On Mechanizing Black-Box Reduction-Based Proofs Using Minimalistic Abstract Interpretation Framework

Alexey Tatuzov[*]

March 24, 2022

Abstract

We introduce a new approach to mechanizing black-box reduction-based proofs in computational settings. At the core of our framework lies a novel concept of iterative automatons that are infinite automatons such that the problem of their equivalence is partially solvable. We construct an algebra of iterative automatons based on iterative composition operation that resembles the substitution of an oracle in a Turing machine. In this algebra, one can verify equational relations using an automatic procedure. These results are inherently computationally sound as we use pure abstract interpretation methodology without relying on a formal calculus.

As an example of an application of our framework, we provide enhanced proofs for some UC-theory statements. In particular, we demonstrate mechanized proof for the general universal composability theorem. Also, we prove the security of an authenticated channel protocol based on a EUF-CMA secure signature scheme.

[*] aatatuzov@gmail.com

Contents

1 Introduction

Theoretical cryptography faces a barrier in providing a way to prove security for complex concurrent protocols in an asynchronous environment. Such proof should deal with an excessive amount of cases arising from an asynchronous nature of communications. It makes human-made proofs hardly achievable in the general case.

The framework of Universal Composable Security [Can00] provides a means to reduce the problem of enumerating all possible security breaches to the question of polynomial indistinguishability of two systems of ITMs. Universal composability theorem allows dividing the task of proving security into simpler subproblems. Unfortunately, this progress seems insufficient to achieve human-provable security for complex protocols.

In the pioneering work [DY83], Dolev and Yao demonstrated how one could put an adversary into a restrictive symbolic model to enumerate all his possible strategies. Abadi and Rogaway showed how to use this method to make computationally sound proofs of security ([AR02]). Results of [CH06] demonstrate how this approach can afford proof of the universal composable security as well.

The π-calculus [MPW92] provides another way to automatize security proofs using formal logic instruments. Blanchet [Bla05] proposed a way to make formal proofs in π-like-calculus sound in computational settings (see CryptoVerif project[1]). In recent work [CSV19] it was shown how to use EasyCrypt, a toolkit based on CryptoVerif, to prove security in UC settings.

1.1 Our contributions

We propose a new framework for mechanizing black-box reduction-based proofs. We provide a means to mechanize only one aspect of proof, namely checking the equivalence of some constructions, without touching the rest of the proof.

We base our work on the barebone abstract interpretation methodology. Although there is ongoing progress in this area starting from the work [CC77], we are unaware of any computationally sound results and start from scratch.

It is convenient to split our work into logical layers so that the results of each layer depend only on the results of previous layers.

Layer 0. Basic sets We propose to replace standard binary string data format with ground terms, which are formulas with an arbitrary number of functional symbols and constants[2]. Although terms are already widely used in all fields of mathematics, the novelty of our proposition is to adopt their usage without semantics. We do not attach any special meaning to functional symbols and constants in those terms and use them just to add a hierarchical structure to data.

A term with variables describes a set of ground terms constructible by substituting various ground terms in place of variables of the original term. Such sets we call term sets.

A union of a finite number of term sets we call a basic set. It seems that basic sets have not yet attracted considerable attention, at least we did not find works where these sets would be mentioned in the explicit definition.

Basic sets have several properties that make them very convenient instrument in describing traces of computations. They are closed under standard set operations (except complement, see Theorem 1). The well-known term unification procedure [PW78] provides a convenient way to make various calculations on term sets. In particular, all mentioned

[1] `https://bblanche.gitlabpages.inria.fr/CryptoVerif/`

[2] One can assume that functional symbols and constants are just strings.

operations on basic sets are efficiently computable[3]. The problem of equality of basic sets is also efficiently solvable.

Layer 1. Iterative automatons An iterative automaton is an infinite automaton, which is a function that maps (state, input) pair to (state, output) pair, defined on ground terms. We restrict our attention to deterministic automatons but permit epsilon transitions. The latter is the reason why we call them "iterative".

We provide a way to define an iterative automaton by basic set. Let's call an automaton defined by a basic set a basic iterative automaton for short.

The main result of this layer is an algorithm for checking the equivalence of basic iterative automatons. We adopt the standard method of checking the equivalence of finite automatons despite infinity of the set of ground terms, see details in Subsection 4.3. Of course, our algorithm does not provide a complete solution for the equivalence problem as it is provably unsolvable[4]. But the specter of automatons for which algorithm works is wide enough to meet our needs in this article.

We provide a simple programming language and a procedure that compiles listings into basic iterative automatons. This makes the language replaceable.

Layer 2. Iterative composition The iterative composition operation combines two iterative automatons f and g into an iterative automaton $h = f\,g$ in the same way that a Turing machine M^O combines with a Turing machine N to form the Turing machine with the oracle M^N. It is important to note that iterative composition operation is describable using only standard set operations and basic sets.

This composition operation has the following properties.

1. The iterative composition of basic iterative automatons is also a basic iterative automaton.

2. The automaton equivalence relation is compatible with the iterative composition operation.

3. For arbitrary expression based on iterative composition operation there exists a basic iterative automaton that realizes this expression, e. g. for expression $((x\,(z\,w))\,(q\,y))$ there exists H such that $(((((H\,x)\,y)\,z)\,w)\,q) = ((x\,(z\,w))\,(q\,y))$ for every x, y, z, w, q. In particular, there are basic iterative automatons that realize combinators (except S) from combinatory logic, see Section 5 for details.

These properties allow us to automatize checking equalities like

$$Z(\mathrm{Exec}(A, N)) =_A \mathrm{AdvZ}(Z, A)(\mathrm{Exec}(\mathrm{DummyAdv}, N)),^5$$

where iterative automatons Exec, AdvZ, and DummyAdv, DummyP are fixed basic iterative automatons, and Z, A, N are arbitrary iterative automatons. Let us demonstrate how this works. We can construct iterative automatons H_1 and H_2 such that

$$(((H_1\,\mathrm{Exec})\,Z)\,A)\,N =_A Z(\mathrm{Exec}(A, N)) \text{ and}$$

$$(((((H_2\,\mathrm{AdvZ})\,\mathrm{Exec})\,\mathrm{DummyAdv})\,Z)\,A)\,N =_A \mathrm{AdvZ}(Z, A)(\mathrm{Exec}(\mathrm{DummyAdv}, N))$$

Then the task of checking original equality is reduced to checking the following equality

$$H_1\,\mathrm{Exec} =_A ((H_2\,\mathrm{AdvZ})\,\mathrm{Exec})\,\mathrm{DummyAdv} \Rightarrow (((H_1\,\mathrm{Exec})\,Z)\,A)\,N =_A (((((H_2\,\mathrm{AdvZ})\,\mathrm{Exec})\,\mathrm{DummyAdv})\,Z)\,A)\,N$$

[3] We assume that basic sets are encoded as lists of terms, and terms are encoded as digraphs.

[4] One can show that the class of basic iterative automatons forms a Turing complete computation model.

where the implication comes from the fact that $=_A$ relation is a congruence.

In practice, we use another method based on replacing variables in expression with some kind of variable-types automatons (see details in the proof of Theorem 5). This approach is more general as it allows us to verify the fact that automatons like H_1 and H_2 do indeed realize corresponding expressions.

Layer 3. Polynomiality We introduce a notion of polynomial iterative automatons that are probabilistic polynomial time Turing machines under the hood of iterative automatons input-output format.

The main ingredient for reasoning about polynomiality in our framework is a concept of a polynomial operator. We call a polynomial iterative automaton f a polynomial operator if it preserves polynomiality after being applied to another polynomial iterative automaton g, i. e. $f\,g$ is also a polynomial automaton. Unfortunately, this part of the framework is not completed yet and may require some tuning.

We base the definition of polynomial indistinguishability on the concept of a polynomial operator. Roughly speaking, two iterative automatons are polynomially indistinguishable if no polynomial operator can output 1 after being applied to one of these automatons significantly more often than to the other.

[Not done] We believe there exists an algorithm that in some cases answers the following question about arbitrary basic iterative automaton h: is $h(x, \ldots, z)$ a polynomial operator / polynomial iterative automaton if iterative automatons $x, y, \ldots, z$ are polynomial operators / polynomial iterative automatons. Unfortunately, this algorithm is to be implemented. The idea is to check whether automaton h routes messages between $x, y, \ldots, z$ in such a way that resembles the hierarchical structure (i.e. digraph without cycles) where those of $x, y, \ldots, z$ which are not polynomial operators are located in places of sink vertexes.

We assume that we already have such a method in the rest of the work. It does not significantly affect the results of the work as all iterative automatons for which we should apply such a test are straightforward. Problems can only arise if there is an unrecoverable flaw in our definition of a polynomial operator.

Layer 4. Memory abstraction We designed iterative automatons and their composition in a way that makes it possible for iterative automatons to do separate queries to something like an oracle that provides random access memory functionality. Those requests are formalized as a special kind of iterative automatons output message. We use the notation $\mathrm{mem}(f)$ to denote an iterative automaton that works like f as if it got linked to the memory oracle.

The algorithm for checking an extended version of equivalence, namely $\mathrm{mem}(f) =_A \mathrm{mem}(g)$, is provided. It is not trivial as it requires an abstract model of a memory state. See details in Subsection 4.3.3.

We use this algorithm to prove results related to an emulation of execution of multiple independent sub-processes (for example, it could be participants of cryptographic protocol).

Layer 5. Applications The problem of mechanizing cryptographic proofs requires extensive use of semantic properties of primitives. Unlike calculus methodology, we don't rely on any formal inference rules and are to base reasoning solely on a game-based approach. One should code every semantic property of a primitive in the form of, say, polynomial indistinguishability of two games. Every proof in this methodology requires constructing a sequence of iterative automaton equalities which step by step incorporates different games describing the semantics of necessary primitives. We gave iterative automatons access to library calls to provide a means for describing such games (in the same way as we did it for memory calls).

[5] Here we use carrying-style notation: $f(g_1, \ldots, g_n) \overset{\mathrm{def}}{=} (\cdots (f\,g_1)\,\ldots)\,g_n)$.

This approach is not much different from that used in recent frameworks based on π-calculus. The main difference is that we root from a much smaller core. So, our framework may require longer chains of equalities, but instead, it grants more flexible models.

For example, we provide a formalization of a variant of the Universally Composable Security with global setup [CDPW07] (see Section 6). Alongside the technical theorems (including the universal composability theorem), we prove the security of the authenticated channel protocol assuming that we have a EUF-CMA secure signature scheme.[6] The proof is done for the case of honest participants, but in full-fledged multiparty and multisession settings.

The Universally Composable Security paradigm [Can00] addresses the problem of reducing the security of a multisession protocol to the security of a single session. We didn't touch on this issue in the current work. To fill this gap, we prepared some instruments as a vision for the future. Namely, we demonstrated on a simple example (see Section 7) how one can make use of the hybrid argument technique in our framework.

All supplementary materials can be found on github[7].

[6] We formalize the assumption in the form of the polynomial indistinguishability of two games.

[7] `https://github.com/LeshaTat/ia-calc`

2 Computation Model

We introduce a computational model that is more suitable for adapting abstract interpretation principles then classical Turing machines. In fact, we only change the input/output format by replacing string monads with the free algebra of terms. It is not a big deal from the computational point of view as one can convert terms to strings and back quite efficiently.[8]

2.1 Term sets

A term is a formula built from function symbols, variables and constants.

Let $\mathbb{F} = \{\mapsto, A, B, \ldots\}$ be a set of functional symbols, $\mathbb{V} = \{\underline{\mathrm{a}}, \underline{\mathrm{b}}, \ldots\}$ — a set of variables, and $\mathrm{Const} = \{\mathrm{err}, 0, 1, \ldots\}$ — a set of constants.

Definition 2.1. *A term is an element of $\mathbb{U}$, a set defined by the following inductive principle.*

- $\mathbb{U}_1 = \mathbb{V} \cup \mathrm{Const}$.
- $\mathbb{U}_{i+1} = \mathbb{U}_i \cup \bigcup_{f \in \mathbb{F}} \{f\} \times \mathbb{U}_i^*$.
- $\mathbb{U} = \mathbb{U}_\infty = \bigcup_{i \in \mathbb{N}} \mathbb{U}_i$.

Let t be a term. We denote the set of variables of t by

$$\mathbb{V}_t = \begin{cases} t, & t \in \mathbb{V}, \\ \emptyset, & t \in \mathrm{Const}, \\ \bigcup_{i=1}^n \mathbb{V}_{t_i}, & t = (f, t_1, \ldots, t_n). \end{cases}$$

We say that a term is a ground term if it does not contain any variables. The set of ground terms is denoted by $\overline{\mathbb{U}}$. For convenience, we sometimes write $F(t_1, \ldots, t_n)$ instead of $(F, t_1, \ldots, t_n)$.

Definition 2.2. *Let $h : \mathbb{V} \to \mathrm{Const}$. Define a mapping $[\cdot]_h : \mathbb{U} \to \overline{\mathbb{U}}$ recurrently:*

$$[t]_h = \begin{cases} h(t), & t \in \mathbb{V} \\ t, & t \in \mathrm{Const} \\ (f, h(t_1), \ldots, h(t_n)), & f \in \mathbb{F}, t = (f, t_1, \ldots, t_n). \end{cases}$$

Definition 2.3. *A term set is a set $[t] = \bigcup_{h \in V \to C} [t]_h$ for $t \in \mathbb{U}$.*

Proposition 2.1 (Terms unification [PW78])**.** *Let $t, s \in \mathbb{U}$. Then one of following condition holds:*

1. $\exists r \in \mathbb{U} : \; [t] \cap [s] = [r]$,
2. $[t] \cap [s] = \emptyset$.

It is computationaly feasible to find a term r if it exists.

It is convenient to consider function like term sets and define apply and composition operations on them. We utilize the functional symbol $\mapsto \in \mathbb{F}$ to make pairs of arguments and values and use a notation $a \mapsto b \overset{\mathrm{def}}{=} (\mapsto, a, b)$.

[8] Note that we assume that terms are coded as digraphs to ensure the polynomiality of some algorithms.

Definition 2.4. *Let $f, g \subseteq \overline{\mathbb{U}}$, and $x \in \overline{\mathbb{U}}$. Then,*

$$f(x) = \begin{cases} y, & \text{for } y \in \overline{\mathbb{U}} \text{ such that } x \mapsto y \in f, \\ \perp, & \text{otherwise;} \end{cases}$$

$$f \circ g = \{x \mapsto z \in \overline{\mathbb{U}} \mid x \mapsto y \in g \ \wedge \ y \mapsto z \in f\}.$$

The operation $f(x)$ extends to sets: $f(\alpha) = \{y \in \overline{\mathbb{U}} \mid \exists x \in \alpha \,.\, x \mapsto y \in f\}$.

Definition 2.5. *A basic set is a finite union of term sets.*

We say that a set of term sets $[t_i]$ is a canonical representation of basic set b if $b = \bigcup_i [t_i]$ and $[t_i] \not\subset [t_j]$ for all i, j.

Proposition 2.2. *For every basic set there exists exactly one canonical representation.*

In the following computational statements we assume terms are implemented as labeled oriented acyclic graphs, and canonical representations are used for basic sets.

Theorem 1. *The family of basic sets is closed under following operations:*

- *union, intersection, and cartesian product;*
- *mapping a set $f(\alpha)$, and composition $f \circ g$.*

All listed operations are efficiently computable.

Theorem 2. *The problem of equality of basic sets is computationally solvable.*

2.2 Iterative automatons

We base our model on the notion of iterative automaton. It is an automaton that use terms as states and input/output values. Also, we allow these automatons to iterate without outputing a value.

A set of all possible output messages of iterative automatons is defined by Output:

$$\mathrm{Output} \overset{\text{def}}{=} [\mathrm{Out}(\underline{\mathrm{mes}})] \cup \mathrm{LibOutput}$$

$$\mathrm{LibOutput} \overset{\text{def}}{=} [\mathrm{Lib}(\underline{\mathrm{name}}, \underline{\mathrm{req}})] \cup \mathrm{MemOutput}$$

$$\mathrm{MemOutput} \overset{\text{def}}{=} [\mathrm{Mem}(\underline{\mathrm{id}}, \mathrm{Get}(\underline{\mathrm{addr}}))] \cup [\mathrm{Mem}(\underline{\mathrm{id}}, \mathrm{Put}(\underline{\mathrm{addr}}, \underline{\mathrm{val}}))].$$

An iterative automaton can be a partially defined. To simplify notation, we use symbol $\perp$ to mark the case of undefined output and denote extended set of possible automaton outputs $\mathrm{Output}_\perp \overset{\text{def}}{=} \mathrm{Output} \cup \{\perp\}$.

Definition 2.6. *An iterative automaton is a subset of $\overline{\mathbb{U}}$ consisting of elements of the form $a \mapsto b$, where*

- $a \in \{\mathrm{StateMesOut}(s, m)] \mid m \in \mathrm{Output},\ s \in \overline{\mathbb{U}}\} \cup [\mathrm{Iter}(\underline{\mathrm{x}})] \cup [\mathrm{err}]$,
- $b \in [\mathrm{StateMesIn}(\underline{\mathrm{s}}, \underline{\mathrm{m}})] \cup [\mathrm{Iter}(\underline{\mathrm{x}})] \cup [\mathrm{err}]$.

Also we require that iterative automaton should be deterministic, i.e. $\forall x \in \overline{\mathbb{U}}\ \# f(x) \leq 1$.

Definition 2.7. *An iterative closure of an iterative automaton f is an iterative automaton $\mathrm{iter}(f) = \bigcup_{i=1}^{\infty} f^i$, where $f^i \overset{\text{def}}{=} \underbrace{f \circ \cdots \circ f}_{i}$.*

We say that an iterative automaton f is closed if $\mathrm{iter}(f) = f$.

Remark 2.1. *Note that input values on which iterations did not halt will drop out from this closure.*

Definition 2.8. *Let f be an iterative automaton and $(x_1, \ldots) \in \overline{\mathbb{U}}^{\infty}$ — some arbitrary input sequence. We call the following recurrently constructed sequence $(y_1, \ldots) \subseteq \mathrm{Output}_{\perp}^{\infty}$ the output sequence of f on the imput $(x_1, \ldots)$.*

- $s_0 = \{0\}$
- $s_i = \begin{cases} s, & \textit{if } s_{i-1} \neq \perp \textit{ and } \exists y \in \mathrm{Output} \ . \ \hat{f}(\mathrm{StateMesIn}(s', x_i)) = \mathrm{StateMesOut}(s, y)\}, \\ \perp, & \textit{otherwise;} \end{cases}$
- $y_i = \begin{cases} y, & \textit{if } s_{i-1} \neq \perp \textit{ and } \hat{f}(\mathrm{StateMesIn}(s', x_i)) = \mathrm{StateMesOut}(s, y), \\ \perp, & \textit{otherwise.} \end{cases}$

A domain of an iterative automaton f, $\mathrm{dom}(f) \subseteq \mathbb{U}^*$, is a set of prefixes of input sequences $(x_1, \ldots, x_n)$ such that there are no $\perp$ symbols in the corresponding output prefix.

Definition 2.9. *Two iterative automatons f and g are equivalent, $f =_A g$, if for every input sequence $(x_1, \ldots) \in \overline{\mathbb{U}}$ output sequences for these automatons match.*

2.3 Iterative composition

Let f and g be two iterative automatons. The composed iterative automaton $h = f\,g$ works by the following scheme. For formal and detailed description see Definition A.2.

When h receives an input message m it runs automaton f with the input message $\mathrm{Out}(\mathrm{Up}(m))$. Automaton f can perform arbitrary number of requests to the automaton g. To make such a request f outputs a message of the form $\mathrm{Out}(\mathrm{Down}(m))$. Then automaton g runs on the input m and outputs m', after that automaton f gets $\mathrm{Out}(\mathrm{Down}(m'))$ as new input message. A serie of these interactions between f and g ends when automaton f outputs a message of the form $\mathrm{Out}(\mathrm{Up}(m))$. In this case automaton h outputs message m.

