The Logic in Computer Science Column

by

Anuj Dawar

Department of Computer Science and Technology
University of Cambridge, Cambridge, CB3 0FD, UK
anuj.dawar@cl.cam.ac.uk

In this issue, we welcome back Yuri Gurevich, who is well known to the readers of this column. He is here with a contribution that is a further development in the realm of Abstract State Machines. As the regular readers of this column know very well, these are a formalization of the notion of an algorithm that can work at any level of abstraction. While the original formulation was for sequential deterministic algorithms, more recent developments by Yuri and his collaborators have included formalizations of nondeterministic and quantum algorithms. In the present version, an algorithm is seen as interacting with an external environment. This environment can be the source of nondeterminism or randomness. In the article presented here, Yuri develops the fomalization of the *inner* view of a deterministic algorithm which sees the input from the environment through oracles. I am excited to share this with you.

ASSISTED COMPUTATIONS: INNER VIEW

Yuri Gurevich
University of Michigan, Ann Arbor, MI, USA

If people do not believe that mathematics is simple,
it is only because they do not realize how complicated life is.
— John von Neumann

1 Introduction

Starting in the 1930s, if not earlier, logicians were interested in "effective methods" of computation, which they eventually renamed "algorithms." By the 1960s, the intuitive notion of algorithm had been substantially clarified — see, for example, [12, §1.1]. To distinguish this notion from later generalizations, we call such algorithms *basic*.

The original Church-Turing thesis was about basic algorithms, which are deterministic and work in sequential time, i.e. proceed step by step. For a long time, the attempts to derive the thesis from first principles met with limited success. According to [7], called "the original study" below, there were two missing ingredients. One was a formalization of abstract state. The emergence of numerous levels of abstraction in software facilitated this formalization. The other ingredient was the bounded exploration principle formalizing Kolmogorov's observation that algorithms compute in steps of bounded complexity [11].

The main theorem of the original study was that three basic principles—sequential time, abstract state, and bounded exploration—imply that every basic algorithm is step-by-step simulated by a basic abstract state machine (ASM). The ASM model of computation was introduced earlier to provide an operational specification of software (see [6] for a standard exposition). The original Church-Turing thesis follows from the three principles plus the assumption that initial states contain only the minimum information necessary to compute a function [4].

This work was supported in part by the U.S. Army Research Office under Grant W911NF-20-1-0297.

I am indebted to Andreas Blass for careful reading of numerous drafts and for many insightful suggestions.

A natural question is whether this progress could be extended to nondeterminism. The answer is yes, and the core idea appears already in the original study [7, §9.1]. Algorithms are inherently deterministic[1] and need an environment to perform non-algorithmic tasks like flipping a coin. This idea was explored in depth in [2, 3]. Later we applied the idea to quantum computing [10], which was not covered in [2, 3].

Eventually we realized that it is beneficial for the algorithm to interact with partners that assist it (and may in turn be assisted by it). Mathematically, these partners/assistants are oracles. This, together with a few additional insights, substantially simplifies and strengthens the analysis of [2, 3]; in particular, it yields an explicit treatment of quantum circuit computations.

This perspective naturally leads to two complementary views of computation: the *inner view* (as seen by the algorithm itself) and the *outer view* (as seen by an external observer). The outer view is richer. An external observer sees how the interaction with partners develops. The outer view analysis was sketched in [8]. A journal version is in preparation [10].

The inner view is much simpler but clarifies many aspects nevertheless. Here we present the inner view, which is closer to the original study. In this view, oracle-managed functions are like other functions managed by the environment (as opposed to program variables managed by the algorithm).

2 Example algorithms

Example 1. A version of Euclid's algorithm for computing the greatest common divisor of two positive integers.

```
a:= Input1; b:= Input2;
until b=0 do (a := b || b := a mod b);
print "GCD(Input1,Input2)="a
```

Here Input1 and Input2 are positive integers provided by the user. The "until h do R" is equivalent to "while $\neg h$ do R" but avoids negating the termination (a.k.a. halting) condition. The sign || denotes parallel composition, which in this case is simultaneous assignment.

Example 2. A well-known ruler-and-compass algorithm known as the tangent algorithm. In a fixed Euclidean plane we are given a circle A, its center p, and a point q outside of A. The algorithm constructs a tangent to A through q.

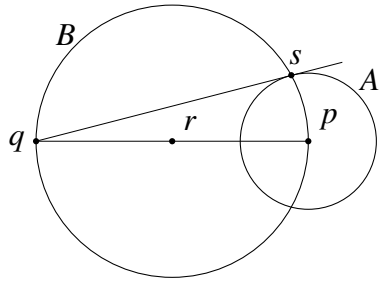
[1] Recall Yogi Berra's instruction: "When you come to a fork in the road, take it."

(1) Draw the midpoint r between p and q.

(2) Draw the r-centered circle B through q.

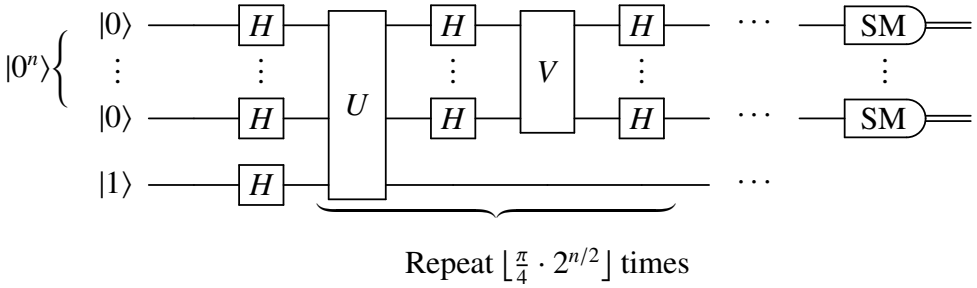
(3) Choose a point s where A and B intersect.

(4) Draw a line through q and s.



The tangent algorithm has no code for drawing or choosing. It delegates all that to the environment. It just orchestrates the process and tells the environment what to do and in what order.

Example 3. Grover's search algorithm [5], one of the most famous quantum circuit algorithms, is given by the following circuit diagram:



Our description of the algorithm involves quantum terminology; if it is unfamiliar to you, don't worry. The algorithmic part should be clear.

The circuit has $n + 1$ wires numbered from top to bottom. Time flows left to right. A state of the algorithm has two separate parts known as the classical and quantum states. The initial quantum state is $|0\rangle \otimes \cdots \otimes |0\rangle \otimes |1\rangle$.

The gates H, U, and V perform unitary transformations of the quantum state. Each SM gate performs a standard measurement and produces a classical bit: 0 or 1. Double lines indicate production of classical data.

3 Algorithm-apt structures

This is an auxiliary and somewhat technical section; but the notion of algorithm-apt structures is key in our analysis of computation.

Informally, a state of an algorithm is a complete description of the result of the computation performed so far. The subsequent computation of the algorithm is determined by the state and the environment.

Reality check 3.1. This is an idealization of course which is largely valid but frays on the margin. In real-world system engineering, in addition to syntax and semantics, there is also pragmatics. Resources are limited, and shortcuts between different levels of abstraction may be beneficial to efficiency. Think about terminal states for example. In many implementations, the algorithm doesn't "know" it has finished; it simply stops receiving CPU cycles. ◁

Formally, states may be viewed as structures in the sense of mathematical logic. The rich experience of mathematics and computer science strongly suggests that any static mathematical setting can be faithfully represented by a structure.

