Fundamenta Informaticae XX (2005) 1–21
IOS Press

Some Things Algorithms Cannot Do

Dean Rosenzweig
University of Zagreb
FSB, I. Lučića 5
10002 Zagreb, Croatia
dean@math.hr

Davor Runje
University of Zagreb
FSB, I. Lučića 5
10002 Zagreb, Croatia
davor.runje@fsb.hr

Abstract. A new, 'behavioral' theory of algorithms, intending to capture algorithms at their intended abstraction level, has been developed in this century in a series of papers by Y. Gurevich, A. Blass and others, motivated initially by the goal of establishing the ASM thesis. A viable theory of algorithms must have its limitative results, algorithms, however abstract, cannot do just anything. We establish some nonclassical limitative results for the behavioral theory:

- algorithms cannot distinguish some distinct states;
- algorithms cannot reach some existing states;
- algorithms cannot access some existing objects.

The algorithms studied are interactive, querying an environment, small–step, operating over different background classes. Since our primary motivation is abstract analysis of cryptographic algorithms, our examples come from this field – we believe however that the potential application field is much broader.

Introduction

Within the framework of the "behavioral theory of algorithms" [10, 2, 3, 4, 5], we look into some limitations of principle:

- no algorithm can distinguish some states;
- no algorithm can access some objects;
- no algorithm can reach some states.

The primary application area we have in mind is abstract cryptography—we feel that the behavioral framework is the right framework for its study, though we believe that the results are of broader interest.

States of an algorithm at a fixed abstraction level can be viewed as (first–order) structures of fixed vocabulary. What is the natural notion of equivalence of such states? One might argue it is isomorphism, claiming that everything relevant for algorithm execution in a state is expressed in terms of a class of structures isomorphic to it. After all, this is the intuition behind the postulates.

We show that isomorphism is too fine–grained for some applications, not relating states that are (in any practical way) behaviorally indistinguishable by algorithms. Following the rich tradition of seeing the objects indistinguishable by a class of algorithms as equal, we introduce the dynamic notion of indistinguishability by algorithms and show its equivalence with the static notion of similarity of structures. This equivalence survives generalization to the case of algorithms which interact with the environment within a step.

In order to make this paper reasonably self-contained, we also list several results which are not new, and which can be found scattered, sometimes inlined in proofs, sometimes without an explicit statement, in the behavioral theory literature. We attempt to attribute such results properly.

We thank Andreas Blass, Matko Botinčan and Yuri Gurevich for very helpful comments on an earlier version of the paper.

1. Non-Interactive Small–Step Algorithms

We take over many notions, notations and conventions on vocabularies, structures and sequential algorithms from [10] without further ado. In particular, we assume the following:

- all structures we consider are purely functional (algebras);
- all vocabularies have distinguished nullary symbols `true`, `false` and `undef`, with the interpretation of `true` distinct from interpretations of `false` and `undef` in all structures considered;
- all vocabularies have the binary function symbol $=$, interpreted as equality in all structures, as well as the usual Boolean connectives under their usual interpretations. If one of the arguments of a Boolean connective is not Boolean, the connective takes the default value of `false`.

Symbols `true`, `false`, `undef`, $=$ and the connectives are the *logical constants*.

Ground terms of vocabulary Υ are defined inductively in the usual way. All terms in this section are assumed to be ground.

1.1. Coincidence and Similarity

The following definitions are taken from [10].

Definition 1.1. Let Υ be a vocabulary and T a set of Υ-terms. Υ–structures X and Y are said to *coincide over* T, denoted with $X =_T Y$, if every term in T has the same value in X and Y.

A structure X induces an equivalence relation E_X on T: $(t_1, t_2) \in E_X$ if and only if $Val(t_1, X) = Val(t_2, X)$.

Definition 1.2. Let Υ be a vocabulary and T a set of Υ-terms. Υ–structures X and Y are *T–similar*, written as $X \sim_T Y$, if they induce the same equivalence relation over T.

Both relations are equivalence relations over Υ–structures for any choice of T. For any fixed set of terms T, coincidence is contained in similarity: if $X =_T Y$, then $X \sim_T Y$. Isomorphic structures are also similar: if $X \cong Y$, then $X \sim_T Y$.

When T is the set of *all* Υ–terms, we suppress it, and speak of coincident and similar structures.

1.2. Factorization

The following theorem reveals the connection between the equivalence relations on structures just mentioned. It is implicit in the proof of one of the key lemmas of [10]—it is actually proved there, although not explicitly stated.

Proposition 1.1. (Factorization)
Let X and Y be structures of a vocabulary Υ, T a set of Υ-terms. Then X, Y are T-similar if and only if there is a structure Z isomorphic to Y which coincides with X over T.

Proof:
One direction is obvious: both coincidence and isomorphism are contained in (transitive) similarity.

To see the other direction, it suffices to consider the special case when base sets of X and Y are disjoint (if not, replace Y below by an isomorphic copy disjoint from X).

We define a map ξ defined on Y as:

$$\xi(y) = \begin{cases} Val(t, X) & \text{if } y = Val(t, Y) \text{ for some } t \in T \\ y & \text{otherwise} \end{cases}$$

By similarity, ξ is well defined and injective on Y.

Since ξ is a total injection respecting the values of all terms, there is a structure Z isomorphic to Y whose base set is the codomain of ξ. For all Υ–terms t, we have: $Val(t, Z) = \xi(Val(t, Y))$. Notice that $\xi(Val(t, Y)) = Val(t, X)$ for all $t \in T$ by the definition of ξ. Hence, $Val(t, Z) = Val(t, X)$ for all $t \in T$, meaning that X and Z coincide over T. □

A useful way to apply factorization is the following technique: to show that X, Y are T-similar, tweak an isomorphic copy Z of Y so as to coincide with X over T while preserving isomorphism to Y. It follows immediately that similarity is the joint transitive closure of isomorphism and coincidence:

Corollary 1.1. Let T be a set of Υ–terms. Similarity $\sim_T$ is the smallest transitive (and equivalence) relation over Υ-structures containing both coincidence $=_T$ and isomorphism $\cong$.

1.3. Postulates

[10] defines a *sequential algorithm* as an object A satisfying a few postulates (see [10, 2] for extended discussion and motivation). For reference, we list the postulates as refactored in [3].

Postulate 1. (State)
Every algorithm A determines

- a nonempty collection $\mathcal{S}(A)$, called *states* of A;

- a nonempty collection $\mathcal{I}(A) \subseteq \mathcal{S}(A)$, called *initial states*; and
- a finite vocabulary Υ such that every $X \in \mathcal{S}(A)$ is an Υ-structure.

The base set of a state remains invariant under the operation of the algorithm; this is a technical choice of convenience. The difference of states $X, Y \in \mathcal{S}(A)$ with the same carrier can be explicitly represented as the *update set*

$$Y - X = \{(f, (a_1, \ldots, a_n), a_0) \mid f_Y(a_1, \ldots, a_n) = a_0 \neq f_X(a_1, \ldots, a_n),\ f \in \Upsilon_n\}.$$

The change the algorithm effects on a state X, turning it into successor state X', is then explicitly represented by the update set of A at X:

$$\Delta_A(X) = X' - X.$$

One–step transformation $X' = \tau_A(X)$ and the update set $\Delta_A(X)$ determine each other: we can write

$$\tau_A(X) = X + \Delta_A(X)$$

with the obvious definition of $+$, in the sense of 'unless overruled by'[1].