In the special case when one of the automatons f or g outputs err interactions are halted and output of h is set to err.

Automaton h also takes account of library requests of f and g. When one of them outputs a message from the set LibOutput, automaton h duplicates these message as its output. After receiving next input it redirects it to the original requestor. A subcase of a message from the set MemOutput $\subset$ LibOutput gets slightly more complicated treatment. A request from f of the form $\mathrm{Mem}(k, a)$ will be transformed to $\mathrm{Mem}(\mathrm{ForkUp}(k), a)$, and a request from g will be transformed to $\mathrm{Mem}(\mathrm{ForkDown}(k), a)$ (see details in Subsection 2.5).

Composed automaton h stores states of automatons f and g; both f and g have no access to the others automaton state.

We will consider iterative composition as left-associative operation, i. e. $f_1\,f_2\,f_3 = (f_1\,f_2)\,f_3$. Also we will use carrying-style notation if it does not allow for ambiguity:

$$f(g_1, \ldots, g_n) \overset{\mathrm{def}}{=} f\,g_1\,\ldots\,g_n.$$

2.4 Linking a library

We provide a way for an iterative automaton to get access to some library functions, for example to make a call to signing and signature verification algorithms. Also, a library is an instrument for passing the secret parameter and random strings to the iterative automatons (see section 3).

Given an automaton f and a library g one can link g to f and get new automaton $f_{\text{lib}\leftarrow g}$. Formal definition of linking is given in Definition A.3.

The scheme behind linking is similar to iterative composition. Iterative automaton $f_{\text{lib}\leftarrow g}$ is based on automaton f. Each output of f of type $\text{Lib}(n, m)$ is redirected to g whose answer v is then redirected back to the f in the form $\text{LibRet}(n, v)$. Other output messages of f are output by $f_{\text{lib}\leftarrow g}$ without modifications.

2.5 Memory

We give an iterative automaton a way to store and get value by a key through explicit memory calls. The purpose of this feature is to simplify the analysis of iterative automatons.

A memory call can be considered as special type of a library call which gets a special treatment during iterative composition (see Subsection 2.3).

The automaton memImpl (see Definition A.4) implements memory functionality. On receiving request $\text{Mem}(k, \text{Get}(a))$ it returns the value v stored for the key pair (k, a) or constant 0 if this value has not yet been stored. After receiving request $\text{Mem}(k, \text{Put}(a, v))$ it stores v as the value for key pair (k, a).

Memory functionality can be linked to an iterative automaton in the same way as above. Iterative automaton $h = \text{mem}(f)$ is based on automaton f where each request of the type $\text{Mem}(k, a)$ is redirected to the memImpl automaton whose answer v is then redirected back to the f of the form $\text{MemRet}(v)$. Other output messages of f are output by h without modification.

Proposition 2.3. *For all iterative automatons f and g*

$$\text{mem}(f\,g) = \text{mem}(f)\,\text{mem}(g).$$

The proposition follows from the structure of the memory automaton and the separation of Mem-type requests we made explicit in the definition of iterative composition.

3 Polynomial Indistinguishability

Our definition of iterative automatons allows them to be partially defined. We do not want to consider such automatons as polynomial. So we start with the definition of a complete iterative automaton.

Let us introduce a set of all possible input message for iterative automatons:

$$\mathrm{CompleteInput} \overset{\mathrm{def}}{=} [\mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{Out}(\underline{\mathrm{mes}}))] \cup [\mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{Lib}(\underline{\mathrm{k}}, \underline{\mathrm{mes}}))] \cup [\mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{Mem}(\underline{\mathrm{k}}, \underline{\mathrm{mes}}))] \cup$$
$$[\mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{LibRet}(\underline{\mathrm{k}}, \underline{\mathrm{mes}}))] \cup [\mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{MemRet}(\underline{\mathrm{mes}}))]$$

Definition 3.1. *An iterative automaton f is complete if $f(x) \neq \perp$ for all $x \in \mathrm{CompleteInput}$.*

We short "closed complete iterative automatons" to CCIA.

Definition 3.2 (Correct Library)**.** *We say CCIA g is a correct library for parameter $n \in \mathbb{N}$ and random string $(w_1, \ldots) \in \{0,1\}^\infty$, if on every input sequence $(x_1, \ldots) \in \mathrm{CompleteInput}^\infty$ the automaton g produces the output sequence $(y_1, \ldots)$, $y_i \subseteq \overline{\mathbb{U}}$, which satisfies next requirements. For every number i*

- *(security parameter) if x_i is of the form $\mathrm{Lib}(\mathrm{Parameter}, t, a)$ then $y_i = \underbrace{\mathrm{U}(\mathrm{U} \cdots (\mathrm{U}) \cdots)}_{n}$ (a term analog of 1^n).*
- *(randomness) if x_i is of the form $\mathrm{Lib}(\mathrm{Random}, t, a)$ then $y_i = \{w_{k+1}\}$.*

Definition 3.3 (Uniform Family of Libraries)**.** *An uniform family of correct libraries is a family of probabilistic distributions $\{D_n\}_{n\in\mathbb{N}}$ on CCIAs, where each CCIA $g \in \operatorname{supp} D_n$ is a correct library for parameter n and random string $\vec{w}_g$, and for every length $m \in \mathbb{N}$ and each prefix $\vec{q} = (q_1, \ldots, q_m) \in \{0,1\}^m$*

$$\mathrm{P}_{g \leftarrow D_n}[\vec{w}_g \in B_{\vec{q}}] = \frac{1}{2^m},$$

where $B_{\vec{q}} = \{(w_1, \ldots) \in \{0,1\}^\infty \mid w_1 = q_1 \wedge \ldots \wedge w_m = q_m\}$.

Definition 3.4. *An uniform family of correct libraries is polynomial if there exists probabilistic polynomial time Turing machine M such that for every $n \in \mathbb{N}$ and every input sequence $(x_1, \ldots, x_m) \in \mathrm{CompleteInput}^m$ random variables*

$$M[1^n, (x_1, \ldots, x_m)] \text{ and } (y^g_1, \ldots, y^g_m)_{g \leftarrow \mathrm{Lib}_n}$$

are equal in distribution, where $(y^g_1, \ldots, y^g_m) \in \mathrm{Output}$, $g \in \mathrm{Lib}_n$, is the output sequence of g on the input sequence $(x_1, \ldots, x_m)$.

Let Lib be a polynomial uniform family of correct libraries.

Definition 3.5. *A CCIA f is polynomial (w.r.t. Lib) if there exists probabilistic polynomial time Turing machine M such that for every $n \in \mathbb{N}$ and every input sequence $(x_1, \ldots, x_m) \in \mathrm{CompleteInput}^m$ random variables*

$$M[1^n, (x_1, \ldots, x_m)] \text{ and } (y^{f_{\mathrm{lib}\leftarrow g}}_1, \ldots, y^{f_{\mathrm{lib}\leftarrow g}}_m)_{g \leftarrow \mathrm{Lib}_n}$$

are equal in distribution, where $(y^{f_{\mathrm{lib}\leftarrow g}}_1, \ldots, y^{f_{\mathrm{lib}\leftarrow g}}_m) \in \mathrm{Output}$, $g \in \mathrm{Lib}_n$, is the output sequence of $f_{\mathrm{lib}\leftarrow g}$ on the input sequence $(x_1, \ldots, x_m)$.

Definition 3.6. *A CCIA f is polynomial operator (w.r.t. Lib) if for every polynomial iterative automaton g the iterative automaton $(f\,g)$ is also polynomial.*

Remark 3.1. *Let's demonstrate some examples of polynomial operators.*

1. *An iterative automaton f that do not make more than fixed number of calls to g during the processing of every outer input of $(f\,g)$.*
2. *An iterative automaton h that restricts total length of messages sent to g by a polynomial in cumulative size of inputs the automaton $(h\,g)$ got so far (i.e. don't count answers from g).*
3. *An hybrid iterative automaton w such that $w\,g = h\,(f\,g)$.*

In the following, we'll use a short notation $\mathrm{P}_n[h] \overset{\text{def}}{=} \mathrm{P}_{l\leftarrow \mathrm{Lib}_n}[h_{\mathrm{lib}\leftarrow l}(\mathrm{StateMesIn}(0, \mathrm{Out}(0))) \in [\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{Out}(1))]]$ when the value of Lib is clear from the context.

Definition 3.7. *Two polynomial iterative automatons f and g are polynomial indistinguishable (w.r.t.* Lib*), $f \simeq_p g$, if for every polynomial operator z*

$$|\mathrm{P}_n[z\,f] - \mathrm{P}_n[z\,g]| = \nu(n).$$

Definition 3.8. *Let M be an oracle Turing machine and f be a CCIA. We denote by M^f the machine M with an oracle, that answers to the serie of requests $(x_1, \ldots)$, $x_i \in \mathrm{CompleteInput}$, with a serie of answers $(y_1, \ldots)$, $y_i \in \overline{\mathbb{U}}$.*

Theorem 3 (Incomplete: Lib has to satisfy some sort of stationary property)**.** *Let* Lib *be a ... (todo). Two polynomial iterative automatons f and g are polynomial indisinguishable (w.r.t.* Lib*) iff for every oracle PPT M*

$$|\mathrm{P}_{l\leftarrow \mathrm{Lib}_n}[M^{f_{\mathrm{lib}\leftarrow l}}(1^n) = 1] - \mathrm{P}_{l\leftarrow \mathrm{Lib}_n}[M^{g_{\mathrm{lib}\leftarrow l}}(1^n) = 1]| = \nu(n).$$

Theorem 4. *The polynomial indistinguishability relation of CCIA satisfies following properties.*

- *$\simeq_p$ is an equivalence relation, i.e. it is reflexive, transitive and symmetric.*
- *$f =_A g \Rightarrow f \simeq_p g$.*
- *$f \simeq_p g \;\Rightarrow\; (q\,f) \simeq_p (q\,g)$ if q is a polynomial operator.*

Elements of the proof. The proof of the last part is quite an automatic one. There exists a CCIA B such that

$$(\mathrm{B}\,z\,q)\,x =_A z\,(q\,x).$$

This equivalence is a corollary of Theorem 5.

If $(q\,f) \not\simeq_p (q\,g)$ then there exists a polynomial operator z such that

$$|\mathrm{P}_n[z\,(q\,f)] - \mathrm{P}_n[z\,(q\,g)]| \neq \nu(n).$$

Then we get

$$|\mathrm{P}_n[(B\,z\,q)\,f] - \mathrm{P}_n[(B\,z\,q)\,g]| \neq \nu(n).$$

It follows from the structure of B that $(B\,z\,q)$ is a polynomial operator (todo: full proof) and so we conduct that $f \not\simeq_p g$. □

4 Calculating Iterative Automatons

We propose a general scheme to define iterative automatons by basic sets. This scheme permits implement of iterative closure and composition operations on behalf of standard set operations listed in Theorem 1.

We present an algorithm that can solve, in some cases, the problem of equality of iterative automatons. The algorithm is based on a combination of abstract interpretation principles and the standard procedure of checking the equivalence of finite automatons. We present it at the end of this section.

4.1 Complete iterative automatons

A complete iterative automaton should define an output for every input. Only a complete iterative automaton can be polynomial.

We use the following principle to define an complete iterative automaton. We define an iterative automaton by a basic set and extend it with a rule to output err on all remaining input values.

Definition 4.1. *Let t be a basic set. We call t an iterative automaton specification (or just IA-specification for shortness) if it is an iterative automaton and*

$$[t] \setminus \{x \mapsto y \mid x \in \mathrm{CompleteInput},\ y \in \overline{\mathbb{U}}\} = \emptyset$$

Definition 4.2. *Let $f \subseteq \overline{\mathbb{U}}$ be an iterative automaton.*

An error complementory to f is a set $f_{\mathrm{err}} = \{x \mapsto \mathrm{err} \mid \forall y \in \overline{\mathbb{U}}\ f(x) =\perp\}$.

Definition 4.3. *Let t be a IA-specification. We say that t defines f if*

$$f = t \cup t_{\mathrm{err}}.$$

In the following propositions, we will show that this principe of defining iterative automatons can be conjugated with iterative composition and closure operations.

Proposition 4.1. *Let t define a complete iterative automaton f. If iterative closure* $\mathrm{iter}(f)$ *is complete and* $\mathrm{iter}(t)$ *is a basic set then* $\mathrm{iter}(t)$ *defines* $\mathrm{iter}(f)$*.*

Proof. Suppose that iterative closure $\mathrm{iter}(f)$ is complete and choose arbitrary $x \in \mathrm{CompleteInput}$. Then there exists n such that $f^n(x) \notin [\mathrm{Iter(x)}]$.

Consider two cases $t(x) =\perp$ and $t(x) \neq\perp$. In the first case, $\mathrm{iter}(f)(x) = f(x) = \mathrm{err}$ and $\mathrm{iter}(t)(x) =\perp$. In the second case, $t^n(x) = f^n(x)$ as f does not extend t on the inputs of the form $\mathrm{Iter}(m)$, and $t^i(x) \in [\mathrm{Iter}(m)]$ for every $i \in \overline{1,n}$.

Consequently, $\mathrm{iter}(f) = \mathrm{iter}(t)_{\mathrm{err}} \cup t_{\mathrm{err}} = \mathrm{iter}(t) \cup \mathrm{iter}(t)_{\mathrm{err}}$. □

Proposition 4.2. *Let t and r define complete iterative automatons f and g respectively. The set $(t\,r)$ is a basic set. If iterative closure* $\mathrm{iter}(f\,g)$ *is complete and* $\mathrm{iter}(t\,r)$ *is a basic set then* $\mathrm{iter}(t\,r)$ *defines* $\mathrm{iter}(f\,g)$*.*

Informal proof. The fact that $(t\,r)$ is a basic set follows from the Theorem 1.

There are two reasons why input x could be undefined for $\mathrm{iter}(t\,r)$.

The first case is that somewhere in the iteration there was an input message for t or r on which those automatons do not define output message. In this case, the corresponding output of $\mathrm{iter}(f\,g)$ would be err due to the construction of the iterative composition.

The second case is that the iterations starting from x do not end because of some kind of cycle. This scenario is impossible as in this case $\mathrm{iter}(f\,g)$ would not be complete as f and g do not differ from t and r structurally.

We conclude that every input lacking in $\mathrm{iter}(t\,r)$ falls into the first case which means that $\mathrm{iter}(t\,r)_{\mathrm{err}} = \mathrm{iter}(f\,g) \setminus \mathrm{iter}(t\,r)$. □

4.2 Iterative closure algorithm

Input: an IA-specification t that defines iterative automaton f.
Output: $\mathrm{iter}(t)$. Will not halt if $\mathrm{iter}(f)$ is not complete.

Let $t_0 = f$ and calculate the sequence of basic sets $t_i = t_{i-1} \cup (f \circ t_{i-1})$. Once the sequence stabilizes output the fixed point t_∞ with all $\mathrm{Iter}(\cdot)$ positions cleared out:

$$t_\infty \cap ([\underline{\mathrm{x}} \mapsto \mathrm{StateMesOut(Out(\underline{m}))}] \cup [\underline{\mathrm{x}} \mapsto \mathrm{StateMesOut(Lib(\underline{k}, \underline{m}))}] \cup [\underline{\mathrm{x}} \mapsto \mathrm{StateMesOut(Mem(\underline{k}, \underline{m}))}] \cup \\ [\mathrm{StateMesIn(Out(\underline{m}))} \mapsto \underline{\mathrm{y}}] \cup [\mathrm{StateMesIn(LibRet(\underline{m}))} \mapsto \underline{\mathrm{y}}] \cup [\mathrm{StateMesIn(MemRet(\underline{m}))} \mapsto \underline{\mathrm{y}}]).$$

□

4.3 Equality checking algorithms

In the following, we will present three algorithms that check equivalence of iterative automatons. One can consider the first two algorithms as preliminary variants of the last one.

The main idea is to adopt a standard procedure for checking finite automaton equivalence. We start from the initial state for both automatons, feed them the same input and get a pair of new states. Repeat the process until the set of pairs of states stabilizes.

In our case, the set of all possible automaton states is not finite. We overcome this by using term sets (e.g., $[t]$) instead of individual values. This method will fail in general, but it works in all cases that we are interested in in this article.

Definition 4.4. *An IA-specification t is a closed iterative automaton specification (cIA-specification) if it is a closed iterative automaton.*

4.3.1 Naive algorithm

Input: cIA-specifications t and r which define complete iterative automatons f and g.
Output: Yes, if for every good input sequence output sequences for automatons f and g match; otherwise - No. May not halt.

We call input sequence a good one if for both automatons f and g corresponding output sequences do not contain err. In other words, during execution on good input sequences automatons t and r should always have their outputs defined.

Construct a directed graph.

Nodes The nodes are identified with terms of the following form:

- $\mathrm{ModeF(StateMesIn(s, \underline{x}), StateMesIn(d, \underline{x}))}$, where $s, d \in \mathbb{U}$ and variable $\underline{\mathrm{x}}$ is not present in s nor d,
- $\mathrm{ResF(StateMesOut(s, y), StateMesIn(d, x))}$, where $s, d, x, y \in \mathbb{U}$,

- $\mathrm{ResG}(\mathrm{StateMesOut}(\mathrm{s}, \mathrm{y}), \mathrm{StateMesOut}(\mathrm{d}, \mathrm{y}'))$, where $s, d, y, y' \in \mathbb{U}$.

Traverse through all nodes of the graph starting from the node $\mathrm{ModeF}(\mathrm{StateMesIn}(0, \underline{\mathrm{x}}), \mathrm{StateMesIn}(0, \underline{\mathrm{x}}))$. If during the process the algorithm walks through the node $\mathrm{ResG}(\mathrm{StateMesOut}(\mathrm{s}, \mathrm{y}), \mathrm{StateMesOut}(\mathrm{d}, \mathrm{y}'))$ such that $y \neq y'$, then it outputs "No". Otherwise, after all nodes are traversed, it outputs "Yes".

Arcs Let

$$t_{\text{ext}} = [(\mathrm{StateMesIn}(\underline{\mathrm{s}}, \underline{\mathrm{x}}) \mapsto \underline{\mathrm{y}}) \mapsto (\mathrm{ModeF}(\mathrm{StateMesIn}(\underline{\mathrm{s}}, \underline{\mathrm{x}}), \mathrm{StateMesIn}(\underline{\mathrm{d}}, \underline{\mathrm{x}})) \mapsto \mathrm{ResF}(\mathrm{StateMesOut}(\underline{\mathrm{s}}, \underline{\mathrm{y}}), \mathrm{StateMesIn}(\underline{\mathrm{d}}, \underline{\mathrm{x}})))](t)$$

$$r_{\text{ext}} = [(\mathrm{StateMesIn}(\underline{\mathrm{d}}, \underline{\mathrm{x}}) \mapsto \underline{\mathrm{y}}) \mapsto (\mathrm{ResF}(\mathrm{StateMesOut}(\underline{\mathrm{s}}, \underline{\mathrm{y}}), \mathrm{StateMesIn}(\underline{\mathrm{d}}, \underline{\mathrm{x}})) \mapsto \mathrm{ResG}(\mathrm{StateMesOut}(\underline{\mathrm{s}}, \underline{\mathrm{y}}), \mathrm{StateMesOut}(\underline{\mathrm{d}}, \underline{\mathrm{z}})))](r).$$

Let's list all arcs of the graph.

- Every node $n = \mathrm{ModeF}(\cdot, \cdot)$ is connected to nodes that constitute basic set $t_{\text{ext}}(n)$: nodes $n_1, \ldots, n_k$ such that $t_{\text{ext}}(n) = [n_1] \cup \cdots \cup [n_k]$[9].
- Every node $n = \mathrm{ResF}(\cdot, \cdot)$ is connected to the nodes that constitute basic set $r_{\text{ext}}(n)$.
- Every node $n = \mathrm{ResG}(\mathrm{StateMesOut}(\mathrm{s}, \mathrm{y}), \mathrm{StateMesOut}(\mathrm{d}, \mathrm{y}'))$ is connected to node $\mathrm{ModeF}(\mathrm{StateMesIn}(\mathrm{s}, \underline{\mathrm{x}}), \mathrm{StateMesIn}(\mathrm{d}, \underline{\mathrm{x}}))$.