Q: It is often said that a state of a program is determined by the values of programming variables. Do you agree with that?

A: Conventional programming variables may not determine the state. For example, the recursion stack of a recursive program contains additional state information. Hence our more general notion of programming variables. ◁

We consider specialized vocabularies and structures. As far as vocabularies are concerned, we stipulate the following.

`Relations` In logic, vocabularies include relation symbols. In programming, a relation is a Boolean-valued function, and that is how we treat relations. But we mark the function symbols intended to be relations; in that sense we have relations in our vocabularies.

`Function labels` We use several kinds of basic functions in our structures, and it is convenient to give names to those kinds, used as labels for the corresponding function symbols.

`Types` We assign types to function symbols, albeit in a rudimentary way sufficient for our purposes.

Definition 3.2. An *algorithm-apt* vocabulary V is a finite set of *function symbols*, subject to the following requirements.

(1) Function symbols may be marked as *relation*.

(2) Every function symbol has one of the following five labels:
program variable, logic, type, plain, oracle.

(3) Each function symbol f is assigned a *function type*:

$$f : T_1 \times \cdots \times T_r \to T_0$$

where r is the *arity* of f, all T_i's are type symbols, symbols $T_1, \ldots, T_r$ are the *argument types* of f, and T_0 the *value type*. If f is a relation then T_0 is `Bool`. If f is nullary, we say that f is of type T_0.

(4) V contains several particular symbols.

- The type symbols `Any`, `Bool`, and `Integer`; all three are relations.
- The equality sign, which is a relation of type $\texttt{Any} \times \texttt{Any} \to \texttt{Bool}$, and the propositional connectives $\wedge$, $\vee$, $\to$ (binary), $\neg$ (unary), and `true`, `false` (nullary) whose argument and value types are `Bool`. All these are logic symbols.
- The symbols for the standard arithmetical operations $0, 1 : \texttt{Integer}$ and $+, -, \times, \div : \texttt{Integer} \times \texttt{Integer} \to \texttt{Integer}$. ◁

Terms in a vocabulary V are defined recursively. If $f : T_1 \times \cdots \times T_r \to T_0$ and $t_1, \ldots, t_r$ are terms of type $T_1, \ldots, T_r$ respectively, then $f(t_1, \ldots, t_r)$ is a term of type T_0. Here the case $r = 0$ is the basis of recursion.

Recall that a relation R on a set S is represented by — and may be identified with — the set $\{s \in S : R(s) \text{ is true}\}$.

Definition 3.3. An *algorithm-apt structure* X of an algorithm-apt vocabulary V consists of the following components.

(1) A nonempty set $|X|$, the *base set* of X.

(2) The interpretations f_X of the V symbols, the *basic functions* of X. If $f : T_1 \times \cdots \times T_r \to T_0$ then f_X is a partial function from $(T_1)_X \times \cdots \times (T_r)_X$ to $(T_0)_X$. If f is a type symbol, it is convenient to view f_X as a subset of $|X|$, an *element type*, or just *type*.

It is required that conditions S1–S3 below are satisfied. ◁

The partiality of basic functions is necessary to deal with oracles. Given a query, an oracle may or may not respond. Also, the reply may be inappropriate and thus illegal.

If $f_X(\bar{x}) = \texttt{true}_X$, we say that $f_X(\bar{x})$ *holds* in X, symbolically $X \models f_X(\bar{x})$.

S1 $\texttt{true}_X$ and $\texttt{false}_X$ are distinct. The equality sign is interpreted as identity: If x_1, x_2 are elements of $|X|$, then $(x_1 =_X x_2)$ produces $\texttt{true}_X$ if x_1, x_2 are the same element, and $\texttt{false}_X$ otherwise.

S2 $\texttt{Any}_X = |X|$. $\texttt{Bool}_X = \{\texttt{true}_X, \texttt{false}_X\}$, $\texttt{Integer}_X$ is (a copy of) the integers, the propositional and arithmetical symbols are interpreted as expected.

S3 If $f : T_1 \times T_2 \times \cdots \times T_r \to T_0$ then f_X is a partial function from $(T_1)_X \times (T_2)_X \times \cdots \times (T_r)_X$ to $(T_0)_X$.

If T is a type then the elements of T_X are of type T. We do not assume that types are mutually exclusive, so an element may have several types.

Concerning the five labels, the intention is that programming variables are managed by the algorithm, whereas all other basic functions are managed by (possibly different parts of) the environment:

Basic functions	**Access mode for algorithms**	**Determinism required?**	**Examples**
Prog vars	Read, write	Yes	Variable arrays
Logic	Read	Yes	Prop connectives
Type	Read	Yes	`Bool`, `Integer`
Plain	Read	Yes	Arithmetic
Oracle	Read	No	User input

4 Sequential Time Postulate

We start with recalling the syntax and semantics of (well formed) terms, called *expressions* in programming.

Definition 4.1. Let V be a vocabulary and X a V structure. By mutual recursion, we define

(i) terms t of vocabulary V,
(ii) the (value) type of term t,
(iii) whether term t is defined in X, and
(iv) if yes, the value $[\![t]\!]_X$ of t in X.

If f is a function symbol in V of type $T_1 \times \cdots \times T_r \to T_0$ and $t_1, \ldots, t_r$ are terms of the types $T_1, \ldots, T_r$ then:

(1) $f(t_1, \ldots, t_r)$ is a *term*,

(2) its *type* is T_0,

(3) $f(t_1, \ldots, t_r)$ *is defined* in X if every t_i is defined in X and f_X is defined at $([\![t_1]\!]_X, \ldots, [\![t_r]\!]_X)$, and

(4) if $f(t_1, \ldots, t_r)$ is defined then its *value* $[\![f(t_1, \ldots, t_r)]\!]_X$ *in* X is $f_X([\![t_1]\!]_X, \ldots, [\![t_r]\!]_X)$ and it is of type T_0.

Here the case $r = 0$ is the basis of recursion. ◁

Remark 4.2 (Strict)**.** In programming terminology, clause (3) says that evaluation of terms is strict. This is a serious restriction — expression evaluation in programming is typically not strict. But the restriction is for a reason. Our functions are mathematical functions in a static structure, not procedures in a program. For illustration, suppose that f is a mathematical function of type $\texttt{Integer} \times \texttt{Integer} \to \texttt{Integer}$. If $f(0, y) = 0$ for all y, your algorithm may take advantage of it. But $\mathrm{Dom}(f)$ consists of pairs (x, y) of integers. If you want to invoke f, you have to provide a pair of integers.

By the way, in our approach, such f may be a single programming variable, a variable array (which may be viewed, if desired, as an array of conventional programming variables). ◁

Remark 4.3 (Ground)**.** Our terms are ground terms, which means that they do not involve variables. We view conventional programming variables as nullary function symbols. While their interpretations may vary from one state to another, they are constant in any given state. ◁

Postulate 1 (Sequential Time)**.** An algorithm $\mathcal{A}$ is given by the following components:

(1) An algorithm-apt vocabulary Voc and a set ImVal of Voc terms, including `true` and `false`.

(2) A nonempty set $\mathcal{S}$ of algorithm-apt Voc structures such that $[\![e]\!]_X = [\![e]\!]_Y$ for all e in ImVal and all $X, Y \in \mathcal{S}$.

