalctx_redex_uniform: Lemma 1.76 page 52
assumes "R : generalized_redex"
and "E : generalized_evalcontext"
shows "ciu_uniform (gp_concatcontext E R,σ,η)"

I.8.6 Finishing the Proof

(483) lemma ciu_uniform_step: Lemma 1.77 page 53
assumes Eeval: "E : generalized_evalcontext"
and Rredex: "R : generalized_redex"
and ciuex: "ciu_counterexample (A,B,gp_concatcontext E R,σ,η,n)"
shows "$\exists$C' σ' η' n'. n=Suc n' $\wedge$ ciu_counterexample(A,B,C',σ',η',n')"

(484) lemma freevars_map_insert_programterm': Section 1.11.5 page 54
assumes "a$\in$generalized_instantiation"
assumes "epsclosed {A,B} n (GPEpsilon a)"
shows "freevars (insert_programterm' (GPEpsilon (map_insert_programterm' A a)) P)
= freevars (insert_programterm' (GPEpsilon (map_insert_programterm' B a)) P)"

(485) lemma eps_no_minimal: Lemma 1.81 page 54
assumes Eeval: "E : generalized_evalcontext"
and aass: "a$\in$generalized_instantiation"
and C_def: "C = gp_concatcontext E (GPEpsilon a)"
and ciuex: "ciu_counterexample (A,B,C,σ,η,n)"
shows "$\exists$C' σ' η' n'. ciu_counterexample(A,B,C',σ',η',n') $\wedge$
(n' < n $\vee$ (n' = n $\wedge$ unprotected_epsilons C' < unprotected_epsilons C))"

(486) lemma no_minimal_ciu_ex: "$\neg$ ($\exists$X. minimal_ciu_counterexample X)" Lemma 1.83 page 55

(487) lemma ciu_counterexample_imp_minimal: Section 1.11.5 page 56
assumes "ciu_counterexample (A,B,C,σ,η,n)"
shows "$\exists$mC. minimal_ciu_counterexample mC"

(488) theorem ciu_approx_imp_obs_approx: Theorem 1.84 page 56
assumes "ciu_approx A B"
shows "observationally_approximated_untyped A B"

Theorem 1.58 page 45
(489) theorem ciu_equiv_imp_obs_equiv:
assumes "ciu_equiv A B"
shows "observationally_equivalent_untyped A B"

Lemma 1.57 page 45
(490) lemma closed_denot_equiv_imp_ciu_equiv:
assumes "∀vl σ η. are_values vl
∧ (∀v∈set vl. freevars v = {} ∧ (∀l∈locations_of v. l<length σ))
∧ are_values σ
∧ fullyclosed (substituteLr 0 A vl, σ, η)
∧ fullyclosed (substituteLr 0 B vl, σ, η)
⟶ apply_kernel denotation (substituteLr 0 A vl, σ, η)
= apply_kernel denotation (substituteLr 0 B vl, σ, η)"
shows "ciu_equiv A B"

I.9 Program Transformations

I.9.1 Transformations based on Computation Rules

Lemma 1.85 page 57
(491) lemma obs_equiv_stepsto_unitkernel:
assumes "∀vl. (are_values vl ⟶ (∀se.
(substituteLr 0 A vl,se) ⇝ apply_kernel unitkernel (substituteLr 0 B vl,se)))"
shows "observationally_equivalent_untyped A B"

Theorem 1.88 page 58
(492) theorem obs_equiv_evaluationcontexts:
assumes "E1 ∈ evaluationcontext"
assumes "E2 ∈ evaluationcontext"
assumes "∀v. v∈values ⟶
observationally_equivalent_untyped (applycontext E1 v) (applycontext E2 v)"
shows "observationally_equivalent_untyped (applycontext E1 p) (applycontext E2 p)"

Lemma 1.86 page 58
(493) lemma obseq_untyped_beta:
assumes "is_value v"
shows "observationally_equivalent_untyped (Application (Abstraction M) v)
(substitute' 0 M v)"

(494) lemma obseq_untyped_rule_beta:
assumes "F : contextfuns ∧ is_value v"
assumes "observationally_equivalent_untyped (F (substitute' 0 M v)) S"
shows "observationally_equivalent_untyped (F (Application (Abstraction M) v)) S"

Lemma 1.87 page 58
(495) lemma obseq_untyped_Fst_PairP':
assumes "is_value v1"
and "is_value v2"
shows "observationally_equivalent_untyped (Fst (PairP v1 v2)) v1"

Lemma 1.89 page 59
(496) lemma obseq_untyped_Fst_PairP:
assumes "is_value v"
shows "observationally_equivalent_untyped (Fst (PairP p v)) p"

(497) lemma obseq_untyped_rule_Fst_PairP:
assumes "F : contextfuns ∧ is_value v"
assumes "observationally_equivalent_untyped (F p) S"
shows "observationally_equivalent_untyped (F (Fst (PairP p v))) S"

Lemma 1.90 page 59
(498) lemma obseq_untyped_Snd_PairP:
assumes "is_value v"
shows "observationally_equivalent_untyped (Snd (PairP v p)) p"

(499) lemma obseq_untyped_rule_Snd_PairP:
assumes "F : contextfuns ∧ is_value v"
assumes "observationally_equivalent_untyped (F p) S"
shows "observationally_equivalent_untyped (F (Snd (PairP v p))) S"

(500) lemma obseq_untyped_Function_deterministic_kernel:
assumes "v ∈ purevalues"
and "is_measurable f"
shows "observationally_equivalent_untyped (Function (deterministic_kernel f) v)
(pureterm_to_term (f (term_to_pureterm v)))"

(501) lemma obseq_untyped_CaseP_InlP: Lemma 1.92 page 59
assumes "is_value vl"
and "is_value vr"
shows "observationally_equivalent_untyped (CaseP (InlP p) vl vr) (Application vl p)"

(502) lemma obseq_untyped_rule_CaseP_InlP:
assumes "F : contextfuns ∧ is_value vl ∧ is_value vr"
assumes "observationally_equivalent_untyped (F (Application vl p)) S"
shows "observationally_equivalent_untyped (F (CaseP (InlP p) vl vr)) S"

(503) lemma obseq_untyped_CaseP_InrP: Lemma 1.93 page 59
assumes "is_value vl"
and "is_value vr"
shows "observationally_equivalent_untyped (CaseP (InrP p) vl vr) (Application vr p)"