  Here we assume that the variable $\underline{\mathrm{x}}$ is not present in term s and term d. If it is not the case then one should replace $\underline{\mathrm{x}}$ with an appropriate fresh variable.

□

This algorithm has a flaw. It does check that iterative automatons will output the same values only when both automatons define outputs. So, we need to add a check for the case if one of the automatons loses output while the other still has it.

4.3.2 Full fledged algorithm

Input: cIA-specifications t and r which define complete iterative automatons f and g.
Output: Yes, if $f =_A g$; otherwise - No. May not halt.

Construct the same graph as in the naive algorithm and remove some arcs from it. Namely, remove arcs that connects node $n = \mathrm{ResF}(\cdot, \cdot)$ to the nodes that constitute basic set $r_{\text{ext}}(n)$ but fails the following contraction check.

Let $r'_{\text{ext}} = [(\underline{\mathrm{x}} \mapsto \underline{\mathrm{y}}) \mapsto (\underline{\mathrm{x}} \mapsto \mathrm{ContCheck}(\underline{\mathrm{x}}, \underline{\mathrm{y}}))](f)$. Apply this function to n and get a basic set $r'_{\text{ext}}(n) = [\mathrm{CheckCont}(n'_1, n_1)] \cup \ldots \cup [\mathrm{CheckCont}(n'_k, n_k)]$. Remove from the graph arcs that connect n with n_i for which $n'_i \neq n$. Add check on existence of outgoing arc for every node of the form $\mathrm{ResF}(\cdot, \cdot)$.

Let's elaborate on this contraction check procedure a little more. Notice that

$$\operatorname{dom} r_{\text{ext}} \cap [n] = [n'_1] \cup \ldots \cup [n'_k].$$

[9] Recall that this representation is unique for the case $[n_i] \not\subset [n_j]$.

To make sure that g defines output to the same extent as f, we should check that $\operatorname{dom} r_{\text{ext}} \cap [n] = [n]$. It follows from the Proposition 2.2 that

$$[n] = [n'_1] \cup \ldots \cup [n'_k] \Leftrightarrow \exists i \; [n] = [n'_i].$$

Note that the procedure described above checks exactly this condition.

Those modifications enable detection of the case when iterative automaton g loses output while f still has it. To make the algorithm symmetric, we rerun it with swapped input. □

4.3.3 Memory-powered algorithm

Input: cIA-specifications t and r which define complete iterative automatons f and g; an advice h.
Output: If Yes then $\text{mem}(f) =_A \text{mem}(g)$. May not halt or output *Error*.

The algorithm is based on the previously described full-fledged algorithm. We modify it by extending the set of graph nodes. Recall that every node was identified by a term. Now we extend each node's identifier with a description of a memory state.

We modify the arcs structure of the graph for the nodes for which term part of the identifier is of the form

$$\mathrm{ResF(StateMesOut(s, Mem(k, m)), StateMesIn(d, x))} \text{ or}$$

$$\mathrm{ResG(StateMesOut(s, y), StateMesOut(d, Mem(k, m)))}.$$

Those nodes are looped back to $\mathrm{MesF}(\cdot, \cdot)$ and $\mathrm{ResF}(\cdot, \cdot)$ nodes as if corresponding automatons have received answers for the memory calls. Also, the algorithm will output *Error* and stop if k is not a ground term or $m \notin [\mathrm{Get}(\underline{\mathrm{a}})] \cup [\mathrm{Put}(\underline{\mathrm{a}}, \underline{\mathrm{v}})]$ for one of the nodes having above form.

Now we will describe how we model memory state. We will need a bunch of definitions.

Definition 4.5. *A memory advice is a set of pair of terms* $(k, a) \in \mathrm{Const} \times \mathbb{U}$.

Definition 4.6. *A set of pair of terms* $(k', a') \in \mathrm{Const} \times \mathbb{U}$ *is compatible with a memory advice h if there exists a mapping on variables $m : \mathbb{V} \to \mathbb{V}$ such that the set of pairs of the form (k', a'') is a subset of h, where terms a'' are modified versions of term a' with all variables replaced using m.*

For the next definitions, we will assume that a memory advice h is fixed. Recall that this advice is a part of the input of the algorithm.

Definition 4.7. *A memory layer l is a set of tuples* $(k, a, v) \in \mathrm{Const} \times \mathbb{U} \times \mathbb{U}$ *such that*

- *the corresponding set of pairs (k, a) is compatible with a memory hint h,*
- *all a has same variable set $\mathbb{V}_a$.*

A variable set of a layer l is the set of variables shared by all a's and is denoted by $\mathbb{V}_l$.

Definition 4.8. *A description of memory state is a tuple* (pool, cache) *where both* pool *and* cache *are sets of memory layers.*

All layers in cache *have unique variable sets.*

Each description of a memory state is coupled with a term to form a node identifier. We maintain the following invariant properties for those pairs.

Definition 4.9. *Let $d = (\mathrm{pool}, \mathrm{cache})$ be a description of memory state and $t \in \mathbb{U}$. The pair (t, d) is a correct node identifier if $\mathbb{V}_l \subseteq \mathbb{V}_t$ for all $l \in \mathrm{cache}$.*

To maintain this invariant, the algorithm make sures that the arcs structure is organized in the following way. Let's consider an arc that connects a source node $(t_1, (\mathrm{pool}_1, \mathrm{cache}_1))$ with a destination node $(t_2, (\mathrm{pool}_2, \mathrm{cache}_2))$. Assume that a layer $l_1 \in \mathrm{cache}_1$ should correspond to the layer $l_2 \in \mathrm{cache}_2$[10], but $\mathbb{V}_{l_2} \not\subseteq \mathbb{V}_{t_2}$; then the algorithm makes sure that l_2 is put into pool_2 instead of cache_2. The reason why the situation $\mathbb{V}_{l_2} \not\subseteq \mathbb{V}_{t_2}$ occurs is that variables become obsolete over time (for example, they got erased from states of iterative automatons during execution).

All requests to memory from both automatons are processed using the same description of memory state, but we modify the k argument of the request $\mathrm{Mem}(k, \cdot)$ in the following fashion: for the first automaton replace k with $\mathrm{FromF}(k)$, for the second replace k with $\mathrm{FromG}(k)$.

Now we will briefly explain how the algorithm imitates answers to memory calls. Every such answer is an arc of the graph that connects a node with a $\mathrm{Mem}(\cdot, \cdot)$ request to a node with an answer. Depending on the description of the memory state in the original node, there could be zero, one, or many outgoing arcs leading to one or many nodes with answers.

Case 1 Request $\mathrm{Mem}(k, \mathrm{Get}(a))$ and there is a tuple of the form (k, a, v) in the layers in the cache.

The nodes will have exactly one outgoing arc that will lead to the node with the received answer v and the same memory description.

Case 2 Request $\mathrm{Mem}(k, \mathrm{Get}(a))$ and there is a layers l in the cache such that $\mathbb{V}_l = \mathbb{V}_a$.

If the layer l extended with the tuple $(k, a, 0)$ is not compatible with the hint h than output *Error* and stop execution.

The node will have exactly one outgoing arc that will lead to the node with the received answer 0 and the same memory description.

Case 3 Request $\mathrm{Mem}(k, \mathrm{Get}(a))$ and there are no layers l in the cache such that $\mathbb{V}_l = \mathbb{V}_a$.

Enumerate all layers in the pool that has the variable set of the same size as $\mathbb{V}_a$. Add the special layer $\{(k, a, 0)\}$ to this listing.

The graph will have one outgoing arc for each position in this list. To construct this arc we will construct a new memory state description with a new layer added to cache. Then the arc is built in the same way as it was in *Case 1* or *Case 2* but with the new memory state description at the target node.

Let l be a layer in the list. Construct the new memory state by adding l to the old one. Modify the tuples (k', a', v') of l in the following way:

- rename variables in v''s to those that are not presented in terms of layers in cache or the term that is part of the node identifier; use the same variables mapping for all tuples of the layer l;
- rename variables in a''s in such a way that after adding tuple $(k, a, 0)$ the layer will stay compatible with a memory hint h and the variable set of the layer will equal to $\mathbb{V}_a$.

Add the resulted layer to cache of the current memory state to form the new memory state.

Case 4 Request $\mathrm{Mem}(k, \mathrm{Put}(a, v))$ and there is a tuple of the form (k, a, v') in the layers l in the cache.

Make a new memory description by replaceing tuple (k, a, v') with (k, a, v) in layer l.

The node will have exactly one outgoing arc that will lead to the node with the received answer v and the new memory description.

[10] l_1 may be not equal to l_2 because t_1 and t_2 could use different sets of variables.

Case 5 Request $\mathrm{Mem}(k, \mathrm{Put}(a, v))$ and there is a layers l in cache such that $\mathbb{V}_l = \mathbb{V}_a$.

If the layer l extended with the tuple $(k, a, 0)$ is not compatible with the hint h than output *Error* and stop execution.

Make a new memory description by adding a tuple (k, a, v) to l.

The node will have exactly one outgoing arc that will lead to the node with the received answer v and the new memory description.

Case 6 Request $\mathrm{Mem}(k, \mathrm{Put}(a, v))$ and there are no layers l in the cache such that $\mathbb{V}_l = \mathbb{V}_a$.

The process is equivalent to *Case 3* followed by *Case 4* or *Case 5*.

The remaining details of the algorithm are currently omitted. They are comprehensive but straightforward. One can find a python implementation of the algorithm in the complementary materials.

5 Example 1. Combinatory Logic

Program 5.1. $\mathrm{I} \overset{\mathrm{def}}{=} \mathrm{build}($

```
  $\mathrm{call}(1, \underline{\mathrm{mes}}, \underline{\mathrm{mes}})$
  $\mathbf{ret_1}$
)
```

Program 5.2. $\mathrm{K} \overset{\mathrm{def}}{=} \mathrm{build}($

```
  $\mathrm{call}(1, \underline{\mathrm{mes}}, \underline{\mathrm{mes}})$
  $\mathbf{ret_2}$
)
```

Program 5.3. $\mathrm{B} \overset{\mathrm{def}}{=} \mathrm{build}($

```
  $\mathrm{callCbk}(1, \underline{\mathrm{mes}}, \underline{\mathrm{mes}}, (m) \longrightarrow$
    $\mathrm{callCbk}(2, m, m, (m') \longrightarrow$
      $\mathrm{call}(3, m', m')$
    )
  )
  $\mathbf{ret_3}$
)
```

Program 5.4. $\mathrm{C} \overset{\mathrm{def}}{=} \mathrm{build}($

```
  $\mathrm{callCbk_2}(1, \underline{\mathrm{mes}}, \underline{\mathrm{mes}},$
    $(m) \longrightarrow \mathrm{call}(3, m, m),$
    $(m) \longrightarrow \mathrm{call}(2, m, m)$
  )
  $\mathbf{ret_3}$
)
```

Program 5.5. $\mathrm{S} \overset{\mathrm{def}}{=} \mathrm{build}($

```
  $\mathrm{callCbk_2}(1, \underline{\mathrm{mes}}, \underline{\mathrm{mes}},$
    $(m) \longrightarrow \mathrm{call}(3, m, m),$
    $(m) \longrightarrow \mathrm{callCbk}(2, m, m, (m') \longrightarrow \mathrm{call}(3, m', m')))$
  )
  $\mathbf{ret_3}$
)
```

In this and the following sections, we use program listings to define iterative automatons. For the detailed description of syntax and semantics see Appendix B.

Theorem 5. *For all iterative automatons f, g, h:*

- $\mathrm{I}\, f =_A f$,
- $\mathrm{K}\, f\, g =_A f$,
- $\mathrm{B}\, f\, g\, h =_A f\,(g\, h)$,
- $\mathrm{C}\, f\, g\, h =_A f\, h\, g$,
- $\mathrm{S}\, f\, g\, h \neq_A (f\, h)\,(g\, h)$.

Proof. One can check all these equations with the algorithm presented in Subsection 4.3.2. For example, we show how to do it for the case of combinator B.

Definition 5.1. *Let $c \in \mathrm{Const}$. The variable iterative automaton for c is the iterative automaton defined by the following IA-specification*

$$v_c = [\mathrm{StateMesIn}(0, \mathrm{Out}(\underline{\mathrm{x}})) \mapsto \mathrm{StateMesOut}(1, \mathrm{Lib}(c, \mathrm{StateMesIn}(0, \mathrm{Out}(\underline{\mathrm{x}}))))] \cup$$
$$[\mathrm{StateMesIn}(\mathrm{VarState}(\underline{\mathrm{s}}), \mathrm{Out}(\underline{\mathrm{x}})) \mapsto \mathrm{StateMesOut}(1, \mathrm{Lib}(c, \mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{Out}(\underline{\mathrm{x}}))))] \cup$$
$$[\mathrm{StateMesIn}(1, \mathrm{LibRet}(c, \mathrm{StateMesOut}(\underline{\mathrm{d}}, \mathrm{Out}(\underline{\mathrm{x}})))) \mapsto \mathrm{StateMesOut}(\mathrm{VarState}(\underline{\mathrm{d}}), \mathrm{Out}(\underline{\mathrm{x}}))]$$

Let $x = v_0$, $y = v_1$ and $z = v_2$.

Lemma 5.1. *If* $\mathrm{B}\, x\, y\, z = x\,(y\, z)$ *then for all iterative automatons* f, g *and* h

$$\mathrm{B}\, f\, g\, h =_A f\,(g\, h).$$

This lemma can be generalized to any expression. We currently omit the proof.

For the last equation $\mathrm{S}\, f\, g\, h \neq_A (f\, h)\,(g\, h)$ algorithm answers No. Indeed, two instances of h in the right part of the equation do not share the same state as they should to match the left side. □

Remark 5.1. *Although combinator* S *is not a correct* S *combinator for iterative automatons algebra, there exists another algebra for which it does fit. Consequently, all reasonable combinator equations*[11] *are correct. One can verify them by applying the algorithm presented in Subsection 4.3.2. For example,*

$$\mathrm{B} =_A \mathrm{S}\,(\mathrm{K}\,\mathrm{S})\,\mathrm{K},$$

$$\mathrm{C} =_A \mathrm{S}\,(\mathrm{S}\,(\mathrm{K}\,(\mathrm{S}\,(\mathrm{K}\,\mathrm{S})\,\mathrm{K}))\,\mathrm{S})\,(\mathrm{K}\,\mathrm{K}).$$

[11] Some constructions do not make sense in our model. For example S I I (S I I) is an completely undefined automaton, i.e. it always returns the error message.

6 Example 2. Universally Composable Security

Recall that we use carrying-style notation for iteration composition operation:

$$f(g_1, \ldots, g_n) \overset{\text{def}}{=} f\, g_1\, \ldots\, g_n.$$

The model we use here is a modified version of Canetti's model of Universally Composable Security [Can00]. More precisely, we use a variant with global setup (see [CDPW07]), i. e. we consider protocols that implement multiple sessions on their own.

We use the methodology of hierarchical calls rather than horizontal connections between elements of the model. We made this change to simplify the reasoning about the polynomiality of interactive systems and to make use of the simple definition of a polynomial operator. But we believe that it is possible to use techniques of this paper to provide a thorough implementation of the original model.

The UC model we use is based on two routing iterative automatons Exec and Net that bind together four parts of a model: Z — environment, A — adversary, P — protocol, and F — ideal functionality. The environment Z is interacting with composition $\mathrm{Exec}(A, \mathrm{Net}(P, F))$ that provides following communication lines:

- Z can send an input directly to A and P, but not to F;
- A can make several calls to P and F each time it gets input from Z;
- P can make requests to F after being called by Z or A;
- F only answers to requests and do not call anybody.

Note that this structure guarantees that if A and P are polynomial operators and F is a polynomial iterative automaton, then the whole construction is a polynomial iterative automaton. The same goes for intermediate constructions we introduce later in this section.

We provide more details on the inner structure of protocol P in Subsection 6.3.

Let us demonstrate how the hierarchical calls principle can provide the means to send a message from a participant to the adversary. Suppose that a participant receives input from the environment and wants to send a message to the adversary. Then he should save the message in an inner state and return some output to the environment. When the environment sends input to an adversary, she will interrogate all of the participants about messages they want to send. This scheme has some drawbacks, but it is good enough for demonstration purposes.

Definition 6.1. *Let P be a polynomial operator and F, G — polynomial iterative automatons. We say that P UC-realizes F using G if for each polynomial operator A there exists a polynomial operator S such that*

$$\mathrm{Exec}(A, \mathrm{Net}(P, G)) \simeq_p \mathrm{Exec}(S, \mathrm{Net}(\mathrm{DummyP}, F)).$$

Program 6.1. Exec $\overset{\text{def}}{=}$ build($\mathbf{Z} = 1$, $\mathbf{Net} = 2$

```
  switch{
    case $\underline{\text{mes}} \to \text{UserMes}(\underline{\text{mes}})$ :
      call($\mathbf{Net}, \underline{\text{mes}}, \text{FromZ}(\text{UserMes}(\underline{\text{mes}}))$)
    case $\underline{\text{mes}} \to \text{AdvMes}(\underline{\text{mes}})$ :
      callCbk($\mathbf{Z}, \underline{\text{mes}}, \text{AdvMes}(\underline{\text{mes}}), (m) \longrightarrow$
        call($\mathbf{Net}, m, \text{FromA}(m)$)
      )
  }
  ret$_{\mathbf{2}}$
)
```

Program 6.2. Net $\overset{\text{def}}{=}$ build($\mathbf{P} = 1$, $\mathbf{F} = 2$

```
  switch{
    case $\underline{\text{mes}} \to \text{FromZ}(\text{UserMes}(\underline{\text{mes}}))$ :
      callCbk($\mathbf{P}, \underline{\text{mes}}, \text{FromZ}(\text{UserMes}(\underline{\text{mes}})), (m) \longrightarrow$
        call($\mathbf{F}, m, \text{FromP}(m)$)
      )
    case $\underline{\text{mes}} \to \text{FromA}(\text{ToP}(\underline{\text{mes}}))$ :
      callCbk($\mathbf{P}, \underline{\text{mes}}, \text{FromA}(\underline{\text{mes}}), (m) \longrightarrow$
        call($\mathbf{F}, m, \text{FromP}(m)$)
      )
    case $\underline{\text{mes}} \to \text{FromA}(\text{ToF}(\underline{\text{mes}}))$ :
      call($\mathbf{F}, \underline{\text{mes}}, \text{FromA}(\underline{\text{mes}})$)
  }
  ret$_{\mathbf{2}}$
)
```

Program 6.3. DummyP $\overset{\text{def}}{=}$ build($\mathbf{F} = 1$

```
  switch{
    case $\underline{\text{mes}} \to \text{FromZ}(\text{UserMes}(\underline{\text{mes}}))$ :
      call($\mathbf{F}, \underline{\text{mes}}, \underline{\text{mes}}$)
      ret$_{\mathbf{1}}$
  }
)
```

Program 6.4. DummyAdv $\overset{\text{def}}{=}$ build($\mathbf{Net} = 1$

```
  switch{
    case $\underline{\text{mes}} \to \text{AdvMes}(\underline{\text{mes}})$ :
      call($\mathbf{Net}, \underline{\text{mes}}, \underline{\text{mes}}$)
      ret$_{\mathbf{1}}$
  }
)
```

6.1 Dummy adversary

Theorem 6. *For any polynomial operators Z and A, there exists polynomial operator Z_A such that for any polynomial iterative automaton N*

$$Z(\text{Exec}(A, N)) \simeq_p Z_A(\text{Exec}(\text{DummyAdv}, N))$$

Proof. Let $Z_A = \text{AdvA}$.

$$Z(\text{Exec}(A, N)) =_A \text{AdvZ}(Z, A)(\text{Exec}(\text{DummyAdv}, N)).$$

This equivalence can be checked using the algorithm presented in Subsection 4.3.3. The procedure is the same as described in the proof of Theorem 5 (replace automatons A, N, and Z with special variable-like automatons). One can find the code in the file "ex-uc-dummy-adv.py".