Postulate 2. (Updates)
For any state X the algorithm provides an update set $\Delta_A(X)$. If the update set is contradictory, the algorithm *fails*; otherwise it produces the *next state* $\tau_A(X)$. If there is a next state X', then it

- has the same base set as X,
- has $f_{X'}(\vec{a}) = b$ if $\langle f, \vec{a}, b\rangle \in \Delta_A(X)$, and
- otherwise interprets function symbols as in X.

States are *abstract*, in the sense that everything must be preserved by isomorphism: if your algorithm can distinguish red integers from green integers, then it is not about integers. This requirement can also be seen as prescriptive: everything relevant to the algorithm must be explicitly represented in the structure. Isomorphism extends to updates pointwise.

Postulate 3. (Isomorphism)
- Any structure isomorphic to a state is a state.
- Any structure isomorphic to an initial state is an initial state.
- If $i :\ X \cong Y$ is an isomorphism of states, then $i[\Delta_A(X)] = \Delta_A(Y)$.

The work performed by an algorithm in a step is bounded and defined by some finite text:

Postulate 4. (Bounded Exploration)
There is a finite set of terms T such that $\Delta_A(X) = \Delta_A(Y)$ whenever X and Y coincide over T.

[1] In the ASM literature [9] it is usual to speak of pairs $(f, (a_1, \ldots, a_n))$, where $f \in \Upsilon_n, a_i \in X$, as *locations* of X, in the 'structures–as–memories' metaphor. Then both the structure X and the update set $\Delta_A(X)$ can be seen as (partial) functions of locations to values, and the above usage of $+$ literally means overriding one partial function by another.

Such a set of terms is a *bounded exploration witness* for A. Notice that a bounded exploration witness is not uniquely determined, eg. any finite superset of a witness would do. Whenever we refer to a bounded exploration witness T, we assume that for a given algorithm we have chosen an arbitrary but fixed set of terms satisfying the postulate. We shall also call terms in T *critical* or *observable*.

Since many tend to understand a sequential algorithm as an object satisfying the other postulates, and something in general weaker then stringent Bounded Exploration, [3] suggest a confusion–preventing shift in terminology: an object satisfying the above postulates could be aptly called a *small–step algorithm*. We will adhere to that here.

An element $a \in X$ is *critical* at X if it is the value of a critical term, given an algorithm A and its fixed bounded exploration witness T. For reference, we list the following lemma, proved in [10]:

Lemma 1.1. If $(f, (a_1, \ldots, a_n), a_0) \in \Delta_A(X)$, then every $a_i,\ i = 0, \ldots n$, is critical at X.

Proof:
If some a_i is not critical, obtain a contradiction by constructing an isomorphic structure Y by replacing a_i by a fresh element: by Bounded Work, the algorithm should affect a non-element of Y, contradicting Updates. □

By the above lemma (and Bounded Exploration postulate), the update set of a small–step algorithm is (uniformly) finite at any state.

1.4. Next Value

The main result of this section is preservation of coincidence and similarity over the set of all terms by a step of a small–step algorithm, proved as consequences of the Next Value theorem: all elements representable by terms in the successor state to X were already so representable at X, uniformly with respect to similarity. We will also show how the Next Value theorem can be used to derive some known results like Linear Speedup.

Fix an algorithm A and its state X. By Lemma 1.1, every element of an update set in X is critical. For an arbitrary bounded exploration witness T and a term t, we can generate a larger set of terms by adding to T all instances of t with some subterms replaced with elements of T—this is a syntactic simulation of possible updates (not necessarily the most efficient one). The value of t in $\tau_A(X)$ must be a value of some term from the generated set in X. In general, for different states, *different terms* picked up from the generated set will have this property. However, if two states coincide over the larger set of terms, then *the same term* works for both states.

Let T be a set of terms and t a term of the same vocabulary. We define a set of terms T^t inductively over the structure of t as

$$T^{f(t_1,\ldots,t_n)} = T \cup \left\{ f(t'_1, \ldots, t'_n) \mid t'_i \in T^{t_i} \right\} \cup \bigcup \left\{ T^{t_i} \mid i = 1, \ldots n \right\}.$$

In the ground case of 0-ary f we have $T^f = T \cup \{f\}$. Obviously, if T is finite, T^t is finite as well.

Theorem 1.1. (Next Value)
Let A be a small–step algorithm, X its state, and T one of its exploration witnesses. Then for every term t of its vocabulary there is a term $\bigcirc_X^A t \in T^t$ such that

- $Val(\bigcirc_X^A t, X) = Val(t, \tau_A(X))$, moreover,
- whenever a state Y coincide with X over T^t we have $Val(t, \tau_A(Y)) = Val(t, \tau_A(X))$.

Proof:
We construct the term $\bigcirc_X^A t$ and prove the statements by induction on the structure of t. Suppose that $t = f(t_1, \ldots, t_n)$. By the induction hypothesis we have $Val(t_i, \tau_A(X)) = a_i$, there is $\bigcirc_X^A t_i \in T^{t_i}$ such that $Val(\bigcirc_X^A t_i, X) = Val(t_i, \tau_A(X))$, and whenever $Y =_{T^{t_i}} X$ then also $Val(t_i, \tau_A(Y)) = a_i$ for every $i = 1, \ldots, n$. Notice that $T^{t_i} \subseteq T^t$, allowing us to use the induction hypothesis as follows:

1. Assume $(f, (a_1, \ldots, a_n), a_0) \in \Delta_A(X)$ for some a_0. By Lemma 1.1, a_0 is critical in X and there is a term $\bigcirc_X^A t \in T$ such that $Val(\bigcirc_X^A t, X) = a_0 = Val(t, \tau_A(X))$.

   Suppose $Y =_{T^t} X$. Then $\Delta_A(Y) = \Delta_A(X)$ and thus $(f, (a_1, \ldots, a_n), a_0) \in \Delta_A(Y)$. We have

$$\begin{aligned} Val(t, \tau_A(Y)) &= f_{\tau_A(Y)}(Val(t_1, \tau_A(Y)) \ldots, Val(t_n, \tau_A(Y))) \\ &= f_{\tau_A(Y)}(a_1, \ldots, a_n) = a_0 = f_{\tau_A(X)}(a_1, \ldots, a_n) \\ &= Val(t, \tau_A(X)) \end{aligned}$$

2. Otherwise, we set $\bigcirc_X^A t = f(\bigcirc_X^A t_1, \ldots, \bigcirc_X^A t_n) \in T^t$, and we have

$$Val(t, \tau_A(X)) = f_{\tau_A(X)}(a_1, \ldots, a_n) = f_X(a_1, \ldots, a_n) = Val(\bigcirc_A^X t, X)$$

   Suppose $Y =_{T^t} X$. Then $(f, (a_1, \ldots, a_n), a_0) \notin \Delta_A(Y)$ for any a_0. Thus

$$\begin{aligned} Val(t, \tau_A(Y)) &= f_{\tau_A(Y)}(Val(t_1, \tau_A(Y)) \ldots, Val(t_n, \tau_A(Y))) \\ &= f_{\tau_A(Y)}(a_1, \ldots, a_n) = f_Y(a_1, \ldots, a_n) \\ &= f_Y(Val(\bigcirc_X^A t_1, Y) \ldots, Val(\bigcirc_X^A t_n, Y)) \\ &= Val(\bigcirc_X^A t, Y) = Val(\bigcirc_X^A t, X) \\ &= Val(t, \tau_A(X)) \end{aligned}$$

□

Remark 1.1. A more general variant of the Next Value theorem in the context of small-step ordinary interactive algorithms can be found in [5] as Lemma 8.8. In order to keep the paper reasonably self-contained, we state and prove the special case here. The proof of the special case is also considerably simpler. See also Theorem 2.1 in the next section for generalization to ordinary interactive small–step algorithms.