(504) lemma obseq_untyped_rule_CaseP_InrP:
assumes "F : contextfuns ∧ is_value vl ∧ is_value vr"
assumes "observationally_equivalent_untyped (F (Application vr p)) S"
shows "observationally_equivalent_untyped (F (CaseP (InrP p) vl vr)) S"

(505) lemma obseq_untyped_Unfold_Fold: Lemma 1.91 page 59
shows "observationally_equivalent_untyped (Unfold (Fold p)) p"

(506) lemma obseq_untyped_LET_flatten: Lemma 1.99 page 63
assumes "C' = lift_vars (Suc 0) C"
shows "observationally_equivalent_untyped
(Application (Abstraction C) (Application (Abstraction B) A))
(Application (Abstraction (Application (Abstraction C') B)) A)"

(507) lemma obseq_untyped_rule_LET_flatten:
assumes "F : contextfuns ∧ (F (Var 0) ≠ F (Var 1))"
assumes "observationally_equivalent_untyped
(F (Application (Abstraction (Application (Abstraction (lift_vars(Suc 0)Y))X))A)) S"
shows "observationally_equivalent_untyped
(F (Application (Abstraction Y) (Application (Abstraction X) A))) S"

(508) lemma obseq_rule_beta_left:
assumes "is_contextfun_typed F ∧ is_value (Rep_program v)"
assumes "observationally_equivalent (F (SUBSTITUTEk 0 p v)) S"
shows "observationally_equivalent (F (APPLY (ABSTRACT x p) v)) S"

(509) lemma obseq_rule_FST_PAIR_left:
assumes "is_contextfun_typed F ∧ is_value (Rep_program v)"
assumes "observationally_equivalent (F p) S"
shows "observationally_equivalent (F (FST (PAIR p v))) S"

(510) lemma obseq_rule_SND_PAIR_left:
assumes "is_contextfun_typed F ∧ is_value (Rep_program v)"
assumes "observationally_equivalent (F p) S"
shows "observationally_equivalent (F (SND (PAIR v p))) S"

(511) lemma obseq_rule_LET_flatten_left:
assumes "is_contextfun_typed F"
assumes "observationally_equivalent
(F (LET A (ABSTRACT a (LET B (ABSTRACT b (LIFTk 1 C)))))) S"
shows "observationally_equivalent (F (LET (LET A (ABSTRACT a B)) (ABSTRACT b C))) S"

I.9.2 Expression Propagation

Definition 1.94, page 60

```
(512) definition "propagatable W ==
  ∀a. are_values a ⟶
  (∀v∈set a. freevars v = {}) ⟶
  freevars (substituteLr 0 W a) = {} ⟶
  (∃w. is_value w ∧ (∀σ η. (are_values σ ∧ fullyclosed(substituteLr 0 W a,σ,η))
                        ⟶ (fullyclosed (w, σ, η)
                           ∧ apply_kernel denotation (substituteLr 0 W a, σ, η)
                            =apply_kernel unitkernel (w, σ, η))))
  ∨ (∀σ η. (are_values σ ∧ fullyclosed (substituteLr 0 W a, σ, η)) ⟶
            apply_kernel denotation (substituteLr 0 W a, σ, η) = 0)"
```

Lemma 1.95, page 61

```
(513) lemma exp_propagation:
assumes "propagatable W"
assumes "∀a σ η. are_values a ∧ (∀v∈set a. freevars v = {}) ∧ are_values σ ∧
                 fullyclosed (substituteLr 0 W a, σ, η) ∧
                 apply_kernel denotation(substituteLr 0 W a, σ, η) = 0 ⟶
                 apply_kernel denotation(substituteLr 0 (substitute' 0 Q W) a,σ,η)=0"
shows "observationally_equivalent_untyped (Application (Abstraction Q) W)
                                          (substitute' 0 Q W)"

(514) lemma obseq_untyped_rule_exp_propagation:
  assumes "F : contextfuns ∧ (propagatable W) ∧
    (∀a σ η. are_values a ∧ (∀v∈set a. freevars v = {}) ∧ are_values σ ∧
             fullyclosed (substituteLr 0 W a, σ, η) ∧
             apply_kernel denotation(substituteLr 0 W a, σ, η) = 0 ⟶
             apply_kernel denotation(substituteLr 0 (substitute' 0 Q W) a,σ,η)=0)"
  assumes "observationally_equivalent_untyped (F (substitute' 0 Q W)) S"
  shows "observationally_equivalent_untyped (F (Application (Abstraction Q) W)) S"
```

Lemma 1.96, page 62

```
(515) lemma propagatable_Fst_Var: "propagatable (Fst (Var n))"
```

Lemma 1.96, page 62

```
(516) lemma propagatable_Snd_Var: "propagatable (Snd (Var n))"
```

```
(517) lemma propagatable_Fst_Snd_Var: "propagatable (Fst (Snd (Var n)))"
```

Section 1.13.1, page 70

```
(518) lemma propagatable_prog_embedding_app:
  fixes f :: "('a::embeddable_pure) ⇒ ('b::embeddable_pure)"
  shows "propagatable (Application (prog_embedding f) (Var n))"
```

I.9.3 Line Swapping

Definition 1.100, page 64

```
(519) definition
"state_independent P =
  (∀s e a. (are_values a ∧
            (∀v∈set a. freevars v = {} ∧ (∀l∈locations_of v. l < length s)) ∧
             fullyclosed(substituteLr 0 P a,s,e)) ⟶
             (apply_kernel denotation (substituteLr 0 P a,s,e)
              =project_measure (λ(V,se). (V,s,e))
               (apply_kernel denotation (substituteLr 0 P a,[],[]))))"
```

Lemma 1.101, page 64

```
(520) lemma state_independent_line_swapping:
  assumes "state_independent A"
  shows "observationally_equivalent_untyped
  (Application (Abstraction(Application (Abstraction(PairP (Var (Suc 0)) (Var 0)))
                                        (lift_vars 0 B))) A)
  (Application (Abstraction(Application (Abstraction(PairP (Var 0) (Var (Suc 0))))
                                        (lift_vars 0 A))) B)"
```

Section 1.12.5, page 66

```
(521) definition
  "uncurry2 P = Abstraction (lift_vars (Suc 0) (lift_vars (Suc 0)
                (Application (Application
```

```
              (Abstraction (Abstraction (lift_vars (Suc(Suc 0)) P)))
              (Fst (Var 0))) (Snd (Var 0)))))"
```