We currently omit the proof that $\text{AdvZ}(Z, A)$ is a polynomial operator. Note that this fact is almost clear from the construction of AdvZ. □

Remark 6.1. *It may not be obvious why $=_A$ relation is stronger than $\simeq_p$, as the former relation have deterministic nature while the latter one is probabilistic. The reason is that if two iterative automatons are equivalent, they query randomness simultaneously, and output the same values having the same random string.*

Corollary 6.1. *P UC-realizes F using G functionality iff there exists a polynomial operator S such that*

$$\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(P, F)) \simeq_p \mathrm{Exec}(S, \mathrm{Net}(\mathrm{DummyP}, G)).$$

6.2 UC-theorem

Program 6.5. UComp $\overset{\text{def}}{=}$ build($\mathbf{P} = 1$, $\mathbf{Q} = 2$, $\mathbf{H} = 3$

```
  **switch**{
    **case** $\underline{\mathrm{mes}} \to \mathrm{FromA}(\mathrm{MesForP}(\underline{\mathrm{mes}}))$ :
      $\mathrm{ucCallP}(\mathbf{P}, \mathbf{Q}, \mathbf{H}, \underline{\mathrm{mes}}, \mathrm{FromA}(\underline{\mathrm{mes}}))$
    **case** $\underline{\mathrm{mes}} \to \mathrm{FromA}(\mathrm{MesForQ}(\underline{\mathrm{mes}}))$ :
      $\mathrm{ucCallQ}(\mathbf{Q}, \mathbf{H}, \underline{\mathrm{mes}}, \mathrm{FromA}(\underline{\mathrm{mes}}))$
    **case** $\underline{\mathrm{mes}} \to \mathrm{FromZ}(\mathrm{UserMes}(\underline{\mathrm{mes}}))$ :
      $\mathrm{ucCallP}(\mathbf{P}, \mathbf{Q}, \mathbf{H}, \underline{\mathrm{mes}}, \mathrm{FromZ}(\mathrm{UserMes}(\underline{\mathrm{mes}})))$
  }
  $\mathbf{ret_3}$
)
```

Program 6.6. ucCallQ($\mathbf{Q}, \mathbf{H}, g, s$) $\overset{\text{def}}{=}$

```
    $\mathrm{callCbk}(\mathbf{Q}, g, s, (m) \longrightarrow$
      $\mathrm{call}(\mathbf{H}, m, m)$
    )
```

Program 6.7. ucCallP($\mathbf{P}, \mathbf{Q}, \mathbf{H}, g, s$) $\overset{\text{def}}{=}$

```
    $\mathrm{callCbk}(\mathbf{P}, g, s, (m) \longrightarrow$
      $\mathrm{ucCallQ}(\mathbf{Q}, \mathbf{H}, m, \mathrm{FromZ}(\mathrm{UserMes}(m)))$
    )
```

Theorem 7. *Let P UC-realizes F using G functionality, and Q UC-realizes G using H functionality. Then* UComp(P, Q) *UC-realizes F using H functionality.*

Proof. From the Corollary 6.1 and the conditions of the theorem we have two polynomial operators SP and SQ such that

$$\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(Q, H)) \simeq_p \mathrm{Exec}(SQ, \mathrm{Net}(\mathrm{DummyP}, G)),$$

$$\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(P, G)) \simeq_p \mathrm{Exec}(SP, \mathrm{Net}(\mathrm{DummyP}, F)).$$

There exists iterative automatons T1, T2, and SPQ (see below for program specification) such that

$$\begin{aligned}
\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(\mathrm{UComp}(P, Q), H)) &=_A \mathrm{T1}(P)(\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(Q, H))) && (1)\\
\mathrm{T1}(P)(\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(Q, H))) &\simeq_p \mathrm{T1}(P)(\mathrm{Exec}(SQ, \mathrm{Net}(\mathrm{DummyP}, G))) && (2)\\
\mathrm{T1}(P)(\mathrm{Exec}(SQ, \mathrm{Net}(\mathrm{DummyP}, G))) &=_A \mathrm{T2}(SQ)(\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(P, G))) && (3)\\
\mathrm{T2}(SQ)(\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(P, G))) &\simeq_p T2(SQ)(\mathrm{Exec}(SP, \mathrm{Net}(\mathrm{DummyP}, F))) && (4)\\
T2(SQ)(\mathrm{Exec}(SP, \mathrm{Net}(\mathrm{DummyP}, F))) &=_A \mathrm{Exec}(\mathrm{SPQ}(SP, SQ), \mathrm{Net}(\mathrm{DummyP}, F)) && (5)
\end{aligned}$$

Equations (1), (3), and (5) are verifiable by the algorithm presented in Subsection 4.3.3. The procedure is the same as described in the proof of Theorem 5. The code can be found in file "ex-uc-composability.py".

Polynomial equivalences (2), (4), and (4) follow from the Theorem 4. We only need to approve (proof is currently omitted) that all of the following constructions can be rewritten in the form $f(\cdot)$ where f is a polynomial operator: $\text{T1}(P)(\cdot)$, $\text{T2}(SQ)(\cdot)$. We will use a similar technique in the following subsection without further explanations.

□

Program 6.8. T1 $\overset{\text{def}}{=}$ build($\mathbf{P} = 1$, $\mathbf{Exec} = 2$

```
  switch{
    case $\underline{\text{mes}} \to \text{UserMes}(\underline{\text{mes}})$ :
      $\text{callPT1}(\mathbf{P}, \mathbf{Exec}, \underline{\text{mes}}, \text{FromZ}(\text{UserMes}(\underline{\text{mes}})))$
    case $\underline{\text{mes}} \to \text{AdvMes}(\text{ToP}(\text{MesForP}(\underline{\text{mes}})))$ :
      $\text{callPT1}(\mathbf{P}, \mathbf{Exec}, \underline{\text{mes}}, \text{FromA}(\underline{\text{mes}}))$
    case $\underline{\text{mes}} \to \text{AdvMes}(\text{ToP}(\text{MesForQ}(\underline{\text{mes}})))$ :
      $\text{call}(\mathbf{Exec}, \underline{\text{mes}}, \text{AdvMes}(\text{ToP}(\underline{\text{mes}})))$
    case $\underline{\text{mes}} \to \text{AdvMes}(\text{ToF}(\underline{\text{mes}}))$ :
      $\text{call}(\mathbf{Exec}, \underline{\text{mes}}, \text{AdvMes}(\text{ToF}(\underline{\text{mes}})))$
  }
  $\mathbf{ret_2}$
)
```

Program 6.9. $\text{ucCallPT1}(\mathbf{P}, \mathbf{Exec}, g, s) \overset{\text{def}}{=}$

```
  $\text{callCbk}(g, \mathbf{P}, s, (m) \longrightarrow$
    $\text{call}(\mathbf{Exec}, m, \text{UserMes}(m))$
  )
```

Program 6.10. T2 $\overset{\text{def}}{=}$ build($\mathbf{SQ} = 1$, $\mathbf{Exec} = 2$

```
  switch{
    case $\underline{\text{mes}} \to \text{UserMes}(\underline{\text{mes}})$ :
      $\text{call}(\mathbf{Exec}, \underline{\text{mes}}, \text{UserMes}(\underline{\text{mes}}))$
    case $\underline{\text{mes}} \to \text{AdvMes}(\text{ToP}(\text{MesForP}(\underline{\text{mes}})))$ :
      $\text{call}(\mathbf{Exec}, \underline{\text{mes}}, \text{AdvMes}(\text{ToP}(\underline{\text{mes}})))$
    case $\underline{\text{mes}} \to \text{AdvMes}(\text{ToP}(\text{MesForQ}(\underline{\text{mes}})))$ :
      $\text{callSQT2}(\mathbf{SQ}, \mathbf{Exec}, \underline{\text{mes}}, \text{AdvMes}(\text{ToP}(\underline{\text{mes}})))$
    case $\underline{\text{mes}} \to \text{AdvMes}(\text{ToF}(\text{MesForQ}(\underline{\text{mes}})))$ :
      $\text{callSQT2}(\mathbf{SQ}, \mathbf{Exec}, \underline{\text{mes}}, \text{AdvMes}(\text{ToF}(\underline{\text{mes}})))$
  }
  $\mathbf{ret_2}$
)
```

Program 6.11. $\text{ucCallSQT2}(\mathbf{SQ}, \mathbf{Exec}, g, s) \overset{\text{def}}{=}$

```
  $\text{callCbk}(g, \mathbf{SQ}, s, (m) \longrightarrow$
    $m \to \text{ToF}(m)$
    $\text{call}(\mathbf{Exec}, m, \text{AdvMes}(\text{ToF}(m)))$
  )
```

Program 6.12. SPQ $\overset{\text{def}}{=}$ build($\mathbf{SP} = 1$, $\mathbf{SQ} = 2$, $\mathbf{Exec} = 3$

```
  switch{
    case $\underline{\text{mes}} \to \text{AdvMes}(\text{ToP}(\text{MesForP}(\underline{\text{mes}})))$ :
      $\text{ucCallSPQ}_{\text{P}}(\mathbf{SP}, \mathbf{Net}, \underline{\text{mes}}, \text{AdvMes}(\text{ToP}(\underline{\text{mes}})))$
    case $\underline{\text{mes}} \to \text{AdvMes}(\text{ToP}(\text{MesForQ}(\underline{\text{mes}})))$ :
      $\text{ucCallSPQ}_{\text{Q}}(\mathbf{SP}, \mathbf{SQ}, \mathbf{Net}, \underline{\text{mes}}, \text{AdvMes}(\text{ToP}(\underline{\text{mes}})))$
    case $\underline{\text{mes}} \to \text{AdvMes}(\text{ToF}(\underline{\text{mes}}))$ :
      $\text{ucCallSPQ}_{\text{Q}}(\mathbf{SP}, \mathbf{SQ}, \mathbf{Net}, \underline{\text{mes}}, \text{AdvMes}(\text{ToF}(\underline{\text{mes}})))$
  }
  $\mathbf{ret_3}$
)
```

Program 6.13. $\text{ucCallSPQ}_{\text{P}}(\mathbf{SP}, \mathbf{Net}, g, s) \overset{\text{def}}{=}$

```
callCbk($g, \mathbf{SP}, s, (m) \longrightarrow$
  $m \to \text{ToF}(m)$
  call($\mathbf{Net}, m, \text{AdvMes}(\text{ToF}(m))$)
)
```

Program 6.14. $\text{ucCallSPQ}_{\text{Q}}(\mathbf{SP}, \mathbf{SQ}, \mathbf{Net}, g, s) \overset{\text{def}}{=}$

```
callCbk($g, \mathbf{SQ}, s, (m) \longrightarrow$
  $m \to \text{ToF}(m)$
  $\text{ucCallSPQ}_{\text{P}}(\mathbf{SP}, \mathbf{Net}, m, \text{AdvMes}(\text{ToF}(m)))$
)
```

6.3 Authenticated channel (with honest participants)

The concept of this example is rooted in the work [Can04].

In this section we assume that Lib[12] includes an implementation for arbitrary polynomial EUF-CMA secure signature scheme. The interface follows.

- Lib(SignKeyGen, 0) → SignKeyPair(pk, sk) — generate a secret and a public key. Note that Lib is a family of distributions parametrized by a security parameter n; so, our model directly approve the standard key generation mechanism based on probabilistic polynomial time Turing machine.

- Lib(SignMakeSign, SignMakeSignArg(sk, m)) → sig — generate a signature sig for message m using secret key sk.

- Lib(SignVerify, SignVerifyArgs(m, pk, sig)) → b — verify a signature sig for message m using public key pk and return the result (one bit).

We utilize an indistinguishability-based definition of signature scheme security, namely the polynomial indistinguishability of two games

$$\text{SignGameIdeal} \overset{\text{def}}{=} \text{SignGame}(\mathit{Ideal}) \text{ and}$$

$$\text{SignGameReal} \overset{\text{def}}{=} \text{SignGame}(\mathit{Real}).$$

The games are based on the following scenario.

[12] Recall that a library Lib is present in definitions of polynomiality and polynomial indistinguishability.

1. Request a public key for *pid*. If it is the first request for *pid*, a key pair is generated and stored.

2. Get signature for a message m parametrized by *pid* and *sid*. For each pair (pid, sid) only one message could be signed. The signature *sig* for message SidMes(sid, m) on a secret key corresponding to *pid* is generated. The player gets *sig*.

3. Get a message corresponding to a pair (pid, sid). It is a technical element of games that is used to simplify the proof.

4. Verify a tuple (pid, sid, m', sig). If verification of message SidMes(sid, m') on key corresponding to *pid* passes, the player gets a message that was supposed to be signed. For SignGameReal it is the message m', for SignGameIdeal it is the message m that was provided in a sign request for (pid, sid); the game errors if there was no such requests.

Program 6.15. SignGame(*mode*) $\stackrel{\text{def}}{=}$ build(

```
switch{
  case $\underline{\text{mes}}$ → GameKeyGen($\underline{\text{pid}}$) :
    getKeyPairGen($\underline{\text{pk}}$, $\underline{\text{sk}}$, $\underline{\text{pid}}$)
    $\underline{\text{mes}}$ ← $\underline{\text{pk}}$
    ret
  case $\underline{\text{mes}}$ → GameSign($\underline{\text{pid}}$, $\underline{\text{sid}}$, $\underline{\text{m}}$) :
    getKeyPair($\underline{\text{pk}}$, $\underline{\text{sk}}$, $\underline{\text{pid}}$)
    memGet(MemSignedSid, $\underline{\text{signed}}$, MemSignedSidKey($\underline{\text{pid}}$, $\underline{\text{sid}}$))
    switch{
      case $\underline{\text{signed}}$ → MemSignedSidVal($\underline{\text{t}}$) :
        mes ← 0
        ret
      case $\underline{\text{signed}}$ → 0 :
    }
    lib(SignMakeSign, $\underline{\text{sig}}$, SignMakeSignArg($\underline{\text{sk}}$, SidMes($\underline{\text{sid}}$, $\underline{\text{m}}$)))
    memSet(MemSignedSid, MemSignedSidKey($\underline{\text{pid}}$, $\underline{\text{sid}}$), MemSignedSidVal($\underline{\text{m}}$))
    $\underline{\text{mes}}$ ← $\underline{\text{sig}}$
    ret
  case $\underline{\text{mes}}$ → GameSignedSid($\underline{\text{pid}}$, $\underline{\text{sid}}$, $\underline{\text{m}}$) :
    memGet(MemSignedSid, $\underline{\text{signed}}$, MemSignedSidKey($\underline{\text{pid}}$, $\underline{\text{sid}}$))
    $\underline{\text{mes}}$ ← $\underline{\text{signed}}$
    ret
  case $\underline{\text{mes}}$ → GameVerify($\underline{\text{pid}}$, $\underline{\text{sid}}$, $\underline{\text{m}}$, $\underline{\text{sig}}$) :
    getKeyPair($\underline{\text{pk}}$, $\underline{\text{sk}}$, $\underline{\text{pid}}$)
    lib(SignVerify, $\underline{\text{r}}$, SignVerifyArgs(SidMes($\underline{\text{sid}}$, $\underline{\text{m}}$), $\underline{\text{pk}}$, $\underline{\text{sig}}$))
```

```
        switch{
            case $\underline{r} \rightarrow 0$ :
                $\underline{mes} \leftarrow 0$
            case $\underline{r} \rightarrow 1$ :
                $\underline{mes} \leftarrow \text{VerifiedMes}(\underline{m})$
        }
        ret    if mode=Real, else run following lines
        switch{
            case $\underline{mes} \rightarrow \text{VerifiedMes}(\underline{m})$ :
                memGet(MemSignedSid, $\underline{m}$, MemSignedSidKey($\underline{pid}$, $\underline{sid}$))
                $\underline{m} \rightarrow \text{MemSignedSidVal}(\underline{m})$
                $\underline{mes} \leftarrow \text{VerifiedMes}(\underline{m})$
                ret
            case $\underline{mes} \rightarrow 0$ :
                $\underline{mes} \leftarrow 0$
                ret
        }
    }
)
```

Program 6.16. getKeyPair$(pk, sk, pid) \overset{\text{def}}{=}$

```
    mem($\underline{r}$, MemKeyPair, Get($pid$))
    switch{
        case $\underline{r} \rightarrow 0$ :
            ret
        case $\underline{r} \rightarrow \text{SignKeyPair}(pk, sk)$ :
    }
```

Program 6.17. getKeyPairGen$(pk, sk, pid) \overset{\text{def}}{=}$

```
    mem($\underline{r}$, MemKeyPair, Get($pid$))
    switch{
        case $\underline{r} \rightarrow 0$ :
            lib(SignKeyGen, $\underline{r}$, 0)
            memSet(MemKeyPair, $\underline{pid}$, $\underline{r}$)
            $\underline{r} \rightarrow \text{SignKeyPair}(pk, sk)$
        case $\underline{r} \rightarrow \text{SignKeyPair}(pk, sk)$ :
    }
```

Definition 6.2. *We say* Lib *incorporates an EUF-CMA secure signature scheme if*

$$\text{SignGameIdeal} \overset{w.r.t\ \text{Lib}}{\simeq_p} \text{SignGameReal}.$$

Theorem 8 (Informal)**.** *If* Lib *implements described signature scheme interface using a polynomial EUF-CMA secure signature scheme then it incorporates an EUF-CMA secure signature scheme.*

Remark 6.2. *Note that an adversary can implement the same signature scheme on their own, without using* Lib*. The cornerstone for proving this theorem is that* SignGameIdeal *and* SignGameReal *keep a secret key secret.*

We present a protocol that UC-realizes functionality of an authenticated channel ($\mathrm{F}_{\mathrm{auth}}$) using certification authority functionality (F_{CA}).

First, we ensure that an iterative automaton emulating the protocol does separate individual states of participants. We gain this by defining the authenticated channel protocol in the form $\mathrm{UCShell}(\mathrm{P}_{\mathrm{auth}})$, where $\mathrm{P}_{\mathrm{auth}}$ does not use memory calls and does not save states between invokes from Z or A (but it does store states while requesting F_{CA}). The shell UCShell provides $\mathrm{P}_{\mathrm{auth}}$ with a wrapper for memory calls such that each call key gets extended with a *pid* value. This technique explicitly separates memory zones for protocol participants. Also the shell ensures that *pid* is added to requests to F_{CA}.

Currently, we lack corruption mechanics in our model. It could be implemented on behalf of UCShell with some modifications in Net. One can find a partial implementation in code in core files “core/ucnet.py” and “core/ucshell.py”.

Let us describe what $\mathrm{F}_{\mathrm{auth}}$ does. A sender can register only one message per session. Anyone can request an access to such a message by providing a pair (pid, sid). After the adversary approves the request, the requestor can call $\mathrm{F}_{\mathrm{auth}}$ to get the message. So, one can think of it as some variant of the billboard functionality.

We provide listings of iterative automatons UCShell, $\mathrm{P}_{\mathrm{auth}}$, F_{CA} and $\mathrm{F}_{\mathrm{auth}}$ at the Appendix C. There are also other listings used in the proof.

We show here a typical line of actions a participant P_A should go through to hand over a message to a participant P_B. Recall that automatons do not send messages in our model, but they make requests and get answers.