Corollary 1.2. (Preserving Coincidence)
Let A be a small–step algorithm and X and Y coincident states. Then $\tau_A(X)$ and $\tau_A(Y)$ coincide.

Theorem 1.2. Let A be a small-step algorithm and T its bounded exploration witness. If states X and Y are T^t–similar, then $Val(\bigcirc_X^A t, Y) = Val(t, \tau_A(Y))$.

Proof:
Use Factorization (Proposition 1.1), Abstract State postulate and Next State (Theorem 1.1). □

Corollary 1.3. (Preserving Similarity)
Let A be a small–step algorithm and X and Y similar states. Then $\tau_A(X)$ and $\tau_A(Y)$ are similar.

The following statement, quoted in [10] and proved for interactive algorithms in [5] (also proved by syntactic means in different places for different kinds of textual programs), states that whatever a small–step algorithm can do in two steps, could be done in one step by another small–step algorithm. By induction the same holds for any finite number of steps — the small steps can be enlarged by any fixed factor. We obtain it as a simple consequence of Next Value.

Proposition 1.2. (Linear Speedup)
Let A be a small–step algorithm, with associated $\mathcal{S}(A), \mathcal{I}(A)$ and τ_A. Then there is a small–step algorithm B, such that $\mathcal{S}(B) = \mathcal{S}(A), \mathcal{I}(B) = \mathcal{I}(A)$, and $\tau_B(X) = \tau_A(\tau_A(X))$ for all $X \in \mathcal{S}(B)$.

Proof:
It suffices to demonstrate a bounded exploration witness for B. Let T be a bounded exploration witness for A, and X and Y be its states. We have

$$\Delta_B(X) = \tau_A(\tau_A(X)) - X = \Delta_A(\tau_A(X)) \cup (\Delta_A(X) \setminus \Delta_A(\tau_A(X))).$$

If X and Y coincide over T, we have $\Delta_A(X) = \Delta_A(Y)$. If they also coincide over a finite set $T^T = \bigcup\{T^t \mid t \in T\}$ extending T, then, by Next Value theorem, $\tau_A(X)$ coincides with $\tau_A(Y)$ over T. Hence, $\Delta_A(\tau_A(X)) = \Delta_A(\tau_A(Y))$ and $\Delta_B(X) = \Delta_B(Y)$. Thus T^T is a bounded exploration witness for B. □

The similarity relation over a finite set of terms T partitions Υ–structures to finitely many equivalence classes — there is a finite set of structures $\{X_1, \ldots, X_n\}$ such that every structure is T–similar to some X_i. For each X_i there is a Boolean term φ^{X_i} such that φ^{X_i} holds in Y if and only if Y is T–similar to X_i.

This was the crucial observation behind the proof of the sequential thesis [10] – it allowed uniformization of local update sets into a finite program. It also allows us to uniformize the $\bigcirc^A_X t$ construction into a finite set of possible terms for all states, given an additional construct on terms.

Let *conditional terms* be terms closed under the ternary `if-then-else` construct, with the usual interpretation.

Corollary 1.4. Let A be a small–step algorithm and t a term of its vocabulary. Then there is a conditional term $\bigcirc^A t$ such that $Val(\bigcirc^A t, X) = Val(t, \tau_A(X))$ for every state X.

Remark 1.2. Using conditional terms is not a serious extension—it is easy (though somewhat tedious, in view of the number of cases) to prove that any ASM program written with conditional terms can be also equivalently rewritten without them, by pushing conditionals to rules.

Different versions of the next-value construction, restricted to Boolean terms (logical formulæ, for which the `if-then-else` construct is definable), and proved over textual programs, have been around in the literature in the form of a 'next-state' modality [7, 6, 12].

1.5. Indistinguishability, Accessibility and Reachability

This section introduces the main contribution of this paper — the notions of indistinguishability, accessibility and reachability and their properties—in the context of non–interactive small–step algorithms. However simple, these notions have not been studied in the literature (though related to the notions of *active* objects of [6] and *exposed* objects of [1], they are not the same). In subsequent sections we will extend these notions and prove the corresponding results for algorithms with intrastep interaction in general, and algorithms creating fresh objects over background structures in particular.

The notion of indistinguishability by a class of algorithms is a well known tool for analyzing behavioral equivalence of objects. The notion of indistinguishability by small–step algorithms, given here, is unashamedly influenced by similar notions widely used in process calculi and probabilistic complexity theory.

The intuition is that an algorithm can distinguish state X from state Y if it can determine in which of them it has executed a step. What does *to determine* mean here? Taking a behavioral view, we can require an algorithm to take different actions depending on whether it is in X or in Y, say by writing $\mathtt{true}_X$ into a specific location if it is in X and $\mathtt{false}_Y$ if it is in Y.

Definition 1.3. (Indistinguishability)
Let A be a small–step algorithm of the vocabulary Υ, whose states include X and Y. We say that A *distinguishes* X from Y if there is a Υ–term t taking the value $\mathtt{true}_X$ in $\tau_A(X)$, and not taking the value $\mathtt{true}_Y$ in $\tau_A(Y)$. Structures X and Y of the same vocabulary are *indistinguishable* by small-step algorithms if no such algorithm can distinguish them.

This is at first glance weaker than requiring of t to take the value of $\mathtt{false}_Y$ in $\tau_A(Y)$, but only at first glance: if t satisfies our requirement, then the term $t = \mathtt{true}$ will satisfy the seemingly stronger requirement. The wording of Indistinguishability definition has been chosen so as to work smoothly also in an interactive situation, where terms can have no value. In spite of the asymmetric wording, it is easy to verify the following

Corollary 1.5. Indistinguishability is an equivalence relation on structures of the same vocabulary.

The dynamic notion of indistinguishability coincides with the static notion of similarity:

Theorem 1.3. Structures X and Y of the same vocabulary Υ are indistinguishable by small–step algorithms if and only if they are similar.

Proof:
Suppose that X and Y are not similar. Then there are Υ-terms t_1 and t_2 having the same value in X and different values in Y. But then a do-nothing algorithm distinguishes them by term $t_1 = t_2$.

Now suppose that X and Y are similar and distinguishable by a term t taking the value $\mathtt{true}_X$ in $\tau_A(X)$ and not $\mathtt{true}_Y$ in $\tau_A(Y)$. Then $\tau_A(X)$ and $\tau_A(Y)$ are not similar, which is a contradiction by Corollary 1.3. □

By Corollary 1.3, similarity is equivalent to indistinguishability in any number of steps. An element of a structure can be, in the small-step case, accessible to an algorithm only if it is the value of some term.

Definition 1.4. (Accessibility)
An element a is *accessible* in a structure X of a vocabulary Υ if there is a Υ–term t such that $Val(t, X) = a$.

Remark 1.3. The reader familiar with logic should have in mind that we are speaking about indistinguishability *by algorithms*, and not about indistinguishability *by logic*: similar (indistinguishable) structures need not be elementarily equivalent. In all our examples of indistinguishable structures below it will be easy to find simple quantified sentences which distinguish them. But small-step algorithms are typically not capable of evaluating quantifiers over their states, unless such a capability is explicitly built in—if an algorithm explored states of unbounded size, the capability to evaluate quantifiers would contradict Bounded Work.

A straightforward consequence of Next Value is

Corollary 1.6. Let A be a small–step algorithm and a an element of its state X. If a is accessible in $\tau_A(X)$, then it is accessible in X.