(522) lemma obseq_untyped_uncurry2_apply: Section 1.12.5 page 66
```
  assumes "P' = uncurry2 P"
  shows "observationally_equivalent_untyped P
    (Application P' (PairP (Var (Suc 0)) (Var 0)))"
```

(523) lemma obseq_untyped_swap_uncurry2: Section 1.104 page 66
```
  assumes "P' = uncurry2 P"
  shows "observationally_equivalent_untyped (swap_vars 0 (Suc 0) P)
    (Application P' (PairP (Var 0) (Var (Suc 0))))"
```

(524) theorem obseq_state_independent_line_swapping: Theorem 1.103 page 65
```
  assumes "state_independent (Rep_program A)"
  shows "observationally_equivalent
  (LET A (ABSTRACT xa (LET (LIFT B) (ABSTRACT xb C))))
  (LET B (ABSTRACT xb (LET (LIFT A) (ABSTRACT xa (SWAPn 0 C)))))"
```

(525) lemma obseq_rule_state_independent_line_swapping_left:
```
  assumes "is_contextfun_typed F ∧ state_independent (Rep_program A) ∧ LIFTk 0 B = LB"
  assumes "observationally_equivalent
    (F (LET B (ABSTRACT xb (LET (LIFTk 0 A) (ABSTRACT xa (SWAPn 0 C)))))) S"
  shows   "observationally_equivalent
    (F (LET A (ABSTRACT xa (LET (LB) (ABSTRACT xb C))))) S"
```

(526) lemma state_independent_Function_v:
```
  assumes "v ∈ values"
  shows "state_independent (Function f v)"
```

(527) lemma state_independent_beta:
```
  assumes "is_value v"
  assumes "state_independent (substitute' 0 p v)"
  shows "state_independent (Application (Abstraction p) v)"
```

I.10 Composing 1-1 One-way Functions

I.10.1 Definitions

(528) definition Section 1.13.1 page 68
```
  "one_way_game f A n ==
  ¨let x ← [mk_kernel2 (λn. uniform_distribution {w. length w = n})] ^n;
       y ← ^f x;
       x'← :A: (:unary_parameter n:, y)
   in ^f x = ^f x'¨"
```

(529) definition strongly_oneway :: "(bitstring ⇒ bitstring) ⇒ bool" **where** Definition 1.108 page 68
```
  "strongly_oneway f == efficiently_computable f ∧
  (∀A. polynomial_time A ⟶ negligible (λn. prog_probability (one_way_game f A n)))"
```

I.10.2 The Sequence of Games

(530) definition Section 1.13.1 page 69
```
  "fof_game1 f A n ==
  ¨let x ← [mk_kernel2 (λn. uniform_distribution {w. length w = n})] ^n;
       y ← ^f(^f x);
       x'← :A: (:unary_parameter n:, y)
   in ^(f o f) x = ^(f o f) x'¨"
```

Section 1.13.1 page 69
(531) definition
```
"fof_game2 f A n ==
¨let x ← [mk_kernel2 (λn. uniform_distribution {w. length w = n})] ^n;
     z ← ^f x;
     y ← ^f z;
     x'← :A: (:unary_parameter n:, y)
 in ^(f o f) x = ^(f o f) x'¨"
```

Section 1.13.1 page 70
(532) definition
```
"fof_game3 f A n ==
¨let x ← [mk_kernel2 (λn. uniform_distribution {w. length w = n})] ^n;
     z ← ^f x;
     y ← ^f z;
     x'← (λy'. :A: y') (:unary_parameter n:, y)
 in ^(f o f) x = ^(f o f) x'¨"
```

Section 1.13.1 page 70
(533) definition
```
"fof_game4 f A n ==
¨let x ← [mk_kernel2 (λn. uniform_distribution {w. length w = n})] ^n;
     z ← ^f x;
     x'← (λy'. :A: y') (:unary_parameter n:, ^f z)
 in ^f x = ^f x'¨"
```

Section 1.13.1 page 71
(534) definition `"adversary_o_f A f = ¨λnz. (λy'. :A: y') (#1 nz, ^f (#2 nz))¨"`

Section 1.13.1 page 71
(535) lemma `obseq_opeq_f:` **shows**
```
"observationally_equivalent_untyped
   (Application
     (Application (prog_embedding (op = :: 'b⇒'b⇒bool))
       (Application (prog_embedding f) (Var m)))
     (Application (prog_embedding f) (Var n)))
   (Application
     (Application (prog_embedding (λx x'. f x = f x'))
       (Var m))
     (Var n))"
```

Section 1.13.1 page 69
(536) lemma `obseq_app_fof:`
```
shows "observationally_equivalent_untyped
   (Application (prog_embedding (f ∘ f)) (Var n))
   (Application (prog_embedding f) (Application (prog_embedding f) (Var n)))"
```

Lemma 1.110 page 69
(537) lemma `fof_step1:`
```
shows "observationally_equivalent (one_way_game (f o f) A n) (fof_game1 f A n)"
```

Lemma 1.111 page 70
(538) lemma `fof_step2:`
```
shows "observationally_equivalent (fof_game1 f A n) (fof_game2 f A n)"
```

Lemma 1.112 page 70
(539) lemma `fof_step3:`
```
shows "observationally_equivalent (fof_game2 f A n) (fof_game3 f A n)"
```

Lemma 1.113 page 70
(540) lemma `fof_step4:`
```
assumes "inj f"
shows "observationally_equivalent (fof_game3 f A n) (fof_game4 f A n)"
```

Lemma 1.115 page 71
(541) lemma `fof_step5:`
```
shows "observationally_equivalent
  (fof_game4 f A n) (one_way_game f (adversary_o_f A f) n)"
```

Theorem 1.116 page 72
(542) theorem `fof_reduction:`
```
assumes "inj f"
shows "prog_probability (one_way_game (f o f) A n)
     = prog_probability (one_way_game f (adversary_o_f A f) n)"
```

(543) theorem `fof_oneway:`
```
assumes "inj f"
assumes "strongly_oneway f"
```

```
  assumes "efficiently_computable (f o f)"
  assumes "!!A::⟦bitstring × bitstring ⇒ bitstring⟧.
    polynomial_time A ⟹ polynomial_time (adversary_o_f A f)"
  shows "strongly_oneway (f o f)"
```