(3) A nonempty subset of $\mathcal{I}$ of $\mathcal{S}$ and a subset $\mathcal{T}$ of $\mathcal{S}$.

(4) A partial map τ on the set $\mathcal{S}$ such that every $\tau(X)$ is an algorithm-apt V structure obtained from X by modifying (only) the interpretations of programming variables. $\mathrm{Dom}(\tau)$ contains no $\mathcal{T}$ states. ◁

Notation and terminology 4.4. The components (1)–(4) are the *basic components* of $\mathcal{A}$. Members of $\mathcal{S}$, $\mathcal{I}$, and $\mathcal{T}$ are *states, initial states*, and *terminal states* respectively. Members e of ImVal are *immediate-value* terms; their values are *immediate* values. The map τ is the *state transformer*. We allow modifications to be vacuous, so $\tau(X)$ may be X.

States in $\mathrm{Dom}(\tau)$ are *actionable*. A state is *normative* if it is actionable or terminal. We write $\mathrm{Voc}(\mathcal{A})$, $\mathrm{ImVal}(\mathcal{A})$, $\mathcal{S}(\mathcal{A})$, $\mathcal{I}(\mathcal{A})$, $\mathcal{T}(\mathcal{A})$, and $\tau(X, \mathcal{A})$ if the algorithm $\mathcal{A}$ is not clear from the context. ◁

Discussion 4.5 (Immediate values)**.**

Q: Immediate-values expressions? Are there any in the real world? In what sense are they immediate?

A: Yes, there are. The Booleans and small integers are typical examples. They are immediate in this sense: if you inspect an immediate value — think about it as a string, for simplicity — then you find the corresponding immediate-value expression immediately. Immediate values are trivial systematic variants of the corresponding expressions

Q: What about larger integers?

A: Larger integers are not immediate. Normally, there are only a few immediate-value terms. Their role will become clear in §6. ◁

Discussion 4.6 (The inner and outer views)**.**

Q: The transformer is single-valued over its domain. How is this possible in the presence of nondeterministic oracles?

A: We distinguish between the *inner view* of the computation (as seen by the algorithm itself) and the *outer view* (as seen by an external observer). The outer view is analyzed in [10]. Here we analyze the inner view.

Let us illustrate the two views on an example where Voc contains a plain function f and a single oracle function O, both of type `Integer` $\to$ `Integer`. Let a state X be in $\mathrm{Dom}(\tau)$.

We begin with the outer view. In the course of the computation, the algorithm may need values $O(x)$ for some integers x. Appropriate queries are issued. Depending on the replies, the algorithm may issue more queries, and so on. Eventually, $\tau(X)$ is produced.

The inner view is very different. The algorithm cannot distinguish between evaluating plain functions and evaluating oracle functions. Whether it evaluates $f(7)$ or $O(7)$, the requested value is provided by the environment. In reality, different parts of the environment may be responsible for computing $f_X(7)$ and $O_X(7)$. For example, $f_X(7)$ may be computed by the local operating system while $O_X(7)$ comes via the internet. But the algorithm is not aware of that. In both cases, the result miraculously appears.

In the next step, $O(7)$ may acquire a new value while $\tau(X)$ retains the old value because τ can change only programming variables. In the inner view, it is the environment that can modify the value of $O(7)$. Such modifications by the environment turn $\tau(X)$ into a successor state, and this happens after the current step of the algorithm is completed and before the following step starts. Nondeterminism enters only via such modifications by the environment. The transition function τ and, indeed, the whole algorithm is deterministic.

Q: It seems that the outer view is more expressive and useful. Why bother with the inner view?

A: Partly for expositional reasons. The inner view is simpler. Some subtle issues are easier in the inner view. But it is also important to understand algorithm's perspective. ◁

Definition 4.7. A *finite run* of the algorithm has the form

$$X_0, \tau(X_0), X_1, \tau(X_1), \ldots, \tau(X_{n-1}), X_n$$

where $X_0, \ldots, X_{n-1}$ are actionable states, and every X_{i+1} is produced from $\tau(X_i)$ by the environment, by modifying only the oracle functions. The run is *failing* if X_n is failing or if the environment fails to turn $\tau(X_n)$ into a state. ◁

5 Abstractness postulate

Postulate 2 (Abstractness)**.** The postulate consists of two clauses: Abstract State and Abstract Step below. ◁

In the Abstract State clause, we abstract from the implementation of states; isomorphic states are identical on the abstraction level of the algorithm In the Abstract Step clause, we abstract from the details of how the algorithm transforms a actionable state X into $\tau(X)$.

We interleave the clauses with auxiliary text. Each of the following two subsections is devoted to a single clause.

Abstract Step

Definition 5.1. An *(atomic) update* of a state X has the form $\langle f, (x_1, \ldots, x_r), x_0 \rangle$ where f is a programming variable of some type $(T_1 \times \cdots T_r) \to T_0$, and every x_i is of type T_i. The intention is that the value of f_X at the argument tuple $(x_1, ..., x_r)$ is to be changed to x_0 in the structure $\tau(X)$.

The update is *trivial* if $x_0 = f_X(x_1, \ldots, x_r)$. Two updates of the same function f_X at the same arguments but with different values are *contradictory*. A set of updates of X, in short an *update set*, is *consistent* if it contains no contradictory updates. ◁

Definition 5.2. If Δ is a consistent update set of a state X, then $X + \Delta$ is the Voc structure with $|X + \Delta| = |X|$ and the basic functions

$$f_{X+\Delta}(\bar{x}) = \begin{cases} y & \text{if the update } \langle f, \bar{x}, y \rangle \in \Delta, \\ f_X(\bar{x}) & \text{otherwise.} \end{cases}$$

If Δ is inconsistent then $X + \Delta$ is undefined. ◁

Corollary 5.3. *(1) If f is a logic, type, plain, or oracle function symbol in Voc — i.e. not a programming variable — then $f_{X+\Delta} = f_X$.*

(2) $X + \Delta$ is algorithm-apt. ◁

Finally, we are ready to formulate the Abstract Step clause.

Abstract Step For every actionable state, the algorithm produces a consistent finite update set $\Delta(X)$ such that $\tau(X) = X + \Delta(X)$. ◁

We write $\Delta(X, \mathcal{A})$ when the algorithm $\mathcal{A}$ is not clear from the context.

Abstract State

Definition 5.4. An *isomorphism* $\mu : X \to Y$ of structures of the same vocabulary V is a bijection from $|X|$ to $|Y|$ that respects the basic functions, that is,

$$\mu(f_X(x_1, \ldots, x_r)) = f_Y(\mu x_1, \ldots, \mu x_r)$$

for every arity r and every r-ary function symbol f in the vocabulary.

