Fachrichtung 6.2 - Informatik
Naturwissenschaftlich-Technische Fakultät I
- Mathemathik und Informatik -
Universität des Saarlandes

Formalization of Game-Transformations

Bachelorarbeit

Angefertigt unter der Leitung von Prof. Dr. Michael Backes

Betreuung durch Prof. Dr. Michael Backes und M.Sc. Matthias Berg

Begutachtet von Prof. Dr. Michael Backes und Dr. Dominique Unruh

vorgelegt von
Jonathan Driedger

am
12. Januar 2010

Eidesstattliche Erklärung

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

Einverständniserklärung

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

Saarbrücken, 12. Januar 2010

Jonathan Driedger

Danksagung

Als erstes möchte ich mich bei Matthias Berg bedanken für die konsequente Unterstüzung durch die vielen Gespräche, die umfangreiche Bereitstellung von Arbeitsmaterial und die Denkanstöße, die es mir erleichtert haben mich in dem Thema dieser Arbeit zurecht zu finden. Des weiteren möchte ich Prof. Backes dafür danken, dass er mein Interesse an der Kryptographie geweckt hat, für das interessante Thema dieser Arbeit und die Möglichkeit selbige an seinem Lehrstuhl zu schreiben.
Weiterer Dank gilt Malte Skoruppa für die vielen konstruktiven Verbesserungsvorschläge, die lebhaften Diskussionen und die angenehme Zeit beim gemeinsamen Arbeiten. Auch möchte ich Marvin Künnemann danken für das Korrekturlesen dieser Arbeit, sowie meiner Familie, die mich immer wieder motiviert und angetrieben hat.

Abstract

Formalizing and verifying proofs in cryptography has become an important task. Backes et al. therefore invented a framework [1] that uses the proof assistant `Isabelle/HOL` [9] to verify game-based proofs. In this framework a powerful probabilistic language allows to formalize games that describe security properties. To show that these security properties hold one can modify the games such that their outcome is not altered. This is done until the games have the form of already known security properties. Such a modification of a game is called a transformation. Transformations are based on relations between games which have to be verified. But verifying such relations is very often a challenging task. To be able to come up with game-based proofs more naturally it is useful to have a collection of relations, and therefore transformations, verified upfront. This thesis presents a couple of different game-transformations, formalizes them, and shows proofs of their correctness.

Contents

1 Introduction

Game-based proofs are a common technique in cryptography to prove certain security properties (see [5]). The desired property is formulated as a probabilistic process that describes which information an adversary has access to according to the protocol, and how this information is computed. Such a process is called a *game*. The output of such a game may describe the success probability of the adversary. Now, one can modify this game in a way such that the success probability of the adversary is changed at most by a negligible fraction. A modification that fulfills this condition is called a *transformation*. If one is able to apply a finite sequence of transformations to the initial game that transforms it into a game in which the success probability of the adversary is known to be negligible, one has constructed a proof by contradiction: Suppose, the adversary in the initial game had a non negligible success probability. Applying transformations to the game changes the outcome of a game at most by a negligible amount. Since only a finite number of transformations are applied, the outcome of the last game can deviate from the outcome of the initial game at most by a negligible fraction. But since the success probability of the adversary in the last game is negligible, this contradicts the assumption. An example of such a proof can be seen in Figure 10. For a deeper insight in this proof method see [11].
The correctness of every single transformation can be proven on its own. This breaks down the complexity of huge cryptographic proofs and furthermore gives the opportunity to reuse already verified transformations. Nevertheless, in most cases games are described in natural language or at best in pseudo-code. This can lead to ambiguities and mistakes. To avoid this, Backes et al. invented a formal language to describe games and relations between them [1]. The language is powerful enough to express all common constructs used in cryptography like probabilistic behavior or oracles. It is implemented in the proof assistant `Isabelle/HOL` [9] and is already equipped with common relations on games, like different kinds of equivalences. The formalization of this language and its implementation in a powerful theorem proving framework like `Isabelle/HOL` allows to verify even huge proofs with guaranteed correctness.
The goal of this thesis is to find, formalize and verify game-transformations. I will investigate a simple reduction proof and split it in as many decent game-transformations as possible. Every single transformation is formalized in the language developed by Backes et al. as a relation between two games. Afterwards I will give for every transformation a proof that shows that the stated relation holds. The structure of the thesis is as follows: First I will give an overview of the framework developed by Backes et al. in Section 2. Section 3 describes the cryptographic proof from which I extracted the transformations presented in Section 4. I will conclude with an outlook in Section 5 and discuss related work in Section 6.

2 The Framework

The language I present in this Section was invented by Backes et al. [1]. It is a probabilistic higher-order functional language which is able to handle continuous probability measures. Therefore it is powerful enough to argue about infinite constructs or to reason about information-theoretic security guarantees. Because it is functional, it can deal with oracles, which can be seen as higher order objects, in a more natural way than imperative languages. Nevertheless it is not purely functional. A purely functional language would not allow an oracle to keep state during several invocations for example. Although it is possible to transform every program that uses state into a functional program by making the state explicit and passing it around as an object this is not appropriate in this application since the state may contain secrecy properties like keys. Therefore the language allows programs to store data in a separate *store*.
Furthermore, the language provides an iso-recursive typesystem (see [10]) that includes product and sum types. Therefore, the language is flexible and able to express arbitrary datatypes.
Finally, the language provides *events*. Events are a common technique in cryptographic game-based proofs. They can be triggered by certain, usually unwanted, conditions and can help to prove that certain properties of games hold up to an error which is bound by the probability that the event is raised.

In the following I will first explain the *de Bruijn notation* which is used by the language. Afterwards, I will discuss the syntax as well as some syntactical constructs which I will use in later Sections. The Section will close with the semantics of the language and the explanation of some program relations which describe different kinds of program equalities.

2.1 De Bruijn Notation

The language defined in Figure 1 is a *lambda-calculus* (see [10]). It uses the nameless representation, the so called *de Bruijn notation* which can be processed much easier by computers than the named representation. In the unnamed representation, variables are not represented by names (as in $(\lambda x.x)$) but by natural numbers. Therefore lambda-binders do not introduce new names either. The natural number that represents a variable is called a *de Bruijn index* or a *de Bruijn variable*. Such an index tells us relatively to which λ it is bound to. In a program P The variable (var n) is bound to the $(\text{n}+1)^{st}$ λ we pass when travelling the syntaxtree of P from (var n) up to the root.
If there exists a λ in a program P, such that a variable (var n) is linked to this λ we call this variable *bound* in P. On the other side, if (var n) is not bound in P, this variable is called *free* in P. In a program of the form $\lambda.P$ we call the first lambda-binder the *highest lambda*. A variable that is free in a program P but bound in the program $\lambda.P$ is called a *lowest free variable in P*.

Example:
$\lambda x.x$ is the same as $\lambda.$var 0

In the program $\lambda.\lambda.$(var 1, var 2) the variable var 1 is bound and var 2 is free. The variable var 2 is also the lowest free variable of the program since it is bound in the program $\lambda.\lambda.\lambda.$(var 1, var 2).

To make programs more readable, it is common to use let constructs. This is absolutely no problem in the named lambda-calculus since let constructs are just syntactic sugar and do not introduce any ambiguities. Let constructs without names do not bring the advantage of better readability. If there are no lambda-binders, and these are exactly what the let constructs are abstracting away from, it is even more difficult to see what a de Bruijn variable is referring to. Therefore I will use let constructs with names whenever giving examples for the sake of readability and understandability. This can lead to confusion since all formal statements are given in nameless de Bruijn notation. But one is able to transform a program expressed with let constructs and names into one without let constructs with names, and a program with names into one in nameless representation (see [10]).

Example:

$$\begin{tikzcd} \boxed{\begin{array}{llll}\text{let} & x & \leftarrow & 5\\ & y & \leftarrow & 2\\ \text{in} & x+y & & \end{array}} \arrow[r] & \boxed{(\lambda x.(\lambda y.x+y)2)5} \arrow[r] & \boxed{(\lambda.(\lambda.\ (\text{var } 1)+(\text{var } 0))2)5} \end{tikzcd}$$

Explanation: First we note, that every let construct of the form 'let $x \leftarrow P$ in Q' is just syntactic sugar for $(\lambda x.Q)P$. Therefore, we can get rid of all let constructs. Afterward, we can replace every named variable by its corresponding de Bruijn variable. All three representations represent the same program.

This was just a brief and informal overview of de Bruijn notation and the different representations of programs. For a more detailed insight see [10].

2.2 Syntax

The syntax of the language used in the framework is defined in Figure 1. It consists of a probabilistic higher-order lambda calculus with references, iso-recursive types and events. Therefore, it provides all features discussed in the introduction to Section 2.

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

Figure 1: defining grammar for the language ($n \in \mathbb{N}$, $v \in \mathbf{B}$, $X \in \Sigma_{\mathbf{B}}$, s is a string, f denotes a submarkov kernel from V_0 to V_0, $\mathbf{B}$ is a type (see below) and $\Sigma_{\mathbf{B}}$ denotes a sigma algebra over $\mathbf{B}$. For more information on sigma algebras see [3])

A term that is derivable from T we call a *type*, a term derivable from T_0 a *pure type*, a term derivable from P a *program*, a term derivable from V a *value*, and a term derivable from V_0 a *pure value.*

Types T are either types of basic values of a measurable set X denoted by Value_X or compound types. The set of basic types is not fixed in the language for the sake of flexibility but a type $\mathbf{B}$ is assumed that contains at least some important types for sure like unit, a type with a single element *unit*. The construct value v introduces a new element $v \in \mathbf{B}$ in programs. For types T and U there exist the following compound types: $T \times U$ denotes the product type, $T + U$ the sum type and $T \rightarrow U$ the function type that consists of all functions from T to U. Ref T denotes the type of references of type T. The construct $\mu.T$ introduces a new binder in de Bruijn notation and (Tvar n) denotes the typevariable with de Bruijn index n. For the sake of readability I will use $\mu x.T$ to denote named types instead of using de Bruijn notation. Intuitively the $\mu x.T$ construct allows us to give recursive definitions of types. This is because $\mu x.T$ denotes the set of terms resulting from substituting $\mu x.T$ for x in T recursively. For an example see the examplebox. The set of pure types T_0 consists of all types not containing reference or function types.

Programs P are constructed in de Bruijn notation. Let P, Q and W be programs and n a natural number. (var n) denotes a de Bruijn variable and $\lambda.P$ introduces a new lambda binder for the program P (see Section 2.1). The construct $\text{fun}(f, P)$ where f is a submarkov kernel from V_0 to V_0 introduces probabilism. It can be interpreted as applying f to P yielding a probability measure on pure values (see the examplebox in Section 2.4 for a better understanding). It allows us to express every deterministic or probabilistic mathematical function. PQ denotes the application of P on Q. To reference the $(n+1)^{st}$ element in the store (which is represented as a list) one uses loc n. Furthermore ref P denotes reference creation and $!P$ dereferencing. $P := Q$ denotes an assignment Q to P. Pairs are denoted by (P, Q) and their projections by fst W and snd W. To raise an event one uses event s and eventlist gives us a list of all previously raised events. Sums, which means programs with a sumtype, can be constructed by using inl P and inr P and destructed using case P_1 P_2 P_3. For a program P of type T the program inl P has type $T + U$ and inr P has type $U + T$. For a program P of type $\mu x.T$ the construction unfold P changes the type of P by substituting the type $\mu x.T$ for x in T. The inverse operation is denoted by fold P. For more information on iso-recursive types see [10].

Values V are programs of the form value v, var n, $\lambda.P$ or loc n. If V_1 and V_2 are values, than (V_1, V_2), inl V_1, inr V_1 and fold V_1 are values as well. Pure values V_0 are values without locations, variables and lambda-binders.

Example:
We can model the type bool by

$\mathsf{bool} := \text{Value}_{\mathsf{unit}} + \text{Value}_{\mathsf{unit}}$

Therefore we can interpret inl(value *unit*) as the truth-value *true* and inr(value *unit*) as *false*.
An *if then else* construct can then be modeled for programs P_1, P_2 and P_3 as

if P_1 then P_2 else P_3 := case P_1 $(\lambda. \uparrow P_2)$ $(\lambda. \uparrow P_3)$

$\uparrow$ denotes a lifting of all free de Bruijn variables by one. for a detailed explanation and definition see Section 2.3.

As an example take the program

P := if inl(value *unit*) then (var 0) else (var 4)

Although I have not explained the semantics of the language yet I will use it here in a simplified way to explain the expressivity of the language and to give a better understanding of the syntax.

$$\begin{array}{cll} & \text{if inl(value } unit\text{) then (var 0) else (var 4)} & \\ = & \text{case (inl(value } unit\text{)) } (\lambda. \uparrow \text{(var 0)}) \; (\lambda. \uparrow \text{(var 4)}) & \text{by definition} \\ \rightsquigarrow_{sem} & (\lambda. \uparrow \text{(var 0)})(inl(\text{value } unit)) & \text{by semantics} \\ = & (\lambda.\text{(var 1)})(inl(\text{value } unit)) & \text{by definition of } \uparrow \\ \rightsquigarrow_{sem} & \text{(var 0)} & \text{by semantics} \end{array}$$

Explanation: Note that the $\rightsquigarrow_{sem}$ relation is not the real semantic reduction relation of the language. It should only give an intuition of the usage of some constructs of the language. Nevertheless the real reduction relation works in a similar way. That is why we need the extra lambda-binder and the $\uparrow$ operator in the model of the if then else construct. Because of the $\uparrow$ operator the application of $(\lambda. \uparrow \text{(var 0)})$ to $(inl(\text{value } unit))$ has no effect on the initial program (var 0).

We can also model recursive types like lists. The type of a list with elements of type T can be defined by

$\text{list}T := \mu x.(\text{Value}_{\mathsf{unit}} + (T \times x))$.

This is just a closed term that describes the set

$\{unit\} \cup \{(t_1,unit) \mid t_1 \in T\} \cup \{(t_1,(t_2,unit)) \mid t_1, t_2 \in T\} \cup ...$

in the language by recursively inserting $\mu x.(\text{Value}_{\mathsf{unit}} + (T \times x))$ for x in list$T$. Intuitively, we use *unit* as the empty list and $t :: list := (t, list)$ as the cons operation. We can model them formally in the language by

nil := fold(inl(value *unit*)) and $P_1 :: P_2$:= fold(inr(P_1, P_2))

Here the fold changes the type of the programs from $(\text{Value}_{\mathsf{unit}}+(T\times \text{list}T))$ to $\mu x.(\text{Value}_{\mathsf{unit}}+(T\times x))$ and therefore to list$T$.