I.11 IND-CPA Security of ElGamal

I.11.1 Definitions

(544) record
```
(’e,’d,’m,’c) encryption_scheme =
  encGen   :: "⟦’e × ’d⟧"
  encEnc   :: "⟦’e ⇒ ’m ⇒ ’c⟧"
  encDec   :: "⟦’d ⇒ ’c ⇒ ’m⟧"
  encDom   :: "’m set"
```

(545) definition Section 1.13.2 page 73
```
"IND_CPA_Game1 ES A (B::nat⇒⟦’e::program_type × bitstring ⇒ bool⟧) n
  == ¨let ed ← :encGen (ES n):;
        m12a ← :A n: (#1 ed);
           c ← :encEnc (ES n): (#1 ed) (#1 m12a)
      in :B n: (c,#2(#2 m12a))¨"
```

(546) definition Section 1.13.2 page 73
```
"IND_CPA_Game2 ES A (B::nat⇒⟦’e::program_type × bitstring ⇒ bool⟧) n
  == ¨let ed ← :encGen (ES n):;
        m12a ← :A n: (#1 ed);
           c ← :encEnc (ES n): (#1 ed) (#1(#2 m12a))
      in :B n: (c,#2(#2 m12a))¨"
```

(547) definition Definition 1.117 page 73
```
"IND_CPA_adversary ES A B ==
  nonuniform_polynomial_time A ∧ nonuniform_polynomial_time B ∧
  (∀n v s e. is_value v ⟶
          (∀vse ← apply_kernel denotation (Application (Rep_program (A n)) v,s,e).
           ∃m12a ∈ (encDom (ES n) × encDom (ES n) × (UNIV::bitstring set)).
           (fst vse) = (prog_embedding m12a)))"
```

(548) definition Definition 1.118 page 73
```
"IND_CPA_secure ES ==
  ∀A B. IND_CPA_adversary ES A B ⟶
  negligible (λn. |Pr[:IND_CPA_Game1 ES A B n:] - Pr[:IND_CPA_Game2 ES A B n:]|)"
```

(549) definition Section 1.119 page 75
```
"ElGamal_Gen G g (q::nat) ==
   ¨let x ← [mk_kernel2 (λu. uniform_distribution {1 .. q})] ^();
      gx ← [deterministic_kernel (λ(a,b). (pow G a b))] (^g,x)
    in (gx,x)¨"
```

(550) definition Section 1.119 page 75
```
"ElGamal_Enc G g (q::nat) ==
¨λgx. λm. (let y ← [mk_kernel2 (λu. uniform_distribution {1 .. q})] ^();
              gy ← [deterministic_kernel (λ(a,b). (pow  G a b))] (^g,y);
             gxy ← [deterministic_kernel (λ(a,b). (pow  G a b))] (gx,y);
            mgxy ← [deterministic_kernel (λ(a,b). (mult G a b))] (m,gxy)
           in (gy,mgxy))¨"
```

(551) definition Section 1.119 page 75
```
"ElGamal_Dec G g (q::nat) ==
¨λx. λgymgxy . (let gxy ← [deterministic_kernel (λ(a,b). (pow G a b))](#1 gymgxy,x);
                   igxy ← [deterministic_kernel (λa. (m_inv G a))] gxy
                in [deterministic_kernel (λ(a,b). (mult G a b))] (#2 gymgxy,igxy))¨"
```

(552) definition
```
"ElGamal G g q (n::nat) ==
  ⦇encGen = ElGamal_Gen (G n) (g n) (q n),
   encEnc = ElGamal_Enc (G n) (g n) (q n),
   encDec = ElGamal_Dec (G n) (g n) (q n),
   encDom = carrier (G n)
```

```
⟧"
```

Section 1.119 page 75

(553) definition

```
"DDHxy G g (q::nat⇒nat)n ==
¨let x ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gx ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),x);
    y ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),y);
  gxy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (gx,y)
 in (gx,gy,gxy)¨"
```

Section 1.119 page 75

(554) definition

```
"DDHz G g (q::nat⇒nat) n ==
¨let x ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gx ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),x);
    y ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),y);
    z ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gz ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),z)
 in (gx,gy,gz)¨"
```

Definition 1.121 page 75

(555) definition `"Diffie_Hellman G g q ==`

```
computationally_indistinguishable (DDHxy G g q) (DDHz G g q)"
```

I.11.2 Cyclic Groups

(556) lemma `project_uniform_distribution:`

```
assumes "finite S ∧ S ≠ {}"
assumes "is_measurable f"
assumes "inj_on f S"
shows "project_measure f (mk_subprobability (uniform_distribution S))
     = mk_subprobability (uniform_distribution (f ‘ S))"
```

(557) lemma `cyclic_mult_inj:`

```
assumes mG: "monoid G"
assumes gen: "carrier G = (pow G g) ‘ {Suc 0 .. q}"
assumes injp: "inj_on (pow G g) {Suc 0 .. q}"
assumes gG: "g ∈ carrier G"
assumes aG: "a ∈ carrier G"
shows "inj_on (mult G a) (carrier G)"
```

Section 1.13.2 page 73

(558) lemma `cyclic_mult_carrier:`

```
assumes mG: "monoid G"
assumes gen: "carrier G = (pow G g) ‘ {Suc 0 .. q}"
assumes injp: "inj_on (pow G g) {Suc 0 .. q}"
assumes gG: "g ∈ carrier G"
assumes aG: "a ∈ carrier G"
shows "(λx. mult G a (pow G g x)) ‘ {Suc 0 .. q} = (carrier G)"
```

Lemma 1.120 page 74

(559) lemma `mult_with_random_element:`

```
assumes mG: "monoid (G n)"
assumes gen: "carrier (G n) = (pow (G n) (g n)) ‘ {Suc 0 .. (q n)}"
assumes injp: "inj_on (pow (G n) (g n)) {Suc 0 .. (q n)}"
assumes gG: "g n ∈ carrier (G n)"
assumes fG: "(fst m12a) ∈ carrier (G n)"
assumes sG: "(fst(snd m12a)) ∈ carrier (G n)"
shows "denotationally_equivalent
(Rep_program
¨let z ← [mk_kernel2 (λu. uniform_distribution {1 .. ((q::nat⇒nat) n)})] ^();
   gz ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),z)
 in [deterministic_kernel (λ(a,b). (mult(G n) a b))] (#1 ^m12a,gz)¨)
(Rep_program
¨let z ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
```

```
    gz ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),z)
 in [deterministic_kernel (λ(a,b). (mult(G n) a b))] (#1(#2 ^m12a),gz)¨)"
```