1. $Z \to P_A$: $\mathrm{PidMes}(P_A, sid, \mathrm{SendReq}(m))))$;
2. $A \to P_A$: $\mathrm{PidMes}(P_A, sid, \mathrm{AuthRegister})$ — P_A generates a key pair (sk, pk) for the signature scheme;
3. $P_A \to \mathrm{F}_{\mathrm{CA}}$: $\mathrm{PidMes}(P_A, 0, \mathrm{RegisterReq}(pk))$;
4. $A \to \mathrm{F}_{\mathrm{CA}}$: $\mathrm{AdvRegisterGrant}(P_A)$;
5. $A \to P_A$: $\mathrm{PidMes}(P_A, sid, \mathrm{GetSendReq})$ — P_A returns $\mathrm{SignedMes}(m, sig)$;
6. $Z \to P_B$: $\mathrm{PidMes}(\mathrm{RetrieveReq}(P_B, sid, \mathrm{RetrieveReq}(P_A)))$;
7. $P_B \to \mathrm{F}_{\mathrm{CA}}$: $\mathrm{PidMes}(P_B, 0, \mathrm{RetrieveReq}(P_A)))$;
8. $A \to \mathrm{F}_{\mathrm{CA}}$: $\mathrm{AdvRetrieveGrant}(P_A, P_B)$;
9. $A \to P_B$: $\mathrm{PidMes}(P_B, sid, \mathrm{TransmitSignedMes}(P_A, m, sig))$
10. $P_B \to \mathrm{F}_{\mathrm{CA}}$: $\mathrm{PidMes}(P_B, 0, \mathrm{RegisterGet}(P_A))$ — F_{CA} returns pk; after that P_B verifies signature sig for message $\mathrm{SidMes}(sid, m)$ and stores m by key (sid, P_A) in local memory (UCShell extends this memory key to (P_B, sid, P_A));
11. $Z \to P_B$: $\mathrm{PidMes}(P_B, sid, \mathrm{SendGet}(P_A))$ — P_B gets m from memory and returns $\mathrm{Sent}(m)$.

Theorem 9. *Let* Lib *incorporate a EUF-CMA secure signature scheme. Then* $\mathrm{UCShell}(\mathrm{P}_{\mathrm{auth}})$ *UC-realizes* $\mathrm{F}_{\mathrm{auth}}$ *using* F_{CA}.

Proof. We'll show that

$$\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(\mathrm{UCShell}(\mathrm{P}_{\mathrm{auth}}), \mathrm{F}_{\mathrm{CA}})) \simeq_p \\ \mathrm{Exec}(\mathrm{lib2call}(\mathrm{GameSign}, \mathrm{SimX}_{\mathrm{auth}}, \mathrm{SignGameIdeal}), \mathrm{Net}(\mathrm{DummyP}, \mathrm{F}_{\mathrm{auth}})),$$

where $\mathrm{SimX_{auth}} \overset{\mathrm{def}}{=} \mathrm{SimBase_{auth}}(\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(\mathrm{UCShell}(\mathrm{PX_{auth}}), \mathrm{F_{CA}})))$. We use the function $\mathrm{lib2call}(n, f)$ that modifies automaton f to f' in such a way that $f'(g)$ would work as f but with all requests to the library functions with name n being redirected to g.

$$
\begin{aligned}
\mathrm{lib2call}(n, f) = \mathrm{applyGently}(& \\
&[(\underline{\mathrm{z}} \mapsto \mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{Out}(\underline{\mathrm{x}}))) \mapsto (\underline{\mathrm{z}} \mapsto \mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{Out}(\mathrm{Up}(\underline{\mathrm{x}}))))] \cup \\
&[(\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{Out}(\underline{\mathrm{x}})) \mapsto \underline{\mathrm{z}}) \mapsto (\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{Out}(\mathrm{Up}(\underline{\mathrm{x}}))) \mapsto \underline{\mathrm{z}})] \cup \\
&[(\underline{\mathrm{z}} \mapsto \mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{Lib}(n, \underline{\mathrm{x}}))) \mapsto (\underline{\mathrm{z}} \mapsto \mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{Out}(\mathrm{Down}(\underline{\mathrm{x}}))))] \cup \\
&[(\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{LibRet}(n, \underline{\mathrm{x}})) \mapsto \underline{\mathrm{z}}) \mapsto (\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{Out}(\mathrm{Down}(\underline{\mathrm{x}}))) \mapsto \underline{\mathrm{z}})], f),
\end{aligned}
$$

where $\mathrm{applyGently}(h, f) = h(f) \cup \{x \mapsto y \mid x \mapsto y \in f \text{ and } h(x \mapsto y) = \perp\}$.

The following sequence of equalities completes the proof. Code for checking all three automaton equivalences can be found in files “ex-uc-auth-1.py” and “ex-uc-auth-2.py”.

$$
\begin{aligned}
&\mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(\mathrm{UCShell}(\mathrm{P_{auth}}), \mathrm{F_{CA}})) =_A && (6)\\
&\mathrm{lib2call}(\mathrm{GameSign}, \mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(\mathrm{UCShell}(\mathrm{PX_{auth}}), \mathrm{F_{CA}})))(\mathrm{SignGameReal}) \simeq_p && (7)\\
&\mathrm{lib2call}(\mathrm{GameSign}, \mathrm{Exec}(\mathrm{DummyAdv}, \mathrm{Net}(\mathrm{UCShell}(\mathrm{PX_{auth}}), \mathrm{F_{CA}})))(\mathrm{SignGameIdeal}) =_A && (8)\\
&\mathrm{lib2call}(\mathrm{GameSign}, \mathrm{Exec}(\mathrm{SimX_{auth}}, \mathrm{Net}(\mathrm{DummyP}, \mathrm{FX_{auth}})))(\mathrm{SignGameIdeal}) =_A && (9)\\
&\mathrm{Exec}(\mathrm{lib2call}(\mathrm{GameSign}, \mathrm{SimX_{auth}})(\mathrm{SignGameIdeal}), \mathrm{Net}(\mathrm{DummyP}, \mathrm{F_{auth}}))
\end{aligned}
$$

□

7 Example 3. Hybrid Argument

Recall that we use carrying-style notation for iteration composition operation:

$$f(g_1, \dots, g_n) \overset{\text{def}}{=} f\, g_1\, \dots\, g_n.$$

Theorem 10 (Hybrid Argument)**.** *Let h be a polynomial operator (hybrid). Then*

$$\text{shift}(h) \simeq_p h \;\Rightarrow\; h(\text{const}(0)) \simeq_p h(\text{const}(1)).$$

Remark 7.1. *Note that for each polynomial operator h an iterative automaton* shift(h) *is a polynomial iterative automaton due to the construction of* shift*.*

Program 7.1. accBitDelta$(b, d, s) \overset{\text{def}}{=}$

```
switch{
    case $s \to 0$, $\underline{\text{acc}} \to 0$ :
        $d \leftarrow 0$
        $b \leftarrow 0$
    case $s \to 0$, $\underline{\text{acc}} \to 1$ :
        $d \leftarrow 0$
        $b \leftarrow 1$
    case $s \to 1$, $\underline{\text{acc}} \to 0$ :
        $d \leftarrow 1$
        $b \leftarrow 1$
    case $s \to 1$, $\underline{\text{acc}} \to 1$ :
        $d \leftarrow 0$
        $b \leftarrow 1$
}
```

accBitDelta$(b, d) \overset{\text{def}}{=}$ accBitDelta(b, d, b).

Program 7.2. shift $\overset{\text{def}}{=}$ build(**H** $= 1$, **Bit** $= 2$

```
    callCbk(H, $\underline{\text{mesA}}$, $\underline{\text{mes}}$, $(m)$ $\longrightarrow$
        call(Bit, $\underline{\text{mesBA}}$, $m$)
        accBitDelta($m$, $\underline{\text{d}}$, $\underline{\text{mesBA}}$)
        switch{
            case $\underline{\text{d}} \to 1$ :
                $m \leftarrow 0$
            case $\underline{\text{d}} \to 0$ :
        }
    )
    ret$_2$
)
```

Program 7.3. const$(k) \overset{\text{def}}{=}$ build(

```
    $\underline{\text{mes}} \leftarrow k$
    ret
)
```

We illustrate an application of this theorem in one example.

Program 7.4. memOnce$(i, b, a, v) \overset{\text{def}}{=}$

```
  $\text{memGet}(i, \underline{\text{cur}}, a)$
  **switch**{
    **case** $\underline{\text{cur}} \to 0$ :
      $b \leftarrow v$
      $\text{memSet}(i, a, \text{Fixed}(v))$
    **case** $\underline{\text{cur}} \to \text{Fixed}(\underline{\text{cur}})$ :
      $b \leftarrow \underline{\text{cur}}$
  }
```

Program 7.5. butOne $\overset{\text{def}}{=}$ build(

```
$\mathbf{MX} = 1$, $\mathbf{X} = 2$, $\mathbf{Bit} = 2$
  $\underline{\text{mes}} \to \text{AddrMes}(\underline{\text{addr}}, \underline{\text{a}})$
  $\text{call}(\mathbf{Bit}, \underline{\text{b}}, 0)$
  $\text{accBitDelta}(\underline{\text{b}}, \underline{\text{d}})$
  $\text{memOnce}(0, \underline{\text{d}}, \underline{\text{addr}}, \underline{\text{d}})$
  **switch**{
    **case** $d \to 0$ :
      $\text{call}(\mathbf{MX}, \underline{\text{mes}}, \text{AddrMes}(\underline{\text{addr}}, \underline{\text{mes}}))$
    **case** $d \to 1$ :
      $\text{call}(\mathbf{X}, \underline{\text{mes}}, \underline{\text{a}})$
  }
  $\mathbf{ret_3}$
)
```

Program 7.6. imitateButOne $\overset{\text{def}}{=}$ build(

```
$\mathbf{MX} = 1$, $\mathbf{Bit} = 2$
  $\underline{\text{mes}} \to \text{AddrMes}(\underline{\text{a}}, \underline{\text{m}})$
  $\text{call}(\mathbf{Bit}, \underline{\text{b}}, 0)$
  $\text{call}(\mathbf{MX}, \underline{\text{mes}}, \underline{\text{mes}})$
  $\mathbf{ret_2}$
)
```

Definition 7.1. *A polynomial iterative automaton M_A is an multiplexor for a polynomial iterative automaton A iff* imitateButOne $M_A \simeq_p$ butOne M_A A.

Theorem 11. *Let an iterative automaton M_A be an multiplexor for a polynomial iterative automaton A and let M_B be an multiplexor for B. Then*

$$A \simeq_p B \;\Rightarrow\; M_A \simeq_p M_B.$$

Proof. We base the proof on a hybrid argument and use the following hybrid.

Program 7.7. hybridM $\overset{\text{def}}{=}$ build($\mathbf{MA} = 1$, $\mathbf{MB} = 2$, $\mathbf{Bit} = 3$

```
    $\underline{\text{mes}} \to \text{AddrMes}(\underline{\text{addr}}, \underline{\text{a}})$
    $\text{call}(\mathbf{Bit}, \underline{\text{b}}, 0)$
    $\text{accBitDelta}(\underline{\text{b}}, \underline{\text{d}})$
    $\text{memOnce}(0, \underline{\text{b}}, \underline{\text{addr}}, \underline{\text{b}})$
    switch{
        case $\underline{\text{b}} \to 0$ :
            $\text{call}(\mathbf{MA}, \underline{\text{mes}}, \underline{\text{mes}})$
        case $\underline{\text{b}} \to 1$ :
            $\text{call}(\mathbf{MB}, \underline{\text{mes}}, \underline{\text{mes}})$
    }
    $\mathbf{ret_3}$
)
```

First, we can check that

$$\text{hybridM}(M_A, M_B)(\text{const}(0)) =_A M_A$$

and

$$\text{hybridM}(M_A, M_B)(\text{const}(1)) =_A M_B.$$

Now we need to show that

$$\text{shift}\,(\text{hybridM}(M_A, M_B)) \simeq_p \text{hybridM}(M_A, M_B).$$

We will define two intermediate iterative automatons hybridM1 and hybridM2 for which the following equivalence chain is satisfied and leads to the desired proposition.

$$\begin{aligned}
\text{shift}\,(\text{hybridM}(M_A, M_B)) &=_A \text{hybridM1}(\text{imitateButOne}(M_A), M_B) && (10)\\
\text{hybridM1}(\text{imitateButOne}(M_A), M_B) &\simeq_p \text{hybridM1}(\text{butOne}(M_A, A), M_B) && (11)\\
\text{hybridM1}(\text{butOne}(M_A, A), M_B) &=_A \text{hybridM2}(M_A, \text{butOne}(M_B, A)) && (12)\\
\text{hybridM2}(M_A, \text{butOne}(M_B, A)) &\simeq_p \text{hybridM2}(M_A, \text{butOne}(M_B, B)) && (13)\\
\text{hybridM2}(M_A, \text{butOne}(M_B, B)) &\simeq_p \text{hybridM2}(M_A, \text{imitateButOne}(M_B)) && (14)\\
\text{hybridM2}(M_A, \text{imitateButOne}(M_B)) &=_A \text{hybridM}(M_A, M_B) && (15)
\end{aligned}$$

Equations (10), (12), and (15) are verifiable by the algorithm presented in Subsection 4.3.3. The procedure is the same as described in the proof of Theorem 5. The code can be found in the file "ex-hybrid.py".

Polynomial equivalences (11), (13), and (14) follow from the condition of the current theorem and the Theorem 4. We only need to approve (proof is currently omitted) that all of the following constructions can be rewritten in the form $f(\cdot)$ where f is a polynomial operator: $\text{hybridM1}(\cdot, M_B)$, $\text{hybridM2}(M_A, \text{butOne}(M_B, \cdot))$, $\text{hybridM2}(M_A, \cdot)$. □

Program 7.8. hybridM1 $\overset{\text{def}}{=}$ build(

```
$\mathbf{IMA} = 1$, $\mathbf{MB} = 2$, $\mathbf{Bit} = 3$
  $\underline{\text{mes}} \to \text{AddrMes}(\underline{\text{addr}}, \underline{\text{a}})$
  $\text{call}(\mathbf{Bit}, \underline{\text{b}}, 0)$
  $\text{accBitDelta}(\underline{\text{b}}, \underline{\text{d}})$
  $\text{memOnce}(0, \underline{\text{b}}, \underline{\text{addr}}, \underline{\text{b}})$
  $\text{memOnce}(1, \underline{\text{d}}, \underline{\text{addr}}, \underline{\text{d}})$
  **switch**{
    **case** $\underline{\text{b}} \to 0$ :
      $\text{callCbk}(\mathbf{IMA}, \underline{\text{mes}}, \underline{\text{mes}}, (b) \longrightarrow$
        $b \leftarrow 0$
      )
    **case** $\underline{\text{b}} \to 1$ :
      **switch**{
        **case** $\underline{\text{d}} \to 0$ :
          $\text{call}(\mathbf{MB}, \underline{\text{mes}}, \underline{\text{mes}})$
        **case** $\underline{\text{d}} \to 0$ :
          $\text{callCbk}(\mathbf{IMA}, \underline{\text{mes}}, \underline{\text{mes}}, (b) \longrightarrow$
            $b \leftarrow 1$
          )
  }
  $\mathbf{ret_3}$
)
```

Program 7.9. hybridM2 $\overset{\text{def}}{=}$ build(

```
$\mathbf{MA} = 1$, $\mathbf{IMB} = 2$, $\mathbf{Bit} = 3$
  $\underline{\text{mes}} \to \text{AddrMes}(\underline{\text{addr}}, \underline{\text{a}})$
  $\text{call}(\mathbf{Bit}, \underline{\text{b}}, 0)$
  $\text{accBitDelta}(\underline{\text{b}}, \underline{\text{d}})$
  $\text{memOnce}(0, \underline{\text{b}}, \underline{\text{addr}}, \underline{\text{b}})$
  **switch**{
    **case** $\underline{\text{b}} \to 0$ :
      $\text{call}(\mathbf{MA}, \underline{\text{mes}}, \underline{\text{mes}})$
    **case** $\underline{\text{b}} \to 1$ :
      $\text{callCbk}(\mathbf{IMB}, \underline{\text{mes}}, \underline{\text{mes}}, (b) \longrightarrow$
        $b \leftarrow \underline{\text{d}}$
      )
  }
  $\mathbf{ret_3}$
)
```

A Computation Model: Details

Definition A.1 (Carthesian composition)**.** *Let f and g be iterative automatons. Their carthesian composition is an iterative automaton $f \times g = \mathrm{iter}_f \cup \mathrm{iter}_g \cup \mathrm{in}_f \cup \mathrm{in}_{0,f} \cup \mathrm{out}_f \cup \mathrm{in}_g \cup \mathrm{in}_{0,g} \cup \mathrm{out}_g \cup \mathrm{out}_f^{\mathrm{err}} \cup \mathrm{out}_g^{\mathrm{err}}$, where*

$$\begin{aligned}
\mathrm{iter}_f &= [(\underline{\mathrm{x}} \mapsto \underline{\mathrm{y}}) \mapsto \big(\mathrm{Iter}(\mathrm{PartF}(\underline{\mathrm{q}}, \underline{\mathrm{x}})) \mapsto (\mathrm{Iter}(\mathrm{PartF}(\underline{\mathrm{q}}, \underline{\mathrm{y}})))\big)]\,(f)\\
\mathrm{iter}_g &= [(\underline{\mathrm{x}} \mapsto \underline{\mathrm{y}}) \mapsto \big(\mathrm{Iter}(\mathrm{PartG}(\underline{\mathrm{q}}, \underline{\mathrm{x}})) \mapsto (\mathrm{Iter}(\mathrm{PartG}(\underline{\mathrm{q}}, \underline{\mathrm{y}})))\big)]\,(g)\\
\mathrm{in}_f &= [\mathrm{StateMesIn}(\mathrm{FGState}(\underline{\mathrm{s}}, \underline{\mathrm{d}}), \mathrm{MesF}(\underline{\mathrm{x}})) \mapsto \mathrm{Iter}(\mathrm{PartF}(\underline{\mathrm{d}}, \mathrm{StateMesIn}(\underline{\mathrm{s}}, \underline{\mathrm{x}})))]\\
\mathrm{in}_{0,f} &= [\mathrm{StateMesIn}(0, \mathrm{MesF}(\underline{\mathrm{x}})) \mapsto \mathrm{Iter}(\mathrm{PartF}(0, \mathrm{StateMesIn}(0, \underline{\mathrm{x}})))]\\
\mathrm{in}_g &= [\mathrm{StateMesIn}(\mathrm{FGState}(\underline{\mathrm{s}}, \underline{\mathrm{d}}), \mathrm{MesG}(\underline{\mathrm{x}})) \mapsto \mathrm{Iter}(\mathrm{PartG}(\underline{\mathrm{s}}, \mathrm{StateMesIn}(\underline{\mathrm{d}}, \underline{\mathrm{x}})))]\\
\mathrm{in}_{0,g} &= [\mathrm{StateMesIn}(0, \mathrm{MesG}(\underline{\mathrm{x}})) \mapsto \mathrm{Iter}(\mathrm{PartG}(0, \mathrm{StateMesIn}(0, \underline{\mathrm{x}})))]\\
\mathrm{out}_f &= [\mathrm{Iter}(\mathrm{PartF}(\underline{\mathrm{d}}, \mathrm{StateMesOut}(\underline{\mathrm{s}}, \underline{\mathrm{x}}))) \mapsto \mathrm{StateMesOut}(\mathrm{FGState}(\underline{\mathrm{s}}, \underline{\mathrm{d}}), \mathrm{MesF}(\underline{\mathrm{x}}))]\\
\mathrm{out}_g &= [\mathrm{Iter}(\mathrm{PartG}(\underline{\mathrm{s}}, \mathrm{StateMesOut}(\underline{\mathrm{d}}, \underline{\mathrm{x}}))) \mapsto \mathrm{StateMesOut}(\mathrm{FGState}(\underline{\mathrm{s}}, \underline{\mathrm{d}}), \mathrm{MesG}(\underline{\mathrm{x}}))]\\
\mathrm{out}_f^{\mathrm{err}} &= [\mathrm{Iter}(\mathrm{PartF}(\underline{\mathrm{d}}, \mathrm{err})) \mapsto \mathrm{err}]\\
\mathrm{out}_g^{\mathrm{err}} &= [\mathrm{Iter}(\mathrm{PartG}(\underline{\mathrm{d}}, \mathrm{err})) \mapsto \mathrm{err}].
\end{aligned}$$