A Context C is a program that contains an arbitrary amount of holes $\Box$ in which another program P can be inserted. For the program resulting by inserting P in C we write $C[P]$. We call $\Box$ the *empty context*. Whenever $C[P]$ contains no free variables, C is called a *closing context* for P.

2.3 Special Constructs

The Lift Operator $\uparrow$ increases all de Bruijn indices of free variables in a program P by one.

Example:
Let P be a program defined by

$P := \lambda.((\text{var } 0),(\text{var } 1))$

Now, we consider the program $\uparrow P$:

$\uparrow P = \lambda.((\text{var } 0),(\text{var } 2))$

Explanation: The variable (var 0) is bound to the lambda in the program. Therefore the lift operator does not change it. The variable (var 1) is free. The lift operator increases it by one to (var 2).

Formally, we first need to define a function $\mathsf{lift\_vars}$ (see Figure 2). This function has an additional argument named k that tells the function up to which index the variables count as bound. Since our lift operation should not change bound variables, the function has to keep track of this value.

$$\mathsf{lift\_vars} : \mathbb{N} \to \text{program} \to \text{program}$$

$$\begin{aligned}
&\mathsf{lift\_vars}\ k\ (\text{var } i) &&= (\text{if } i < k \text{ then var } i \text{ else var } (i+1))\\
&\mathsf{lift\_vars}\ k\ (\lambda.S) &&= \lambda.(\mathsf{lift\_vars}\ (k+1)\ S)\\
&\mathsf{lift\_vars}\ k\ (ST) &&= (\mathsf{lift\_vars}\ k\ S)\ (\mathsf{lift\_vars}\ k\ T)\\
&\mathsf{lift\_vars}\ k\ (\text{value } v) &&= \text{value } v\\
&\mathsf{lift\_vars}\ k\ (\text{fun}(f,p)) &&= \text{fun}(f,(\mathsf{lift\_vars}\ k\ p))\\
&\mathsf{lift\_vars}\ k\ ((p1,p2)) &&= ((\mathsf{lift\_vars}\ k\ p1),(\mathsf{lift\_vars}\ k\ p2))\\
&\mathsf{lift\_vars}\ k\ (\text{loc } n) &&= \text{loc } n\\
&\mathsf{lift\_vars}\ k\ (\text{ref } p1) &&= \text{ref } (\mathsf{lift\_vars}\ k\ p1)\\
&\mathsf{lift\_vars}\ k\ (!p1) &&= !(\mathsf{lift\_vars}\ k\ p1)\\
&\mathsf{lift\_vars}\ k\ (p1 := p2) &&= (\mathsf{lift\_vars}\ k\ p1) := (\mathsf{lift\_vars}\ k\ p2)\\
&\mathsf{lift\_vars}\ k\ (\text{event } e) &&= \text{event } e\\
&\mathsf{lift\_vars}\ k\ \text{eventlist} &&= \text{eventlist}\\
&\mathsf{lift\_vars}\ k\ (\text{fst } p1) &&= \text{fst } (\mathsf{lift\_vars}\ k\ p1)\\
&\mathsf{lift\_vars}\ k\ (\text{snd } p1) &&= \text{snd } (\mathsf{lift\_vars}\ k\ p1)\\
&\mathsf{lift\_vars}\ k\ (\text{case } p1\ p2\ p3) &&= \text{case } (\mathsf{lift\_vars}\ k\ p1)\ (\mathsf{lift\_vars}\ k\ p2)\ (\mathsf{lift\_vars}\ k\ p3)\\
&\mathsf{lift\_vars}\ k\ (\text{inl } p1) &&= \text{inl } (\mathsf{lift\_vars}\ k\ p1)\\
&\mathsf{lift\_vars}\ k\ (\text{inr } p1) &&= \text{inr } (\mathsf{lift\_vars}\ k\ p1)\\
&\mathsf{lift\_vars}\ k\ (\text{fold } p1) &&= \text{fold } (\mathsf{lift\_vars}\ k\ p1)\\
&\mathsf{lift\_vars}\ k\ (\text{unfold } p1) &&= \text{unfold } (\mathsf{lift\_vars}\ k\ p1)
\end{aligned}$$

Figure 2: definition of $\mathsf{lift\_vars}$

Only the first two cases of the definition are non trivial:

- In case our program is simply a variable we use the function's additional argument k to check whether the variable is bound or not. If $i < k$ it is bound and we do not change it. Otherwise, we increase it by one.

- In case of an abstraction, we go down recursively into the body of the abstraction. Since we do not want to change variables relating to the lambda of the abstraction, we increase our argument k by one.

In all the other cases, we either simply push the recursion down the program term or we are in a base case and do nothing.

Finally, we can define the lift operator by:
$\uparrow_m P :=$ lift_vars m P

Since in most cases we have that $m = 0$ we define
$\uparrow P := \uparrow P0$

The Substitution Operator {} can be used to model a β-reduction. If applied to a program P with an argument program c, it has the following tasks:

- Replace in P all lowest free variables by c.[1]
- While substituting c in P lift c as necessary to ensure that all variables in c still refer to their old lambda-binders after the substitution.
- Shift every occurrence of a de Bruijn index that is free in $\lambda.P$ down by one.
- Keep every occurrence of a de Bruijn index that is bound in P as it is.

If we now have a term of the form $(\lambda.P)c$, this term would β-reduce to $P\{c\}$.

[1]Note that this means, that the operator substitutes every occurrence of the lowest possible variable that is **not** bound in P. This are all variables of the form (var i) satisfying that the variable itself is free in P, but if it were the variable (var i-1) it would be bound in P. This possibly unexpected specification is necessary since we want to use the substitution operator to model β-reductions for programs of the form $\lambda.P$.

Example:
Let "+" be the addition function that adds two values of a pair and P be the program

$$P{:=}\ \lambda.\text{fun}\Big(+, \big(\text{var } 1\ ,\ \text{fun}(+, (\text{var } 0\ ,\ \text{var } 3))\big)\Big)$$

Now we consider the substitution $P\{\text{value } 5\}$:

$$P\{\text{value } 5\} = \lambda.\text{fun}\Big(+, \big(\text{value } 5\ ,\ \text{fun}(+, (\text{var } 0\ ,\ \text{var } 2))\big)\Big)$$

Explanation: In our example, only var 1 would refer to the highest lambda in the program $\lambda.P$. Therefore, we replace it with the substitution operator's argument value 5. The variable var 0 refers to a lambda in P and is therefore bound, so it is not changed. The variable var 3 is free in the program $\lambda.P$. The substitution operator decreases it by one. Note, that the program $(\lambda.P)$(value 5) would reduce to $P\{\text{value } 5\}$:

$$\begin{array}{ll} & (\lambda.P)(\text{value } 5) \\ = & \Big(\lambda.\lambda.\text{fun}\Big(+, \big(\text{var } 1\ ,\ \text{fun}(+, (\text{var } 0\ ,\ \text{var } 3))\big)\Big)\Big)(\text{value } 5) \\ \rightsquigarrow_{sem} & \lambda.\text{fun}\Big(+, \big(\text{value } 5\ ,\ \text{fun}(+, (\text{var } 0\ ,\ \text{var } 2))\big)\Big) \\ = & P\{\text{value } 5\} \end{array}$$

Note that $\rightsquigarrow_{sem}$ is not the real semantic reduction relation of the language. But in this example it gives an intuition how the semantic reduction relation works.

To define the operator formally, we first have to define a function substitute' (see Figure 3) that is able to substitute not only the 0^{th} de Bruijn index, but any index of a program.
The substitute' function gets three arguments. The first one is the program in which we want to substitute a variable. The second one is the program we want to substitute with. The last argument tells us to which lambda we refer as the highest lambda and therefore which variable we want to substitute. We need it, because we have to keep track of the lambda we referred to when calling the substitution-operator. Since de Bruijn indices are relative and not absolute, this can only be done by using this additional function argument that "remembers" the initial lambda.

Again, only the first two cases of the definition are non trivial:
Suppose, we initially called substitute' on a program P

- If we want to substitute in a variable (var i) and $k < i$, then we know, that the variable is free in the program $\lambda.P$. Therefore we shift it down by one, such that it still refers to its old lambda when we remove the highest lambda during the β-reduction.
  If $k = i$, then this is the variable we want to replace, so we simply give back the functions second argument.
  In the last case, when $k > i$, the variable refers to a lambda in P. Therefore, the variable is bound and we do not change it.

- In case of a lambda-abstraction, we push the function recursively into the body of the abstraction. Now, the distance to the highest lambda increased by one. Therefore, we have to lift all free de Bruijn indices in the functions second argument by one such that they still refer to

$$\mathsf{substitute'} : \text{program} \to \text{program} \to \mathbb{N} \to \text{program}$$

$$\begin{aligned}
\mathsf{substitute'}\ (\text{var } i)\ p\ k &= (\text{if } k < i \text{ then } (\text{var } i-1) \text{ else if } i = k \text{ then } p \text{ else } (\text{var } i))\\
\mathsf{substitute'}\ (\lambda.p1)\ p\ k &= \lambda.(\mathsf{substitute'}\ p1\ (\uparrow p)\ (k+1))\\
\mathsf{substitute'}\ (ST)\ p\ k &= (\mathsf{substitute'}\ S\ p\ k)\ (\mathsf{substitute'}\ T\ p\ k)\\
\mathsf{substitute'}\ (\text{value } v)\ p\ k &= \text{value } v\\
\mathsf{substitute'}\ (\text{fun}(f,p1))\ p\ k &= \text{fun}(f,(\mathsf{substitute'}\ p1\ p\ k))\\
\mathsf{substitute'}\ ((p1,p2))\ p\ k &= ((\mathsf{substitute'}\ p1\ p\ k),(\mathsf{substitute'}\ p2\ p\ k))\\
\mathsf{substitute'}\ (\text{loc } n)\ p\ k &= \text{loc } n\\
\mathsf{substitute'}\ (\text{ref } p1)\ p\ k &= \text{ref } (\mathsf{substitute'}\ p1\ p\ k)\\
\mathsf{substitute'}\ (!p1)\ p\ k &= !(\mathsf{substitute'}\ p1\ p\ k)\\
\mathsf{substitute'}\ (p1 := p2)\ p\ k &= (\mathsf{substitute'}\ p1\ p\ k) := (\mathsf{substitute'}\ p2\ p\ k)\\
\mathsf{substitute'}\ (\text{event } e)\ p\ k &= \text{event } e\\
\mathsf{substitute'}\ \text{eventlist}\ p\ k &= \text{eventlist}\\
\mathsf{substitute'}\ (\text{fst } p1)\ p\ k &= \text{fst } (\mathsf{substitute'}\ p1\ p\ k)\\
\mathsf{substitute'}\ (\text{snd } p1)\ p\ k &= \text{snd } (\mathsf{substitute'}\ p1\ p\ k)\\
\mathsf{substitute'}\ (\text{case } p1\ p2\ p3)\ p\ k &= \text{case } (\mathsf{substitute'}\ p1\ p\ k)\ (\mathsf{substitute'}\ p2\ p\ k)\ (\mathsf{substitute'}\ p3\ p\ k)\\
\mathsf{substitute'}\ (\text{inl } p1)\ p\ k &= \text{inl } (\mathsf{substitute'}\ p1\ p\ k)\\
\mathsf{substitute'}\ (\text{inr } p1)\ p\ k &= \text{inr } (\mathsf{substitute'}\ p1\ p\ k)\\
\mathsf{substitute'}\ (\text{fold } p1)\ p\ k &= \text{fold } (\mathsf{substitute'}\ p1\ p\ k)\\
\mathsf{substitute'}\ (\text{unfold } p1)\ p\ k &= \text{unfold } (\mathsf{substitute'}\ p1\ p\ k)
\end{aligned}$$

Figure 3: definition of substitute'

the same lambdas. To "remember" the position of the highest lambda, we increase the third argument by one.

All other cases are again trivial and either push down the recursion or evaluate a base case.

Finally, we can define the substitution operator by
$P\{c\} :=$ substitute' $P\ c\ 0$

The Instantiation Operator. Sometimes, we need to instantiate free variables in a program P with values. This can be done with the instantiation operation P^a where a is a list of the values we want the free variables to be replaced with. In a, the first element of the list is considered the value all variables referring to the highest lambda in $\lambda.P$ should be replaced with, all variables referring to the hightest lambda in $\lambda.\lambda.P$ should be replaced with the second element of a and so on. Note, that a variable bound to the highest lambda in $\lambda.P$ is a lowest free variable in P.

Example:
Let

P:= ((var 2),λ.((var 0),(var 1)))

Now let us consider the instantiation $P^{[\text{value 1, value 2, value 3}]}$:

$P^{[\text{value 1, value 2, value 3}]}$ = ((value 3),λ.((var 0),(value 1)))

Explanation: First, we replace every occurrence of a variable relating to the highest lambda in

$\lambda.P = \lambda$.((var 2),λ.((var 0),(var 1)))

by (value 1). this is only (var 1). Since there is no variable that would refer to the highest lambda in $\lambda.\lambda.P$, there is no substitution with (value 2). For $\lambda.\lambda.\lambda.P$, (var 2) would relate to the highest lambda, so we replace it by (value 3).

The formal definition of the instantiation operator can be done very simple, using the foldl function (recall the definition of foldl in Figure 4) and the already defined {} function by:
P^a := foldl $(\lambda p.\lambda v.p\{v\})$ P a

Intuitively, this is just sequentially using the substitution operator on P with every single element of the list a.

$$\begin{array}{lcl} \mathsf{foldl} : (\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \mathsf{list} \to \alpha & & \\ & & \\ \mathsf{foldl}\ f\ z\ \text{nil} & = & \text{z} \\ \mathsf{foldl}\ f\ z\ (a :: ar) & = & \mathsf{foldl}\ f\ (f\ z\ a)\ ar \end{array}$$

Figure 4: definition of foldl

2.4 Semantics

In this Section, I will give the definition of the semantic reduction relation $\rightsquigarrow$ as well as some other important constructs of the language. Since we are dealing with a probabilistic language we can not just define a reduction relation mapping programs to programs. Furthermore, the reduction relation has to deal with locations and events. Therefore the language uses specialized structures.

The Store is a list of programs and usually denoted by σ. In the language it is used to allow programs to store data in an extra memory which is not the case in purely functional programming languages. At the beginning of an execution of a program it is usually empty. When a new reference is created during a program run, the referenced program is just inserted at the end of the list.

The Eventlist is a list of strings and usually denoted by η. It contains all previously raised events during a program run. Whenever a new event is raised during an execution of a program it is inserted at the end of the list.