I.11.3 The Sequence of Games

(560) definition Section 1.119 page 75

```
"ElGamal_Game1a G g q A B n ==
¨let x ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gx ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),x);
  m12a ← :A n: gx;
    y ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),y);
  gxy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (gx,y);
  mgxy ← [deterministic_kernel (λ(a,b). (mult(G n) a b))] (#1 m12a,gxy);
    c ← (gy,mgxy)
 in :B n: (c,#2(#2 m12a))¨"
```

(561) definition

```
"ElGamal_Game2a G g q A B n ==
¨let x ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gx ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),x);
  m12a ← :A n: gx;
    y ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),y);
  gxy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (gx,y);
  mgxy ← [deterministic_kernel (λ(a,b). (mult(G n) a b))] (#1 (#2 m12a),gxy);
    c ← (gy,mgxy)
 in :B n: (c,#2(#2 m12a))¨"
```

(562) lemma `ElGamal_step1a:` Lemma 1.123 page 77

```
"observationally_equivalent (IND_CPA_Game1 (ElGamal G g q) A B n)
                            (ElGamal_Game1a G g q A B n)"
```

(563) lemma `ElGamal_step2a:`

```
"observationally_equivalent (IND_CPA_Game2 (ElGamal G g q) A B n)
                            (ElGamal_Game2a G g q A B n)"
```

(564) definition Section 1.119 page 77

```
"ElGamal_Game1b G g q A B n ==
¨let x ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gx ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),x);
    y ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),y);
  gxy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (gx,y);
  m12a ← :A n: gx;
  mgxy ← [deterministic_kernel (λ(a,b). (mult(G n) a b))] (#1 m12a,gxy);
    c ← (gy,mgxy)
 in :B n: (c,#2(#2 m12a))¨"
```

(565) definition

```
"ElGamal_Game2b G g q A B n ==
¨let x ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gx ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),x);
    y ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),y);
  gxy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (gx,y);
  m12a ← :A n: gx;
  mgxy ← [deterministic_kernel (λ(a,b). (mult(G n) a b))] (#1 (#2 m12a),gxy);
    c ← (gy,mgxy)
 in :B n: (c,#2(#2 m12a))¨"
```

(566) lemma `ElGamal_step1b:` Lemma 1.124 page 77

```
"observationally_equivalent (ElGamal_Game1a G g q A B n)
```

```
                                (ElGamal_Game1b G g q A B n)"
```

(567) lemma ElGamal_step2b:

```
  "observationally_equivalent (ElGamal_Game2a G g q A B n)
                              (ElGamal_Game2b G g q A B n)"
```

Section 1.119 page 77

(568) definition

```
  "DDH_adversary1 G A B n == ¨λx.
  (let m12a ← :A n: (#1 x);
       mgxy ← [deterministic_kernel(λ(a,b).(mult(G n) a b))](#1 m12a,#2(#2 x));
          c ← (#1(#2 x),mgxy)
   in :B n: (c,#2(#2 m12a)))¨"
```

(569) definition

```
  "DDH_adversary2 G A B n == ¨λx.
  (let m12a ← :A n: (#1 x);
       mgxy ← [deterministic_kernel(λ(a,b).(mult(G n) a b))] (#1 (#2 m12a),#2(#2 x));
          c ← (#1(#2 x),mgxy)
   in :B n: (c,#2(#2 m12a)))¨"
```

Section 1.119 page 77

(570) definition

```
  "ElGamal_Game1c G g q A B n == (¨:DDH_adversary1 G A B n: :DDHxy G g q n:¨)"
```

(571) definition

```
  "ElGamal_Game2c G g q A B n == (¨:DDH_adversary2 G A B n: :DDHxy G g q n:¨)"
```

Lemma 1.125 page 77

(572) lemma ElGamal_step1c:

```
  "observationally_equivalent (ElGamal_Game1b G g q A B n)
                              (ElGamal_Game1c G g q A B n)"
```

(573) lemma ElGamal_step2c:

```
  "observationally_equivalent (ElGamal_Game2b G g q A B n)
                              (ElGamal_Game2c G g q A B n)"
```

Section 1.119 page 78

(574) definition

```
  "ElGamal_Game1d G g q A B n == (¨:DDH_adversary1 G A B n: :DDHz G g q n:¨)"
```

(575) definition

```
  "ElGamal_Game2d G g q A B n == (¨:DDH_adversary2 G A B n: :DDHz G g q n:¨)"
```

Lemma 1.126 page 78

(576) lemma ElGamal_step1d:

```
  assumes "Diffie_Hellman G g q"
  assumes "nonuniform_polynomial_time (DDH_adversary1 G A B)"
  shows   "negligible (λn. | prog_probability (ElGamal_Game1c G g q A B n)
                           - prog_probability (ElGamal_Game1d G g q A B n) | )"
```

(577) lemma ElGamal_step2d:

```
  assumes "Diffie_Hellman G g q"
  assumes "nonuniform_polynomial_time (DDH_adversary2 G A B)"
  shows   "negligible (λn. | prog_probability (ElGamal_Game2c G g q A B n)
                           - prog_probability (ElGamal_Game2d G g q A B n) | )"
```

Section 1.119 page 78

(578) definition

```
  "ElGamal_Game1e G g q A B n ==
  ¨let x ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
      gx ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),x);
       y ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
      gy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),y);
    m12a ← :A n: gx;
     mgz ← (let z ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
                gz ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),z)
            in [deterministic_kernel (λ(a,b). (mult(G n) a b))] (#1 m12a,gz));
       c ← (gy,mgz)
   in :B n: (c,#2(#2 m12a))¨"
```

(579) definition Section 1.119 page 78

```
"ElGamal_Game2e G g (q::nat⇒nat) A B n ==
¨let x ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gx ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),x);
    y ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
   gy ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),y);
 m12a ← :A n: gx;
  mgz ← (let z ← [mk_kernel2 (λu. uniform_distribution {1 .. (q n)})] ^();
             gz ← [deterministic_kernel (λ(a,b). (pow (G n) a b))] (^(g n),z)
         in [deterministic_kernel (λ(a,b). (mult(G n) a b))] (#1(#2 m12a),gz));
    c ← (gy,mgz)
 in :B n: (c,#2(#2 m12a))¨"
```