Thus in a sense algorithms cannot learn anything by execution: they cannot learn how to make finer distinctions, and they cannot learn how to access more elements (but they can lose both kinds of knowledge). The only possibility of learning open to algorithms seems to be interaction with the environment, but this is the subject of subsequent sections. What states can algorithms reach?

Definition 1.5. (Reachability)
A structure Y is *reachable* from a structure X of the same vocabulary and same base set by small–step algorithms if there is a small–step algorithm A such that $X, Y \in \mathcal{S}(A)$ and $Y = \tau_A(X)$.

By Linear Speedup, reachability in $\leq n$ steps is the same as reachability in one step, for any n. The notion of accessibility suffices to analyze reachability:

Theorem 1.4. Let X, Y be structures of a vocabulary Υ with the same base set. Then Y is reachable from X by small–step algorithms if and only if

- $Y - X$ is finite,
- all function symbols occurring in $Y - X$ are dynamic in Υ, and
- all objects in the common base set, occurring in $Y - X$, are accessible in X.

Proof:
If Y is reachable from X by A, it follows from Lemma 1.1 that $\Delta_A(X)$ is finite, and that all objects occurring there are critical at X, hence also accessible.

To see that the other direction holds, let, by the assumption,

$$Y - X = \{(f_j, (a_1^j, \ldots, a_{n_j}^j), a_0^j) \mid j = 1, \ldots k\}$$

and, by assumption of accessibility, let t_i^j be Υ–terms such that $Val(t_i^j, X) = a_i^j$, for $j = 1, \ldots, k$, $i = 0, \ldots, n_j$. Fix $\mathcal{I}(A)$ so as to satisfy the postulates and to include X, and $\mathcal{S}(A)$ so as to satisfy the postulates and to be closed under τ_A as defined below. Set, for any $Z \in \mathcal{S}(A)$,

$$\Delta_A(Z) = \{(f_j, (Val(t_1^j, Z), \ldots, Val(t_{n_j}^j, Z)), Val(t_0^j, Z)) \mid j = 1, \ldots, k\}.$$

Then the set $\{t_i^j \mid j = 1, \ldots, k,\ i = 0, \ldots, n_j\}$ is a bounded exploration witness for A, and A is a small–step algorithm reaching Y from X.

□

Example 1.1. (Indistinguishable Structures)
Let X, Y be two structures of the same nonlogical vocabulary $\{\mathit{decrypt}, \mathit{fst}, \mathit{snd}, \mathit{op}, c, k\}$ over the same carrier

$$\{\mathrm{Pri}, \mathrm{Pub}, \mathrm{C}, \mathrm{P}, \mathrm{N}, \mathrm{T}, \mathrm{F}, \mathrm{U}\}$$

with the interpretation of nonlogical function symbols as given in the table

Υ	X	Y
$\mathit{decrypt}$	$\mathrm{Pri}, \mathrm{C} \to \mathrm{P}$	$\mathrm{Pri}, \mathrm{C} \to \mathrm{N}$
fst	$\mathrm{P} \to \mathrm{T}$	$\mathrm{P} \to \mathrm{T}$
snd	$\mathrm{P} \to \mathrm{F}$	$\mathrm{P} \to \mathrm{F}$
op	$\mathrm{Pri} \to \mathrm{Pub}$	$\mathrm{Pri} \to \mathrm{Pub}$
c	C	C
k	Pub	Pub

understanding that non-nullary functions take the value U on all arguments not shown in the table. Logical constants `true`, `false`, `undef` are interpreted as $\mathrm{T}, \mathrm{F}, \mathrm{U}$ in both X and Y, respectively. States X and Y are far from being isomorphic, yet they are similar (even coincident) for all terms of the vocabulary, and hence indistinguishable by small-step algorithms.

If element Pri became accessible, say through interaction with environment, by the same term t_{Pri} in both states, they would be easily be distinguished by say term $\mathit{fst}(\mathit{decrypt}(t_{\mathrm{Pri}}, c))$.

The function symbols $\mathit{snd}, \mathit{op}, k$ and their interpretations play no role here, and they could easily be dropped without spoiling the example. We include them to make the transition to further examples below smoother.

Notice that the first-order sentence $\exists x.\, \mathit{fst}(\mathit{decrypt}(x, c)) = \texttt{true}$ would distinguish X from Y.

2. Ordinary Interactive Small–Step Algorithms

In [3, 4, 5] the theory was extended to algorithms interacting with the environment, also within a step. Algorithms might toss coins, consult oracles or databases, send/receive messages... also within a step. We refer the reader to [3] for full explication and motivation—it will have to suffice here to say that the essential goal of behavioral theory, that of capturing algorithms at arbitrary levels of abstraction, cannot be smoothly achieved if interaction with the environment is confined to happen only between the steps of the algorithm. The "step" is in the eye of beholder: what is say from socket abstraction seen as a single act of sending a byte-array may on a lower layer of TCP/IP look as a sequence of steps of sending and resending individual packets until an acknowledgment for each individual packet has arrived. In order to sail smoothly between levels of abstraction, we need the freedom to view several lower-level steps as compressed into one higher-level step when this is natural, even if the lower-level steps are punctured with external interaction. The Bounded Work postulate serves as a guard ensuring that this freedom is not misused.

The syntax of interaction can be, without loss in generality, given by a finite number of *query-templates* $\hat{f}\,\#1 \ldots \#n$, each coming with a fixed arity. If $b_1, \ldots, b_n$ are elements of a state X, a *potential query* $\hat{f}[b_1, \ldots, b_n]$ is obtained by instantiating the template positions $\#i$ by b_i[2]. The environment behavior can be, for the class of "ordinary" interactive algorithms, represented by an *answer function* over X: a partial function mapping potential queries to elements of X, see [3, 4, 5] for extensive discussion and motivation.

All algorithms in the rest of this paper are small-step ordinary interactive algorithms in this sense—in the sequel, we shall skip all these adjectives except possibly for "interactive", to stress the difference with respect to algorithms of the previous section.

The interactive behavior of an algorithm is abstractly represented by a *causality relation*, between finite answer functions and potential queries. We have the following additional postulate:

Postulate 5. (Interaction)
The algorithm determines, for each state X, a *causality relation* $\vdash_X$ between finite answer functions and potential queries.

The intuition of $\alpha \vdash_X q$ is: if the environment, in state X, behaves according to α, then the algorithm will issue q. A *context* for an algorithm is a minimal answer function that saturates the algorithm, in the sense that it would issue no more queries: α is a context if it is a minimal answer function with the following property: if $\beta \vdash_X q$ for some $\beta \subseteq \alpha$, then $q \in \mathrm{Dom}(\alpha)$.

The Updates Postulate is modified by

- associating either failure or an update set Δ^+_A to pairs X, α, where α is a context over X;
- the update set $\Delta^+_A(X, \alpha)$ may also include trivial updates — in an interactive multi–algorithm situation trivial updates may express conflict with another component.

The Isomorphism Postulate is extended to preservation of causality, failure and updates, where $i : X \cong Y$ is extended to "extended states" X, α as $i : X, \alpha \cong Y, i \circ \alpha \circ i^{-1}$.