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

$$\begin{array}{rcll l} E[(\lambda.P)V]|\sigma|\eta & \rightsquigarrow & \delta_{E[P\{V\}]|\sigma|\eta} & & \textsc{Appl} \\ E[\mathrm{ref}\ V]|\sigma|\eta & \rightsquigarrow & \delta_{E[\mathrm{loc}\ (|\sigma|)]|\sigma @ [V]|\eta} & & \textsc{Ref} \\ E[!(\mathrm{loc}\ l)]|\sigma|\eta & \rightsquigarrow & \delta_{E[\sigma_l]|\sigma|\eta} & ,\text{if } l < |\sigma| & \textsc{Deref} \\ E[\mathrm{loc}\ l := V]|\sigma|\eta & \rightsquigarrow & \delta_{E[\mathrm{value}\ unit]|\sigma[l:=V]|\eta} & ,\text{if } l < |\sigma| & \textsc{Assign} \\ E[\mathrm{fst}(V,V')]|\sigma|\eta & \rightsquigarrow & \delta_{E[V]|\sigma|\eta} & & \textsc{Fst} \\ E[\mathrm{snd}(V,V')]|\sigma|\eta & \rightsquigarrow & \delta_{E[V']|\sigma|\eta} & & \textsc{Snd} \\ E[\mathrm{fun}(f,V)]|\sigma|\eta & \rightsquigarrow & (\lambda x.E[x]|\sigma|\eta)(fV) & & \textsc{Fun} \\ E[\mathrm{event}\ s]|\sigma|\eta & \rightsquigarrow & \delta_{E[\mathrm{value}\ unit]|\sigma|\eta @ [s]} & & \textsc{Ev} \\ E[\mathrm{eventlist}]|\sigma|\eta & \rightsquigarrow & \delta_{E[\overline{\eta}]|\sigma|\eta} & & \textsc{EvList} \\ E[\mathrm{case}(\mathrm{inl}\ V)V_LV_R]|\sigma|\eta & \rightsquigarrow & \delta_{E[V_LV]|\sigma|\eta} & & \textsc{CaseL} \\ E[\mathrm{case}(\mathrm{inr}\ V)V_LV_R]|\sigma|\eta & \rightsquigarrow & \delta_{E[V_RV]|\sigma|\eta} & & \textsc{CaseR} \\ E[\mathrm{unfold}(\mathrm{fold}\ V)]|\sigma|\eta & \rightsquigarrow & \delta_{E[V]|\sigma|\eta} & & \textsc{Fold} \end{array}$$

Figure 5: reduction rules

A Program State is a triple $(P|\sigma|\eta)$ of a program P, a store σ and an eventlist η. Intuitively, P is the remaining program of a program execution, σ contains all previously introduced references and η contains all previously raised events.

The Semantic Reduction Relation $\rightsquigarrow$ can now be defined as a relation between program states and subprobability measures over program states. Intuitively $\rightsquigarrow$ relates a program state S with a measure that tells us for every set of program states X the probability that S reduces semantically to one of the program states in X. Often these steps are deterministic and the measure a program state $(P|\sigma|\eta)$ is related to is just a Dirac measure $\delta_{P'|\sigma'|\eta'}$ (for a Dirac measure it holds that $\delta_x(D) = 1 \Leftrightarrow x \in D$ and $\delta_x(D) = 0 \Leftrightarrow x \notin D$). Figure 5 gives the full reduction relation of the language. The definition uses *evaluation contexts*. An evaluation context E is a context in which the hole is always located at the position where the program should be evaluated first according to the call-by-value strategy. Figure 5 also shows a grammar that defines all evaluation contexts. Evaluation contexts constitute an elegant way to define the structural reduction rules since one can be sure that the program will be executed at exactly this point when making the next reduction step and one can omit a lot of less interesting structural reduction rules.
Whenever $\rightsquigarrow$ can not be applied to a program state, we say that the program execution got *stuck*.

Example:
For a better understanding let us have a look at the following program:

$P := (\lambda.\text{var } 0)(\text{fun}(f, \mathit{unit}))$

where f is a submarkovkernel partially defined by:
$f(\mathit{unit})(X) := \frac{|X \cap \{\text{value } 0, \text{value } 1\}|}{2}$

On input *unit* the function f yields a probability measure on pure values that simulates a coin toss that returns value 0 and value 1 both with probability $\frac{1}{2}$. The program P reduces to an application of the identity function $\lambda.\text{var } 0$ to the outcome of this coin toss. Note, that $\lambda.\text{var } 0$ is a value and therefore the context $E := (\lambda.\text{var } 0)\Box$ is an evaluation context. We can now use our semantic reduction relation $\rightsquigarrow$ to execute the program (with an empty store [] and an empty eventlist []).

$$\begin{array}{lll} & (E[(\text{fun}(f, \mathit{unit}))] \mid [] \mid []) & \\ \rightsquigarrow & (\lambda x.E[x] \mid [] \mid [])(f(\mathit{unit})) & \text{by } \textsc{Fun} \\ = & \lambda S.\frac{|S \cap \{(E[\text{value } 0] \mid [] \mid []), (E[\text{value } 1] \mid [] \mid [])\}|}{2} & \end{array}$$

The last line is the resulting measure where S ranges over sets of program states.
Now we can continue with the execution of each branch. Note that $\Box$ is an evaluation context as well.

$$\begin{array}{lll} & (E[\text{value } 0] \mid [] \mid []) & \\ = & (\lambda.\text{var } 0)(\text{value } 0) & \text{by definition of } E \\ \rightsquigarrow & \delta_{((\text{var } 0)\{\text{value } 0\} \mid [] \mid [])} & \text{by } \textsc{Appl} \text{ using the evaluation context } \Box \\ = & \delta_{(\text{value } 0 \mid [] \mid [])} & \text{by the definition of } \{\} \end{array}$$

The execution of the other program with value 1 works in exactly the same way.

The following graphic can give an intuition of the execution.

$$\begin{tikzcd} & \boxed{(\lambda.\text{var } 0)(\text{fun}(f, \mathit{unit}))} \arrow[dl, "\frac{1}{2}"'] \arrow[dr, "\frac{1}{2}"] & \\ \boxed{(\lambda.\text{var } 0)(\text{value } 0)} \arrow[d, "1"] & & \boxed{(\lambda.\text{var } 0)(\text{value } 1)} \arrow[d, "1"] \\ \boxed{\text{value } 0} & & \boxed{\text{value } 1} \end{tikzcd}$$

The arrows are labeled with the probability that the program on the beginning of the arrow reduces to the program the arrow points to. Note that one is not always able to draw such treelike execution diagrams. In case the fun construct yields not a finite but a continuous measure there is no finite diagram that represents the execution.

The Denotation of a program state $(P|\sigma|\eta)$ is denoted by $[\![P|\sigma|\eta]\!]$. It is a measure on program states. The denotation $[\![P|\sigma|\eta]\!]$ takes a set of program states Ω as an input and returns the probability that the program state $(P|\sigma|\eta)$ evaluates to a program state $(v|\sigma'|\eta') \in \Omega$ where v is a value. To define the denotation formally we use a *step-function.* Note that it is now important to distinguish between program states and measures on program states since our semantic reduction relation $\rightsquigarrow$ relates program states with measures on program states. Therefore we have to define what it means to make a semantic reduction step on a measure on program states. For a program state $(P|\sigma|\eta)$ we define $\mathit{step}(P|\sigma|\eta) := \mu$ if $(P|\sigma|\eta) \rightsquigarrow \mu$ and $\mathit{step}(P|\sigma|\eta) := \delta_{P|\sigma|\eta}$ otherwise. Further we define $\mathit{step}_0(P|\sigma|\eta) := \delta_{P|\sigma|\eta}$ and $\mathit{step}_{n+1}(P|\sigma|\eta) := \mathit{step} \circ \mathit{step}_n(P|\sigma|\eta)$. Now we can define the distribution of the program state $(P|\sigma|\eta)$ after n steps as $\mu^n := \mathit{step}_n(P|\sigma|\eta)$. Let $\mathcal{V}$ be the set of all program states $(V|\sigma|\eta)$ where V is a value. The distribution of the program state $(P|\sigma|\eta)$ after n steps restricted to values is then defined as $\overline{\mu}^n := \mu^n|_{\mathcal{V}}$. Finally we can define the denotation by $[\![P|\sigma|\eta]\!] := \sup_n \overline{\mu}^n$ which is the distribution of the program state $(P|\sigma|\eta)$ after the execution of the program P restricted to values.

Example:
Again consider the program

$P := (\lambda.\text{var } 0)(\text{fun}(f, \mathit{unit}))$

with f being a submarkovkernel partially defined by:

$f(\mathit{unit})(X) := \frac{|X \cap \{\text{value } 0, \text{value } 1\}|}{2}$

Recall that this was just a program that simulated a coin toss. The denotation of this program now takes a set of program states and returns the probability that P evaluates to one of the program states in the set restricted to states $(v|\sigma|\eta)$ where v is a value.

$[\![P|[]|[]]\!](\{(\text{value } 0|[]|[])\}) = \frac{1}{2}$
Explanation: We designed the program in a way such that the outcome of P is (value 0) with probability exactly a half.

$[\![P|[]|[]]\!](\{(\text{value } 0|[]|[]), (\text{value } 1|[]|[])\}) = 1$
Explanation: Since (value 0) and (value 1) are the only two possible output values of P and P can not diverge the probability that P evaluates to either (value 0) or (value 1) is one.

$[\![P|[]|[]]\!](\{(\text{value } 5|[]|[])\}) = 0$
Explanation: P can not evaluate to (value 5).

$[\![P|[]|[]]\!](\{(\lambda x.(\lambda.\text{var } 0)x|[]|[])(f(\mathit{unit}))\}) = 0$
Explanation: Since the denotation of a program state is restricted to values this probability is zero.

The Probability of Termination $T(P|\sigma|\eta)$ of a program state $(P|\sigma|\eta)$ is the probability that the program state $(P|\sigma|\eta)$ evaluates to any program state $(v|\sigma'|\eta')$ where v is a value. Intuitively this is just the probability that the program P evaluates to any value and therefore terminates. We can use the denotation to define the probability of termination formally. Let $\mathcal{PS}$ denote the set of all program states. Then we can define for a program state $(P|\sigma|\eta)$

$T(P|\sigma|\eta) := [\![P|\sigma|\eta]\!](\mathcal{PS})$

The denotation $[\![P|\sigma|\eta]\!]$ applied to $\mathcal{PS}$ gives us the probability that $(P|\sigma|\eta)$ evaluates to any program state $(v|\sigma'|\eta') \in \mathcal{PS}$ where v is a value since the denotation is restricted to values. Therefore $[\![P|\sigma|\eta]\!](PS)$ yields the probability of termination.

2.5 Program Relations

Game transformations base on program relations. If two programs P and Q can be regarded "equivalent" and therefore are in a certain relation, one can transform program P into program Q and vice versa. In this Section I will define several program relations that allow program transformations or may help arguing about program equalities.

The Denotational Equivalence states that for two programs P and Q the denotations $[\![P|[]|[]]\!]$ and $[\![Q|[]|[]]\!]$ are exactly the same what means that they evaluate to the same distribution over program states. Nevertheless in many applications the denotational equivalence is too strong since even two programs that are not distinguishable for an observer can be not denotational equivalent. For example consider two programs that compute the same function but one uses the store and the other does not. Therefore the programs are not denotational equivalent even though they evaluate to the same distribution over values. But since they differ in their stores, they are not denotationally equivalent.

The Observational Equivalence $\equiv_{obs}$ is a slightly weaker equivalence relation amongst programs. The idea behind it is that two programs P and Q are equivalent if their behavior is the same in any context C that binds all free variables of P and Q and that contains no locations outside the store. To formalize this, we call a program state $(P|\sigma|\eta)$ *fully closed* if P and σ contain no free variables and P and σ contain no locations greater than $|\sigma|$. Now we define the observational equivalence of two programs P and Q as $P \equiv_{obs} Q :\Leftrightarrow \forall C$ s.t. $(C[P]|[]|[])$ and $(C[Q]|[]|[])$ are fully closed it holds that $T(C[P]|[]|[]) = T(C[Q]|[]|[])$. In case it even holds that $\forall \sigma, \eta, C$ s.t. $(C[P]|\sigma|\eta)$ and $(C[Q]|\sigma|\eta)$ are fully closed it holds that $T(C[P]|\sigma|\eta) = T(C[Q]|\sigma|\eta)$ we call P and Q *observationally equivalent for arbitrary state* and write $P \equiv_{obs}^{\forall\sigma,\eta} Q$. Note that of course observational equivalence for arbitrary state implies observational equivalence.
Intuitively two programs P and Q being observationally equivalent means that there exists no context that is able to distinguish between P and Q. If any difference between P and Q is observable we can find a context C such that $C[P]$ terminates and $C[Q]$ diverges. Therefore we have modeled a different termination behavior artificially. Only if there exists no context that is able to model such a behavior the two programs can be regarded equivalent.
Two programs that are observationally equivalent can be transformed into each other. Most transformations I define in this thesis are based on the observational equivalence of programs.

The CIU Theorem and CIU Equivalence $\equiv_{CIU}$ are very helpful in case one wants to show the observational equivalence of two programs. It is rather hard to work with the definition of the observational equivalence like stated above since it deals with general contexts (and not evaluation contexts). Therefore it is hard to argue about the semantic behavior of a program of the form $C[P]$ where C is a context and P is a program.

The CIU equivalence describes equivalent behavior of programs in a different way. We call two programs P and Q *CIU equivalent* if for all evaluation contexts E, all lists of values a, all stores σ and all eventlists η it holds that if $(E[P^a]|\sigma|\eta)$ and $(E[Q^a]|\sigma|\eta)$ are fully closed then $T(E[P^a]|\sigma|\eta) = T(E[Q^a]|\sigma|\eta)$. Intuitively instead of looking at all possible contexts we restrict ourselves only to evaluation context but in addition we also take into account everything that could help to distinguish P and Q and what a general context could have had modeled like instantiating free variables or allocating a store σ.
The CIU Theorem states that CIU equivalence implies observational equivalence for arbitrary state. Since CIU equivalence is much easier to handle than observational equivalence I will most often show CIU equivalence of programs and then conclude their observational equivalence.

The Chaining Denotation is a very useful tool when arguing about denotational equivalences. For an evaluation context E and a program P the semantics of the language defined in Figure 5 will evaluate the program $E[P]$ by first evaluating P to a value v and afterwards evaluate $E[v]$. To reflect this we are able to investigate the denotation of P first and chain it with the remaining denotation. Therefore it holds that

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

where we define the $\cdot$ operation for a kernel g and a measure μ by

$$(g \cdot \mu)(A) := \int g(w)(A) d\mu(w)$$

For more information on the notation and measure theory see [3].