In the case where V is Voc, μ is *immediacy-compliant* if it is the identity map on $\{[\![e]\!]_X : e \text{ is immediate-value}\}$. ◁

Lemma 5.5. *If $\mu : X \to Y$ is an isomorphism of structures of the same vocabulary V then $\mu([\![t]\!]_X) = [\![\mu(t)]\!]_Y$ for all V terms t.*

Proof. Induction on t. □

Definition 5.6. Let $\mu : X \to Y$ be a state isomorphism (i.e. an isomorphism of states). If $\Delta(X)$ is an update set of X then

$$\mu\langle f, (x_1, \ldots, x_r), x_0\rangle = \langle f, (\mu x_1, \ldots, \mu x_r), \mu x_0\rangle,$$
$$\mu(\Delta X) = \{\mu(u) : u \in \Delta(X)\}.$$ ◁

Lemma 5.7. *Every state isomorphism $\mu : X \to Y$ is immediacy-compliant.*

Proof. For every immediate-value term e,

$$\begin{aligned} \mu([\![e]\!]_X) &= [\![e]\!]_Y && \text{because } \mu \text{ is an isomorphism} \\ &= [\![e]\!]_X && \text{because } e \text{ is immediate-value.} \end{aligned}$$ □

Abstract State The sets of states, initial states, and terminal states are closed under immediacy-compliant isomorphisms. If $\mu : X \to Y$ is a state isomorphism then $\mu(\Delta(X)) = \Delta(Y)$. ◁

6 Bounded Exploration Postulate

Notation and terminology 6.1. If f and g are partial functions from a set A to a set B, we write $f(x) \simeq g(x)$ to mean that, for any argument a, the two values $f(a)$ and $g(a)$ are either both defined and equal or both undefined. The sign $\simeq$ is known as *Kleene equality*. ◁

Definition 6.2. Structures X, Y of the same vocabulary V *agree on a* V *term* t if $[\![t]\!]_X \simeq [\![t]\!]_Y$. They *agree on a set* S *of terms* if they agree on all terms in S. ◁

Definition 6.3. A set S of terms is *closed under negation* if, together with every unnegated Boolean term β, it contains $\neg\beta$. It is *closed under immediate equalities* if it contains every equality $s = e$ where $s \in S$ and e is an immediate-value term. S is *rounded* if it is closed under subterms, closed under negation, and under immediate equalities (and thus contains the immediate-value terms unless $S = \varnothing$). The *rounded version* $\overline{S}$ of S is the smallest rounded superset of S. ◁

Postulate 3 (Bounded Exploration)**.** There exist a finite, rounded set Tags of $\mathrm{Voc}(\mathcal{A})$ terms and an assignment, to every actionable state X, of a rounded *explore set* $\mathcal{E}(X) \subseteq \{t \in \mathrm{Tags} : [\![t]\!]_X \text{ is defined}\}$ satisfying the following requirements where X, Y range over normative states.

Determine If X and Y have the same explore set and agree on it then either both states are terminal or both states are actionable and $\Delta(X) = \Delta(Y)$.

Discriminate There is a partial *exploration order* on $\mathcal{E}(X)$ such that if t is a tag in $\mathcal{E}(X) - \mathcal{E}(Y)$, then X, Y disagree on a Boolean term in $\mathcal{E}(X) \cap \mathcal{E}(Y)$ that precedes t in the exploration order for X. ◁

Corollary 6.4. *(1) If normative states* X *and* Y *agree on the Boolean terms in* $\mathcal{E}(X) \cap \mathcal{E}(Y)$ *then* $\mathcal{E}(X) = \mathcal{E}(Y)$*.*

(2) Isomorphic actionable states have the same explore set.

Proof. (1) follows from the Discriminate clause.

(2) Isomorphic states agree on Boolean terms. Apply (1). □

Definition 6.5. An (abstract basic) algorithm is an entity that satisfies Postulates 1–3. ◁

In the rest of the paper, $\mathcal{A}$ is a fixed algorithm. Further, Tags and sets $\mathcal{E}(X)$ are as in the postulate (relative to $\mathcal{A}$); elements of Tags are *tags* (for $\mathcal{A}$).

Commentary 6.6 (Intuition and intent)**.**

`Tags`. Consider an algorithm written in some programming language. Let S be the set of terms (expressions) that occur in the program of our algorithm, and T the rounded version of S. Think of tags as members of T. This presumes that all hidden arguments are made explicit.

The most important issue is the finiteness of the set of tags, but that part of the postulate was justified in [7]. Here we will address newer issues.

Let X, Y be actionable states.

`Explore sets`. The explore set $\mathcal{E}(X)$ of X is the set of tags that the algorithm evaluates in X. Since X is actionable, the tags in $\mathcal{E}(X)$ have well-defined values in X.

Explore sets were introduced in [1] to bolster the Bounded Exploration Postulate by requiring that, in every state X, the algorithm explores only terms t that are defined at X.

`Determine`. Consider an arbitrary moment during the algorithm's calculation of $\tau(X)$ from X, and let T be the set of terms evaluated so far. What terms, if any, will the algorithm evaluate next? The next action depends on the values $[\![t]\!]_X$ of tags t in T.

Importantly, it depends only on those values. For suppose that, at a certain moment in the calculation of $\tau(Y)$ from Y, exactly the tags in T have been evaluated with exactly the same results. Thus, the algorithm has not yet detected any difference between the two calculations. Accordingly, the next action will be the same.

If $\mathcal{E}(X) = \mathcal{E}(Y)$ and X, Y agree on the common explore set then the calculations never diverge, and so the two calculations look identical to the algorithm. Hence, $\Delta(X) = \Delta(Y)$.

`Discriminate`. The exploration order is the order in which the tags are evaluated in X in the process of computing $\Delta(X, \mathcal{A})$. Assume that $\mathcal{E}(X) - \mathcal{E}(Y) \neq \emptyset$ and $t \in \mathcal{E}(X) - \mathcal{E}(Y)$. Obviously, the calculations of $\tau(X)$ and $\tau(Y)$ have diverged at some point. Let's examine the situation at that point.

The two calculations have evaluated the same set T of tags and still pursue the same course of action in the sense that the tags to be evaluated next are exactly the same. Notice that it is not necessarily the case that $[\![t]\!]_X = [\![t]\!]_Y$ for all $t \in T$. For example, suppose that the state Y is obtained from X by replacing the elements $[\![t]\!]_X$, $t \in T$, with fresh elements. Since the nature of state elements is implementation dependent and not visible at the abstraction level of the algorithm, the algorithm would pursue the same course of action in both cases.

How can the two calculations diverge at this point in a way that leads to different explore sets? It is here that immediate values come into play. Suppose that, after evaluating the same set T of tags, the two calculations diverge and that, for simplicity, a single tag s is evaluated at this point and $[\![s]\!]_X \neq [\![s]\!]_Y$. We have $s \in \mathcal{E}(X) \cap \mathcal{E}(Y)$. If Y is obtained from X by replacing the elements $[\![u]\!]_X$, $u \in (T \cup \{s\})$, with fresh elements, one still can argue as above that the replacement is immaterial to the course of action. And this is correct, unless the value of s happens to be an immediate value, the value of some immediate-value term e, so that $[\![s]\!]_X = [\![e]\!]_X = [\![e]\!]_Y \neq [\![s]\!]_Y$. You can't get Y if the replacement by fresh elements involves immediate values.

Since the immediate-value terms are trivially reconstructible from their values, the algorithm discovers that the equality $s = e$ holds in X but not in Y. In that sense, it explores this equality, so it is a Boolean tag that belongs to $\mathcal{E}(X) \cap \mathcal{E}(Y)$. (If s happens to be Boolean, it is already a Boolean tag in $\mathcal{E}(X) \cap \mathcal{E}(Y)$ on which X and Y disagree.) ◁

Q: It seems to me that you use only the explore sets, not Tags. Can't you define Tags to be the union of the explore sets?