We can access elements of "extended states" X, α by "extended terms", allowing also query-templates in the formation rules (the extended terms correspond to "e-tags" of [5]). Given vocabularies Υ of function symbols, and E of query-templates disjoint from Υ, we can (partially) evaluate extended terms as

$$\begin{aligned} \mathit{Val}(f(t_1, \ldots, t_n), X, \alpha) &= f_X(\,\mathit{Val}(t_1, X, \alpha), \ldots, \mathit{Val}(t_n, X, \alpha)) && \text{if } f \in \Upsilon \\ \mathit{Val}(\hat{f}(t_1, \ldots, t_n), X, \alpha) &= \alpha(\hat{f}[\,\mathit{Val}(t_1, X, \alpha), \ldots, \mathit{Val}(t_n, X, \alpha)]) && \text{if } f \in E \end{aligned}$$

under the condition that $\mathit{Val}(t_i, X, \alpha)$ are all defined, and also $\hat{f}[\,\mathit{Val}(t_1, X, \alpha), \ldots, \mathit{Val}(t_n, X, \alpha)] \in \mathrm{Dom}(\alpha)$ in the latter case.

Thus the value of an extended term containing query templates can be undefined at X, α, which is different than being defined with the value $\mathtt{undef}_X$. We shall in the sequel use equality of partially defined expressions in the usual Kleene-sense: either both sides are undefined, or they are both defined and equal.

[2]The sole purpose of the $\hat{f}[b_1, \ldots, b_n]$ notation is to be optically distinct from notation for function value $f(b_1, \ldots, b_n)$ when $f \in \Upsilon$.

Remark 2.1. (Kleene Equality)
This means that we lose something of the tight correspondence that the meta-statement $Val(t_1, X) = Val(t_2, X)$ and the Boolean term $t_1 = t_2$ had in the noninteractive case: the former was true if and only if the latter had the (same) value (as) True. Now if say $Val(t_1, X, \alpha)$ is undefined, then also $Val(t_1 = t_2, X, \alpha)$ will be undefined, and the meta-statement $Val(t_1, X, \alpha) = Val(t_2, X, \alpha)$ will be either true or false, depending on whether $Val(t_2, X, \alpha)$ is also undefined. The reader should be aware of this when parsing the meta-statements about coincidence and similarity below.

The Bounded Work Postulate can be (equivalently to the formulation of [3, 4, 5] formulated as before, applying to extended terms, see [5] for extended discussion of "e-tags".

The definition of critical elements must take into account answer functions attached to the state [3, Definition 3.5]: if α is an answer function for a state X, an element of X is *critical* for α if it is the value of some term in a bounded exploration witness T.

All elements in the update set for a given context are critical [3, Proposition 5.24]:

Lemma 2.1. Let X be a state and α a context for X. For any update $\langle f, \vec{a}, b\rangle \in \Delta_A^+(X, \alpha)$, all the components of $\vec{a}$ as well as b are critical for α.

2.1. Coincidence and Similarity

In this subsection, we will extend the notions of coincidence and similarity of extended terms to structures equipped with answer functions.

Definition 2.1. (Coincidence and Similarity)
Let X, Y be Υ-structures, α, β answer functions for X, Y, respectively, and T a set of extended terms. We say that

- X, α and Y, β *coincide over* T, and write $X, \alpha =_T Y, \beta$, if $Val(t, X, \alpha) = Val(t, Y, \beta)$ for every $t \in T$;
- X, α and Y, β are T-*similar*, written as $X, \alpha \sim_T Y, \beta$, if they induce the same equivalence relation on T: $Val(t_1, X, \alpha) = Val(t_2, X, \alpha)$ if and only if $Val(t_1, Y, \beta) = Val(t_2, Y, \beta)$ for all $t_1, t_2 \in T$.

In illustration of Kleene Equality remark 2.1 above, note that if X, Y are coincident/similar for the set T of all Υ-terms, then $X, \emptyset$ and $Y, \emptyset$ are coincident/similar for the set of all extended terms (since the extended terms proper will be undefined under the empty answer function $\emptyset$).

Proposition 2.1. (Factorization for Specific Interactions)
Let X, Y be Υ-structures, α, β answer functions for X, Y, respectively, and T a set of extended terms. Then $X, \alpha \sim_T Y, \beta$ if and only if there is a structure Z and answer function γ for it such that $X, \alpha =_T Z, \gamma \cong Y, \beta$.

Proof:
Define the map ξ as:

$$\xi(y) = \begin{cases} Val(t, X, \alpha) & \text{if } y = Val(t, Y, \beta) \text{ for some } t \in T \\ y & \text{otherwise} \end{cases}$$

and proceed as in the proof of the proposition 1.1. □

An intrastep interaction variant of the Next Value Theorem is proven in [5, Lemma 8.8]. We shall use a variant adapted to our purpose of relating notions of similarity and indistinguishability:

Theorem 2.1. (Next Value)
Let X be a state, T its bounded exploration witness and α a context for X. For every *ground* term t there is a (possibly extended) term $\bigcirc_A^X t \in T^t$ such that $Val(t, \tau_A(X, \alpha)) = Val(\bigcirc_A^X t, X)$. Moreover, if β is a context for a state Y and Y, β coincide with X, α over T^t, then $Val(t, \tau_A(Y, \beta)) = Val(t, \tau_A(X, \alpha))$.

As in the non-interactive case, in consequence to Next Value and Factorization we have preservation of coincidence and similarity:

Corollary 2.1. (Preserving Coincidence and Similarity)
Let X, Y be states and α, β contexts for X, Y, respectively.

- If X, α and Y, β are coincident (over all extended terms), then $\tau_A(X, \alpha)$ and $\tau_A(Y, \beta)$ are coincident (over all ground terms).
- If X, α and Y, β are similar (over all extended terms), then $\tau_A(X, \alpha)$ and $\tau_A(Y, \beta)$ are similar (over all ground terms).

Reasoning about what an algorithm can do in a state, we will have to take into account all possible behaviors of the environment. Typically we will assume some contract with the environment, there will be assumptions on possible environment behaviors. Thus we define what it means for two structures to be similar for given sets of possible answer functions.

Definition 2.2. (Similarity under a Contract)
Let X, Y be Υ-structures, $\mathcal{A}, \mathcal{B}$ sets of answer functions for X, Y respectively, and T a set of extended terms. We say that $X, \mathcal{A}$ and $Y, \mathcal{B}$ are *T-similar*, writing $X, \mathcal{A} \sim_T Y, \mathcal{B}$, if

- for every $\alpha \in \mathcal{A}$ there is a $\beta \in \mathcal{B}$ such that $X, \alpha \sim_T Y, \beta$, and
- for every $\beta \in \mathcal{B}$ there is $\alpha \in \mathcal{A}$ such that $X, \alpha \sim_T Y, \beta$.

The idea is again that, by testing terms for equality, an algorithm cannot determine whether it is operating with X, α for some $\alpha \in \mathcal{A}$ or with Y, β for some $\beta \in \mathcal{B}$. If $\mathcal{A}$ resp. $\mathcal{B}$ are seen as representing the degree of freedom that the environment has in fulfillment of its contract, similarity to the notion of bisimulation of transition systems need not be surprising.

Corollary 2.2. (Factorization under a Contract)
Let X, Y be Υ-structures, $\mathcal{A}, \mathcal{B}$ sets of answer functions for X, Y respectively, and T a set of extended terms. Then $X, \mathcal{A} \sim_T Y, \mathcal{B}$ if and only if

- for every $\alpha \in \mathcal{A}$ there is $\beta \in \mathcal{B}$, Υ-structure Z and answer function γ over Z such that $X, \alpha =_T Z, \gamma \cong Y, \beta$, and
- for every $\beta \in \mathcal{B}$ there is $\alpha \in \mathcal{A}$, Υ-structure Z and answer function γ over Z such that $Y, \beta =_T Z, \gamma \cong A, \alpha$.