Definition A.2 (Iterative composition)**.** *Let f and g be iterative automatons.*

$$f\,g \overset{\mathrm{def}}{=} \big(((f\times g)\circ(\mathrm{req}_{\mathrm{call}}\cup\mathrm{req}_{\mathrm{ret}}))\cup\mathrm{out}\cup\mathrm{out}_{\mathrm{LibF}}\cup\mathrm{out}_{\mathrm{LibG}}\cup\mathrm{out}_{\mathrm{MemF}}\cup\mathrm{out}_{\mathrm{MemG}}\cup\mathrm{out}_{\mathrm{err}}\big)\circ(f\times g)\circ(\mathrm{in}_0\cup\mathrm{in}\cup\mathrm{in}_{\mathrm{LibF}}\cup\mathrm{in}_{\mathrm{LibG}}),$$

where

$$\begin{aligned}
\mathrm{in}_0 &= [\mathrm{StateMesIn}(0, \mathrm{Out}(\underline{\mathrm{x}})) \mapsto \mathrm{StateMesIn}(0, \mathrm{MesF}(\mathrm{Out}(\mathrm{Up}(\underline{\mathrm{x}}))))]\\
\mathrm{in} &= [\mathrm{StateMesIn}(\mathrm{SOut}(\underline{\mathrm{s}}), \mathrm{Out}(\underline{\mathrm{x}})) \mapsto \mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{MesF}(\mathrm{Out}(\mathrm{Up}(\underline{\mathrm{x}}))))] \cup [\mathrm{Iter}(\underline{\mathrm{x}}) \mapsto \mathrm{Iter}(\underline{\mathrm{x}})]\\
\mathrm{in}_{\mathrm{LibF}} &= [\mathrm{StateMesIn}(\mathrm{SLibF}(\underline{\mathrm{s}}), \mathrm{Ret}(\underline{\mathrm{x}})) \mapsto \mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{MesF}(\mathrm{Ret}(\underline{\mathrm{x}})))]\\
\mathrm{in}_{\mathrm{LibG}} &= [\mathrm{StateMesIn}(\mathrm{SLibG}(\underline{\mathrm{s}}), \mathrm{Ret}(\underline{\mathrm{x}})) \mapsto \mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{MesG}(\mathrm{Ret}(\underline{\mathrm{x}})))]\\
\mathrm{out} &= [\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{MesF}(\mathrm{Out}(\mathrm{Up}(\underline{\mathrm{x}})))) \mapsto \mathrm{StateMesOut}(\mathrm{SOut}(\underline{\mathrm{s}}), \mathrm{Out}(\underline{\mathrm{x}}))] \cup [\mathrm{Iter}(\underline{\mathrm{x}}) \mapsto \mathrm{Iter}(\underline{\mathrm{x}})]\\
\mathrm{req}_{\mathrm{call}} &= [\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{MesF}(\mathrm{Out}(\mathrm{Down}(\underline{\mathrm{x}})))) \mapsto \mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{MesG}(\mathrm{Out}(\underline{\mathrm{x}})))]\\
\mathrm{req}_{\mathrm{ret}} &= [\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{MesG}(\mathrm{Out}(\underline{\mathrm{x}}))) \mapsto \mathrm{StateMesIn}(\underline{\mathrm{s}}, \mathrm{MesF}(\mathrm{Out}(\mathrm{Down}(\underline{\mathrm{x}}))))]\\
\mathrm{out}_{\mathrm{LibF}} &= [\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{MesF}(\mathrm{Lib}(\underline{\mathrm{k}}, \underline{\mathrm{x}}))) \mapsto \mathrm{StateMesOut}(\mathrm{SLibF}(\underline{\mathrm{s}}), \mathrm{Lib}(\underline{\mathrm{k}}, \underline{\mathrm{x}}))]\\
\mathrm{out}_{\mathrm{LibG}} &= [\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{MesG}(\mathrm{Lib}(\underline{\mathrm{k}}, \underline{\mathrm{x}}))) \mapsto \mathrm{StateMesOut}(\mathrm{SLibG}(\underline{\mathrm{s}}), \mathrm{Lib}(\underline{\mathrm{k}}, \underline{\mathrm{x}}))]\\
\mathrm{out}_{\mathrm{MemF}} &= [\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{MesF}(\mathrm{Mem}(\underline{\mathrm{k}}, \underline{\mathrm{x}}))) \mapsto \mathrm{StateMesOut}(\mathrm{SLibF}(\underline{\mathrm{s}}), \mathrm{Mem}(\mathrm{Fork}(\mathrm{F}, \underline{\mathrm{k}}), \underline{\mathrm{x}}))]\\
\mathrm{out}_{\mathrm{MemG}} &= [\mathrm{StateMesOut}(\underline{\mathrm{s}}, \mathrm{MesG}(\mathrm{Mem}(\underline{\mathrm{k}}, \underline{\mathrm{x}}))) \mapsto \mathrm{StateMesOut}(\mathrm{SLibG}(\underline{\mathrm{s}}), \mathrm{Mem}(\mathrm{Fork}(\mathrm{G}, \underline{\mathrm{k}}), \underline{\mathrm{x}}))]\\
\mathrm{out}_{\mathrm{err}} &= [\mathrm{err} \mapsto \mathrm{err}]
\end{aligned}$$

Remark A.1. *It may be not obvious why $(f \times g) \circ (\mathrm{req}_{\mathrm{call}} \cup \mathrm{req}_{\mathrm{ret}})$ part of the $f\,g$ definition do not require in-out processing similar to another $f \times g$ part at the end of the formula. The reason is that this construction produces only* $\mathrm{Iter}(\cdot)$*-type output due to the structure of $f \times g$.*

Definition A.3 (Linking a library)**.** *Let f and g be iterative automatons. We denote by $f_{\mathrm{lib}\leftarrow g}$ a modified verison of iterative automaton f where all library subcalls are proceeded by iterative automaton g.*

$$f_{\mathrm{lib}\leftarrow g} \overset{\mathrm{def}}{=} \big(((f \times g) \circ (\mathrm{req}_{\mathrm{call}} \cup \mathrm{req}_{\mathrm{ret}})) \cup \mathrm{out} \cup \mathrm{out}_{\mathrm{mem}} \cup \mathrm{out}_{\mathrm{err}}\big) \circ (f \times g) \circ (\mathrm{in} \cup \mathrm{in}_0 \cup \mathrm{in}_{\mathrm{lib}}),$$

where

$$\begin{aligned}
\mathrm{in}_0 &= [\mathrm{StateMesIn}(0, \mathrm{Out}(\underline{x})) \mapsto \mathrm{StateMesIn}(\mathrm{FState}(\underline{s}), \mathrm{MesF}(\mathrm{Out}(\underline{x})))] \cup [\mathrm{Iter}(\underline{x}) \mapsto \mathrm{Iter}(\underline{x})]\\
\mathrm{in} &= [\mathrm{StateMesIn}(\mathrm{FState}(\underline{s}), \mathrm{Out}(\underline{x})) \mapsto \mathrm{StateMesIn}(\mathrm{FState}(\underline{s}), \mathrm{MesF}(\mathrm{Out}(\underline{x})))] \cup [\mathrm{Iter}(\underline{x}) \mapsto \mathrm{Iter}(\underline{x})]\\
\mathrm{in}_{\mathrm{lib}} &= [\mathrm{StateMesIn}(\mathrm{FState}(\underline{s}), \mathrm{LibRet}(\underline{k}, \underline{x})) \mapsto \mathrm{StateMesIn}(\mathrm{FState}(\underline{s}), \mathrm{MesF}(\mathrm{LibRet}(\underline{k}, \underline{x})))]\\
\mathrm{in}_{\mathrm{mem}} &= [\mathrm{StateMesIn}(\mathrm{FState}(\underline{s}), \mathrm{MemRet}(\underline{x})) \mapsto \mathrm{StateMesIn}(\mathrm{FState}(\underline{s}), \mathrm{MesF}(\mathrm{MemRet}(\underline{x})))]\\
\mathrm{req}_{\mathrm{call}} &= [\mathrm{StateMesOut}(\mathrm{FState}(\underline{s}), \mathrm{MesF}(\mathrm{Lib}(\underline{k}, \underline{x}))) \mapsto \mathrm{StateMesIn}(\mathrm{LibState}(\underline{k}, \underline{s}), \mathrm{MesG}(\mathrm{Lib}(\underline{k}, \underline{x})))]\\
\mathrm{req}_{\mathrm{ret}} &= [\mathrm{StateMesOut}(\mathrm{LibState}(\underline{k}, \underline{s}), \mathrm{MesG}(\underline{x})) \mapsto \mathrm{StateMesIn}(\mathrm{FState}(\underline{s}), \mathrm{MesF}(\mathrm{LibRet}(\underline{k}, \underline{x})))]\\
\mathrm{out} &= [\mathrm{StateMesOut}(\mathrm{FState}(\underline{s}), \mathrm{MesF}(\mathrm{Out}(\underline{x}))) \mapsto \mathrm{StateMesOut}(\mathrm{FState}(\underline{s}), \mathrm{Out}(\underline{x}))] \cup [\mathrm{Iter}(\underline{x}) \mapsto \mathrm{Iter}(\underline{x})]\\
\mathrm{out}_{\mathrm{mem}} &= [\mathrm{StateMesOut}(\mathrm{FState}(\underline{s}), \mathrm{MesF}(\mathrm{Mem}(\underline{k}, \underline{x}))) \mapsto \mathrm{StateMesOut}(\mathrm{FState}(\underline{s}), \mathrm{Mem}(\underline{k}, \underline{x}))]\\
\mathrm{out}_{\mathrm{err}} &= [\mathrm{err} \mapsto \mathrm{err}]
\end{aligned}$$

A.1 Memory

We say that an automaton implements the memory functionality if the following condition is satisfied for every input sequence $(x_1, \ldots)$. Let x_i be of the form $\mathrm{Mem}(t, \mathrm{Get}(a))$. Then $y_i = \{v_i\}$ and the value of v_i is determined by the content of the subsequence $(x_1, \ldots, x_{i-1})$. In the case there was an x_j of the form $\mathrm{Mem}(k, \mathrm{Put}(a, v))$ in this subsequence, the value of v_i must be equal to v from the latest x_j of this form. Otherwise, $v_i = 0$.

Definition A.4 (Memory storage implementation)**.**

$$\mathrm{memImpl}(\mathrm{StateMesIn}(s, \mathrm{Mem}(k, \mathrm{Get}(a)))) =$$

$$= \begin{cases} \mathrm{StateMesOut}(s, 0), & s \notin \{\mathrm{Store}(s_1, \ldots, \mathrm{KeyVal}(k_a, v), \ldots, s_n) \mid s_i, v \in \overline{\mathbb{U}}\};\\ (s, v), & s \in \{\mathrm{Store}(s_1, \ldots, \mathrm{KeyVal}(k_a, v), \ldots, s_n) \mid s_i \in \overline{\mathbb{U}}\},\ \textit{for some } v \in \overline{\mathbb{U}}. \end{cases}$$

$$\mathrm{memImpl}(\mathrm{StateMesIn}(s, \mathrm{Mem}(k, \mathrm{Put}(a, v)))) =$$

$$= \begin{cases} \mathrm{StateMesOut}(\mathrm{Store}(\mathrm{KeyVal}(k_a, v)), 0), & s = 0;\\ \mathrm{StateMesOut}(\mathrm{Store}(s_1, \ldots, s_n, \mathrm{KeyVal}(k_a, v)), 0), & s = \mathrm{Store}(s_1, \ldots, s_n)\ \textit{and}\ s_i \notin [\mathrm{KeyVal}(k_a, \underline{x})];\\ \mathrm{StateMesOut}(\mathrm{Store}(s_1, \ldots, s_{i-1}, \mathrm{KeyVal}(k_a, v), s_{i+1}, \ldots, s_n), 0), & s = \mathrm{Store}(s_1, \ldots, s_n)\ \textit{and}\ s_i \in [\mathrm{KeyVal}(k_a, \underline{x})]. \end{cases}$$

Here we use short notation k_a for term $\mathrm{KeyPair}(k, a)$*.*

Definition A.5. *Let f and g be iterative automatons.*

$$\mathrm{mem}(f) \overset{\mathrm{def}}{=} \big(((f \times \mathrm{memImpl}) \circ (\mathrm{req}_{\mathrm{call}} \cup \mathrm{req}_{\mathrm{ret}})) \cup \mathrm{out} \cup \mathrm{out}_{\mathrm{lib}} \cup \mathrm{out}_{\mathrm{err}}\big) \circ (f \times \mathrm{memImpl}) \circ (\mathrm{in} \cup \mathrm{in}_{\mathrm{lib}}),$$

where

$$\begin{aligned}
\mathrm{in} &= [\mathrm{StateMesIn}(\underline{s}, \mathrm{Out}(\underline{x})) \mapsto \mathrm{StateMesIn}(\underline{s}, \mathrm{MesF}(\mathrm{Out}(\underline{x})))] \cup [\mathrm{Iter}(\underline{x}) \mapsto \mathrm{Iter}(\underline{x})]\\
\mathrm{in}_{\mathrm{lib}} &= [\mathrm{StateMesIn}(\underline{s}, \mathrm{LibRet}(\underline{k}, \underline{x})) \mapsto \mathrm{StateMesIn}(\underline{s}, \mathrm{MesF}(\mathrm{LibRet}(\underline{k}, \underline{x})))]\\
\mathrm{req}_{\mathrm{call}} &= [\mathrm{StateMesOut}(\underline{s}, \mathrm{MesF}(\mathrm{Mem}(\underline{k}, \underline{x}))) \mapsto \mathrm{StateMesIn}(\underline{s}, \mathrm{MesG}(\mathrm{Mem}(\underline{k}, \underline{x})))]\\
\mathrm{req}_{\mathrm{ret}} &= [\mathrm{StateMesOut}(\underline{s}, \mathrm{MesG}(\underline{x})) \mapsto \mathrm{StateMesIn}(\underline{s}, \mathrm{MesF}(\mathrm{Ret}(\underline{x})))]\\
\mathrm{out} &= [\mathrm{StateMesOut}(\underline{s}, \mathrm{MesF}(\mathrm{Out}(\underline{x}))) \mapsto \mathrm{StateMesOut}(\underline{s}, \mathrm{Out}(\underline{x}))] \cup [\mathrm{Iter}(\underline{x}) \mapsto \mathrm{Iter}(\underline{x})]\\
\mathrm{out}_{\mathrm{lib}} &= [\mathrm{StateMesOut}(\underline{s}, \mathrm{MesF}(\mathrm{Lib}(\underline{k}, \underline{x}))) \mapsto \mathrm{StateMesOut}(\underline{s}, \mathrm{MesF}(\mathrm{Lib}(\underline{k}, \underline{x})))]\\
\mathrm{out}_{\mathrm{err}} &= [\mathrm{err} \mapsto \mathrm{err}]
\end{aligned}$$

B Programs

Definition B.1. *A variables list is a tuple $\vec{v} = (v_1, \ldots, v_n)$ of distinct variables from $\mathbb{V}$.*

A term $t \in \mathbb{U}$ is consistent with $\vec{v}$ if all variable from $\mathbb{V}_t$ are listed in $\vec{v}$.

Definition B.2. *Let $\vec{v} = (v_1, \ldots, v_n)$ be a variables list, $i \in \{1, \ldots, n\}$ and a term $t \in \mathbb{U}$ consistent with $\vec{v}$.*

- *A set operation $v_i \leftarrow t$ for variables list v is a mapping term*

$$\mathrm{VarsList}(v_1, \ldots, v_n) \mapsto \mathrm{VarsList}(v_1, \ldots, v'_i, \ldots, v_n),$$

where $v'_i = t$.

- *A parse operation $v_i \rightarrow t$ for variables list v is a mapping term*

$$\mathrm{VarsList}(v'_1, \ldots, v'_n) \mapsto \mathrm{VarsList}(v_1, \ldots, v_n),$$

where $v'_i = t$ and for $j \neq i$

$$v'_j = \begin{cases} v_j, & v_j \notin \mathbb{V}_t, \\ \hat{v}_j, & v_j \in \mathbb{V}_t, \end{cases}$$

where $\hat{v}_j$ are arbitrary distinct variables that do not occur in $\vec{v}$.

A serie of operations $(p_1, \ldots, p_n)$ is called correct if composition of its elements as term mappings is not an empty set:

$$[p_n] \circ \ldots \circ [p_1] = [a \mapsto b].$$

We denote the mapping term $a \mapsto b$ by $\mathrm{op}(p_1, \ldots, p_n)$.

The set of all operations is denoted by Op.

Definition B.3. *A program graph is a tuple $P = (G, n_1, \ldots, n_k, l)$, where $G = (V, E)$ is an oriented graph and other items in the tuple defines the graph's features:*

- *$n_1, \ldots, n_k \in V$ - a subset of nodes of the graph G (in-out points),*
- *$l : E \rightarrow \mathrm{Op}$ - a partially defined function that labels edges E.*

Nodes of the graph are identified with term constants, i. e. $V \subset \mathrm{Const}$.

A program graph is consistent with variables list $\vec{v}$ if all terms in edge labels of program graph are consistent with $\vec{v}$.

A serie of operations $\vec{p}$ connects nodes n and n' in a program graph P, $\vec{p} \in \mathrm{Ops}_P[n, n']$, if $\vec{p}$ is a correct serie of operations and there exists a path $(e_1, \ldots, e_m) \in E^m$ that starts at the node n, ends at the node n', and $\vec{p} = (l(e_{i_1}), \ldots, l(e_{i_m}))$, where e_{i_j} are labeled edges along the path.

Definition B.4. *Let $P = (G = (V, E), n_1, \ldots, n_k, l)$ be a program graph. We say that an iterative automaton* buildFromGraph(P) *is built from a program graph if it is a closure of the complete iterative automaton defined by IA-specification constructed by the following procedure. Prepare a variables list $\vec{v} = (v_1, \ldots, v_n)$ where first variable is fixed and equals $\underline{\mathrm{mes}}$ and others are all distinct variables occured in edge's labels of the graph listed in alphabetic order.*

$$\mathrm{buildFromGraph}(P) \overset{\mathrm{def}}{=} \Big(\bigcup_{i,j=1}^{k} \bigcup_{\vec{p} \in \mathrm{Ops}_P[n_i, n_j]} \mathrm{opath}_{i,j}(\vec{p}) \Big) \circ \mathrm{in},$$

Here we use the following notation.

- $\mathrm{in} = [\mathrm{StateMesIn}(0, \underline{\mathrm{x}}) \mapsto \mathrm{StateMesIn}(\mathrm{Prog}(c_1, \mathrm{VarsList}(0, \dots, 0), \underline{\mathrm{x}}))] \cup [x \mapsto x]$,
- $\mathrm{opath}_{i,j}(\vec{p}) = [\mathrm{StateMesIn}(\mathrm{ProgramState}(c_i, a), v'_1) \mapsto \mathrm{StateMesOut}(\mathrm{ProgramState}(c_j, b), v''_1)]$, *where terms* a, b, v'_1 *and* v''_1 *are from the following the equality*

$$\mathrm{op}(\vec{p}) = a \mapsto b = \mathrm{VarsList}(v'_1, \dots, v'_n) \mapsto \mathrm{VarsList}(v''_1, \dots, v''_n).$$

Definition B.5. *A program listing is an element of* Listing*, a set defined by the following inductive principle.*

- $\mathrm{Listing}_1 = \mathrm{Op} \cup \{\mathbf{call}, \mathbf{back}, \mathbf{ret}\}$.
- $\mathrm{Listing}_{i+1} = \mathrm{Listing}_i \cup (\mathrm{Listing}_i^*)^*$.
- $\mathrm{Listing} = \bigcup_{i \in \mathbb{N}} \mathrm{Listing}_i$.

A program listing is consistent with variables list $\vec{v}$ *if all operations listed in the listing it are consistent with* $\vec{v}$.