3 The Problem

One of the advantages of the framework presented in Chapter 2 is that once proven Lemmas and Theorems can be used over and over again . For reasoning about security properties, it is useful to have a large amount of Theorems that state relations between programs. These Theorems can be used to transform programs like explained in Chapter 1. The task of this thesis is to take a rather simple cryptographic proof and first split it in as small transformations as possible. Afterwards I prove that an initial program and the program after having applied a transformation are observationally equivalent. The goal is to keep these transformations as generic as possible, such that they can be easily applied to other game-based proofs.

The simple cryptographic proof which I work with is the proof for the claim: "If $f(x)$ is a one-way permutation, than $g(x) := f(f(x))$ is a one-way permutation as well". Intuitively a one-way permutation is a permutation that is easy to compute, but hard to invert. Formally we call a permutation p one-way iff there exists a deterministic polynomial-time algorithm $Perm$ s.t. on input x the algorithm $Perm$ outputs $p(x)$ and additionally it holds that for every probabilistic polynomial-time algorithm A the outcome of the game shown in Figure 6 is negligible in n (see [8]).

$$\boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & \{0,1\}^n \\ & y & \leftarrow & p(x) \\ & x' & \leftarrow & A(1^n, y) \\ \text{in} & & x = x' & \end{array}}$$

Figure 6: for p being a one-way permutation the outcome of this game has to be negligible in n

The statement itself can be proven pretty simple by contradiction: "If we had an adversary A that was able to invert g with non negligible-probability, we could model an adversary B that breaks the one-wayness of f with non-negligible probability as well". For the intuition, we can have a look at the graphical reduction proof in Figure 7. Note that I omitted the security parameter n in this and all the following pictures of games since I do not need it to define the transformations.

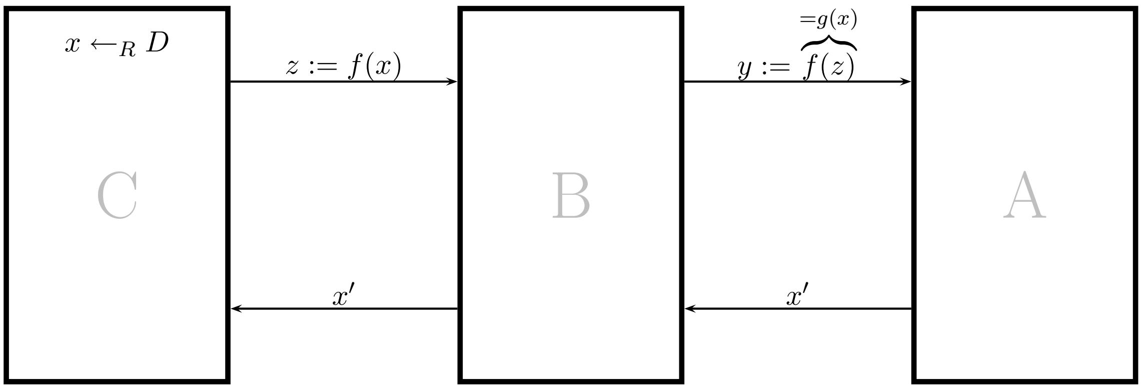


Figure 7: graphical reduction proof

From this, we can get an intuition of the games we will have to model, of the transformations that we will have to apply, and of the definition of the adversary B. When we focus on the right part of the reduction proof, we see the initial game we want to model. This is the game, in which the adversary A has to give the preimage of $y := g(x)$ for a random x in the domain of g. We can model this game in the lambda-calculus like seen in Figure 8.

$$\boxed{\begin{array}{lcccc} \text{let} & x & \leftarrow_R & D \\ & y & \leftarrow & g(x) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array}}$$

Figure 8: initial game

Note, that like explained in Section 2.1, we use the named lambda-calculus with let constructs for the sake of readability here.

If we concentrate on the left part of the graphical proof, we see the final game we want our initial game to transform to. This is the game in which we built a new adversary B from A that is able to break the one-wayness of f, assuming that A can invert g with non negligible probability. In the lambda calculus, this final game looks like seen in Figure 9.

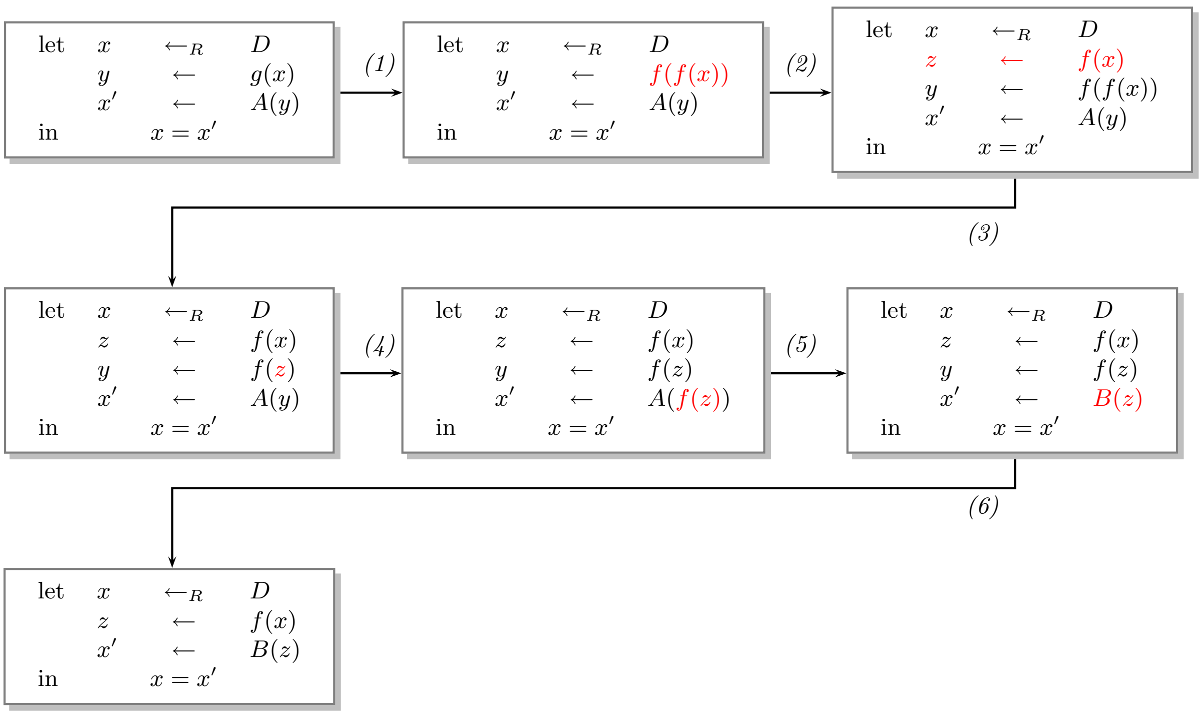
$$\boxed{\begin{array}{lcccc} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & x' & \leftarrow & B(z) \\ \text{in} & & x = x' & \end{array}}$$

Figure 9: final game

In both games as well as in the graphical proof, D is the domain (and consequently also the range) of f. A sequence of transformations that transforms the initial game into the final game can be seen in Figure 10. The transformations applied are explained and the observational equivalence of the games is proven in Section 4.

definitons:
f is a one-way permutation
D is the domain of f
$g(x) := f(f(x))$
$B(x) := A(f(x))$

(1)	*inline propagation*
(2)	*insert independent code*
(3)	*intuitive expression subsumption*
(4)	*intuitive expression propagation*
(5)	*inline subsumption*
(6)	*dead code elimination*



```
let  x   ←_R  D
     y   ←    g(x)
     x'  ←    A(y)
in       x = x'
```

```
let  x   ←_R  D
     y   ←    f(f(x))
     x'  ←    A(y)
in       x = x'
```

```
let  x   ←_R  D
     z   ←    f(x)
     y   ←    f(f(x))
     x'  ←    A(y)
in       x = x'
```

```
let  x   ←_R  D
     z   ←    f(x)
     y   ←    f(z)
     x'  ←    A(y)
in       x = x'
```

```
let  x   ←_R  D
     z   ←    f(x)
     y   ←    f(z)
     x'  ←    A(f(z))
in       x = x'
```

```
let  x   ←_R  D
     z   ←    f(x)
     y   ←    f(z)
     x'  ←    B(z)
in       x = x'
```

```
let  x   ←_R  D
     z   ←    f(x)
     x'  ←    B(z)
in       x = x'
```

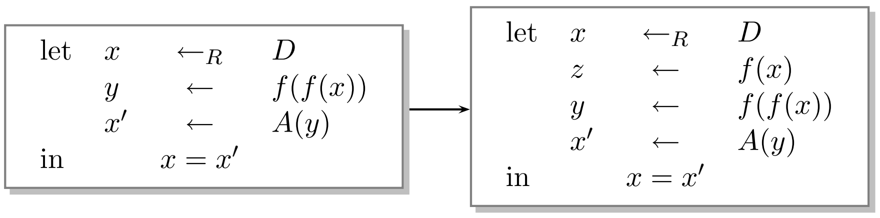
Figure 10: the complete transformation

4 Transformations

In Figure 10 we have seen a sequence of transformations to transform the initial game against the adversary A (Figure 8) into the final game against the adversary B (Figure 9). In this section I will formalize these transformations in the unnamed lambda calculus and prove that for each single transformation a game and its transformed version are at least observationally equivalent. I will always give some intuitions for the formalization first and do the formal proof of observational equivalence of the two formalized programs afterwards. The Theorems that state the equivalences are held as general as possible such that they can be easily reused in other situations. In addition I tried to keep the transformation as intuitive as possible to ensure that even someone who is not familiar with the language can apply these transformations to high level style programs without getting unwanted artifacts. I will give a "transformation info box" at the end of each section that summarizes the transformation described.

4.1 Insert Independent Code

Intuitively, inserting a new variable in a program in which the variable is never used, should not change the behavior of the program at all. Therefore, the transformation seen in Figure 11 should not change the success probability of the adversary.



$$\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & y & \leftarrow & f(f(x)) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array} \longrightarrow \begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & y & \leftarrow & f(f(x)) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array}$$

Figure 11: insert independent code

To model this transformation in the lambda calculus, we let P be an arbitrary program. Further we define Q as the program that we get, when we insert the definition of a fresh variable that is never used in P at the beginning of P. Formally, this can be achieved by defining $Q := (\lambda. \uparrow P)W$. Here W denotes the definition of the new variable. Note that since we lifted all free de Bruijn indices in P there is no variable referring to the highest lambda in $\lambda.(\uparrow P)$. So we modeled a program in which we inserted the definition of a new variable at the beginning of P to which P can not possibly refer.
Note further that now showing the observational equivalence of P and Q suffices to prove that one can insert such a fresh variable in any place in the program without changing its behavior. This is due to the fact that we had no assumptions on P, especially not that it contains no free variables. Therefore, we can have a program $P' = C[P]$ and a program $Q' = C[Q]$ where C is a context such that $(C[P]|[]|[])$ and $(C[Q]|[]|[])$ are fully closed. Intuitively $C[P]$ is the whole program and P is the subprogram in which we want to insert a new variable definition at the beginning of the program. Since P may still refer to lambda binders outside of P we can choose P, and therefore the position where we insert the new variable definition, almost arbitrarily. The definition of the observational equivalence says that if P and Q are observationally equivalent, then their behavior is

equivalent in any closing context. This implies that P' and Q' are observationally equivalent as well.

There are two cases, in which P and Q would not be observationally equivalent:

- It could happen, that W does not terminate with probability 1. In this case, P and Q could have a different termination behavior.
- W could have sideeffects, i.e. it could affect the store or the eventlist.

To avoid these cases, we have to formalize requirements on W. First, we have to ensure, that W always terminates so that inserting the new variable does not change the termination behavior of the resulting program. This is done by requiring that the probability of termination is 1. Formally we require that for all lists of values a, all stores σ, and all eventlists η it holds that $T(W^a|\sigma|\eta) = 1$. For the next requirement, namely that W has no sideeffects, we have to introduce some notation. Let μ be a measure over X and $p : X \to \mathbb{B}$ be a measurable predicate. We write $\forall x \leftarrow \mu.p(x)$ to denote $\exists A \in \Sigma_X.\mu(A) = 0 \wedge \forall x \in (X \backslash A).p(x)$. Intuitively this just means that a value x sampled accordingly to μ always satisfies the predicate p. Now we require for our program W that for all lists of values a, all stores σ, and all eventlists η it holds that $\forall(w, s, e) \leftarrow [\![W^a|\sigma|\eta]\!].\sigma = s \wedge \eta = e$.