(580) lemma `ElGamal_step1e:` Lemma 1.127 page 78

```
"observationally_equivalent (ElGamal_Game1d G g q A B n)
                            (ElGamal_Game1e G g q A B n :: ⟦bool⟧)"
```

(581) lemma `ElGamal_step2e:`

```
"observationally_equivalent (ElGamal_Game2d G g q A B n)
                            (ElGamal_Game2e G g q A B n :: ⟦bool⟧)"
```

(582) lemma `ElGamal_step12:` **assumes** `mG: "monoid (G n)"` Lemma 1.128 page 78

```
assumes gen: "carrier (G n) = (pow (G n) (g n)) ‘ {Suc 0 .. (q n)}"
assumes injp: "inj_on (pow (G n) (g n)) {Suc 0 .. ((q::nat⇒nat) n)}"
assumes gG: "g n ∈ carrier (G n)"
assumes cpaadv: "IND_CPA_adversary (ElGamal G g q) A B"
shows "observationally_equivalent (ElGamal_Game1e G g q A B n)
                                  (ElGamal_Game2e G g q A B n)"
```

(583) theorem `ElGamal_IND_CPA:` Theorem 1.129 page 79

```
assumes mG:  "!!n. monoid (G n)"
assumes gen: "!!n. carrier (G n) = (pow (G n) (g n)) ‘ {Suc 0 .. (q n)}"
assumes injp:"!!n. inj_on (pow (G n) (g n)) {Suc 0 .. ((q::nat⇒nat) n)}"
assumes gG:  "!!n. g n ∈ carrier (G n)"
assumes DDH: "Diffie_Hellman G g q"
assumes poly:"!!(A::nat⇒⟦'a⇒'a×'a×bitstring⟧)
               (B::nat ⇒ ⟦('a × 'a) × bitstring ⇒ bool⟧).
                nonuniform_polynomial_time A ∧ nonuniform_polynomial_time B
            ⟹ (nonuniform_polynomial_time (DDH_adversary1 G A B) ∧
                nonuniform_polynomial_time (DDH_adversary2 G A B))"
shows "IND_CPA_secure (ElGamal G g q)"
```

Bibliography

[1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[2] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography: The computational soundness of formal encryption. In *Theoretical Computer Science – IFIP TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2000.

[3] Reynald Affeldt, Miki Tanaka, and Nicolas Marti. Formal proof of provable security by game-playing in a proof assistant. In *Provable Security – ProvSec 2007*, volume 4784 of *Lecture Notes in Computer Science*, pages 151–168. Springer, 2007.

[4] José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In *Computer and Communications Security – CCS 2012*, pages 488–500. ACM, 2012.

[5] Elena Andreeva, Atul Luykx, and Bart Mennink. Provable security of BLAKE with non-ideal compression function. In *Selected Areas in Cryptography – SAC 2012*, volume 7707 of *Lecture Notes in Computer Science*, pages 321–338. Springer, 2013.

[6] Elena Andreeva, Bart Mennink, and Bart Preneel. On the indifferentiability of the Grøstl hash function. In *Security in Communication Networks – SCN 2010*, volume 6280 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2010.

[7] Elena Andreeva, Bart Mennink, and Bart Preneel. Security reductions of the second round SHA-3 candidates. In *Information Security Conference – ISC 2010*, volume 6531 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2011.

[8] Elena Andreeva, Bart Mennink, and Bart Preneel. The parazoa family: generalizing the sponge hash functions. *International Journal of Information Security*, 11(3):149–165, 2012.

[9] Elena Andreeva, Bart Mennink, Bart Preneel, and Marjan Škrobot. Security analysis and comparison of the SHA-3 finalists BLAKE, Grøstl, JH, Keccak, and Skein. In *Progress in Cryptology – AFRICACRYPT 2012*, volume 7374 of *Lecture Notes in Computer Science*, pages 287–305. Springer, 2012.

[10] Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-property-preserving iterated hashing: ROX. In *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2007.

[11] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, and M. Llanos Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: Breaking the SAML-based single sign-on for Google apps. In *Formal Methods in Security Engineering – FMSE 2008*, pages 1–10. ACM, 2008.

[12] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE, 2010.

[13] Michael Backes, Gilles Barthe, Matthias Berg, Benjamin Grégoire, César Kunz, Malte Skoruppa, and Santiago Zanella Béguelin. Verified security of Merkle-Damgård. In *Computer Security Foundations Symposium – CSF 2012*, pages 354–368. IEEE, 2012.

[14] Michael Backes, Matthias Berg, and Dominique Unruh. A formal language for cryptographic pseudocode. In *Logic for Programming, Artificial Intelligence, and Reasoning – LPAR 2008*, volume 5330 of *Lecture Notes in Computer Science*, pages 353–376. Springer, 2008.

[15] Michael Backes, Dennis Hofheinz, and Dominique Unruh. CoSP: a general framework for computational soundness proofs. In *Computer and Communications Security – CCS 2009*, pages 66–78. ACM, 2009.

[16] Michael Backes and Christian Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *Symposium on Theoretical Aspects of Computer Science – STACS 2003*, volume 2607 of *Lecture Notes in Computer Science*, pages 675–686. Springer, 2003.

[17] Michael Backes, Christian Jacobi, and Birgit Pfitzmann. Deriving cryptographically sound implementations using composition and formally verified bisimulation. In *Formal Methods Europe – FME 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 310–329. Springer, 2002.

[18] Michael Backes and Peeter Laud. Computationally sound secrecy proofs by mechanized flow analysis. In *Computer and Communications Security – CCS 2006*, pages 370–379. ACM, 2006.

[19] Michael Backes, Matteo Maffei, and Dominique Unruh. Computationally sound verification of source code. In *Computer and Communications Security – CCS 2010*, pages 387–398. ACM, 2010.

[20] Michael Backes and Birgit Pfitzmann. Computational probabilistic noninterference. In *European Symposium on Research in Computer Security – ESORICS 2002*, volume 2502 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2002.

[21] Michael Backes and Birgit Pfitzmann. Symmetric encryption in a simulatable Dolev-Yao style cryptographic library. In *Computer Security Foundations Workshop – CSFW 2004*, pages 204–218. IEEE Computer Society, 2004.

[22] Michael Backes and Birgit Pfitzmann. Relating cryptographic und symbolic key secrecy. *IEEE Transactions on Dependable and Secure Computing*, 2(2):109–123, 2005.

[23] Michael Backes, Birgit Pfitzmann, Michael Steiner, and Michael Waidner. Polynomial fairness and liveness. In *Computer Security Foundations Workshop – CSFW 2002*, pages 160–174. IEEE Computer Society, 2002.