Definition B.6. *Let* $P_1 = (G_1 = (V_1, E_1), n^1_1, \dots, n^1_{k_1}, l_1)$ *and* $P_2 = (G_2 = (V_2, E_2), n^2_1, \dots, n^2_{k_2}, l_2)$ *be program graphs. Their union* $P_1 \cup P_2$ *is a program graph*

$$(G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2), n^1_1, \dots, n^1_{k_1}, n^2_1, \dots, n^2_{k_2}, l_1 \cup l_2).$$

Definition B.7. *We define a function* compileToGraph *that compiles a program listing to a program graph as following*

$$\mathrm{compileToGraph}(p) \overset{\mathrm{def}}{=} \mathrm{cl}(0, 0, 0, 0, p) \cup ((\{0\}, \emptyset), 0, \emptyset),$$

where $\mathrm{cl}(b, e, r, p, m)$ *maps* $\mathrm{Const}^4 \times \mathrm{Listing}$ *to a program graph using the following recurrent principle.*

- $\mathrm{cl}(b, e, r, p, ((m^1_1, \dots, m^1_{k_1}), \dots, (m^t_1, \dots, m^t_{k_t}))) = ((V, E), \emptyset) \cup \bigcup_{i=1}^{t} \bigcup_{j=1}^{k_t} \mathrm{cl}(\mathrm{V}(p, i, j-1), \mathrm{V}(p, i, j), b, \mathrm{V}(p, i, j-1), m^i_j)$, *where*

$$V = \{b, e\} \cup \{\mathrm{V}(p, i, j) \mid 1 \le i \le t \ \wedge \ 0 \le j \le k_i\},$$

$$E = \{(b, \mathrm{Node}(p, i, 0)) \mid 1 \le i \le t\} \cup \{(\mathrm{Node}(p, i, k_i), e) \mid 1 \le i \le t\}.$$

- $\mathrm{cl}(b, e, r, p, o) = \{(\{b, e\}, \{(b, e)\}), \{((b, e), o)\}\}$ *if* $o \in \mathrm{Op}$.
- $\mathrm{cl}(b, e, r, p, \mathbf{call}) = \{(\{b, e\}, \{(b, e)\}), b, \emptyset\}$.
- $\mathrm{cl}(b, e, r, p, \mathbf{back}) = \{(\{b, r\}, \{(b, r)\}), \emptyset\}$.
- $\mathrm{cl}(b, e, r, p, \mathbf{ret}) = \{(\{b, 0\}, \{(b, 0)\}), \emptyset\}$.

Let p be a program listing. We use the notation $\mathrm{build}(p)$ *to denote the corresponding iterative automaton*

$$\mathrm{build}(p) \overset{\mathrm{def}}{=} \mathrm{mem}(\mathrm{buildFromGraph}(\mathrm{compileToGraph}(p))).$$

For convenience, we use the following short notations.

- A simple sequence of statements $((m_1, \dots, m_n))$ will be denoted by a simple sequence

m_1

$\dots$

m_k

- A switch statement $((m^1_1, \ldots, m^1_{k_1}, m'^1_1, \ldots, m'^1_{k'_1}), \ldots, (m^t_1, \ldots, m^t_{k_t}, m'^t_1, \ldots, m'^t_{k'_t}))$ will be denoted by

```
switch{
  case $m^1_1, \ldots, m^1_{k_1}$ :
    $m'^1_1$
    $\ldots$
    $m'^1_{k'_1}$
  $\ldots$
  case $m^t_1, \ldots, m^t_{k_1}$ :
    $m'^t_1$
    $\ldots$
    $m'^t_{k'_1}$
}
```

Every individual m in those notations may be an element of Op or may be a list of elements of Op. In the latter case we substitute the symbol m with this list.

In some cases we will preface a program with a list of constants for easy reading, e.g. $\mathbf{A} = 1, \mathbf{B} = 2, \cdots$. See Program B.6 for an example.

B.1 Standard programs

We will define some listings to be used as subprocedures in more sophisticated programs. In order to avoid a collision of variable names, we will use a convention that a context for a variable is bound to the program definition block. That is, we consider variables in different program blocks as different objects even if they have similar names.

Program B.1. $\mathrm{call}(i, g, p) \overset{\mathrm{def}}{=}$

```
$\underline{\mathrm{mes}} \leftarrow \mathrm{Out}(\overbrace{\mathrm{Up}(\ldots\mathrm{Up}(}^{i-1}\mathrm{Down}(p))\ldots))$
call
$\underline{\mathrm{mes}} \rightarrow \mathrm{Out}(\underbrace{\mathrm{Up}(\ldots\mathrm{Up}(}_{i-1}\mathrm{Down}(g))\ldots))$
```

Program B.2. $\mathbf{ret_n} \overset{\mathrm{def}}{=}$

```
$\underline{\mathrm{mes}} \leftarrow \mathrm{Out}(\overbrace{\mathrm{Up}(\ldots\mathrm{Up}(}^{n}p)\ldots))$
ret
```

Program B.3. $\mathrm{lib}(k, g, p) \overset{\mathrm{def}}{=}$

```
$\underline{\mathrm{mes}} \leftarrow \mathrm{Lib}(k, p)$
call
$\underline{\mathrm{mes}} \rightarrow \mathrm{LibRet}(k, g)$
```

Program B.4. $\text{memGet}(k, g, p) \overset{\text{def}}{=}$

```
$\underline{\text{mes}} \leftarrow \text{Mem}(k, \text{Get}(p))$
**call**
$\underline{\text{mes}} \rightarrow \text{MemRet}(g)$
```

Program B.5. $\text{memSet}(k, a, v) \overset{\text{def}}{=}$

```
$\underline{\text{mes}} \leftarrow \text{Mem}(k, \text{Put}(a, v))$
**call**
```

Program B.6. $\text{composition} \overset{\text{def}}{=} \text{build}(\ \mathbf{X} = 1,\ \mathbf{Y} = 2$

```
    $\text{call}(\mathbf{X}, \underline{\text{mesA}}, \text{Up}(\underline{\text{mes}}))$
    **switch**{
        **case** $\underline{\text{mesA}} \rightarrow \text{Down}(\underline{\text{mesAB}})$ :
            $\text{call}(\mathbf{Y}, \underline{\text{mesAB}}, \underline{\text{mesBA}})$
            $\text{call}(\mathbf{X}, \text{Down}(\underline{\text{mesBA}}), \underline{\text{mesA}})$
            **back**
        **case** $\underline{\text{mesA}} \rightarrow \text{Up}(\underline{\text{mes}})$ :
            $\mathbf{ret_2}$
    }
)
```

Proposition B.1. *$x(y) =_A \text{composition}(x, y)$ for all iterative automatons x and y.*

Program B.7. $\text{callCbk}(i, g, p, cbk) \overset{\text{def}}{=}$

```
$\text{call}(i, \underline{\text{mesA}}, \text{Up}(\underline{\text{p}}))$
**switch**{
    **case** $\underline{\text{mesA}} \rightarrow \text{Down}(\underline{\text{mesAB}})$ :
        $cbk(\underline{\text{mesAB}})$
        $\text{call}(i, \underline{\text{mesA}}, \text{Down}(\underline{\text{mesAB}}))$
        **back**
    **case** $\underline{\text{mesA}} \rightarrow \text{Up}(g)$ :
}
```

Program B.8. $\mathrm{callCbk}_2(i, g, p, cbk_1, cbk_2) \overset{\mathrm{def}}{=}$

```
$\mathrm{call}(i, \underline{\mathrm{mesA}}, \mathrm{Up}(\underline{\mathrm{p}}))$
**switch**{
    **case** $\underline{\mathrm{mesA}} \to \mathrm{Down}(\underline{\mathrm{mesAB}})$ :
        $cbk_1(\underline{\mathrm{mesAB}})$
        $\mathrm{call}(i, \underline{\mathrm{mesA}}, \mathrm{Down}(\underline{\mathrm{mesAB}}))$
        **back**
    **case** $\underline{\mathrm{mesA}} \to \mathrm{Up}(\mathrm{Down}(\underline{\mathrm{mesAC}}))$ :
        $cbk_2(\underline{\mathrm{mesAC}})$
        $\mathrm{call}(i, \underline{\mathrm{mesA}}, \mathrm{Up}(\mathrm{Down}(\underline{\mathrm{mesAC}})))$
        **back**
    **case** $\underline{\mathrm{mesA}} \to \mathrm{Up}(\mathrm{Up}(g))$ :
}
```

Program B.9. $\mathrm{composition}' \overset{\mathrm{def}}{=} \mathrm{build}(\ \mathbf{X} = 1,\ \mathbf{Y} = 2$

```
$\mathrm{callCbk}(\mathbf{X}, \underline{\mathrm{mes}}, \underline{\mathrm{mes}}, (m) \longrightarrow \mathrm{call}(\mathbf{Y}, m, m))$
$\mathbf{ret_2}$
)
```

Here and afterwards we use a notation $(x_1, \ldots, x_k) \longrightarrow$ *<some expression depending on* x_i*> to define the corresponding function.*

Proposition B.2. $\mathrm{composition}' =_A \mathrm{composition}$

C Listings for UC Example

Program C.1. $\mathrm{sendToP}(\mathbf{P}, \mathbf{F}, pid, sid, ret, snd) \overset{\text{def}}{=}$

```
  $\mathrm{callCbk2}(\mathbf{P}, ret, snd,$
    $(m) \longrightarrow$
      **switch**{
        **case** $m \to \mathrm{LocalGet}(\underline{\mathrm{ind}}, \underline{\mathrm{addr}})$
          $\mathrm{memGet}(\mathrm{LocalInd}(\underline{\mathrm{ind}}), m, \mathrm{PidAddr}(pid, \underline{\mathrm{addr}}))$
        **case** $m \to \mathrm{LocalSet}(\underline{\mathrm{ind}}, \underline{\mathrm{addr}}, \underline{\mathrm{val}})$
          $\mathrm{memSet}(\mathrm{LocalInd}(\underline{\mathrm{ind}}), \mathrm{PidAddr}(pid, \underline{\mathrm{addr}}), \underline{\mathrm{val}})$
          $m \leftarrow 0$
      }
    $(m) \longrightarrow$
      $m \to \mathrm{Parse}(\mathrm{SidMes}(\underline{\mathrm{sidF}}, \underline{\mathrm{mF}}))$
      $\mathrm{call}(\mathbf{F}, m, \mathrm{PidMes}(pid, \underline{\mathrm{sidF}}, \underline{\mathrm{mF}}))$
  )
```

Program C.2. $\mathrm{UCShell} \overset{\text{def}}{=} \mathrm{build}(\ \mathbf{P} = 1,\ \mathbf{F} = 2$

```
  **switch**{
    **case** $\underline{\mathrm{mes}} \to \mathrm{FromZ}(\mathrm{UserMes}(\mathrm{PidMes}(\underline{\mathrm{pid}}, \underline{\mathrm{sid}}, \underline{\mathrm{mes}}))) :$
      $\mathrm{sendToP}(\mathbf{P}, \mathbf{F}, \underline{\mathrm{pid}}, \underline{\mathrm{sid}}, \underline{\mathrm{mes}}, \mathrm{FromZ}(\mathrm{PidMes}(\underline{\mathrm{pid}}, \underline{\mathrm{sid}}, \underline{\mathrm{mes}})))$
    **case** $\underline{\mathrm{mes}} \to \mathrm{FromA}(\mathrm{PidMes}(\underline{\mathrm{pid}}, \underline{\mathrm{sid}}, \underline{\mathrm{mes}})) :$
      $\mathrm{sendToP}(\mathbf{P}, \mathbf{F}, \underline{\mathrm{pid}}, \underline{\mathrm{sid}}, \underline{\mathrm{mes}}, \mathrm{FromA}(\mathrm{PidMes}(\underline{\mathrm{pid}}, \underline{\mathrm{sid}}, \underline{\mathrm{mes}})))$
  }
  $\mathbf{ret_2}$
)
```

Program C.3. $\mathrm{F}_{\mathrm{auth}} \overset{\text{def}}{=} \mathrm{build}($

```
  **switch**{
    **case** $\underline{\mathrm{mes}} \to \mathrm{FromP}(\mathrm{PidMes}(\underline{\mathrm{pid}}, \underline{\mathrm{sid}}, \mathrm{SendReq}(\underline{\mathrm{m}}))) :$
      $\mathrm{memGet}(\mathrm{MemAuth}, \underline{\mathrm{mem}}, \mathrm{ExtIdentity}(\underline{\mathrm{sid}}, \underline{\mathrm{pid}}))$
      **switch**{
        **case** $\underline{\mathrm{mem}} \to 0 :$
        **case** $\underline{\mathrm{mem}} \to \mathrm{SendReqInfo}(\underline{\mathrm{m}}) :$
          $\underline{\mathrm{mes}} \leftarrow 0$
          **ret**
      }
      memSet(MemAuth, ExtIdentity($\underline{\text{sid}}$, $\underline{\text{pid}}$), SendReqInfo($\underline{\text{m}}$))
      $\underline{\text{mes}}$ ← 0
      **ret**
    **case** $\underline{\text{mes}}$ → FromA(AdvGetInfo($\underline{\text{sid}}$, $\underline{\text{pid}}$)) :
      memGet(MemAuth, $\underline{\text{mem}}$, ExtIdentity($\underline{\text{sid}}$, $\underline{\text{pid}}$))
      mes ← $\underline{\text{mem}}$
      **ret**
    **case** $\underline{\text{mes}}$ → FromA(AdvGrant($\underline{\text{sid}}$, $\underline{\text{pid}}$, $\underline{\text{pid2}}$)) :
      memGet(MemAuth, $\underline{\text{mem}}$, ExtIdentity($\underline{\text{sid}}$, $\underline{\text{pid}}$))
      **switch**{
        **case** $\underline{\text{mem}}$ → 0 :
          $\underline{\text{mes}}$ ← 0
          **ret**
        **case** $\underline{\text{mem}}$ → SendReqInfo($\underline{\text{m}}$) :
      }
      memSet(MemAuthGrant, AuthIdentity($\underline{\text{sid}}$, $\underline{\text{pid}}$, $\underline{\text{pid2}}$), SendGrantedInfo($\underline{\text{m}}$))
      $\underline{\text{mes}}$ ← 0
      **ret**
    **case** $\underline{\text{mes}}$ → FromP(PidMes($\underline{\text{pid2}}$, $\underline{\text{sid}}$, SendGet($\underline{\text{pid}}$))) :
      memGet(MemAuthGrant, $\underline{\text{memGrant}}$, AuthIdentity($\underline{\text{sid}}$, $\underline{\text{pid}}$, $\underline{\text{pid2}}$))
      **switch**{
        **case** $\underline{\text{memGrant}}$ → 0 :
          $\underline{\text{mes}}$ ← 0
          **ret**
        **case** $\underline{\text{memGrant}}$ → SendGrantedInfo($\underline{\text{m}}$) :
      }
      $\underline{\text{mes}}$ ← Sent($\underline{\text{m}}$)
      **ret**
  }
)
```

Program C.4. $\mathrm{FX}_{\mathrm{auth}} \overset{\mathrm{def}}{=}$ build(

```
  **switch**{
    **case** $\underline{\text{mes}}$ → FromP(PidMes($\underline{\text{pid}}$, $\underline{\text{sid}}$, SendReq($\underline{\text{m}}$))) :
      memGet(MemAuth, $\underline{\text{mem}}$, ExtIdentity($\underline{\text{sid}}$, $\underline{\text{pid}}$))
```

```
        switch{
            case $\underline{\text{mem}} \to 0$ :
            case $\underline{\text{mem}} \to \text{SendReqInfo}(\underline{\text{m}})$ :
                $\underline{\text{mes}} \leftarrow 0$
                ret
        }
        $\text{memSet}(\text{MemAuth}, \text{ExtIdentity}(\underline{\text{sid}}, \underline{\text{pid}}), \text{SendReqInfo}(\underline{\text{m}}))$
        $\underline{\text{mes}} \leftarrow 0$
        ret
    case $\underline{\text{mes}} \to \text{FromA}(\text{AdvGetInfo}(\underline{\text{sid}}, \underline{\text{pid}}))$ :
        $\text{memGet}(\text{MemAuth}, \underline{\text{mem}}, \text{ExtIdentity}(\underline{\text{sid}}, \underline{\text{pid}}))$
        $\text{mes} \leftarrow \underline{\text{mem}}$
        ret
    case $\underline{\text{mes}} \to \text{FromA}(\text{AdvGrant}(\underline{\text{sid}}, \underline{\text{pid}}, \underline{\text{pid2}}))$ :
        $\text{lib}(\text{SignLib}, \underline{\text{mem}}, \text{GameSignedSid}(\underline{\text{sid}}, \underline{\text{pid}}))$
        switch{
            case $\underline{\text{mem}} \to 0$ :
                $\underline{\text{mes}} \leftarrow 0$
                ret
            case $\underline{\text{mem}} \to \text{SendReqInfo}(\underline{\text{m}})$ :
        }
        $\text{memSet}(\text{MemAuthGrant}, \text{AuthIdentity}(\underline{\text{sid}}, \underline{\text{pid}}, \underline{\text{pid2}}), \text{SendGrantedInfo}(\underline{\text{m}}))$
        $\underline{\text{mes}} \leftarrow 0$
        ret
    case $\underline{\text{mes}} \to \text{FromP}(\text{PidMes}(\underline{\text{pid2}}, \underline{\text{sid}}, \text{SendGet}(\underline{\text{pid}})))$ :
        $\text{memGet}(\text{MemAuthGrant}, \underline{\text{memGrant}}, \text{AuthIdentity}(\underline{\text{sid}}, \underline{\text{pid}}, \underline{\text{pid2}}))$
        switch{
            case $\underline{\text{memGrant}} \to 0$ :
                $\underline{\text{mes}} \leftarrow 0$
                ret
            case $\underline{\text{memGrant}} \to \text{SendGrantedInfo}(\underline{\text{m}})$ :
        }
        $\underline{\text{mes}} \leftarrow \text{Sent}(\underline{\text{m}})$
        ret
}
```

```
)
```