For the overall proof, we will further need some small helping lemmas:

Lemma 4.1.1. *For all programs P', P'' and lists of values a, it holds that $(P'P'')^a = P'^aP''^a$.*

Proof. Proven by Matthias Berg in `Isabelle/HOL`. □

Lemma 4.1.2. *For all programs P and lists of values a, it holds that $(\lambda. \uparrow P)^a = \lambda. \uparrow (P^a)$.*

Proof. Proven by Matthias Berg in `Isabelle/HOL`. □

Lemma 4.1.3. *For all programs P, evaluation contexts E, values v, stores σ and event lists η, it holds that $[\![E[(\lambda. \uparrow P)v]|\sigma|\eta]\!] = [\![E[P]|\sigma|\eta]\!]$.*

Proof. From the semantics of the language, we know that for an arbitrary evaluation context E, a program P, a value V, a store σ and an event list η it holds that

$E[(\lambda.P)V]|\sigma|\eta \rightsquigarrow \delta_{E[P\{V\}]|\sigma|\eta}$

Therefore, we have:

$$\begin{aligned} & (E[(\lambda. \uparrow P)v]|\sigma|\eta) \\ \rightsquigarrow\ & \delta_{(E[(\uparrow P)\{v\}]|\sigma|\eta)} \\ =\ & \delta_{(E[P]|\sigma|\eta)} && \text{by definition of } \{\} \text{ and } \uparrow \end{aligned}$$

It holds that for all programs P and P', stores σ and event lists η

$(P|\sigma|\eta) \rightsquigarrow \delta_{(P'|\sigma'|\eta')} \Rightarrow [\![P|\sigma|\eta]\!] = [\![P'|\sigma'|\eta']\!]$

Therefore it also holds that $[\![E[(\lambda. \uparrow P)v]|\sigma|\eta]\!] = [\![E[P]|\sigma|\eta]\!]$. □

In the proof I will also use some results from measure and integration theory (see [3]). Let μ be a measure, N a measurable set, and 1_N the function defined by

$$1_N(x) := \begin{cases} 1 & x \in N \\ 0 & x \notin N \end{cases}$$

Then the following equation holds.

$$\int 1_N(w) d\mu(w) = \mu(N) \tag{1}$$

Theorem 4.1.4 (insert independent code)**.** *Let P, Q, and W be programs s.t.* $Q := (\lambda. \uparrow P)W$*. Assume that for all stores* σ*, all eventlists* η*, and all lists of values* a *it holds that* $T(W^a|\sigma|\eta) = 1$ *and* $\forall (w, s, e) \leftarrow [\![W^a|\sigma|\eta]\!] . \sigma = s \wedge \eta = e$*. Then it holds, that* $P \equiv_{obs} Q$

Proof. We show that $P \equiv_{CIU} Q$. Let E be an evaluation context, a a list of values, σ a store, and η an event list s.t. $(E[P^a]|\sigma|\eta)$ and $(E[Q^a]|\sigma|\eta)$ are fully closed. Note that, because Q^a contains no free variables, W^a contains no free variables as well. Let $\mathcal{PS}$ denote the set of all program states and Ω denote the set of program states $(v|s|e)$ where v is a value.

$$
\begin{aligned}
& T(E[Q^a]|\sigma|\eta) && \\
= \ & \llbracket E[Q^a]|\sigma|\eta \rrbracket(\mathcal{PS}) && \\
= \ & \llbracket E[Q^a]|\sigma|\eta \rrbracket(\Omega) && \text{since denotation is restricted to values} \\
= \ & \llbracket E[((\lambda. \uparrow P)W)^a]|\sigma|\eta \rrbracket(\Omega) && \text{by definition of } Q \\
= \ & \llbracket (E[(\lambda. \uparrow P)^a W^a]|\sigma|\eta \rrbracket(\Omega) && \text{by Lemma 4.1.1} \\
= \ & \llbracket (E[(\lambda. \uparrow (P^a))W^a]|\sigma|\eta \rrbracket(\Omega) && \text{by Lemma 4.1.2} \\
= \ & (\lambda(v', \sigma', \eta').\llbracket E[(\lambda. \uparrow (P^a))v']|\sigma'|\eta' \rrbracket) \cdot \llbracket W^a|\sigma|\eta \rrbracket(\Omega) && \text{by chaining denotation [1]} \\
= \ & \textstyle\int (\lambda(v', \sigma', \eta').\llbracket E[(\lambda. \uparrow (P^a))v']|\sigma'|\eta' \rrbracket(w|s|e)(\Omega) d\llbracket W^a|\sigma|\eta \rrbracket(w|s|e) && \text{by definition of } \cdot \\
= \ & \textstyle\int 1_\Omega(w|s|e)(\lambda(v', \sigma', \eta').\llbracket E[(\lambda. \uparrow (P^a))v']|\sigma'|\eta' \rrbracket(w|s|e)(\Omega) d\llbracket W^a|\sigma|\eta \rrbracket(w|s|e) && \text{since denotation is restricted to values} \\
= \ & \textstyle\int 1_\Omega(w|s|e)\llbracket E[(\lambda. \uparrow (P^a))w]|s|e \rrbracket(\Omega) d\llbracket W^a|\sigma|\eta \rrbracket(w|s|e) && \text{by evaluating the } \lambda\text{-term} \\
= \ & \textstyle\int 1_\Omega(w|s|e)\llbracket E[P^a]|s|e \rrbracket(\Omega) d\llbracket W^a|\sigma|\eta \rrbracket(w|s|e) && \text{by Lemma 4.1.3 [2]} \\
= \ & \textstyle\int 1_\Omega(w|s|e)\llbracket E[P^a]|\sigma|\eta \rrbracket(\Omega) d\llbracket W^a|\sigma|\eta \rrbracket(w|s|e) && \text{by assumption [3]} \\
= \ & \llbracket E[P^a]|\sigma|\eta \rrbracket(\Omega) \quad \textstyle\int 1_\Omega(w|s|e) d\llbracket W^a|\sigma|\eta \rrbracket(w|s|e) && \\
= \ & \llbracket E[P^a]|\sigma|\eta \rrbracket(\Omega) && \text{since the integral evaluates to one [4]} \\
= \ & \llbracket E[P^a]|\sigma|\eta \rrbracket(\mathcal{PS}) && \\
= \ & T(E[P^a])|\sigma|\eta) &&
\end{aligned}
$$

Therefore $Q \equiv_{CIU} P$ which implies $Q \equiv_{obs} P$. □

insert independent code

abstract:	Inserts a fresh variable that is never used in the program at the beginning of the program.
model:	$P \rightarrow (\lambda. \uparrow P)W$
preconditions:	For all stores σ, all eventlists η and all lists of values a it has to hold that $T(W^a|\sigma|\eta) = 1$ and $\forall(w, s, e) \leftarrow \llbracket W^a|\sigma|\eta \rrbracket.\sigma = s \wedge \eta = e$.

Drawbacks

Unfortunately the just defined transformation can not be applied in most real world applications. This is because the precondition that for every list of values a such that the program W^a contains no free variables this program has to terminate with probability one is much too strong. In practice this means that even when W gets inputs not in the domain of the function that W computes it

[1] Here we use the chaining denotation not with the evaluation context E, but with the context $E[(\lambda. \uparrow (P^a))\square]$ which is an evaluation context as well.

[2] By definition of 1_Ω and Ω the program w has to be a value.

[3] By assumption we know that $\forall(w, s, e) \leftarrow \llbracket W^a|\sigma|\eta \rrbracket.\sigma = s \wedge \eta = e$. Therefore we can plug in σ for s and η for e.

[4] By equation 1 the integral evaluates to $\llbracket W^a|\sigma|\eta \rrbracket(\Omega)$. It holds that $\llbracket W^a|\sigma|\eta \rrbracket(\Omega) = \llbracket W^a|\sigma|\eta \rrbracket(\mathcal{PS}) = 1$ by assumption.

has to terminate. Even though it is possible to satisfy this condition in a real application it might be very hard to prove it. Matthias Berg and Dominique Unruh from the Information Security and Cryptography Group at Saarland University are currently working on a technique, namely on a new version of the CIU Theorem that allows to argue about the observational equivalence of programs under the condition that one only uses lists of values a as a variable instantiation that satisfy a certain invariant. This invariant may for example be that one uses only values in the domain of the function a program computes.
To apply this technique to my example transformation from Figure 10 we would have to proceed in two steps. First we had to show that the CIU equivalence of the two programs P and Q defined above still holds when we restrict the lists of values a to just lists of values that satisfy the invariant. Since in this particular example the program W computes $f(x)$ and the domain of f is D the invariant would be that we instantiate only with lists of values from D. Proving this is easy since in the proof of Theorem 4.1.4 we do not touch a and have no requirements on it.
The next step is much more challenging and out of the scope of my Bachelor Thesis. It requires a huge expansion of the framework I described in Chapter 2. In this step we have to prove that whenever the program W is called, it can be called just with values that satisfy the invariant. What seems absolutely trivial in my example (first drawing x randomly out of D and than computing $f(x)$ intuitively ensures that f is only called with values in D) is an extremely difficult task to formalize. The core idea is to invent a new type of context hole ϵ^a that is able to keep track of all substitutions a that are applied to it. If we are able to prove that at the point the semantics evaluate ϵ^a then a contains only values that satisfy the invariant we can apply the transformation.

4.2 Dead Code Elimination

Instead of inserting a fresh variable in the program that is never used, one should also be able to remove the definition of a variable that is never used in a program without changing the success probability of the adversary. This would allow the transformation shown in Figure 12.

$$\boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & y & \leftarrow & f(z) \\ & x' & \leftarrow & B(z) \\ \text{in} & & x = x' & \end{array}} \longrightarrow \boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & x' & \leftarrow & B(z) \\ \text{in} & & x = x' & \end{array}}$$

Figure 12: dead code elimination

We can model this transformation by defining a program $Q := (\lambda. \uparrow P)W$ where W is the definition of the variable we want to remove and $\uparrow P$ is the remaining program. Since we lifted all free variables in P, the variable to which W is bound does not occur in $\uparrow P$. The result of the transformation is P.
Note, that these definitions are exactly the same as the definitions for the *insert independent code* transformation. We want to show that $Q \equiv_{obs} P$. But since we have already proven that $P \equiv_{obs} Q$ in Theorem 4.1.4 and since the $\equiv_{obs}$ relation is symmetric, the proof is already done. Therefore all other properties that we have shown for the *insert independent code* transformation also hold for

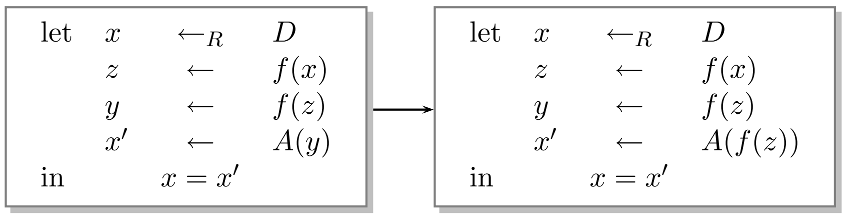
the *dead code elimination* transformation like for example that we are not only able to remove a line of independent code at the beginning of a program, but in any place.

dead code elimination

abstract:	Removes the definition of a variable that is never used from the program.
model:	$(\lambda. \uparrow P)W \to P$
preconditions:	For all stores σ, all eventlists η and all lists of values a it has to hold that $T(W^a\|\sigma\|\eta) = 1$ and $\forall (w, s, e) \leftarrow [\![W^a\|\sigma\|\eta]\!].\sigma = s \wedge \eta = e$.

4.3 Expression Propagation

To transform a game by replacing every occurrence of a variable by the definition of the variable should not change the behavior of the program. Consequently, the transformation presented in Figure 13 should not change the success probability of the adversary.



$$\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & y & \leftarrow & f(z) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array} \longrightarrow \begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & y & \leftarrow & f(z) \\ & x' & \leftarrow & A(f(z)) \\ \text{in} & & x = x' & \end{array}$$

Figure 13: intuitive expression propagation

To prove this, we have to show, that for an arbitrary program Q and a definition of a variable W it holds, that $(\lambda.Q)W \equiv_{obs} Q\{W\}$. Note, that the equivalence is trivial in the case that W is a value. Then, $(\lambda.Q)W$ would just reduce deterministically to $Q\{W\}$ by the semantics of the language (rule Appl in Figure 5).
The transformation we modeled generalizes this. It tells us that we can do such a β-reduction style step even if W is not a value if we meet certain preconditions. Again, we consider the case where the variable that we want to substitute into the program is defined at the beginning of the program. But since we had no assumptions on Q and observational equivalence is defined on equivalent behavior in every context, one could apply the transformation where the definition does not take place at the beginning of the program, too.

Note, that the equivalence we modeled formally is not exactly the equivalence we wanted to have for the games. What the model actually gives us can be seen in Figure 14. There the definition of the variable we want to propagate disappeared. This is due to the substitution operator we used to model the transformation. It works, but in my opinion it makes the transformation rather unintuitive to handle. In section 4.5 we will later model a transformation called *intuitive expression propagation* that looks like what we initially wanted.

Again, we have some cases in which the equivalence does not hold:

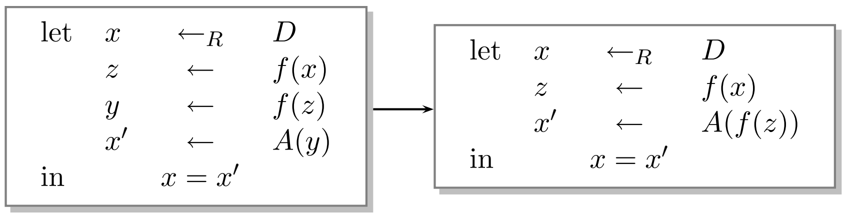


Figure 14: expression propagation

- W could be probabilistic and therefore yield different values at each occurrence, if it were replaced several times in the program.
- W could affect or be affected by the store or the eventlist.
- W could diverge and there is no occurrence of the variable defined by W in Q. In this case $(\lambda.Q)W$ and $Q\{W\}$ would have a different termination behavior.

We formulate requirements for W to avoid these cases. To ensure, that W is not probabilistic, we need W for every input to either yield a certain value deterministically or not to terminate at all (by allowing W not to terminate at all we avoid getting the same application problems as for the *insert independent code* transformation I described in the *Drawbacks* paragraph in Section 4.1). Furthermore we do not want W to interfere with the store or the eventlist in any way. Formally we combine these two requirements by requiring that for all lists of values a s.t. W^a contains no free variables there either exists a value w s.t. for all stores σ, and all eventlists η for which it holds that $(W^a|\sigma|\eta)$ and $(w|\sigma|\eta)$ are fully closed it holds that $[\![W^a|\sigma|\eta]\!] = [\![w|\sigma|\eta]\!]$, or for all stores σ, and all eventlists η for which it holds that $(W^a|\sigma|\eta)$ is fully closed it holds that $[\![W^a|\sigma|\eta]\!] = 0$.
Note that $[\![W^a|\sigma|\eta]\!] = [\![w|\sigma|\eta]\!] \Rightarrow \forall(w,s,e) \leftarrow [\![W^a|\sigma|\eta]\!].\sigma = s \wedge \eta = e$ which was our requirement on W in Theorem 4.1.4.
Finally we require, that if W does not terminate, $Q\{W\}$ must not terminate as well. The formalization of this is: for all lists of values a, all stores σ and all event lists η it holds that $[\![W^a|\sigma|\eta]\!] = 0 \Rightarrow [\![(Q\{W\})^a|\sigma|\eta]\!] = 0$.

Again, we will need some small helping lemmas:

Lemma 4.3.1. *For all programs P, values v, stores σ, event lists η and lists of values a it holds that $[\![((\lambda.P)v)^a|\sigma|\eta]\!] = [\![(P\{v\})^a|\sigma|\eta]\!]$.*

Proof. Proven by Matthias Berg in `Isabelle/HOL` . □

Lemma 4.3.2. *For all lists of values a there exists a context C_a s.t. for all programs P, stores σ and event lists η it holds that $[\![P^a|\sigma|\eta]\!] = [\![C_a[P]|\sigma|\eta]\!]$.*

Proof. Let the context C_a be inductively defined by

$$\begin{array}{lcl} C_{nil} & := & \Box \\ C_{a::ar} & := & C_{ar}[(\lambda.\Box)a)] \end{array}$$

where a is a value, ar is a list of values and :: denotes the cons operation.