A: Yes, that would work, mathematically speaking.

Q: So why bother with Tags?

A: Suppose that we don't use Tags and just require that the union of the explore sets is finite. Then we need to add this additional requirement to Postulate 3, as we did in [1]. A question arises how to motivate the new requirement in this foundational study.

On the other hand, motivating the finiteness of Tags is natural and has been done already in the original study. In fact, we mentioned this motivation in the commentary above. Intuitively, if all programming variables are made explicit, then Tags is the rounded version of the set of expressions that actually occur in the program.

7 Explore sets

This is an auxiliary technical section. States are states of the fixed algorithm $\mathcal{A}$.

Definition 7.1 (Similarity)**.** (1) For a set S of normative states, $\mathcal{E}(S) = \bigcap \{\mathcal{E}(X) : X \in S\}$ and $\mathcal{B}(S)$ is the set of Boolean tags in $\mathcal{E}(S)$.

(2) Normative states X and Y are *similar*, symbolically $X \sim Y$, if they agree on $\mathcal{B}(\{X, Y\})$. If $X \sim Y$ then, by Corollary 6.4, $\mathcal{E}(X) = \mathcal{E}(Y)$. By the Determinate clause, both states are terminal or both are actionable and $\Delta(X) = \Delta(Y)$.

(3) $[X]$ is the similarity class of X, which is *actionable* (resp. *terminal*) if X is so.

(4) The conjunction $G_\sigma = \bigwedge \{\beta \in \mathcal{B}(X) : X \models \beta\}$ is the *guard* of σ.

Corollary 7.2. *If $Y \models G_{[X]}$ then $X \sim Y$.*

Corollary 7.3. *There are only finitely many similarity classes.*

Proof. Explore sets consist of tags, and the set of tags is finite. □

Definition 7.4. A set A of normative states is *agreeable* if they agree on $\mathcal{B}(A)$.

Lemma 7.5. *All members of an agreeable set A have the same explore set.*

Proof. Suppose, toward a contradiction, that despite A's agreeability, not all states in A have the same explore set. Then there are $X, Y \in A$ with $\mathcal{E}(X) - \mathcal{E}(A) \neq \emptyset$. Fix an exploration order $<$ on $\mathcal{E}(X)$ as in the Discriminate clause and choose a tag $t \in \mathcal{E}(X) - \mathcal{E}(A)$ that is minimal respect to $<$.

Since $t \in \mathcal{E}(X) - \mathcal{E}(A)$, there is some $Y \in A$ such that $t \in \mathcal{E}(X) - \mathcal{E}(Y)$. By the Discriminate clause, X and Y disagree on some $s < t$ in $\mathcal{B}(\{X, Y\})$. Recall that A is agreeable. If s were to belong to $\mathcal{E}(A)$, then all members of A, including X and Y, would agree on s. Hence $s \in \mathcal{E}(X) - \mathcal{E}(A)$, which contradicts the choice of t. □

Corollary 7.6. *If A is agreeable then all states of A are similar.*

This corollary will be called the **alternate Discriminate clause**.

Construction 7.7. We construct the *explore tree* ET whose nodes are sets of states closed under similarity. ET is ordered by inclusion. The root is the set $\mathcal{S}$ of all states.

If a node A is agreeable then, by the alternate Discriminate clause, all states in A are similar. Since A is closed under similarity, A is a similarity class. Similarity classes are the leaves of ET.

Suppose that a node A is not agreeable. Then the children of A are the maximal proper subsets B of A whose members agree on $\mathcal{B}(A)$. Obviously, each child B is closed under similarity. ◁

Along every branch of ET from the root to a leaf, nodes shrink but their explore sets grow. The intention is that the branches of ET reflect the paths that the algorithm's exploration takes in different states. In every state X, the algorithm explores $\mathcal{E}(\mathcal{S})$. If A is a non-leaf node in ET then, in any $X \in A$, the algorithm explores $\mathcal{E}(A)$ and then, depending on $\{[\![t]\!]_X : t \in \mathcal{B}(A)\}$, the algorithm explores $\mathcal{E}(B) - \mathcal{E}(A)$ for a child B of A.

Note that different nodes have different explore sets. Since all explore sets are subsets of the finite set Tags, the tree is finite.

Proposition 7.8. *Modify Postulate 3 by replacing the Discriminate clause with the alternate one. Then the original Discriminate clause is derivable.*

Proof. First, given an arbitrary state X, we construct an exploration order $<_X$ on $\mathcal{E}(X)$. Consider the branch in ET from the root to the leaf $[X]$, and define $s <_X t$ if the explore set of some node on this branch contains s but not t.

Second, we verify the original Discriminate clause. Suppose that $t \in (\mathcal{E}(X) - \mathcal{E}(Y))$ for some states X and Y. Then X and Y aren't similar and belong to different leaves of ET. Let A be the bifurcation node, so that A contains both X and Y but the two states belong to different children of A. Note that $t \notin \mathcal{E}(A)$ because $t \notin \mathcal{E}(Y)$. By the construction, there is a tag $s \in \mathcal{B}(A)$ such that $[\![s]\!]_X \neq [\![s]\!]_Y$. By the definition of the exploration order, $s <_X t$. □

8 ASM Rules

Let V be an algorithm-apt vocabulary. By default, terms are V terms and structures are algorithm-apt V structures. For each term t, let $\mathrm{Sub}(t)$ be the set of subterms of t, including t itself.

Definition 8.1 (Syntax)**.** We use induction to define *(ASM) rules* R of the fixed vocabulary V. The intention is that R is a one-step algorithm capable of working in any algorithm-apt structure.

- `skip` is a rule.
- An *assignment rule* has the form $f(t_1, \ldots, t_r) := t_0$
  where f is a programming variable, the terms $t_1, \ldots, t_r$ have the appropriate argument types for f, and the type of t_0 is the value type of f.
- A *parallel rule* has the form $R_1 \parallel R_2$
  where R_1, R_2 are rules.
- A *conditional rule* has the form `if` γ `then` R
  where γ is a Boolean term and R is a rule. ◁

Definition 8.2 (Status)**.** Let X be a structure. By induction on rule R we define whether X is normative for R.

- X is *normative* for `skip`.
- X is *normative* for an assignment $f(t_1, \ldots, t_r) := t_0$ if all $[\![t_i]\!]_X$ are defined.
- X is *normative* for $P \parallel Q$ if it is normative for both P and for Q and $\Delta(X, P) \cup \Delta(X, Q)$ is consistent.

- X is *normative* for a conditional $\texttt{if } \gamma \texttt{ then } Q$ if $[\![\gamma]\!]_X$ is defined and either $X \models \gamma$ and X is normative for Q or else $X \not\models \gamma$. ◁

Recall that $\overline{S}$ is the rounded version of a set S of terms.

Definition 8.3 (Exploration)**.** By induction on rule R we define the explore sets $\mathcal{E}(X, R)$ for R; here X is a normative structure for R.