Proof:
Use definitions and Proposition 2.1. □

Remark 2.2. (Contracts)
We use a notion of contract heuristically here, we did not define contracts. A proper definition should certainly require that contracts are *abstract*: it should associate a set of answer functions $\mathcal{A}_X$ to any state X in an isomorphism-invariant way. But our results would certainly carry over to such a definition. We are not going to pursue a theory of contracts in this paper.

2.2. Indistinguishability

The notion of indistinguishable states splits here to two notions: states indistinguishable under specific environment behaviors, and states indistinguishable under classes of environment behaviors. We need the former notion in order to formulate the latter.

Definition 2.3. (Indistinguishability under Specific Interactions)
Let X, Y be Υ structures, and α, β answer functions over X, Y respectively, given query templates from E. We say that

- an interactive algorithm A *distinguishes* X, α *from* Y, β if there is a *ground* Υ-term t such that one of the following holds (but not both):
  - either α is a context for A over X and $Val(t, \tau_A(X, \alpha)) = \mathtt{true}_X$, or if this is not true,
  - β is a context for A over Y and $Val(t, \tau_A(Y, \beta)) = \mathtt{true}_Y$.
- X, α and Y, β are indistinguishable if there is no algorithm distinguishing them.

This definition requires an algorithm, if it is to distinguish X, α from Y, β, to complete its step with at least one of them. Weaker requirements might be argued for, but the intuition that we wish to maintain here is that, in order to distinguish two candidate situations, an algorithm should be able to *determine* that it is running in one of them and not in the other—but in order to determine anything an algorithm must complete its step.

The distinguishing term t is required to be ground. The result of the distinguishing algorithm must be contained in the resulting state, and the value of t in it must not depend on any future interaction. Otherwise, even identical states provided with identical answer functions could be distinguishable.

We also assume that vocabulary of each algorithm contains at least one dynamic function symbol. Algorithms with no such symbols are clearly not very useful, but nevertheless allowed by the postulates.

Anyway, the choice of this definition is confirmed by the connection to similarity established below. The following corollary is as simple as it was in the previous section:

Corollary 2.3. Indistinguishability is an equivalence relation on Υ-structures equipped with E-answer functions.

Theorem 2.2. X, α and Y, β are indistinguishable by interactive algorithms if and only if they are similar.

Proof:
Suppose that X, α and Y, β are not similar. Without loss of generality, then there are terms t_1, t_2 such that $Val(t_1, Y, \beta) \neq Val(t_2, Y, \beta)$, whereas $Val(t_1, X, \alpha) = Val(t_2, X, \alpha)$, and $Val(t_1, Y, \beta)$ is defined. If $Val(t_2, Y, \beta)$ is also defined, then an algorithm A computing t_1, t_2, and then completing the step with the update set $\Delta_A(Y, \beta) = \{(f, \texttt{true}, \ldots, \texttt{true}, Val(t_1 \neq t_2, Y, \beta)\}$ distinguishes X, α from Y, β by term $f(\texttt{true}, \ldots, \texttt{true})$. If $Val(t_2, Y, \beta)$ is not defined, we have two distinct cases:

1. Both $Val(t_1, X, \alpha)$ and $Val(t_2, X, \alpha)$ are undefined. In that case, an algorithm evaluating the term t_1 and then concluding the step distinguishes X, α from Y, β by term `true`.

2. Both $Val(t_1, X, \alpha)$ and $Val(t_2, X, \alpha)$ are defined and equal. Then an algorithm evaluating terms t_1, t_2 and then concluding the step distinguishes X, α from Y, β by term `true`.

For the other direction, suppose that X, α and Y, β are similar. By Corollary 2.1, $\tau_A(X, \alpha)$ and $\tau_A(Y, \beta)$ must be similar as well. □

Indistinguishability of states for concrete answer functions is thus equivalent to their similarity under the same answer functions. But what we are really interested in is indistinguishability of states for all possible reactions of the environment. The following definition reflects this consideration.

Definition 2.4. (Indistinguishability under a Contract)
Let X and Y be Υ-structures and let $\mathcal{A}$ and $\mathcal{B}$ be sets of answer functions for X and Y, respectively.

- An algorithm A *distinguishes* $X, \mathcal{A}$ *from* $Y, \mathcal{B}$ if either
  - there is $\alpha \in \mathcal{A}$ such that A distinguishes X, α from Y, β for all $\beta \in \mathcal{B}$, or
  - there is $\beta \in \mathcal{B}$ such that A distinguishes Y, β from X, α for all $\alpha \in \mathcal{A}$.
- $X, \mathcal{A}$ and $Y, \mathcal{B}$ are *indistinguishable* if there is no algorithm distinguishing them.

The intuition here is again that, for an algorithm to distinguish $X, \mathcal{A}$ from $Y, \mathcal{B}$ it must be possible to detect that it is operating in one of them and not in the other. Indistinguishability means here that this is not at all possible, an algorithm can never tell for sure in which of the two worlds it is. It is easy to see that indistinguishability is an equivalence relation on pairs $X, \mathcal{A}$, where X is an Υ-structure and $\mathcal{A}$ a set of E-answer functions over X.

Corollary 2.4. Let $X, \mathcal{A}$ and $Y, \mathcal{B}$ be structures of the same vocabulary, equipped with sets of possible answer functions over the same vocabulary of query-templates. Then they are indistinguishable by interactive ordinary small–step algorithms if and only if they are similar.

Proof:
Use the definitions and theorem 2.2. □

2.3. Accessibility and Reachability

Definition 2.5. (Accessibility and Reachability under Interaction)
Let x be an element of a state X, Y another state of the same vocabulary with the same carrier, $\mathcal{A}$ a set of answer functions for X and $\alpha \in \mathcal{A}$. We say that

- x is *accessible for* X, α if there is an extended term t denoting it at X, α;
- x is *accessible for* $X, \mathcal{A}$ if there is $\alpha \in \mathcal{A}$ such that x is accessible for X, α;
- Y is *reachable from* X, α if there is an algorithm A such that $\tau_A(X, \alpha) = Y$;
- Y is *reachable from* $X, \mathcal{A}$ if there is $\alpha \in \mathcal{A}$ such that Y is reachable from X, α.

Corollary 2.5. (Accessibility)
If X is a structure and $\mathcal{A}$ a set of answer functions over it, any element of X in the range of an $\alpha \in \mathcal{A}$ is accessible for $X, \mathcal{A}$.

Theorem 2.3. Let X, Y be structures of a vocabulary Υ with the same base sets and $\mathcal{A}$ be a set of possible answer functions for X. Then Y is reachable from $X, \mathcal{A}$ by ordinary interactive small–step algorithms if and only if

- $Y - X$ is finite,
- all function symbols occurring in $Y - X$ are dynamic in Υ, and
- there is an $\alpha \in \mathcal{A}$ such that all objects in the common base set occurring in $Y - X$ are also accessible for X, α.

Proof:
Proceed as in the proof of Theorem 1.4. □

2.4. Algorithms with Import

The idea of modeling creation of new objects, often needed for algorithms, by importing fresh objects from a reserve of naked, amorphous objects devoid of nontrivial properties, has been present in the ASM literature since [8].

An answer function α is *importing* for a state if it has only reserve elements in its codomain. We specialize notions of accessibility, reachability and indistinguishability under a contract to *importing small-step algorithm*, meaning that answer functions allowed by a contract are importing.

We need the notions and results of the previous sections in particular for algorithms which import new elements, over a background structure [1]. This case is special, since nondeterminism introduced by a choice of reserve element to be imported is inessential up to isomorphism; see [9] for import from a naked set and [1] for import over a background structure.