[24] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A composable cryptographic library with nested operations. In *Computer and Communications Security – CCS 2003*, pages 220–230. ACM, 2003.

[25] Michael Backes and Dominique Unruh. Computational soundness of symbolic zero-knowledge proofs against active attackers. In *Computer Security Foundations Symposium – CSF 2008*, pages 255–269. IEEE Computer Society, 2008.

[26] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. Computer-aided cryptographic proofs. In *Interactive Theorem Proving – ITP 2012*, volume 7406 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2012.

[27] Gilles Barthe, Marion Daubignard, Bruce M. Kapron, and Yassine Lakhnech. Computational indistinguishability logic. In *Computer and Communications Security – CCS 2010*, pages 375–386. ACM, 2010.

[28] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, Federico Olmedo, and Santiago Zanella Béguelin. Verified indifferentiable hashing into elliptic curves. In *Principles of Security and Trust – POST 2012*, volume 7215 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2012.

[29] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Formal certification of ElGamal encryption. A gentle introduction to CertiCrypt. In *Formal Aspects in Security and Trust – FAST 2008*, volume 5491 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2009.

[30] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[31] Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. Beyond provable security. Verifiable IND-CCA security of OAEP. In *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2011.

[32] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Principles of Programming Languages – POPL 2009*, pages 90–101. ACM, 2009.

[33] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Programming language techniques for cryptographic proofs. In *Interactive Theorem Proving – ITP 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2010.

[34] Gilles Barthe, Daniel Hedin, Santiago Zanella Béguelin, Benjamin Grégoire, and Sylvain Heraud. A machine-checked formalization of Sigma-protocols. In *Computer Security Foundations Symposium – CSF 2010*, pages 246–260. IEEE Computer Society, 2010.

[35] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic reasoning for differential privacy. In *Principles of Programming Languages – POPL 2012*, pages 97–110. ACM, 2012.

[36] Gilles Barthe, Federico Olmedo, and Santiago Zanella Béguelin. Verifiable security of Boneh-Franklin identity-based encryption. In *Provable Security – ProvSec 2011*, volume 6980 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2011.

[37] Gilles Barthe, David Pointcheval, and Santiago Zanella Béguelin. Verified security of redundancy-free encryption from Rabin and RSA. In *Computer and Communications Security – CCS 2012*, pages 724–735. ACM, 2012.

[38] David Basin, Sebastian Mödersheim, and Luca Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[39] Heinz Bauer. *Wahrscheinlichkeitstheorie*. de Gruyter, fourth edition, 1991.

[40] Mihir Bellare, Tadayoshi Kohno, Stefan Lucks, Niels Ferguson, Bruce Schneier, Doug Whiting, Jon Callas, and Jesse Walker. Provable security support for the Skein hash family, 2009.

[41] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Computer and Communications Security – CCS 1993*, pages 62–73. ACM, 1993.

[42] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006.

[43] Guido Bertoni, Joan Daemen, Michaël Peeters, Assche, and Gilles Van. The KECCAK reference, 2011.

[44] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[45] Rishiraj Bhattacharyya, Avradip Mandal, and Mridul Nandi. Security analysis of the mode of JH hash function. In *Fast Software Encryption – FSE 2010*, volume 6147 of *Lecture Notes in Computer Science*, pages 168–191. Springer, 2010.

[46] John Black, Martin Cochran, and Thomas Shrimpton. On the impossibility of highly-efficient blockcipher-based hash functions. *Journal of Cryptology*, 22(3):311–329, 2009.

[47] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Computer Security Foundations Workshop – CSFW 2001*, pages 82–96. IEEE Computer Society, 2001.

[48] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security & Privacy – S&P 2006*, pages 140–154. IEEE Computer Society, 2006.

[49] Bruno Blanchet. Automatically verified mechanized proof of one-encryption key exchange. In *Computer Security Foundations Symposium – CSF 2012)*, pages 325–339. IEEE, 2012.

[50] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In *Principles of Security and Trust – POST 2012*, volume 7215 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2012.

[51] Bruno Blanchet, Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay. Computationally sound mechanized proofs for basic and public-key Kerberos. In *ACM Symposium on Information, Computer and Communications Security – ASIACCS 2008*, pages 87–99. ACM, 2008.

[52] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 2006.

[53] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology – CRYPTO 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.

[54] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2010.

[55] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science – FOCS 2001*, pages 136–145. IEEE Computer Society, 2001.

[56] Ran Canetti and Marc Fischlin. Universally composable commitments. In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2001.

[57] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *Journal of the ACM*, 51(4):557–594, 2004.

[58] Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of mutual authentication and key exchange protocols. In *Theory of Cryptography Conference – TCC 2006*, volume 3876 of *Lecture Notes in Computer Science*, pages 380–403. Springer, 2006.

[59] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2002.

[60] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Symposium on Theory of Computing – STOC 2002*, pages 494–503. ACM, 2002.

[61] Anne Canteaut, Thomas Fuhr, María Naya-Plasencia, Pascal Paillier, Jean-René Reinhard, and Marion Videau. A unified indifferentiability proof for permutation- or block cipher-based hash functions. Cryptology ePrint Archive, Report 2012/363, 2012.

[62] Donghoon Chang, Mridul Nandi, and Moti Yung. Indifferentiability of the hash algorithm BLAKE. Cryptology ePrint Archive, Report 2011/623, 2011.

[63] The Coq development team. The Coq proof assistant reference manual, Version 8.3. `http://coq.inria.fr`, 2010.

[64] Pierre Corbineau, Mathilde Duclos, and Yassine Lakhnech. Certified security proofs of cryptographic protocols in the computational model: An application to intrusion resilience. In *Certified Programs and Proofs – CPP 2011*, volume 7086 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2011.

[65] Ricardo Corin and J. den Hartog. A probabilistic hoare-style logic for game-based cryptographic proofs. In *International Colloquium on Automata, Languages and Programming – ICALP 2006*, volume 4052 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2006.

[66] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[67] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning*, 46(3–4):225–259, 2011.

[68] Véronique Cortier and Bogdan Warinschi. Computationally sound, automated proofs for security protocols. In *European Symposium on Programming – ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2005.

[69] Judicaël Courant, Marion Daubignard, Cristian Ene, Pascal Lafourcade, and Yassine Lakhnech. Towards automated proofs for asymmetric encryption schemes in the random oracle model. In *Computer and Communications Security – CCS 2008*, pages 371–380. ACM, 2008.