Program C.5. $\mathrm{F_{CA}} \overset{\mathrm{def}}{=}$ build(

```
switch{
  case $\underline{\text{mes}} \to \text{FromP(PidMes(}\underline{\text{pid}}\text{, 0, RegisterReq(}\underline{\text{v}}\text{)))}$ :
    $\text{memGet(MemCAReg, }\underline{\text{mem}}\text{, }\underline{\text{pid}}\text{)}$
    switch{
      case $\underline{\text{mem}} \to 0$ :
        $\text{memSet(MemCAReg, }\underline{\text{pid}}\text{, RegReq(}\underline{\text{v}}\text{))}$
        $\underline{\text{mes}} \leftarrow 0$
        ret
      case $\underline{\text{mem}} \to \text{RegReq(}\underline{\text{x}}\text{)}$ :
        $\underline{\text{mes}} \leftarrow 0$
        ret
      case $\underline{\text{mem}} \to \text{Registered}$ :
        $\underline{\text{mes}} \leftarrow 0$
        ret
    }
  case $\underline{\text{mes}} \to \text{FromA(AdvRegisterGrant(}\underline{\text{pid}}\text{))}$ :
    $\text{memGet(MemCAReg, }\underline{\text{mem}}\text{, }\underline{\text{pid}}\text{)}$
    switch{
      case $\underline{\text{mem}} \to \text{RegReq(}\underline{\text{x}}\text{)}$ :
        $\text{memSet(MemCA, }\underline{\text{pid}}\text{, RegisteredVal(}\underline{\text{v}}\text{))}$
        $\text{memSet(MemCAReg, }\underline{\text{pid}}\text{, Registered)}$
        $\underline{\text{mes}} \leftarrow 0$
        ret
      case $\underline{\text{mem}} \to 0$ :
        $\underline{\text{mes}} \leftarrow 0$
        ret
      case $\underline{\text{mem}} \to \text{Registered}$ :
        $\underline{\text{mes}} \leftarrow 0$
        ret
    }
  case $\underline{\text{mes}} \to \text{FromP(PidMes(}\underline{\text{pid}}\text{, 0, RetrieveReq(}\underline{\text{pid2}}\text{)))}$ :
    $\text{memGet(MemCARet, }\underline{\text{mem}}\text{, RetReq(}\underline{\text{pid}}\text{, }\underline{\text{pid2}}\text{))}$
    switch{
        case $\underline{\text{mem}} \to 0$ :
            memSet(MemCARet, RetReq($\underline{\text{pid}}$, $\underline{\text{pid2}}$), RetrieveRequested)
            $\underline{\text{mes}} \leftarrow 0$
            ret
        case $\underline{\text{mem}} \to$ RetrieveRequested :
            $\underline{\text{mes}} \leftarrow 0$
            ret
        case $\underline{\text{mem}} \to$ RetrieveGranted :
            $\underline{\text{mes}} \leftarrow 0$
            ret
    }
case $\underline{\text{mes}} \to$ FromA(AdvRetrieveGrant($\underline{\text{pid}}$, $\underline{\text{pid2}}$)) :
    memGet(MemCARet, $\underline{\text{memRet}}$, RetReq($\underline{\text{pid}}$, $\underline{\text{pid2}}$))
    switch{
        case $\underline{\text{memRet}} \to 0$ :
            $\underline{\text{mes}} \leftarrow 0$
            ret
        case $\underline{\text{memRet}} \to$ RetrieveRequested :
            memSet(MemCARet, RetReq($\underline{\text{pid}}$, $\underline{\text{pid2}}$), RetrieveGranted)
            $\underline{\text{mes}} \leftarrow 0$
            ret
        case $\underline{\text{memRet}} \to$ RetrieveGranted :
            $\underline{\text{mes}} \leftarrow 0$
            ret
    }
case $\underline{\text{mes}} \to$ FromP(PidMes($\underline{\text{pid}}$, 0, RetrieveGet($\underline{\text{pid2}}$))) :
    memGet(MemCARet, $\underline{\text{memRet}}$, RetReq($\underline{\text{pid}}$, $\underline{\text{pid2}}$))
    switch{
        case $\underline{\text{memRet}} \to 0$ :
            $\underline{\text{mes}} \leftarrow 0$
            ret
        case $\underline{\text{memRet}} \to$ RetrieveRequested :
            $\underline{\text{mes}} \leftarrow 0$
            ret
        case $\underline{\text{memRet}} \to$ RetrieveGranted :
          memGet(MemCA, $\underline{\text{mem}}$, $\underline{\text{pid}}$)
          $\underline{\text{mes}} \leftarrow \underline{\text{mem}}$
          **ret**
        }
  }
)
```

Program C.6. $\mathrm{P}_{\text{auth}} \overset{\text{def}}{=}$ build(**LocMem** = 1, $\mathbf{F_{CA}}$ = 2

```
  **switch**{
    **case** $\underline{\text{mes}} \rightarrow$ FromA(PidMes($\underline{\text{pid}}$, $\underline{\text{sid}}$, AuthRegister)) :
      call(**LocMem**, $\underline{\text{mem}}$, LocalGet(MemReg, 0))
      **switch**{
        **case** $\underline{\text{mem}} \rightarrow 0$ :
        **case** $\underline{\text{mem}} \rightarrow$ Registered($\underline{\text{x}}$) :
          $\underline{\text{mes}} \leftarrow 0$
          $\mathbf{ret_2}$
      }
      lib(SignKeyGen, $\underline{\text{keys}}$, 0)
      $\underline{\text{keys}} \rightarrow$ SignKeyPair($\underline{\text{pk}}$, $\underline{\text{sk}}$)
      call($\mathbf{F_{CA}}$, 0, SidMes(0, RegisterReq($\underline{\text{pk}}$)))
      call(**LocMem**, 0, LocalSet(MemReg, 0, Registered($\underline{\text{sk}}$)))
      $\underline{\text{mes}} \leftarrow 0$
      $\mathbf{ret_2}$
    **case** $\underline{\text{mes}} \rightarrow$ FromZ(PidMes($\underline{\text{pid}}$, $\underline{\text{sid}}$, SendReq($\underline{\text{m}}$))) :
      call(**LocMem**, $\underline{\text{mem}}$, LocalGet(MemSend, $\underline{\text{sid}}$))
      **switch**{
        **case** $\underline{\text{mem}} \rightarrow 0$ :
        **case** $\underline{\text{mem}} \rightarrow$ SignedMes($\underline{\text{m}}$, $\underline{\text{sign}}$) :
          $\underline{\text{mes}} \leftarrow 0$
          $\mathbf{ret_2}$
        **case** $\underline{\text{mem}} \rightarrow$ NotSignedMes($\underline{\text{m}}$) :
          $\underline{\text{mes}} \leftarrow 0$
          $\mathbf{ret_2}$
      }
      call(**LocMem**, $\underline{\text{mem}}$, LocalSet(MemSend, $\underline{\text{sid}}$, NotSignedMes($\underline{\text{m}}$)))
      $\underline{\text{mes}} \leftarrow 0$
    ret$_2$
case $\underline{\text{mes}}$ → FromA(PidMes($\underline{\text{pid}}$, $\underline{\text{sid}}$, $\underline{\text{GetSendReq}}$)) :
    call($\mathbf{LocMem}$, $\underline{\text{mem}}$, LocalGet(MemSend, $\underline{\text{sid}}$))
    switch{
        case $\underline{\text{mem}}$ → SignedMes($\underline{\text{m}}$, $\underline{\text{sign}}$) :
            $\underline{\text{mes}}$ ← 0
            ret$_2$
        case $\underline{\text{mem}}$ → NotSignedMes($\underline{\text{m}}$) :
            call($\mathbf{LocMem}$, $\underline{\text{memReg}}$, LocalGet(MemReg, 0))
            switch{
                case $\underline{\text{memReg}}$ → 0 :
                    $\underline{\text{mes}}$ ← 0
                    ret$_2$
                case $\underline{\text{memReg}}$ → Registered($\underline{\text{sk}}$) :
            }
            lib(SignMakeSign, $\underline{\text{sign}}$, SignMakeSignArg($\underline{\text{sk}}$, SidMes($\underline{\text{sid}}$, $\underline{\text{m}}$)))
            call($\mathbf{LocMem}$, $\underline{\text{mem}}$, LocalSet(MemSend, $\underline{\text{sid}}$, SignedMes($\underline{\text{m}}$, $\underline{\text{sign}}$)))
    }
    mes ← SignedMes($\underline{\text{m}}$, $\underline{\text{sign}}$)
    ret$_2$
case $\underline{\text{mes}}$ → FromA(PidMes($\underline{\text{pid2}}$, $\underline{\text{sid}}$, RetrieveReq($\underline{\text{pid}}$))) :
    call($\mathbf{F_{CA}}$, 0, SidMes(0, RetrieveReq($\underline{\text{pid}}$)))
    mes ← 0
    ret$_2$
case $\underline{\text{mes}}$ → FromA(PidMes($\underline{\text{pid2}}$, $\underline{\text{sid}}$, TransmitSignedMes($\underline{\text{pid}}$, $\underline{\text{m}}$, $\underline{\text{sig}}$))) :
    call($\mathbf{F_{CA}}$, $\underline{\text{ca}}$, SidMes(0, RegisterGet($\underline{\text{pid}}$)))
    switch{
        case $\underline{\text{ca}}$ → 0 :
            $\underline{\text{mes}}$ ← 0
            ret$_2$
        case $\underline{\text{ca}}$ → Retrieved(0) :
            $\underline{\text{mes}}$ ← 0
            ret$_2$
        case $\underline{\text{ca}}$ → Retrieved(RegisteredVal($\underline{\text{pk}}$)) :
    }
```

```
      lib(SignVerify, $\underline{\mathrm{ver}}$, SignVerifyArgs(SidMes($\underline{\mathrm{sid}}$, $\underline{\mathrm{m}}$), $\underline{\mathrm{pk}}$, $\underline{\mathrm{sig}}$))
      switch{
        case $\underline{\mathrm{ver}} \to 1$ :
        case $\underline{\mathrm{ver}} \to 0$ :
          $\underline{\mathrm{mes}} \leftarrow 0$
          $\mathbf{ret_2}$
      }
      call($\mathbf{LocMem}$, $\underline{\mathrm{mem}}$, LocalSet(MemRecv, SidPid($\underline{\mathrm{sid}}$, $\underline{\mathrm{pid}}$), AuthedMes($\underline{\mathrm{m}}$)))
      $\underline{\mathrm{mes}} \leftarrow$ VerifiedMes($\underline{\mathrm{m}}$)
      $\mathbf{ret_2}$
    case $\underline{\mathrm{mes}} \to$ FromZ(PidMes($\underline{\mathrm{pid2}}$, $\underline{\mathrm{sid}}$, SendGet($\underline{\mathrm{pid}}$))) :
      call($\mathbf{LocMem}$, $\underline{\mathrm{mem}}$, LocalGet(MemRecv, SidPid($\underline{\mathrm{sid}}$, $\underline{\mathrm{pid}}$))
      switch{
        case $\underline{\mathrm{mem}} \to$ AuthedMes($\underline{\mathrm{m}}$) :
        case $\underline{\mathrm{mem}} \to 0$ :
          $\underline{\mathrm{mes}} \leftarrow 0$
          $\mathbf{ret_2}$
      }
      $\underline{\mathrm{mes}} \leftarrow$ Sent($\underline{\mathrm{m}}$)
      $\mathbf{ret_2}$
  }
)
```

Program C.7. $\mathrm{PX_{auth}} \overset{\mathrm{def}}{=}$ build($\mathbf{LocMem} = 1$, $\mathbf{F_{CA}} = 2$

```
  switch{
    case $\underline{\mathrm{mes}} \to$ FromA(PidMes($\underline{\mathrm{pid}}$, $\underline{\mathrm{sid}}$, AuthRegister)) :
      call($\mathbf{LocMem}$, $\underline{\mathrm{mem}}$, LocalGet(MemReg, 0))
      switch{
        case $\underline{\mathrm{mem}} \to 0$ :
        case $\underline{\mathrm{mem}} \to$ Registered($\underline{\mathrm{x}}$) :
          $\underline{\mathrm{mes}} \leftarrow 0$
          $\mathbf{ret_2}$
      }
      lib(SignLib, $\underline{\mathrm{pk}}$, GameKeyGen($\underline{\mathrm{pid}}$))
      call($\mathbf{F_{CA}}$, 0, SidMes(0, RegisterReq($\underline{\mathrm{pk}}$)))
      call($\mathbf{LocMem}$, 0, LocalSet(MemReg, 0, RegisteredX))
  $\underline{\text{mes}} \leftarrow 0$
  $\mathbf{ret_2}$
$\textbf{case } \underline{\text{mes}} \rightarrow \text{FromZ}(\text{PidMes}(\underline{\text{pid}}, \underline{\text{sid}}, \text{SendReq}(\underline{\text{m}}))) :$
  $\text{call}(\textbf{LocMem}, \underline{\text{mem}}, \text{LocalGet}(\text{MemSend}, \underline{\text{sid}}))$
  $\textbf{switch}\{$
    $\textbf{case } \underline{\text{mem}} \rightarrow 0 :$
    $\textbf{case } \underline{\text{mem}} \rightarrow \text{SignedMes}(\underline{\text{m}}, \underline{\text{sign}}) :$
      $\underline{\text{mes}} \leftarrow 0$
      $\mathbf{ret_2}$
    $\textbf{case } \underline{\text{mem}} \rightarrow \text{NotSignedMes}(\underline{\text{m}}) :$
      $\underline{\text{mes}} \leftarrow 0$
      $\mathbf{ret_2}$
  $\}$
  $\text{call}(\textbf{LocMem}, \underline{\text{mem}}, \text{LocalSet}(\text{MemSend}, \underline{\text{sid}}, \text{NotSignedMes}(\underline{\text{m}})))$
  $\underline{\text{mes}} \leftarrow 0$
  $\mathbf{ret_2}$
$\textbf{case } \underline{\text{mes}} \rightarrow \text{FromA}(\text{PidMes}(\underline{\text{pid}}, \underline{\text{sid}}, \underline{\text{GetSendReq}})) :$
  $\text{call}(\textbf{LocMem}, \underline{\text{mem}}, \text{LocalGet}(\text{MemSend}, \underline{\text{sid}}))$
  $\textbf{switch}\{$
    $\textbf{case } \underline{\text{mem}} \rightarrow \text{SignedMes}(\underline{\text{m}}, \underline{\text{sign}}) :$
      $\underline{\text{mes}} \leftarrow 0$
      $\mathbf{ret_2}$
    $\textbf{case } \underline{\text{mem}} \rightarrow \text{NotSignedMes}(\underline{\text{m}}) :$
      $\text{call}(\textbf{LocMem}, \underline{\text{memReg}}, \text{LocalGet}(\text{MemReg}, 0))$
      $\textbf{switch}\{$
        $\textbf{case } \underline{\text{memReg}} \rightarrow 0 :$
          $\underline{\text{mes}} \leftarrow 0$
          $\mathbf{ret_2}$
        $\textbf{case } \underline{\text{memReg}} \rightarrow \text{RegisteredX} :$
      $\}$
      $\text{lib}(\text{SignLib}, \underline{\text{sign}}, \text{GameSign}(\underline{\text{pid}}, \underline{\text{sid}}, \underline{\text{m}}))$
      $\text{call}(\textbf{LocMem}, \underline{\text{mem}}, \text{LocalSet}(\text{MemSend}, \underline{\text{sid}}, \text{SignedMes}(\underline{\text{m}}, \underline{\text{sign}})))$
  $\}$
  $\text{mes} \leftarrow \text{SignedMes}(\underline{\text{m}}, \underline{\text{sign}})$
  $\mathbf{ret_2}$
```

```
case $\underline{mes} \rightarrow \mathrm{FromA}(\mathrm{PidMes}(\underline{pid2}, \underline{sid}, \mathrm{RetrieveReq}(\underline{pid})))$ :
    $\mathrm{call}(\mathbf{F_{CA}}, 0, \mathrm{SidMes}(0, \mathrm{RetrieveReq}(\underline{pid})))$
    $\mathrm{mes} \leftarrow 0$
    $\mathbf{ret_2}$
case $\underline{mes} \rightarrow \mathrm{FromA}(\mathrm{PidMes}(\underline{pid2}, \underline{sid}, \mathrm{TransmitSignedMes}(\underline{pid}, \underline{m}, \underline{sig})))$ :
    $\mathrm{call}(\mathbf{F_{CA}}, \underline{ca}, \mathrm{SidMes}(0, \mathrm{RegisterGet}(\underline{pid})))$
    switch{
        case $\underline{ca} \rightarrow 0$ :
            $\underline{mes} \leftarrow 0$
            $\mathbf{ret_2}$
        case $\underline{ca} \rightarrow \mathrm{Retrieved}(0)$ :
            $\underline{mes} \leftarrow 0$
            $\mathbf{ret_2}$
        case $\underline{ca} \rightarrow \mathrm{Retrieved}(\mathrm{RegisteredVal}(\underline{pk}))$ :
    }
    $\mathrm{lib}(\mathrm{SignLib}, \underline{ver}, \mathrm{GameVerify}(\underline{pid}, \underline{sid}, \underline{m}, \underline{sig}))$
    switch{
        case $\underline{ver} \rightarrow \mathrm{VerifiedMes}(\underline{m})$ :
        case $\underline{ver} \rightarrow 0$ :
            $\underline{mes} \leftarrow 0$
            $\mathbf{ret_2}$
    }
    $\mathrm{call}(\mathbf{LocMem}, \underline{mem}, \mathrm{LocalSet}(\mathrm{MemRecv}, \mathrm{SidPid}(\underline{sid}, \underline{pid}), \mathrm{AuthedMes}(\underline{m})))$
    $\underline{mes} \leftarrow \mathrm{VerifiedMes}(\underline{m})$
    $\mathbf{ret_2}$
case $\underline{mes} \rightarrow \mathrm{FromZ}(\mathrm{PidMes}(\underline{pid2}, \underline{sid}, \mathrm{SendGet}(\underline{pid})))$ :
    $\mathrm{call}(\mathbf{LocMem}, \underline{mem}, \mathrm{LocalGet}(\mathrm{MemRecv}, \mathrm{SidPid}(\underline{sid}, \underline{pid}))$
    switch{
        case $\underline{mem} \rightarrow \mathrm{AuthedMes}(\underline{m})$ :
        case $\underline{mem} \rightarrow 0$ :
            $\underline{mes} \leftarrow 0$
            $\mathbf{ret_2}$
    }
    $\underline{mes} \leftarrow \mathrm{Sent}(\underline{m})$
    $\mathbf{ret_2}$
```

```
  }
)
```

Program C.8. $\mathrm{SimBase}_{\mathrm{auth}} \overset{\mathrm{def}}{=}$ build(**SimNet** = 1, **Net** = 2

```
  switch{
    case mes → AdvMes(ToP(PidMes(pid, sid, AuthRegister))) :
      call(SimNet, ret, FromA(ToP(PidMes(pid, sid, AuthRegister))))
      mes ← ret
      ret_2
    case mes → AdvMes(ToP(PidMes(pid, sid, GetSendReq))) :
      call(Net, info, ToF(AdvGetInfo(sid, pid)))
      switch{
        case info → 0 :
        case info → SendReqInfo(m) :
          call(SimNet, t, FromZ(UserMes(PidMes(pid, sid, SendReq(m)))))
      }
      call(SimNet, ret, FromA(ToP(PidMes(pid, sid, GetSendReq))))
      mes ← ret
      ret_2
    case mes → AdvMes(ToP(PidMes(pid, sid, RetrieveReq(pid)))) :
      call(SimNet, ret, FromA(ToP(PidMes(pid, sid, RetrieveReq(pid)))))
      mes ← 0
      ret_2
    case mes → AdvMes(ToP(PidMes(pid, sid, TransmitSignedMes(pid, m, sig)))) :
      call(SimNet, ret, FromA(ToP(PidMes(pid, sid, TransmitSignedMes(pid, m, sig)))))
      switch{
        case ret → 0 :
          mes ← 0
          ret_2
        case ret → VerifiedMes(m) :
          call(Net, info, ToF(AdvGrant(sid, pid, pid2)))
      }
      mes ← ret
      ret_2
    case mes → AdvMes(ToF(AdvRegisterGrant(pid))) :
      call(SimNet, ret, FromA(ToF(AdvRegisterGrant(pid))))
      $\underline{\text{mes}} \leftarrow \underline{\text{ret}}$
      $\mathbf{ret_2}$
    $\mathbf{case}$ $\underline{\text{mes}} \rightarrow \text{AdvMes}(\text{ToF}(\text{AdvRetrieveGrant}(\underline{\text{pid}}, \underline{\text{pid2}})))$ :
      $\text{call}(\mathbf{SimNet}, \underline{\text{ret}}, \text{FromA}(\text{ToF}(\text{AdvRetrieveGrant}(\underline{\text{pid}}, \underline{\text{pid2}}))))$
      $\underline{\text{mes}} \leftarrow \underline{\text{ret}}$
      $\mathbf{ret_2}$
  }
)
```

References

[AR02] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption)*. *J. Cryptol.*, 15(2):103–127, jan 2002.

[Bla05] Bruno Blanchet. A computationally sound automatic prover for cryptographic protocols. In *Workshop on the link between formal and computational models, Paris, France*, 2005.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `https://ia.cr/2000/067`.

[Can04] R. Canetti. Universally composable signature, certification, and authentication. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 219–233, 2004.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *Proceedings of the 4th Conference on Theory of Cryptography*, TCC'07, page 61–85, Berlin, Heidelberg, 2007. Springer-Verlag.

[CH06] Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of mutual authentication and key-exchange protocols. In *Proceedings of the Third Conference on Theory of Cryptography*, TCC'06, page 380–403, Berlin, Heidelberg, 2006. Springer-Verlag.

[CSV19] Ran Canetti, Alley Stoughton, and Mayank Varia. EasyUC: Using easycrypt to mechanize proofs of universally composable security. *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 167–16716, 2019.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.

[PW78] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.