We prove the statement by induction over the list of values. For all programs P, stores σ and event lists η we have that

BC: $[\![P^{nil}|\sigma|\eta]\!]$

$=$	$[\![P|\sigma|\eta]\!]$	by definition of the instantiation operator
$=$	$[\![C_{nil}[P]|\sigma|\eta]\!]$	by definition of C_{nil}

IH: $[\![P^{ar}|\sigma|\eta]\!] = [\![C_{ar}[P]|\sigma|\eta]\!]$ for arbitrary P

IS: $[\![C_{a::ar}[P]|\sigma|\eta]\!]$

$=$	$[\![C_{ar}[(\lambda.P)a)]|\sigma|\eta]\!]$	by definition of $C_{a::ar}$
$=$	$[\![((\lambda.P)a)^{ar}|\sigma|\eta]\!]$	by IH
$=$	$[\![(P\{a\})^{ar}|\sigma|\eta]\!]$	by Lemma 4.3.1
$=$	$[\![\mathsf{foldl}\ (\lambda p.\lambda v.p\{v\})\ (P\{a\})\ ar|\sigma|\eta]\!]$	by definition of the instantiation operator
$=$	$[\![\mathsf{foldl}\ (\lambda p.\lambda v.p\{v\})\ P\ (a :: ar)|\sigma|\eta]\!]$	by definition of foldl
$=$	$[\![P^{a::ar}|\sigma|\eta]\!]$	by definition of the instantiation operator

□

Lemma 4.3.3. *For all programs Q there exists a context C_Q s.t. for all programs P without free variables it holds that $Q\{P\} = C_Q[P]$.*

Proof. We define the context C_Q by using a function similar to the substitution function substitute'. We just modify the function a little bit such that instead of substituting a program for the least free variable, we are able to insert a context. we call this context constructing function csubstitute. Its definition can be seen in Figure 15.

Note, that the only difference to the substitute' function is in the type of the function and in the case the program is a lambda-abstraction. Instead of lifting the second argument of the function like in the substitute' function, the csubstitute function omits this operation (a lift on □ is not defined anyway). But since the program we will insert in the context has no free variables, a lift would not have an effect either. On this kind of programs substitute' and csubstitute indeed behave the same.

We define the context C_Q as:
$C_Q :=$ csubstitute Q □ 0

First we show that for a program P without free variables substitute' $Q\ P\ k =$ (csubstitute Q □ k)[P] by induction over Q. Note that all the equalities are well typed since a context applied to a program yields a program.

$\mathsf{csubstitute} :$ program $\rightarrow$ context $\rightarrow \mathbb{N} \rightarrow$ context		
$\mathsf{csubstitute}$ (var i) c k	$=$	(if $k < i$ then (var $i-1$) else if $i = k$ then c else (var i))
$\mathsf{csubstitute}$ ($\lambda.p1$) c k	$=$	λ.($\mathsf{csubstitute}$ $p1$ c $(k+1)$)
$\mathsf{csubstitute}$ (ST) c k	$=$	($\mathsf{csubstitute}$ S c k) ($\mathsf{csubstitute}$ T c k)
$\mathsf{csubstitute}$ (value v) c k	$=$	value v
$\mathsf{csubstitute}$ (fun(f,$p1$)) c k	$=$	fun(f,($\mathsf{csubstitute}$ $p1$ c k)
$\mathsf{csubstitute}$ (($p1$,$p2$)) c k	$=$	(($\mathsf{csubstitute}$ $p1$ c k),($\mathsf{csubstitute}$ $p2$ c k))
$\mathsf{csubstitute}$ (loc n) c k	$=$	loc n
$\mathsf{csubstitute}$ (ref $p1$) c k	$=$	ref ($\mathsf{csubstitute}$ $p1$ c k)
$\mathsf{csubstitute}$ (!$p1$) c k	$=$	!($\mathsf{csubstitute}$ $p1$ c k)
$\mathsf{csubstitute}$ ($p1 := p2$) c k	$=$	($\mathsf{csubstitute}$ $p1$ c k) $:=$ ($\mathsf{csubstitute}$ $p2$ c k)
$\mathsf{csubstitute}$ (event e) c k	$=$	event e
$\mathsf{csubstitute}$ eventlist c k	$=$	eventlist
$\mathsf{csubstitute}$ (fst $p1$) c k	$=$	fst ($\mathsf{csubstitute}$ $p1$ c k)
$\mathsf{csubstitute}$ (snd $p1$) c k	$=$	snd ($\mathsf{csubstitute}$ $p1$ c k)
$\mathsf{csubstitute}$ (case $p1$ $p2$ $p3$) c k	$=$	case ($\mathsf{csubstitute}$ $p1$ c k) ($\mathsf{csubstitute}$ $p2$ c k) ($\mathsf{csubstitute}$ $p3$ c k)
$\mathsf{csubstitute}$ (inl $p1$) c k	$=$	inl ($\mathsf{csubstitute}$ $p1$ c k)
$\mathsf{csubstitute}$ (inr $p1$) c k	$=$	inr ($\mathsf{csubstitute}$ $p1$ c k)
$\mathsf{csubstitute}$ (fold $p1$) c k	$=$	fold ($\mathsf{csubstitute}$ $p1$ c k)
$\mathsf{csubstitute}$ (unfold $p1$) c k	$=$	unfold ($\mathsf{csubstitute}$ $p1$ c k)

Figure 15: definition of $\mathsf{csubstitute}$

BC: *case* $Q =$ (var i), $i = k$

	$\mathsf{substitute'}$ (var i) P k	
$=$	P	by definition of {}
$=$	($\mathsf{csubstitute}$ (var i) $\square$ k)[P]	by definition of $\mathsf{csubstitute}$

BC: *case* $Q =$ (var i), $i < k$

	$\mathsf{substitute'}$ (var i) P k	
$=$	(var i)	by definition of {}
$=$	($\mathsf{csubstitute}$ (var i) $\square$ k)[P]	by definition of $\mathsf{csubstitute}$

BC: *case* $Q =$ (var i), $i < k$

	$\mathsf{substitute'}$ (var i) P k	
$=$	(var i - 1)	by definition of {}
$=$	($\mathsf{csubstitute}$ (var i) $\square$ k)[P]	by definition of $\mathsf{csubstitute}$

The remaining base cases are obvious since the definitions of substitute' and csubstitute in these cases are the same.

$\boxed{\textbf{IH:}\quad \textsf{substitute'}\ W\ P\ k = (\textsf{csubstitute}\ W\ \square\ k)[P] \quad \text{for arbitrary } k}$

$\boxed{\textbf{IS:}\quad \textit{case}\ Q = (\lambda.W)}$

$$\begin{aligned}
& \textsf{substitute'}\ (\lambda.W)\ P\ k \\
= \ & \lambda.(\textsf{substitute'}\ W\ (\uparrow P)\ (k+1)) && \text{by definition of substitute'} \\
= \ & \lambda.(\textsf{substitute'}\ W\ P\ (k+1)) && \text{by definition of } \uparrow \text{ and since } P \text{ contains no free variables} \\
= \ & \lambda.((\textsf{csubstitute}\ W\ \square\ (k+1))[\mathrm{P}]) && \text{by IH} \\
= \ & (\textsf{csubstitute}\ (\lambda.W)\ \square\ k)[\mathrm{P}] && \text{by definition of csubstitute}
\end{aligned}$$

$\boxed{\textbf{IS:}\quad \textit{case}\ Q = \mathrm{fst}\ W}$

$$\begin{aligned}
& \textsf{substitute'}\ (\mathrm{fst}\ W)\ P\ k \\
= \ & \mathrm{fst}\ (\textsf{substitute'}\ \ W\ P\ k) && \text{by definition of substitute'} \\
= \ & \mathrm{fst}\ ((\textsf{csubstitute}\ \ W\ \square\ k)[\mathrm{P}]) && \text{by IH} \\
= \ & (\textsf{csubstitute}\ (\mathrm{fst}\ W)\ \square\ k)[\mathrm{P}] && \text{by definition of csubstitute}
\end{aligned}$$

The remaining cases work similar to the fst case.

Now we can conclude that:

$$\begin{aligned}
& Q\{P\} \\
= \ & \textsf{substitute'}\ Q\ P\ 0 && \text{by definition by definition of } \{\} \\
= \ & (\textsf{csubstitute}\ Q\ \square\ 0)[P] \\
= \ & C_Q[P] && \text{by definition of } C_Q
\end{aligned}$$

$\square$

Lemma 4.3.4. *Let*
$P^{(a)_n}$*:=* foldl *(*$\lambda p.\lambda v.$ substitute' $p\ v\ n$*)* $P\ a$ *(the instantiation operation not beginning with the lowest free variable, but with the* n^{th} *free variable).*
($\uparrow_m a$*):=* foldl *(*$\lambda xs.\lambda x.xs@[\uparrow_m x]$*)* $[]\ a$ *where* a *is a list of values (the list in which the* $\uparrow_m$ *operator is applied to every single value).*

For an arbitrary program P and a list of values a it holds that
$m \leq n \Rightarrow \uparrow_m (P^{(a)_n}) = (\uparrow_m P)^{(\uparrow_m a)_{n+1}}$.

Proof. Proven by Matthias Berg in `Isabelle/HOL`. □

Lemma 4.3.5. *For a program Q and a list of values a it holds that $(\lambda.Q)^{(a)_n} = \lambda.(Q^{(\uparrow a)_{n+1}})$.*

Proof. Proven by Matthias Berg in `Isabelle/HOL`. □

Lemma 4.3.6. *Let $Q\{P\}_m$ denote* `substitute'` *Q P m.*
For all programs P and Q, all natural numbers l and u and all lists of values a s.t. $P^{(a)_l}$ contains no free variables, it holds that $(Q\{P^{(a)_l}\}_u)^{(a)_l} = (Q\{P\}_u)^{(a)_l}$.

Proof. We prove the statement by induction over Q. Let P be a program and a a list of values.

BC: *case* $Q =$ (var k), $k = u$

	$((\text{var k})\{P^{(a)_l}\}_u)^{(a)_l}$	
$=$	$(P^{(a)_l})^{(a)_l}$	by definition of $\{\}_u$
$=$	$P^{(a)_l}$	since $P^{(a)_l}$ contains no free variables
$=$	$((\text{var k})\{P\}_u)^{(a)_l}$	by definition of $\{\}_u$

BC: *case* $Q =$ (var k), $k > u$

	$((\text{var k})\{P^{(a)_l}\}_u)^{(a)_l}$	
$=$	$(\text{var k-1})^{(a)_l}$	by definition of $\{\}_u$
$=$	$((\text{var k})\{P\}_u)^{(a)_l}$	by definition of $\{\}_u$

BC: *case* $Q =$ (var k), $k < u$

	$((\text{var k})\{P^{(a)_l}\}_u)^{(a)_l}$	
$=$	$(\text{var k})^{(a)_l}$	by definition of $\{\}_u$
$=$	$((\text{var k})\{P\}_u)^{(a)_l}$	by definition of $\{\}_u$

The remaining base cases are trivial since the instantiation operator has no effect on them at all.

IH: $(W\{P^{(a)_l}\}_u)^{(a)_l} = (W\{P\}_u)^{(a)_l}$ for arbitrary P, a, l and u

$$\boxed{\textbf{IS:}\quad \textit{case } Q = (\lambda.W)}$$

$$\begin{aligned}
& ((\lambda.W)\{P^{(a)_l}\}_u)^{(a)_l} & \\
= {} & (\lambda.W\{\uparrow (P^{(a)_l})\}_{u+1})^{(a)_l} & \text{by definition of } \{\}_u \\
= {} & (\lambda.W\{(\uparrow P)^{(\uparrow a)_{l+1}}\}_{u+1})^{(a)_l} & \text{by Lemma 4.3.4 with } m = 0 \text{ and } n = l \\
= {} & \lambda.((W\{(\uparrow P)^{(\uparrow a)_{l+1}}\}_{u+1})^{(\uparrow a)_{l+1}}) & \text{by Lemma 4.3.5} \\
= {} & \lambda.((W\{\uparrow P\}_{u+1})^{(\uparrow a)_{l+1}}) & \text{by IH} \\
= {} & (\lambda.W\{\uparrow P\}_{u+1})^{(a)_l} & \text{by Lemma 4.3.5} \\
= {} & ((\lambda.W)\{P\}_u)^{(a)_l} & \text{by definition of } \{\}_u
\end{aligned}$$

$$\boxed{\textbf{IS:}\quad \textit{case } Q = \text{fst } W}$$

$$\begin{aligned}
& ((\text{fst } W)\{P^{(a)_l}\}_u)^{(a)_l} & \\
= {} & (\text{fst } (W\{P^{(a)_l}\}_u))^{(a)_l} & \text{by definition of } \{\}_u \\
= {} & \text{fst } (W\{P^{(a)_l}\}_u)^{(a)_l} & \\
= {} & \text{fst } (W\{P\}_u)^{(a)_l} & \text{by IH} \\
= {} & (\text{fst } (W\{P\}_u))^{(a)_l} & \\
= {} & ((\text{fst } W)\{P\}_u)^{(a)_l} &
\end{aligned}$$

The remaining cases can be proven similar to the fst case.

□

Lemma 4.3.7. *Let P and Q be programs. If for all states σ, all eventlists η, and all lists of values a s.t. $(P^a|\sigma|\eta)$ and $(Q^a|\sigma|\eta)$ are fully closed it holds that $[\![P^a|\sigma|\eta]\!] = [\![Q^a|\sigma|\eta]\!]$, then it also holds that $P \equiv_{obs}^{\forall\sigma,\eta} Q$.*

Proof. We show that $P \equiv_{CIU} Q$. Let $\mathcal{PS}$ denote the set of all program states. For all evaluation contexts E, all stores σ, all eventlists η and all lists of values a such that $(E[P^a]|\sigma|\eta)$ and $(E[Q^a]|\sigma|\eta)$ are fully closed it holds that

$$\begin{aligned}
& T(E[P^a]|\sigma|\eta) & \\
= {} & [\![E[P^a])]|\sigma|\eta]\!](\mathcal{PS}) & \\
= {} & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![P^a|\sigma|\eta]\!](\mathcal{PS}) & \text{by chaining denotation} \\
= {} & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![Q^a|\sigma|\eta]\!](\mathcal{PS}) & \text{by assumption} \\
= {} & [\![E[Q^a]|\sigma|\eta]\!](\mathcal{PS}) & \text{by chaining denotation} \\
= {} & T(E[Q^a]|\sigma|\eta) &
\end{aligned}$$

Therefore it holds that $P \equiv_{CIU} Q$ which implies $P \equiv_{obs}^{\forall\sigma,\eta} Q$. □

Theorem 4.3.8 (expression propagation)**.** *Let Q and W be programs. Assume that for all lists of values a s.t. W^a contains no free variables there either exists a value w s.t. for all stores σ, and all eventlists η for which it holds that $(W^a|\sigma|\eta)$ and $(w|\sigma|\eta)$ are fully closed it holds that $[\![W^a|\sigma|\eta]\!] = [\![w|\sigma|\eta]\!]$, or for all stores σ, and all eventlists η for which it holds that $(W^a|\sigma|\eta)$ is fully closed it holds that $[\![W^a|\sigma|\eta]\!] = 0$. Further assume that $[\![W^a|\sigma|\eta]\!] = 0 \Rightarrow [\![(Q\{W\})^a|\sigma|\eta]\!] = 0$.*
Then it holds, that $(\lambda.Q)W \equiv_{obs} Q\{W\}$.