The reserve of a state was originally defined to be a naked set. In applications, it is usually convenient, and sometimes even necessary, to have some structure like tuples, sets, lists etc. predefined on *all* elements of a state, including the ones in the reserve. The notion of *background structure* [1] makes

precise what sort of structure can exist above a set of atoms without imposing any properties on the atoms themselves, except for their identity.

In this section, we assume that each vocabulary contains a unary predicate $Atomic$. This predicate and the logical constants are called *obligatory* and all other symbols are called *non-obligatory*. The set of atoms of a state X, denoted with $Atoms(X)$, are elements of X for which $Atomic$ holds.

Definition 2.6. A class K of structures over a fixed vocabulary is called a *background class* if the following requirements are satisfied:
BC0 K is closed under isomorphisms.
BC1 For every set U, there is a $X \in K$ with $Atoms(X) = U$.
BC2 For all $X, Y \in K$ and every embedding (of sets) $\zeta : Atoms(X) \to Atoms(Y)$, there is a unique embedding (of structures) η of X into Y that extends ζ.
BC3 For all $X \in K$ and every $x \in Base(X)$, there is a smallest K–substructure Y of X that contains x.

Suppose that K is a background class. Let S be a subset of a base set of structure $X \in K$. If there is a smallest K–substructure of X containing S, then it is called the *envelope* $E_X(S)$ of S in X and the set of its atoms is called the *support* $Sup_X(S)$ of S in X. In every $X \in K$, every $S \subseteq Base(X)$ has an envelope [1].

A structure X is *explicitly atom-generated* if the smallest substructure of X that includes all atoms is X itself, and a background class BC is explicitly atom-generated if all of its structures are. A background class is *finitary* if the support of every singleton is finite.

Lemma 2.2. Every explicitly atom-generated background class is finitary.

Definition 2.7. (Backgrounds of Algorithms)
We say that a background class K with vocabulary Υ_0 is the *background* of an algorithm A over Υ if

- vocabulary Υ_0 is included in Υ and every symbol in Υ_0 is static in Υ;
- for every $X \in \mathcal{S}(A)$, the Υ_0–reduct of X is in K.

The vocabulary Υ_0 is the *background vocabulary* of A, and the vocabulary $\Upsilon - \Upsilon_0$ is the *foreground vocabulary* of A. We say that an element of a state is *exposed*, if it is in a range of a foreground function, or if it occurs in a tuple in the domain of a foreground function. The *active part* of a state is the envelope of the set of its exposed elements and the *reserve* of a state is the set of non-active atoms.

If the algorithm is not fixed, we say that a state X is *over* a background class BC of vocabulary Υ_0, if Υ_0–reduct of X is in BC.

The freedom the environment has in choice of reserve elements to import induces *inessential nondeterminism*, resulting in isomorphic states [1]:

Proposition 2.2. Every permutation of the reserve of a state can be uniquely extended to an automorphism that is the identity on the active part of the state.

Intuitively, this means that whatever an algorithm could learn by importing new elements from the reserve does not depend on the particular choice of elements imported. Similarly, one might conjecture that an algorithm cannot learn by importing at all, but this is in general not the case:

Example 2.1. Up to isomorphism, the non-logical part of a background structure X consists of hereditarily finite sets over its atoms. The only non-obligatory functions are the containment relation $\in$ and a binary relation P: $P(x, y)$ holds in X if $\text{rank}_X(x) = \text{rank}_X(y) + 1$, where rank_X is defined as:

$$\text{rank}_X(x) = \begin{cases} 0 & \text{if } x \in Atoms(X) \\ \max\{\text{rank}(y) \mid y \in x\} + 1 & \text{if } x \text{ is a set} \end{cases} .$$

The foreground vocabulary contains only one nullary function symbol f, denoting $\{a\}$ in X and $\{\{a\}\}$ in Y for some atom a (for simplicity, we assume that X and Y have the same reduct over the background vocabulary). Structures X and Y are similar, but for all answer functions α, β evaluating the query $\hat{g}$ to a reserve element, X, α and Y, β are not similar, since $Val(P(f, g), X, \alpha) = \texttt{true}$ and $Val(P(f, g), Y, \beta) = \texttt{false}$.

By theorem 1.3 and corollary 2.4, structures X and Y are indistinguishable by non-interactive small-step algorithms, but distinguishable by small-step algorithm importing from the reserve. Somewhat surprisingly, it follows that import of a reserve element can increase the "knowledge" of an algorithm.

In many common background classes, such as sets, sequences and lists, algorithms *cannot* learn by creation. It is important to have in mind that this property is not guaranteed by the postulates of background classes, and that it must be proved for a concrete background class.

2.4.1. Reachability from Empty States

Given a state over a background class, could the state be a result of calculation of some algorithm A, starting from a state with no exposed elements? The issue matters in applications such as [11]. Initial states of an algorithm must be constructed somehow, and it is sometimes important in applications to know can they be constructed by other algorithms starting from scratch. With few additional assumptions imposed on states, it turns out that every state can be constructed by some importing small–step algorithm. Notice that does not mean that a single algorithm, starting from empty states, could be used to construct *all* states of an algorithm.

Let X be a state over a background BC. We will denote with $\mathbf{0}_X$ the unique state obtained from X by "resetting memory" in X: $\mathbf{0}_X$ is the unique state with no exposed elements, of the same vocabulary and over the same background reduct as X. $\mathbf{0}_X$ is an empty state, in the sense that the "memory" of any algorithm is empty in it.

We assume that all foreground functions in all states are marked as dynamic. This assumption is purely technical, made only to simplify the wording and proofs of the results bellow. The results can easily be generalized to cases where this assumption does not hold, but we found no need for that in our applications. Foreground functions are viewed as the modifiable memory of an algorithm.

We also assume that the set of exposed elements in every state is finite. Again, this is a consequence of our intuition of foreground functions as a representation of the finite memory of an algorithm.

Lemma 2.3. Let X be a state over an explicitly atom–generated background BC. Then every element $x \in \mathbf{0}_X$ is accessible by an importing small-step algorithm in $\mathbf{0}_X$.

Proof:
By Lemma 2.2, $\mathbf{0}_X$ is finitary. Hence atomic support $Sup_{\mathbf{0}_X}(\{x\})$ is finite. Since $\mathbf{0}_X$ is explicitly atom–generated, x is accessible by a term from $\mathbf{0}_X, \alpha$ for every α containing the finite $Sup_{\mathbf{0}_X}(\{x\})$ in its codomain. Every atom in $\mathbf{0}_X$ is in reserve, thus there is such an importing α. □

Theorem 2.4. Let X be a state over an explicitly atom–generated background BC. Then X is reachable from $\mathbf{0}_X$ by importing small–step algorithms.

Proof:
$\Delta = X - \mathbf{0}_X$ is finite, since it contains exposed elements only and the vocabulary is finite. Use Lemma 2.3 and Theorem 2.3 to conclude the proof. □

Intuitively, this means that every state can be seen as a result of computation of some algorithm from an empty initial state.

The above theorem was put into practical use in relating abstract and computational model in cryptography in [11].

Example 2.2. We define a background class which can serve as an abstract model of public key cryptography. We do not argue here for naturality of this model, or its appropriateness for any purpose. An interested reader should consult [11] for details. The only role this model has here is as a source of examples for things that even abstract algorithms cannot do.

Take $Coins_X$ as synonymous with $Atoms(X)$. The non-logical part of the background vocabulary contains

- *constructors* binary $\langle \_, \_ \rangle$, unary $nonce$, $privateKey$ and $publicKey$, and ternary $encrypt$,
- unary *predicates* $Nonce$, $PrivateKey$, $PublicKey$, $Encryption$ and $Pair$,
- *selectors* unary fst, snd and binary $decrypt$.