- $\mathcal{E}(X, \texttt{skip}) = \overline{\varnothing} = \mathit{ImVal}$.
  (Rounding is superfluous here, but it is used everywhere to simplify exposition.)
- $\mathcal{E}(X, f(t_1, \ldots, t_r) := t_0) = \bigcup \left\{ \overline{\mathrm{Sub}(t_i)} : \ 0 \le i \le r \right\}$.
- $\mathcal{E}(X, P \parallel Q) = \mathcal{E}(X, P) \cup \mathcal{E}(X, Q)$.
- $\mathcal{E}(X, \texttt{if } \gamma \texttt{ then } Q) = \overline{\mathrm{Sub}(\gamma)} \cup \begin{cases} \mathcal{E}(X, Q) & \text{if } X \models \gamma, \\ \varnothing & \text{otherwise.} \end{cases}$ ◁

Definition 8.4 (Update sets)**.** By induction on rule R we define an update set $\Delta(X, R)$ for every structure X normative for R.

- $\Delta(X, \texttt{skip}) = \varnothing$.
- $\Delta(X, f(t_1, \ldots, t_r) := t_0) = \{\langle f, ([\![t_1]\!]_X, \ldots, [\![t_r]\!]_X), [\![t_0]\!]_X \rangle\}$.
- $\Delta(X, P \parallel Q) = \Delta(X, P) \cup \Delta(X, Q)$.
- $\Delta(X, \texttt{if } \gamma \texttt{ then } Q) = \begin{cases} \Delta(X, Q) & \text{if } X \models \gamma, \\ \varnothing & \text{otherwise.} \end{cases}$ ◁

Lemma 8.5. *The Determine clause holds for every rule R.*

Proof. Induction on R. □

Lemma 8.6. *The Discriminate clause holds for every rule R.*

Proof. By Proposition 7.8, it suffices to prove that, for every agreeable set S of structures normative for R, all states in S have the same explore set.

We prove the claim by induction on R. The cases of `skip` and assignment rules are trivial because any two normative structures have the same explore set.

Let $R = (P \parallel Q)$ and S a set of structures normative for R. that agree on the Boolean tags in

$$\begin{aligned} \mathcal{E}(S, P \parallel Q) &= \bigcap \{\mathcal{E}(X, P \parallel Q) : X \in S\} = \bigcap \{\mathcal{E}(X, P) \cup \mathcal{E}(X, Q) : X \in S\} \\ &\supseteq \bigcap \{\mathcal{E}(X, P) : X \in S\} = \mathcal{E}(S, P) \end{aligned}$$

By induction hypothesis, all S set have the same explore set relative to P. Similarly, they all have the same explore set relative to Q. It follows that they have the same explore set relative to R.

Let R be $\texttt{if } \gamma \texttt{ then } Q$, $t \in \mathcal{E}(X,R) - \mathcal{E}(Y,R)$, and S be a set of structures normative for R that agree on the Boolean tags in

$$\mathcal{E}(S,R) = \bigcap \{\mathcal{E}(X, \texttt{if } \gamma \texttt{ then } Q)\} = \bigcap \left\{ \mathrm{Sub}(\gamma) \cup \begin{cases} \mathcal{E}(X,Q) & \text{if } X \models \gamma, \\ \varnothing & \text{otherwise.} \end{cases} \right\}$$

In particular, they agree on γ. Two cases arise depending on they agree on γ being true or being false. Both cases are straightforward. Only one needs induction hypothesis. □

9 A generic ASM rule

In this section, we prove our Main Theorem which says essentially that every algorithm can be seen as iteration of a single ASM rule.

As before, $\mathcal{A}$ is a fixed algorithm. Throughout the section, by default, the vocabulary is $\mathrm{Voc}(\mathcal{A})$; in particular ASM rules are in the vocabulary $\mathrm{Voc}(\mathcal{A})$. Further, Tags and explore sets are as in Postulate 3, and states are those of $\mathcal{A}$.

Proposition 9.1. *For every similarity class σ of actionable states of $\mathcal{A}$, there is an ASM rule R_σ such that every $X \in \sigma$ is normative for R_σ, $\mathcal{E}(X, R_\sigma) \subseteq \mathcal{E}(X, \mathcal{A})$, and $\Delta(X, \mathcal{A}) = \Delta(X, R_\sigma)$.*

Proof. We start with an auxiliary construction which we call *point surgery*.

Point surgery. Given a structure X and a one-to-one correspondence $\mu_0 : S \to S'$ from a subset S of $|X|$ to a set S' disjoint from $|X|$, extend μ_0 to the map

$$\mu(x) = \begin{cases} \mu_0(x) & \text{if } x \in S, \\ x & \text{otherwise.} \end{cases}$$

from X to the structure X' of the same vocabulary obtained from X by replacing every $x \in S$ with $\mu_0(x)$. Every equality $f(x_1, \dots, x_n) = x_0$ is preserved in the sense that it becomes $f(\mu x_1, \dots, \mu x_n) = \mu x_0$ and no new such relationships arise. Thus, μ is an isomorphism. ◁

Lemma 9.2. *For every update $\langle f, (x_1, \dots, x_r), x_0 \rangle$ in the update set $\Delta(X, \mathcal{A})$ of a actionable state S there are tags $t_0, \dots, t_r$ in $\mathcal{E}(X)$ such that each $[\![t_i]\!]_X$ is x_i.*

Proof of Lemma. Suppose, toward a contradiction, that some x_i, say x_1, is not the value of any explored tag in X. Let y be an element outside of $|X|$. Use point surgery with $S = \{x_1\}$ and $S' = \{y\}$, and let μ and X' be as in point surgery above. Since $\mathcal{E}(X)$ contains all immediate terms, μ is immediacy compliant. By the Abstract State clause of Postulate 2, X' is a state. By Corollary 6.4(1), $\mathcal{E}(X) = \mathcal{E}(X')$. By the construction of X', μ is the identity of $\mathcal{E}(X)$. By the Determine clause of Postulate 3, $\Delta(X) = \Delta(X')$, so that $\langle f, (x_1, \ldots, x_r), x_0\rangle$ is in $\Delta(X')$ and therefore $x_1 \in |X'|$, which gives the desired contradiction. □

Lemma 9.3. *For every actionable state X of $\mathcal{A}$, there is an ASM rule R_X such that X is normative for R_X, $\mathcal{E}(X, R_X) \subseteq \mathcal{E}(X, \mathcal{A})$, and $\Delta(X, \mathcal{A}) = \Delta(X, R_X)$.*

Proof of Lemma. Let u be an update $\langle f, (x_1, \ldots, x_r), x_0\rangle$ in $\Delta(X, \mathcal{A})$. By Lemma 9.2, there are tags t_i such that every $[\![t_i]\!]_X = x_i$, so that every $[\![t_i]\!]_X$ is defined. Let Q_u be the assignment rule $f(t_1, \ldots, t_r) := t_0$. The desired R_X is $\|\{Q_u : u \in \Delta(X, \mathcal{A})\}$.