[70] Ivan Damgård. A design principle for hash functions. In *Advances in Cryptology – CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1990.

[71] Marion Daubignard, Pierre-Alain Fouque, and Yassine Lakhnech. Generic indifferentiability proofs of hash designs. In *Computer Security Foundations Symposium – CSF 2012*, pages 340–353. IEEE, 2012.

[72] Nicolas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicæ*, 34:381–392, 1972.

[73] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[74] Jonathan Driedger. Formalization of game-transformations. Bachelor's thesis, Saarland University, 2010.

[75] Shimon Even and Oded Goldreich. On the security of multi-party ping-pong protocols. In *Symposium on Foundations of Computer Science – FOCS 1983*, pages 34–39. IEEE Computer Society, 1983.

[76] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whithing, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein hash function family, 2008.

[77] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2001.

[78] David Galindo. Boneh-Franklin identity based encryption revisited. In *International Colloquium on Automata, Languages and Programming – ICALP 2005*, volume 3580 of *Lecture Notes in Computer Science*, pages 791–802. Springer, 2005.

[79] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology – CRYPTO 1984*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1985.

[80] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. `Grøstl` – a SHA-3 candidate, 2011.

[81] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.

[82] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[83] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005.

[84] Paul R. Halmos. *Measure Theory*, volume 18 of *Graduate Texts in Mathematics*. Springer-Verlag, 1974.

[85] Russell Impagliazzo and Bruce M. Kapron. Logics for reasoning about cryptographic constructions. *Journal of Computer and System Sciences*, 72(2):286–320, 2006.

[86] Richard Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, 1989.

[87] Dmitry Khovratovich, Ivica Nikolić, and Christian Rechberger. Rotational rebound attacks on reduced Skein. In *Advances in Cryptology – ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2010.

[88] Peeter Laud. Semantics and program analysis of computationally secure information flow. In *European Symposium on Programming – ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2001.

[89] Peeter Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *IEEE Symposium on Security & Privacy – S&P 2004*, pages 71–85. IEEE Computer Society, 2004.

[90] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for Construction and Analysis of Systems – TACAS 1996*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[91] Stefan Lucks. A failure-friendly design principle for hash functions. In *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.

[92] Stéphane Manuel. Classification and generation of disturbance vectors for collision attacks against SHA-1. *Designs, Codes and Cryptography*, 59(1–3):247–263, 2011.

[93] Ian Mason and Carolyn Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

[94] Ueli Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *Theory of Cryptography Conference – TCC 2004*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2004.

[95] Catherine Meadows. Using narrowing in the analysis of key management protocols. In *IEEE Symposium on Security & Privacy – S&P 1989*, pages 138–147. IEEE Computer Society, 1989.

[96] Ralph Merkle. One way hash functions and DES. In *Advances in Cryptology – CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1990.

[97] Michael Merritt. *Cryptographic Protocols*. PhD thesis, Georgia Institute of Technology, 1983.

[98] Daniele Micciancio and Bogdan Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Theory of Cryptography Conference – TCC 2004*, volume 2951 of *Lecture Notes in Computer Science*, pages 133–151. Springer, 2004.

[99] Jonathan K. Millen. The interrogator: A tool for cryptographic protocol security. In *IEEE Symposium on Security & Privacy – S&P 1984*, pages 134–141. IEEE Computer Society, 1984.

[100] Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In *Theory and Applications of Satisfiability Testing – SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2006.

[101] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

[102] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[103] David Nowak. A framework for game-based security proofs. In *Information Security and Cryptology – ICISC 2007*, volume 4861 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008.

[104] David Nowak. On formal verification of arithmetic-based cryptographic primitives. In *Information Security and Cryptology – ICISC 2008*, volume 5461 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2009.

[105] David Nowak and Yu Zhang. A calculus for game-based security proofs. In *Provable Security – ProvSec 2010*, volume 6402 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2010.

[106] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[107] Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *IEEE Symposium on Security & Privacy – S&P 2001*, pages 184–200. IEEE Computer Society, 2001.

[108] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[109] Stefan Richter. Integration theory and random variables. *Archive of Formal Proofs*, 2004.

[110] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 487–506. Springer, 2011.

[111] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Fast Software Encryption – FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 371–388. Springer, 2004.

[112] Adi Shamir. A polynomial-time algorithm for breaking the basic Merkle-Hellman cryptosystem. *IEEE Transactions on Information Theory*, 30(5):699–704, 1984.

[113] Victor Shoup. OAEP reconsidered. In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 239–259. Springer, 2001.

[114] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004.

[115] Malte Skoruppa. Formal verification of ElGamal encryption using a probabilistic lambda-calculus. Bachelor's thesis, Saarland University, 2010.

[116] Malte Skoruppa. Verifiable security of prefix-free Merkle-Damgård. Master's thesis, Saarland University, 2012.

[117] Christoph Sprenger, Michael Backes, David A. Basin, Birgit Pfitzmann, and Michael Waidner. Cryptographically sound theorem proving. In *Computer Security Foundations Workshop – CSFW 2006*, pages 153–166. IEEE Computer Society, 2006.

[118] Christoph Sprenger and David A. Basin. Cryptographically-sound protocol-model abstractions. In *Computer Security Foundations Symposium – CSF 2008*, pages 115–129. IEEE Computer Society, 2008.

[119] F. Javier Thayer Fabrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *IEEE Symposium on Security & Privacy – S&P 1998*, pages 160–171. IEEE Computer Society, 1998.

[120] Diana Toma and Dominique Borrione. Formal verification of a SHA-1 circuit core using ACL2. In *Theorem Proving in Higher Order Logics – TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2005.

[121] Xiaoyun Wang, Yiqun Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[122] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[123] Hongjun Wu. The hash function JH, 2011.

[124] Santiago Zanella Béguelin. *Formal Certification of Game-Based Cryptographic Proofs*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2010.

[125] Santiago Zanella Béguelin, Benjamin Grégoire, Gilles Barthe, and Federico Olmedo. Formally certifying the security of digital signature schemes. In *IEEE Symposium on Security & Privacy – S&P 2009*, pages 237–250. IEEE Computer Society", 2009.

[126] Yu Zhang. The computational SLR: a logic for reasoning about computational indistinguishability. *Mathematical Structures in Computer Science*, 20(5):951–975, 2010.