All structures of the background class further satisfy the following constraints:

- the constructors are injective (in all arguments) with pairwise disjoint codomains;
- the predicates $Pair$, $Nonce$, $PrivateKey$, $PublicKey$, $Encryption$ hold exactly on the codomains of $\langle \_, \_ \rangle$, $nonce$, $privateKey$, $publicKey$, $encrypt$ respectively;
- domains of the functions are restricted as follows (in the sense that they take value `undef` elsewhere):

$$\begin{aligned} nonce &: Coins \longrightarrow Nonce \\ privateKey &: Coins \longrightarrow PrivateKey \\ publicKey &: PrivateKey \longrightarrow PublicKey \\ encrypt &: PublicKey \times Msg \times Coins \longrightarrow Encryption \end{aligned}$$

  where Msg is used as shorthand for $Nonce \cup PrivateKey \cup PublicKey \cup Encryption \cup (Msg \times Msg) \cup Boole$, but it is not explicitly represented in the structure;

- the selectors are the least partial functions satisfying the constraints
  - $\langle fst(z), snd(z)\rangle = z$ for each pair z;
  - $decrypt(e, k) = m$ if and only if $e = encrypt(publicKey(privateKey(r_1)), m, r_2)$ for some message m and coins r_1 and r_2.

By definition, the predicates and the selectors are determined given the base set, the atoms and the constructors; thus by BC2 the base set of the structure is freely generated from $Coins$ by the above constructors: it is the minimal set containing $Coins$ and closed under the functions.

This background class will be denoted with BC_{PUB} in the following examples. We will consider algorithms working with answer functions which, over a state X, return only reserve atoms, "fresh coins" of X. Let us, for state X, denote the set of such answer functions with $\mathcal{C}_X$.

Example 2.3. (Inaccessible Objects and Unreachable States)
We will reconsider the situation from Example 1.1 once again, embedding it in BC_{PUB}. To recall, we have states X and Y over BC_{PUB} with the same base set. For simplicity, we will also assume X and Y have the same background reduct. Only elements C, Pub are accessible by nullary foreground functions c, k respectively. Function op of example 1.1 is just a respective alias for the background function $publicKey$ of BC_{PUB}.

According to the table of Example 1.1, the element P must be the value of the (background) term $\langle \texttt{true}, \texttt{false}\rangle$ in both states, while $\mathrm{Pub} = publicKey(\mathrm{Pri})$ must be a $PublicKey$, whereas Pri must be a $PrivateKey$, which means that it must be the value of $privateKey(r_{\mathrm{Pri}})$ for some coin r_{Pri}. We can easily assume r_{Pri} to be the same in both states. Since $decrypt(\mathrm{Pri}, \mathrm{C})$ should have a value distinct from `undef` in both states, C must be an $Encryption$:

- in state X we have $\mathrm{C} = encrypt_X(\mathrm{Pri}, \mathrm{P}, r_{\mathrm{C}})$ for some coin r_{C};
- in state Y we have $encrypt_X(\mathrm{Pri}, \mathrm{N}, r_{\mathrm{C}})$, where we can assume that r_{C} is the same in both states.

We further assume the element N to be a $Nonce$ in both states, which means $\mathrm{N} = nonce(r_{\mathrm{N}})$ for some coin r_{N}, where again we can assume r_{N} to be the same in both states.

The status of element N in the two states is different. Consider the support of exposed object C in the two states:

$$Sup_X(\{\mathrm{C}\}) = \{r_{\mathrm{Pri}}, r_{\mathrm{C}}\}, \quad Sup_Y(\{\mathrm{C}\}) = \{r_{\mathrm{Pri}}, r_{\mathrm{N}}, r_{\mathrm{C}}\}$$

which means that $\mathrm{N}, r_{\mathrm{N}}$ are active in Y, but not in X.

Like in Example 1.1, N is not exposed in either state, which also means not accessible by any foreground term. But in state X an answer function from $\mathcal{C}_X$ is free to respond to a query with the reserve atom r_{N}, which means that N is accessible—since it is inactive, we say that N *can be created* in X. In Y on the other hand r_{N} is not reserve, and an answer function form $\mathcal{C}_Y$ is not free to return r_{N}. This means that N is not accessible in Y at all. For the same reason no fresh (different from C) encryption with N as subject can be created (accessed) in Y.

This is something algorithms just cannot do.

But are background structures needed here at all? Why would the functions $encrypt, decrypt$ be needed in the background, could we not just consider them as dynamic functions in the ASM tradition, to be updated as needed, i.e. as encryptions get created? This way we might, in Example 1.1, obtain

isomorphism of X, Y, instead of just similarity. Of course, requirement of isomorphism would exclude a background containing $encrypt$, $decrypt$.

Such an approach, suggested by some studies in (statics of) abstract cryptography, involves a problem arising only in the dynamics: assume that in such a model an algorithm *learns* the private key Pri, say by environment interaction, as a value of a term t_{Pri}. Then X and Y must become distinguishable by term $decrypt(t_{\mathrm{Pri}}, c)$, which means we would have to *create* the distinction by updating $decrypt$. A technical problem arises with public key encryption: the act of encrypting involves updating both $encrypt$ and $decrypt$, but in order to update $decrypt$ we would need to access the *private* key, which is definitely not allowed by the usual assumptions on public key cryptography.

With background structures learning new information does not change anything, we might just *uncover* differences which were there all the time. The natural interpretation of indistinguishability (similarity) of two states is then: information available to algorithms is not sufficient to distinguish them.

References

[1] Blass, A., Gurevich, Y.: Background, Reserve, and Gandy Machines, *Proceedings of CSL '00*, 1862, 2000.

[2] Blass, A., Gurevich, Y.: Algorithms: A Quest for Absolute Definitions, *Bulletin of the European Association for Theoretical Computer Science*, (81), October 2003, 195–225.

[3] Blass, A., Gurevich, Y.: Ordinary Interactive Small–Step Algorithms I, *ACM Transactions on Computaional Logic*, to appear.

[4] Blass, A., Gurevich, Y.: Ordinary Interactive Small–Step Algorithms II, *ACM Transactions on Computaional Logic*, to appear.

[5] Blass, A., Gurevich, Y.: Ordinary Interactive Small–Step Algorithms III, *ACM Transactions on Computaional Logic*, to appear.

[6] Blass, A., Gurevich, Y., Shelah, S.: Choiceless Polynomial Time, *Annals of Pure and Applied Logic*, **100**(1-3), 1999.

[7] Glavan, P., Rosenzweig, D.: Communicating Evolving Algebras, in: *Computer Science Logic*, vol. 702 of *LNCS*, 1993, 182–215.

[8] Gurevich, Y.: Evolving Algebras. A Tutorial Introduction, *Bulletin of the European Association for Theoretical Computer Science*, **43**, 1991, 264–284.

[9] Gurevich, Y.: Evolving Algebras 1993: Lipari Guide, in: *Specification and Validation Methods*, Oxford University Press, 1995, 9–36.

[10] Gurevich, Y.: Sequential Abstract State Machines Capture Sequential Algorithms, *ACM Transactions on Computational Logic*, **1**(1), 2000, 77–111.

[11] Rosenzweig, D., Runje, D., Schulte, W.: Model-Based Testing of Cryptographic Protocols, in: *TGC 2005*, vol. 3705 of *LNCS*, 2005, 33–60.

[12] Staerk, R., Nanchen, S.: A Logic for Abstract State Machines, *Universal Journal of Computer Science*, **11**(7), 2001, 981–1006.