By construction, all terms in R_X are defined in X. By Defintion 8.2, R_X is normative. By Lemma 9.2, $\mathcal{E}(X, R_X) \subseteq \mathcal{E}(X, \mathcal{A})$. By the construction of R_X, if X is actionable for $\mathcal{A}$, then it is so for R_X and $\Delta(X, R_X) = \Delta(X, \mathcal{A})$. □

Lemma 9.4. *Suppose that $\Delta(Z, R_X) = \Delta(Z, \mathcal{A})$ for some state Z isomorphic to a state Y actionable for $\mathcal{A}$. Then $\Delta(Y, R_X) = \Delta(Y, \mathcal{A})$.*

Proof of Lemma. It is easy to check that an isomorphism $\mu : Y \to Z$ transforms $\Delta(Y, \mathcal{A})$ to $\Delta(Z, \mathcal{A})$ and $\Delta(Y, R_X)$ to $\Delta(Z, R_X)$. Since $\Delta(Z, R_X) = \Delta(Z, \mathcal{A})$, we have $\mu\Delta(Y, R_X) = \mu\Delta(Y, \mathcal{A})$. It remains to apply μ^{-1} to the last equality. □

Now we prove claims (P1) and (P3). It suffices to find a state Z isomorphic to Y with $\Delta(Z, R_X) = \Delta(Z, \mathcal{A})$. First we consider the case where $|Y|$ is disjoint from $|X|$. Use point surgery, to construct a state Z isomorphic to Y by replacing $[\![t]\!]_Y$ with $[\![t]\!]_X$ for all t in $\mathcal{E}(Y)$. Since all immediates are explored, the states X, Z have the same explore set and agree on it. By the Determine clause of Postulate 3 — relative to $\mathcal{A}$ — $\Delta(X, \mathcal{A}) = \Delta(Z, \mathcal{A})$. Since R_X explores only terms in $\mathcal{E}(Z, \mathcal{A})$ in Z, we have that $\Delta(X, R_X) = \Delta(Z, R_X)$. By Lemma 9.3, $\Delta(X, R_X) = \Delta(X, \mathcal{A}) = \Delta(Z, \mathcal{A})$. It follows that $\Delta(Z, R_X) = \Delta(Z, \mathcal{A})$.

Second we consider the general case. Point surgery allows us to construct a state Z isomorphic to Y by replacing every non-immediate value in Y that belongs to X with a fresh element. Since Z is isomorphic to Y, it is similar to Y and therefore to X. Since $|Z|$ is disjoint from $|X|$, $\Delta(Z, R_X) = \Delta(Z, \mathcal{A})$ by the first part of proof. □

Definition 9.5. An ASM rule π is *generic* for algorithm $\mathcal{A}$ if every actionable state X of $\mathcal{A}$ is normative for π, $\mathcal{E}(X, \pi) \subseteq \mathcal{E}(X, \mathcal{A})$, and $\Delta(X, \mathcal{A}) = \Delta(X, \pi)$. ◁

Theorem 9.6 (Main Theorem)**.** *For every algorithm $\mathcal{A}$, there is a generic ASM rule R for $\mathcal{A}$.*

Proof. Let the exploration tree ET be as in §7. We write $A \prec B$ if B is a child of A. The following definition generalize Definition 7.1(4).

Definition 9.7. Let A be a node in ET and $X \in A$. A tag $g \in \mathcal{B}(A)$ is a *guardian* for A if $X \models g$. The conjunction of the guardians is the *guard* G_A of A. ◁

Unfortunately the guard may not belong to $\mathcal{E}(A)$, and we can't use it in contexts restricted to tags in $\mathcal{E}(A)$. To this end, we introduce the following abbreviation. If $G = g_1 \wedge g_2 \wedge \cdots \wedge g_n$ and R an ASM rule, then "`If` G_A `then` R" abbreviates

$$\texttt{if}\ g_1\ \texttt{then}\ \texttt{if}\ g_2\ \texttt{then}\ \cdots \texttt{if}\ g_n\ \texttt{then}\ R$$

By induction on ET, from leaves down to the root, we define an ASM rule R_A for each node A.

$$R_A = \begin{cases} \texttt{If}\ G_A\ \texttt{then}\ R_\sigma & \text{if } A = \{\sigma\} \text{ and } \sigma \text{ is actionable,} \\ \texttt{||}\ \{\texttt{If}\ G_A\ \texttt{then}\ R_B :\ A \prec B\} & \text{otherwise} \end{cases}$$

Lemma 9.8. *Let A be a node in ET, and X an $\mathcal{A}$-actionable state in A. Then X is normative for R, $\mathcal{E}(X, R_A) \subseteq \mathcal{E}(X, \mathcal{A})$, and $\Delta(X, \mathcal{A}) = \Delta(X, R)_A$.*

Proof of Lemma. Induction on ET. If A is a leaf, the claim is Proposition 9.1. Otherwise A is the union its children in ET, and so X belongs to a child B of A. Note that the guardians of A are also those of B,. By induction hypothesis, $\mathcal{E}(X, R_A) \subseteq \mathcal{E}(X, R_B) \subseteq \mathcal{E}(X, \mathcal{A})$ and $\Delta(X, R_A) = \Delta(X, R_B) = \Delta(X, \mathcal{A})$. □

The desired generic rule π is the rule R_S for the root of ET. □

10 Representation Theorem, and termination

ASM programs are composed from ASM rules. ASM rules are ASM programs, the standard loop constructs may be used to produce ASM programs, and the sequential composition of ASM programs is an ASM program. But, for the representation theorem, we need a special kind of ASM programs. In our context, a representation theorem says that arbitrary algorithms can be step-by-step simulated by appropriate ASMs. To this end, we need to address the termination issue: While algorithms may have terminal states, ASM rules of §8 do not.

Definition 10.1. A *termination aware ASM rule* is an ASM rule R, as in §8, endowed with a set of terminal states closed under the similarity for R. ◁

Our Main Theorem, Theorem 9.6, can be tightened by requiring that the terminal states of $\mathcal{A}$ are exactly the terminal states of π. Just endow the rule constructed in the proof with the set of terminal states of $\mathcal{A}$.

Definition 10.2. ○ An *iterate ASM program* has the form `iterate` R where R is an ASM rule. ◁

○ An *iterate ASM* is given by basic components Voc, ImVal, $\mathcal{S}$, $\mathcal{I}$, and $\mathcal{T}$, as in Postulate 1, and by an iterate ASM program `iterate` R; its state transformer is $\tau(X) = X + \Delta(R)$. ◁

It is routine to check that iterate ASM programs comply with Postulates 1–3.

Definition 10.3. Let $\mathcal{A}$ be an algorithm with basic components Voc, ImVal, $\mathcal{S}$, $\mathcal{I}$, $\mathcal{T}$, and τ, as in Postulate 1, and with explore sets $\mathcal{E}(X, \mathcal{A})$. An algorithm $\mathcal{B}$ with explore sets $\mathcal{E}(X, \mathcal{B})$ *step-by-step simulates* $\mathcal{A}$ if $\mathcal{A}$ and $\mathcal{B}$ have the same basic components, normative states, and terminal states, and, in addition, $\mathcal{E}(X, \mathcal{B}) \subseteq \mathcal{E}(X, \mathcal{A})$. ◁

Theorem 10.4 (Representation Theorem)**.** *For every algorithm $\mathcal{A}$ with explore sets $\mathcal{E}(X, \mathcal{A})$, there is an iterate ASM $\mathcal{B}$ with explore sets $\mathcal{E}(X, \mathcal{B})$ that step-by-step simulates $\mathcal{A}$.*

Proof. Let π be a (termination aware version of a) generic ASM rule for the algorithm $\mathcal{A}$. The desired $\mathcal{B}$ is the ASM with program `iterate` π with the basic components of $\mathcal{A}$, where $\tau(X, \mathcal{B}) = X + \Delta(X, \pi)$, and with explore sets $\mathcal{E}(X, \pi)$. □

Q: Let me understand "`iterate` R" better. R is repeatedly executed. Forever?