Proof. We show, that $(\lambda.Q)W \equiv_{CIU} Q\{W\}$. Let E be an evaluation context, a a list of values, σ a store, and η an event list s.t. $(E[((\lambda.Q)W)^a]|\sigma|\eta)$ and $(E[(Q\{W\})^a]|\sigma|\eta)$ are fully closed. It follows, that W^a contains no free variables and that W contains no locations greater then $|\sigma|$. Therefore $(W^a|\sigma|\eta)$ is fully closed. Since W^a contains no free variables there either is a value w s.t. $[\![W^a|\sigma'|\eta']\!] = [\![w|\sigma'|\eta']\!]$ for all stores σ' and eventlists η' for which $(W^a|\sigma'|\eta')$ and $(w|\sigma'|\eta')$ are fully closed, or $[\![W^a|\sigma'|\eta']\!] = 0$ for all stores σ' and eventlists η' for which $(W^a|\sigma'|\eta')$ is fully closed. We now make a case distinction over the two possible measures that $[\![W^a|\sigma|\eta]\!]$ may describe.

case 1:
In case, that $[\![W^a|\sigma|\eta]\!] = 0$ we have $[\![(Q\{W\})^a|\sigma|\eta]\!] = 0$ by assumption.
Let $\mathcal{PS}$ be the set of all program states. The goal $(\lambda.Q)W \equiv_{CIU} Q\{W\}$ follows from:

$$\begin{aligned}
& T(E[((\lambda.Q)W)^a]|\sigma|\eta) && \\
= \ & [\![E[((\lambda.Q)W)^a]|\sigma|\eta]\!](\mathcal{PS}) && \\
= \ & [\![E[(\lambda.Q)^a W^a]|\sigma|\eta]\!](\mathcal{PS}) && \text{by Lemma 4.1.1} \\
= \ & (\lambda(v',\sigma',\eta').[\![E[(\lambda.Q)^a v']|\sigma'|\eta']\!]) \cdot [\![W^a|\sigma|\eta]\!](\mathcal{PS}) && \text{by chaining denotation} \\
= \ & 0(\mathcal{PS}) && \text{by assumption} \\
= \ & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![(Q\{W\})^a|\sigma|\eta]\!](\mathcal{PS}) && \text{by assumption} \\
= \ & [\![E[(Q\{W\})^a]|\sigma|\eta]\!](\mathcal{PS}) && \text{by chaining denotation} \\
= \ & T(E[(Q\{W\})^a]|\sigma|\eta) &&
\end{aligned}$$

case 2:
In the other case where $[\![W^a|\sigma|\eta]\!] = [\![w|\sigma|\eta]\!]$ first note that Lemma 4.3.7 applies since $[\![W^a|\sigma'|\eta']\!] = [\![w|\sigma'|\eta']\!]$ for all stores σ' and eventlists η'. Therefore we know that $W^a \equiv_{obs}^{\forall\sigma,\eta} w$. Now we show that $(\lambda.Q)W \equiv_{CIU} Q\{W\}$.

$$\begin{array}{lll}
& T(E[((\lambda.Q)W)^a]|\sigma|\eta) & \\
= & [\![E[((\lambda.Q)W)^a]|\sigma|\eta)]\!](\mathcal{PS}) & \\
= & [\![E[(\lambda.Q)^a W^a]|\sigma|\eta)]\!](\mathcal{PS}) & \text{by Lemma 4.1.1} \\
= & (\lambda(v',\sigma',\eta').[\![E[(\lambda.Q)^a v']|\sigma'|\eta')]\!]) \cdot [\![W^a|\sigma|\eta]\!](\mathcal{PS}) & \text{by chaining denotation }^{1} \\
= & (\lambda(v',\sigma',\eta').[\![E[(\lambda.Q)^a v']|\sigma'|\eta')]\!]) \cdot [\![w|\sigma|\eta]\!](\mathcal{PS}) & \text{by assumption} \\
= & [\![E[(\lambda.Q)^a w]|\sigma|\eta]\!](\mathcal{PS}) & \text{by chaining denotation} \\
= & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![(\lambda.Q)^a w|\sigma|\eta]\!](\mathcal{PS}) & \text{by chaining denotation }^{2} \\
= & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![((\lambda.Q)w)^a|\sigma|\eta]\!](\mathcal{PS}) & \text{since } w \text{ contains no free variables} \\
= & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![(Q\{w\})^a|\sigma|\eta]\!](\mathcal{PS}) & \text{by Lemma 4.3.1} \\
= & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![C_a[Q\{w\}]|\sigma|\eta]\!](\mathcal{PS}) & \text{by Lemma 4.3.2 for a suitable } C_a \\
= & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![C_a[C_Q[w]]|\sigma|\eta]\!](\mathcal{PS}) & \text{by Lemma 4.3.3 for a suitable } C_Q \\
= & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![C_a[C_Q[W^a]]|\sigma|\eta]\!](\mathcal{PS}) & \text{since } W^a \equiv_{obs}^{\forall\sigma,\eta} w \\
= & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![C_a[Q\{W^a\}]|\sigma|\eta]\!](\mathcal{PS}) & \text{by Lemma 4.3.3} \\
= & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![(Q\{W^a\})^a]|\sigma|\eta]\!](\mathcal{PS}) & \text{by Lemma 4.3.2} \\
= & (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![(Q\{W\})^a]|\sigma|\eta]\!](\mathcal{PS}) & \text{by Lemma 4.3.6 with } l=0 \text{ and } u=0 \\
= & [\![E[(Q\{W\})^a]|\sigma|\eta]\!](\mathcal{PS}) & \text{by chaining denotation} \\
= & T(E[(Q\{W\})^a]|\sigma|\eta) &
\end{array}$$

Therefore we have proven, that $(\lambda.Q)W \equiv_{CIU} Q\{W\}$ which implies $(\lambda.Q)W \equiv_{obs} Q\{W\}$.

□

expression propagation

abstract:	Replacing every occurrence of a variable at the beginning of a program by the program with which it is defined. After the transformation the definition of the variable is not part of the program anymore.
model:	$(\lambda.Q)W \to Q\{W\}$
preconditions:	For all lists of values a s.t. W^a contains no free variables there either exists a value w s.t. for all stores σ, and all eventlists η for which it holds that $(W^a|\sigma|\eta)$ and $(w|\sigma|\eta)$ are fully closed it holds that $[\![W^a|\sigma|\eta]\!] = [\![w|\sigma|\eta]\!]$, or for all stores σ, and all eventlists η for which it holds that $(W^a|\sigma|\eta)$ is fully closed it holds that $[\![W^a|\sigma|\eta]\!] = 0$. Furthermore it has to hold that $[\![W^a|\sigma|\eta]\!] = 0 \Rightarrow [\![(Q\{W\})^a|\sigma|\eta]\!] = 0$.

[1] Here we use the chaining denotation not with the evaluation context E, but with the context $E[(\lambda Q)^a\square]$ which is an evaluation context as well.

[2] Here we use in fact the evaluation context E.

4.4 Expression Subsumption

The proof of Theorem 4.3.8 again gives us a second transformation for free. Since we showed, that $(\lambda.Q)W \equiv_{obs} Q\{W\}$, it also holds that $Q\{W\} \equiv_{obs} (\lambda.Q)W$ because of the symmetry of the $\equiv_{obs}$ relation.
Like in the *expression propagation* transformation the way of modeling this transformation leads to a perhaps unexpected resulting program when applying this transformation to an initial program. From an *expression subsumption* transformation, one would probably expect a transformation like in Figure 17. But what applying our transformation actually yields can be seen in Figure 16.

$$\boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & y & \leftarrow & f(f(x)) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array}} \longrightarrow \boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & y & \leftarrow & f(z) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array}}$$

Figure 16: expression subsumption

The reason why our transformation looks so unintuitive in the named representation of the language is, that we have to “create” a fresh variable with a certain definition. It even looks a little bit like the *insert independent code* transformation. In section 4.6 we will later model a transformation that looks like the transformation in Figure 17.

$$\boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & y & \leftarrow & f(f(x)) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array}} \longrightarrow \boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & y & \leftarrow & f(z) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array}}$$

Figure 17: intuitive expression subsumption

expression subsumption

abstract:	Subsuming occurrences of a program by a fresh variable which definition is inserted at the beginning of the overall program.
model:	$Q\{W\} \rightarrow (\lambda.Q)W$
preconditions:	For all lists of values a s.t. W^a contains no free variables there either exists a value w s.t. for all stores σ, and all eventlists η for which it holds that $(W^a\vert\sigma\vert\eta)$ and $(w\vert\sigma\vert\eta)$ are fully closed it holds that $[\![W^a\vert\sigma\vert\eta]\!] = [\![w\vert\sigma\vert\eta]\!]$, or for all stores σ, and all eventlists η for which it holds that $(W^a\vert\sigma\vert\eta)$ is fully closed it holds that $[\![W^a\vert\sigma\vert\eta]\!] = 0$. Furthermore it has to hold that $[\![W^a\vert\sigma\vert\eta]\!] = 0 \Rightarrow [\![(Q\{W\})^a\vert\sigma\vert\eta]\!] = 0$.

4.5 Intuitive Expression Propagation

The *expression propagation* transformation discussed so far has some drawbacks in points of intuition. Applying the transformation to a program removes a line of code from it (see Figure 14). Normally this is not what one would expect an *expression propagation* transformation to do. To make things a little bit easier, we define a new transformation *intuitive expression propagation.* This transformation first applies an *expression propagation* to the program and reinserts the lost line of code with an *insert independent code* transformation afterwards. An Example of the functioning of the transformation can be seen in Figure 18.

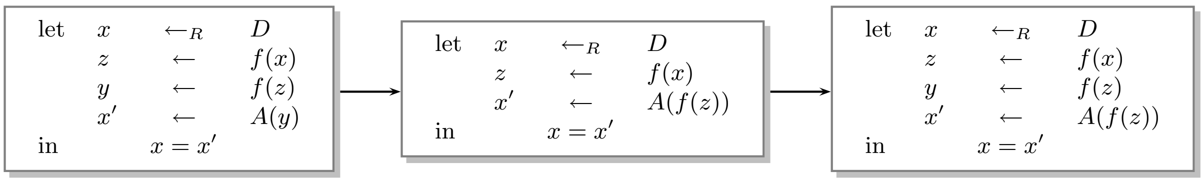


Figure 18: functioning of intuitive expression propagation

Since we discussed the *insert independent code* and the *expression propagation* transformations in detail in the Sections 4.1 and 4.3, we can model this new transformation now straightforward in the lambda calculus as $(\lambda.Q)W \equiv_{obs} (\lambda. \uparrow (Q\{W\}))W$. Here Q and W are programs. W is the definition of a variable at the beginning of $(\lambda.Q)W$, and Q is the remaining program. Again we can exploit the fact that observational equivalence states equivalent behavior in every context to apply this transformation not only to programs, where the variable is defined at the beginning of the program.
Formally, this yields the following Theorem.

Theorem 4.5.1 (intuitive expression propagation). *Let Q and W be programs. Assume that for all lists of values a s.t. W^a contains no free variables there either exists a value w s.t. for all stores σ, and all eventlists η for which it holds that $(W^a|\sigma|\eta)$ and $(w|\sigma|\eta)$ are fully closed it holds that $[\![W^a|\sigma|\eta]\!] = [\![w|\sigma|\eta]\!]$, or for all stores σ, and all eventlists η for which it holds that $(W^a|\sigma|\eta)$ is fully closed it holds that $[\![W^a|\sigma|\eta]\!] = 0$. Furthermore assume, that $[\![W^a|\sigma|\eta]\!] = 0 \Rightarrow [\![(Q\{W\})^a|\sigma|\eta]\!] = 0$. Then it holds, that $(\lambda.Q)W \equiv_{obs} (\lambda. \uparrow (Q\{W\}))W$.*

Proof. By Theorem 4.3.8 we know, that $(\lambda.Q)W \equiv_{obs} Q\{W\}$. Now we want to show that $Q\{W\} \equiv_{CIU} (\lambda. \uparrow (Q\{W\}))W$. Let a be a list of values, E an evaluation context, σ a store, and η an event list s.t. $(E[(Q\{W\})^a]|\sigma|\eta)$ and $(E[((\lambda. \uparrow (Q\{W\}))W)^a]|\sigma|\eta)$ are fully closed. We have that

$$\begin{aligned}
& T(E[((\lambda. \uparrow (Q\{W\}))W)^a]|\sigma|\eta) \\
= \; & [\![E[((\lambda. \uparrow (Q\{W\}))W)^a]|\sigma|\eta]\!](\mathcal{PS}) \\
= \; & [\![E[(\lambda. \uparrow (Q\{W\}))^a W^a]|\sigma|\eta]\!](\mathcal{PS}) && \text{by lemma 4.1.1} \\
= \; & (\lambda(v',\sigma',\eta').[\![E[(\lambda. \uparrow (Q\{W\}))^a v']|\sigma'|\eta']\!]) \cdot [\![W^a|\sigma|\eta]\!](\mathcal{PS}) && \text{by chaining denotation}
\end{aligned}$$

Since we know that W^a contains no free variables, there either exists a value w s.t $[\![W^a|\sigma'|\eta']\!] = [\![w|\sigma'|\eta']\!]$ for all stores σ' and eventlists η' for which $(W^a|\sigma'|\eta')$ and $(w|\sigma'|\eta')$ are fully closed or $[\![W^a|\sigma'|\eta']\!] = 0$ for all stores σ' and eventlists η' for which $(W^a|\sigma'|\eta')$ is are fully closed. Now we can make a case destinction over the two possible measures $[\![W^a|\sigma|\eta]\!]$ may describe depending on a.