A: Not necessarily. You may arrive at a terminal state. Alternatively the environment may stop you.

Q: I see. Termination may be internal or external. Can you make the internal termination condition explicit, like in a while loop?

A: Yes, but .... Recall Definition 7.1, and let H be the set of terminating similarity classes. Then $h = \bigvee \{G_\sigma : \sigma \in H\}$ is a termination condition. A normative state X is terminal if and only if $X \models h$. And the program could be written in the form "`until` h `do` R."

Q: What about that "but"?

A: It should be possible to check termination conditions in every normative state. But the explore sets of some states may not contain h. Due to possible partiality of basic functions, this is a serious problem. Another problem is that h is may be too large to be practical.

11 Example ASM programs

Notation and terminology 11.1.

- `if` γ `then` R_1 `else` R_2 abbreviates
  `if` γ `then` R_1 $\parallel$ `if` $\neg(\gamma)$ `then` R_2.
- Semantically, the binary operation $R_1 \parallel R_2$ is commutative, is associative, and has a unit: `skip`. By analogy with sum and product operations, we write

$$R_1 \parallel R_2 \parallel \cdots \parallel R_k \quad \text{and} \quad \Big\|\{R_i \colon i \in I\}$$

  where I is a finite index set and $\Big\|\{R_i \colon i \in \varnothing\} = \texttt{skip}$. ◁

Example 1, continuation

The program of Example 1 in §2 is already an ASM program where `a,b` are program variables, functions `mod` and `print` are plain, and `Input1, Input2` are oracles.

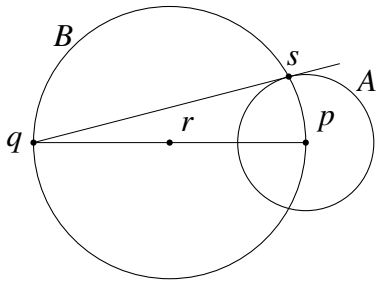
Example 2, continuation.

Introduce:

- Program variables r, s of type `point`, B of type `circle`, and T of type `line`.
- Deterministic functions $C(x, y)$, $L(x, y)$, and $M(x, y)$ such that if x, y are distinct points then $C(x, y)$ is the circle with center x through y, $L(x, y)$ the line through x and y, and $M(x, y)$ the midpoint between x and y.
- An oracle $I(A, B)$ which, given distinct intersecting circles A, B, chooses an intersection point.

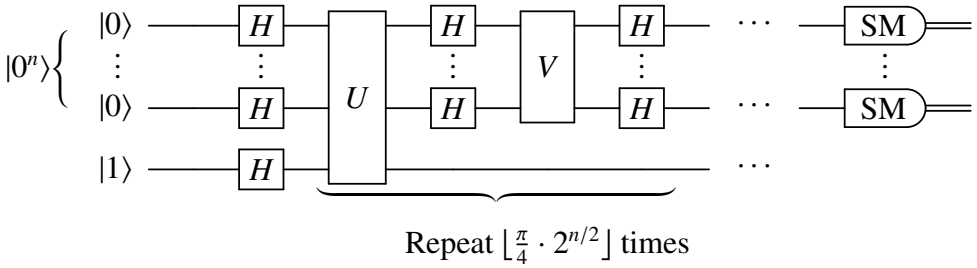
The desired ASM program:

$$\begin{aligned} r &:= M(p, q); \\ B &:= C(r, q); \\ s &:= I(A, B); \\ T &:= L(q, s) \end{aligned}$$



Here `circle`, `line`, and `point` are type symbols; B, r, s, and T are program variables; C and L are plain; and I, p, q are oracles.

Example 3, continuation. The diagram



specifies the algorithm but gives no indication how the gates H, U, V and SM work. In effect these gates are (operated by) oracles. Here is a corresponding ASM program:

```
if ⋀_{i=1}^{n+1} H1(i) then skip;
repeat ⌊(π/4) · 2^{n/2}⌋ times
    if U(1, 2, ..., n + 1) then if ⋀_{i=1}^{n} H1(i) then
    if V(1, 2, ..., n) then if ⋀_{i=1}^{n} H2(i) then skip;
‖_{i=1}^{n} Readout(i) := SM(i)
```

Each gate transforms the quantum state. All the quantum work is done by oracles. The algorithm just tells them what to do. In describing the types of oracle functions, we suppress the quantum part. In the case of unitary transformations, the only classical part to report is that the work has been done. To this end, we introduce the one-element type $\{\texttt{true}\}$, called T below. The measurement gates produce classical bits which — to avoid introducing an additional type — may be treated as Boolean values; in programming, `Bool` is often identified with $\{0, 1\}$.

The first and second columns of H gates are operated by an oracle managing function H_1 : `Integer` $\to T$. Since the second and third columns of H gates are executed within the same step, the third column of H gates is operated by another oracle managing function H_2 : `Integer` $\to T$. The gates U, V, and SM are operated by oracles managing functions of types $(\texttt{Integer})^{n+1} \to T$, $(\texttt{Integer})^n \to T$, and `Integer` $\to$ `Bool` respectively.

The conditionals make the oracles do their job. In the final step, we make n calls to the measurement oracle and collect the results in the Readout function.

References

[1] A. Blass, N. Dershowitz, and Y. Gurevich, "Exact exploration and hanging algorithms," arXiv:2410.10706, ext. abstract in *Springer LNCS* 6247 (2010) 140–154

[2] A. Blass and Y. Gurevich, "Ordinary interactive small-step algorithms," Parts I,II, and III, *ACM ToCL* 7:2 (2006) 363–419 and 8:3 (2007) articles 15 and 16.

[3] A. Blass, Y. Gurevich, D. Rosenzweig, and B. Rossman, "Interactive small-step algorithms", Parts I and II, *LMCS* 3:4 (2007) papers 3 and 4

[4] N. Dershowitz and Y. Gurevich, "A natural axiomatization of computability and proof of Church's Thesis," *BSL* 14:3 (2008) 299–350

[5] L. Grover, "A fast quantum mechanical algorithm for database search," *STOC* 1996

[6] Y. Gurevich, "Evolving algebra 1993: Lipari guide," in E. Börger (ed), "Specification and Validation Methods," Oxford U. Press, 9–36 (1995), also arXiv:1808.06255

[7] Y. Gurevich, "Sequential abstract state machines capture sequential algorithms," *ACM ToCL* 1:1 (2000) 77–111

[8] Y. Gurevich, "The nature of nondeterministic (probabilistic, quantum, etc.) algorithms," YuriFest 2025, `sites.google.com/view/yurifest2025/home`

[9] Y. Gurevich and A. Blass, "Software science view on quantum circuit algorithms," *Information and Computation* 292 (2023), article 105024, arXiv:2209.13731

[10] Y. Gurevich and A. Blass, "Assisted computations: Outer view" [tentative title]

[11] A. Kolmogorov, "On the concept of algorithm," *Uspekhi Mat. Nauk* VIII 4(56) (1953) 175–176, abstract, skipped in *Russian Mathematical Surveys*

[12] H. Rogers, "Theory of recursive functions and effective computability," McGraw-Hill 1967

[13] A. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proc. Lond. Math. Soc*, Ser. 2, vol. 42, parts 3 and 4 (1936), 230–265