case 1:
In case $[\![W^a|\sigma|\eta]\!] = 0$ we have that

$$\begin{aligned}
& (\lambda(v',\sigma',\eta').[\![E[(\lambda.\uparrow(Q\{W\}))^a v']|\sigma'|\eta']\!]) \cdot [\![W^a|\sigma|\eta]\!](\mathcal{PS}) \\
= \;& 0(\mathcal{PS}) \\
= \;& (\lambda(v',\sigma',\eta').[\![E[v']|\sigma'|\eta']\!]) \cdot [\![(Q\{W\})^a|\sigma|\eta]\!](\mathcal{PS}) && \text{by assumption} \\
= \;& [\![E[(Q\{W\})^a]|\sigma|\eta]\!](\mathcal{PS}) && \text{by chaining denotation} \\
= \;& T(E[(Q\{W\})^a]|\sigma|\eta)
\end{aligned}$$

case 2:
In case that $[\![W^a|\sigma|\eta]\!] = [\![w|\sigma|\eta]\!]$ we have that

$$\begin{aligned}
& (\lambda(v',\sigma',\eta').[\![E[(\lambda.\uparrow(Q\{W\}))^a v']|\sigma'|\eta']\!]) \cdot [\![W^a|\sigma|\eta]\!](\mathcal{PS}) \\
= \;& (\lambda(v',\sigma',\eta').[\![E[(\lambda.\uparrow(Q\{W\}))^a v']|\sigma'|\eta']\!]) \cdot [\![w|\sigma|\eta]\!](\mathcal{PS}) \\
= \;& [\![E[(\lambda.\uparrow(Q\{W\}))^a w]|\sigma|\eta]\!](\mathcal{PS}) && \text{by chaining denotation} \\
= \;& [\![E[(\lambda.\uparrow((Q\{W\})^a))w]|\sigma|\eta]\!](\mathcal{PS}) && \text{by Lemma 4.1.2} \\
= \;& [\![E[(Q\{W\})^a]|\sigma|\eta]\!](\mathcal{PS}) && \text{by Lemma 4.1.3} \\
= \;& T(E[(Q\{W\})^a]|\sigma|\eta)
\end{aligned}$$

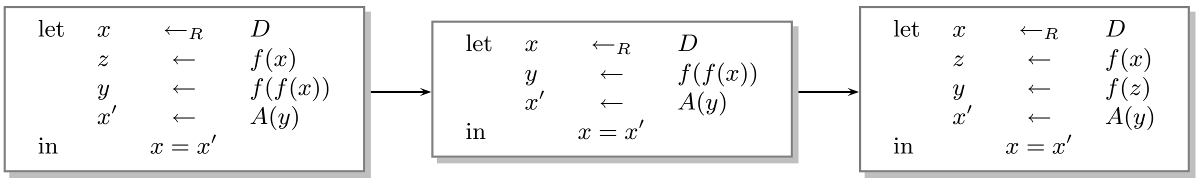
Therefore we have shown that $Q\{W\} \equiv_{CIU} (\lambda.\uparrow(Q\{W\}))W$ what implies $Q\{W\} \equiv_{obs} (\lambda.\uparrow(Q\{W\}))W$. Since the $\equiv_{obs}$ relation is transitive and we know that $(\lambda.Q)W \equiv_{obs} Q\{W\}$ we have shown that $(\lambda.Q)W \equiv_{obs} (\lambda.\uparrow(Q\{W\}))W$. □

intuitive expression propagation

abstract:	Replacing every occurrence of a variable at the beginning of a program by the program with which it is defined. After the transformation the definition of the variable is still part of the program.
model:	$(\lambda.Q)W \rightarrow (\lambda.\uparrow(Q\{W\}))W$
preconditions:	For all lists of values a s.t. W^a contains no free variables there either exists a value w s.t. for all stores σ, and all eventlists η for which it holds that $(W^a|\sigma|\eta)$ and $(w|\sigma|\eta)$ are fully closed it holds that $[\![W^a|\sigma|\eta]\!] = [\![w|\sigma|\eta]\!]$, or for all stores σ, and all eventlists η for which it holds that $(W^a|\sigma|\eta)$ is fully closed it holds that $[\![W^a|\sigma|\eta]\!] = 0$. Furthermore it has to hold that $[\![W^a|\sigma|\eta]\!] = 0 \Rightarrow [\![(Q\{W\})^a|\sigma|\eta]\!] = 0$.

4.6 Intuitive Expression Subsumption

The same problem we had for the *expression propagation* transformation, we also have for the *expression subsumption* transformation. The behaviour of the transformation is even more unexpected since in the transformed program occurs a new variable never used before. This might be acceptable in the nameless lambda calculus, but when it comes to a named representation, and this is normally one would like to work with, this is rather confusing. Therefore, We define a new transformation as a sequence of a *dead code elimination* transformation and an *expression subsumption* transformation. The *dead code elimination* transformation simply first removes the definition of the variable that we would like to subsume. The *expression subsumption* transformation reinserts this definition afterwards again. An example of the functioning can be seen in Figure 19. Since



```
let  x   ←R  D
     z   ←   f(x)
     y   ←   f(f(x))
     x'  ←   A(y)
in       x = x'
```

```
let  x   ←R  D
     y   ←   f(f(x))
     x'  ←   A(y)
in       x = x'
```

```
let  x   ←R  D
     z   ←   f(x)
     y   ←   f(z)
     x'  ←   A(y)
in       x = x'
```

Figure 19: functioning of intuitive expression subsumption

we can model this new transformation as $(\lambda. \uparrow (Q\{W\}))W \equiv_{obs} (\lambda.Q)W$ and the $\equiv_{obs}$ relation is symmetric, we have already proven the correctness of the transformation in Theorem 4.5.1.

intuitive expression subsumption

abstract:	Subsuming all occurrences of a program by a variable that is already defined by this program.
model:	$(\lambda. \uparrow (Q\{W\}))W \rightarrow (\lambda.Q)W$
preconditions:	For all lists of values a s.t. W^a contains no free variables there either exists a value w s.t. for all stores σ, and all eventlists η for which it holds that $(W^a\|\sigma\|\eta)$ and $(w\|\sigma\|\eta)$ are fully closed it holds that $[\![W^a\|\sigma\|\eta]\!] = [\![w\|\sigma\|\eta]\!]$, or for all stores σ, and all eventlists η for which it holds that $(W^a\|\sigma\|\eta)$ is fully closed it holds that $[\![W^a\|\sigma\|\eta]\!] = 0$. Furthermore it has to hold that $[\![W^a\|\sigma\|\eta]\!] = 0 \Rightarrow [\![(Q\{W\})^a\|\sigma\|\eta]\!] = 0$.

4.7 Inline Propagation

This transformation is very simple. One should be able to replace a constant by its definition that was made outside the game. Since in this case, the constant is nothing else than a placeholder for the program defined, we have to meet no preconditions because in an execution one would execute the program anyway. In fact the *inline propagation* transformation is not even a real transformation since the initial and the final program are exactly the same. An example can be seen in Figure 20.

definitons:
f is a one-way permutation
D is the domain of f
$B(x) := A(f(x))$

$$\boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & y & \leftarrow & g(x) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array}} \longrightarrow \boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & y & \leftarrow & f(f(x)) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array}}$$

Figure 20: inline propagation

inline propagation

abstract:	Replacing a placeholder by the program it was defined with outside the game.
model:	$P \to P$ where the placeholder has been replaced with its definition
preconditions:	None.

4.8 Inline Subsumption

In opposite to the *inline propagation* transformation, one should also be able to replace a program by a placeholder program if this program was defined outside the game. Again, this is just a syntactical rewriting of the program and does not change the behavior of the program at all. For an example, see Figure 21.

definitons:
f is a one-way permutation
D is the domain of f
$g(x) := f(f(x))$

$$\boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & y & \leftarrow & f(z) \\ & x' & \leftarrow & A(f(z)) \\ \text{in} & & x = x' & \end{array}} \qquad \boxed{\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & y & \leftarrow & f(z) \\ & x' & \leftarrow & B(z) \\ \text{in} & & x = x' & \end{array}}$$

Figure 21: inline subsumption

inline subsumption

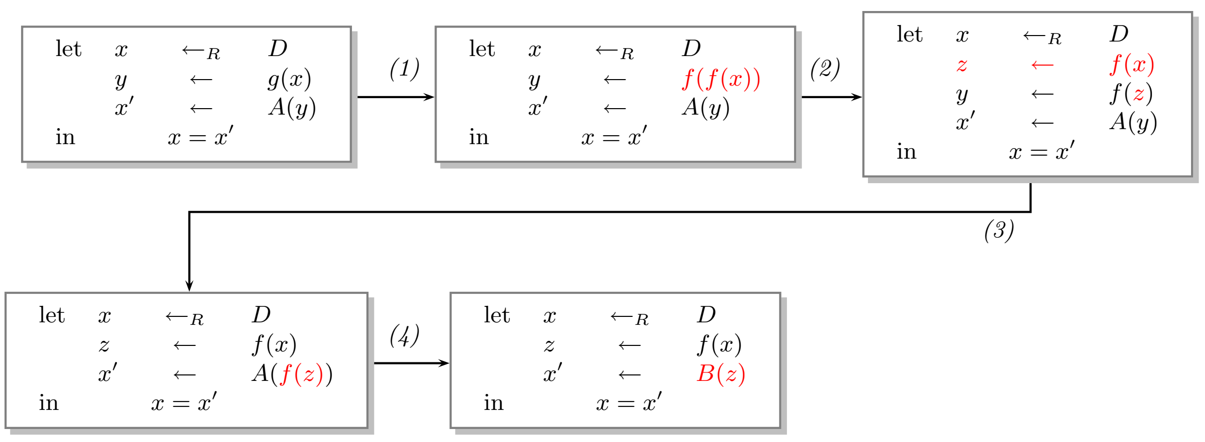
abstract:	Replacing a subprogram by a placeholder that was defined outside the game to stand for this subprogram.
model:	$P \to P$ where we replaced a subprogram by a placeholder
preconditions:	None.

4.9 Remarks

One would be able to do the overall transformation without using the *insert independent code* transformation and the *dead code elimination* transformation at all (Figure 22). This is because the *expression propagation* transformation erases one line of code and the *expression subsumption* inserts a new line of code just by the way we modeled them. Nevertheless, the version presented in Section 3 is in my opinion much more intuitive and easier to understand than this version. Therefore it makes sense to have the *insert independent code* transformation and the *dead code elimination* transformation properly defined on their own.

definitons:
f is a one-way permutation
D is the domain of f
$g(x) := f(f(x))$
$B(x) := A(f(x))$

(1)	*inline propagation*
(2)	*expression subsumption*
(3)	*expression propagation*
(4)	*inline subsumption*



$$\begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & y & \leftarrow & g(x) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array} \xrightarrow{(1)} \begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & y & \leftarrow & f(f(x)) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array} \xrightarrow{(2)} \begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & y & \leftarrow & f(z) \\ & x' & \leftarrow & A(y) \\ \text{in} & & x = x' & \end{array}$$

$$\xrightarrow{(3)} \begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & x' & \leftarrow & A(f(z)) \\ \text{in} & & x = x' & \end{array} \xrightarrow{(4)} \begin{array}{llll} \text{let} & x & \leftarrow_R & D \\ & z & \leftarrow & f(x) \\ & x' & \leftarrow & B(z) \\ \text{in} & & x = x' & \end{array}$$

Figure 22: The complete transformation without *insert independent code* and *dead code elimination*

5 Outlook

The goal of the work presented in [1] is to offer a framework in which one can easily define new games that describe security properties and then transform these games by simply applying transformations from a large collection. The next steps after this thesis are implementing the shown transformations in `Isabelle/HOL` and to prove them by following the proof steps shown in this thesis. This might sound like a simple task, but in general it takes a lot of effort to transfer more involved proofs like those shown in Chapter 4 into a proof assistant like `Isabelle/HOL`.
Furthermore the set of formalized transformations is relatively small at the moment. One also has to find more transformations that might help in other situations and formalize them.

6 Related Work

The machine-based verification of cryptographic proofs is an active field. Backes et al. presented some approaches and tools to define and prove game-transformations in [1]. Malte Skoruppa is currently writing his bachelor thesis on the definition of game-transformations, using the language given in [1], to prove the `IND-CPA` security of `ElGamal` (see [12]).
There are also other frameworks that deal with game-based cryptographic proofs. `CryptoVerif` (see [6]) was the first approach to build an automated prover that is able to verify the security of cryptographic systems. In [7] the authors show how to verify the `EF-CMA` security (secure against existential forgery under an adaptive chosen message attack) of the `Full Domain Hash` signature scheme. Another framework is `CertiCrypt` (see [2]) which is based on the `Coq` proof assistant (see [13]). In [2] the authors present a game-based proof for the `IND-CPA` security of `ElGamal` which they have also verified in the `CertiCrypt` framework. A game-based proof for the `EF-CMA` security of the `Full Domain Hash` signature scheme using `CertiCrypt` is shown in [4]. Even though there are already a lot of cryptographic proofs verified in `CertiCrypt`, in contrast to the language by Backes et al. the `CertiCrypt` language is not higher order and deals only with discrete probability measures. Therefore, reasoning about oracles and information-theoretic security might be difficult. Fundamental principles of game-based cryptographic proofs are explained and discussed in [5]. The authors also give suggestions what kind of game-transformations might be useful in a framework to verify cryptographic proofs.

References

[1] Michael Backes, Matthias Berg, and Dominique Unruh. A formal language for cryptographic pseudocode. In *LPAR*, pages 353–376, November 2008.

[2] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Formal certification of elgamal encryption. In *Formal Aspects in Security and Trust*, pages 1–19, 2008.

[3] Heinz Bauer. *Maß- und Integrationstheorie.* Walter de Gruyter, Berlin, New York, 2nd edition, 1992.

[4] Santiago Zanella Béguelin, Gilles Barthe, Benjamin Grégoire, and Federico Olmedo. Formally certifying the security of digital signature schemes. In *IEEE Symposium on Security and Privacy*, pages 237–250. IEEE Computer Society, 2009.

[5] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006.

[6] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 140–154, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Bruno Blanchet and David Pointcheval. Automated security proofs with sequences of games. In Cynthia Dwork, editor, *CRYPTO'06*, volume 4117 of *Lecture Notes on Computer Science*, pages 537–554, Santa Barbara, CA, August 2006. Springer Verlag.

[8] Oded Goldreich. *Foundations of Cryptography: Basic Tools.* Cambridge University Press, New York, NY, USA, 2000.

[9] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[10] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1 edition, February 2002.

[11] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, Nov 2004.

[12] Malte Skoruppa. Ind-cpa security of elgamal - a theoretical approach using game-based transformations, Jan 2010. Bachelor Thesis.

[13] Coq D. Team. The Coq proof assistant reference manual, version 8.2, August 2009.