The Logic in Computer Science Column

by

Yuri Gurevich

Computer Science and Engineering
University of Michigan, Ann Arbor, MI 48109, USA
gurevich@umich.edu

MEANS-FIT EFFECTIVITY

Yuri Gurevich

Abstract

Historically, the notion of effective algorithm is closely related to the Church-Turing thesis. But effectivity imposes no restriction on computation time or any other resource; in that sense, it is incompatible with engineering or physics. We propose a natural generalization of it, *means-fit effectivity*, which is effectivity relative to the (physical or abstract) underlying machinery of the algorithm. This machinery varies from one class of algorithms to another. Think for example of ruler-and-compass algorithms, arithmetical algorithms, and Blum-Shub-Smale algorithms. We believe that means-fit effectivity is meaningful and useful independently of the Church-Turing thesis. Means-fit effectivity is definable, at least in the theory of abstract state machines (ASMs). The definition elucidates original effectivity as well. Familiarity with the ASM theory is not assumed. We tried to make the paper self-contained.

1 Introduction

How can you prove the Church-Turing thesis? Here is one idea from our favorite logic text:

> "We get more evidence [for Church's thesis] if we try to define *calculable* directly. For simplicity, consider a unary calculable function F. It is reasonable to suppose that the calculation consists of writing expressions on a sheet of paper (or that it can be reduced to this). As will become clear in the next section, there is no loss of generality in supposing that the expressions written are numbers (more precisely, expressions which designate numbers). We therefore write $a_0, a_1, \ldots, a_n$, where a_0 is a and a_n is $F(a)$. Now the decision method tells us how to derive a_i from $a_0, \ldots, a_{i-1}$ or, equivalently, from $\langle a_0, \ldots, a_{i-1}\rangle$. Hence there is a calculable function G such that $G(\langle a_0, \ldots, a_{i-1}\rangle) = a_i$. The decision method also tells us when the computation is complete; so there is a calculable predicate P such that $P(\langle a_0, \ldots, a_i\rangle)$ is false for $i < n$ and true for $i = n$.
>
> Our attempt to define calculability thus ends in circularity, since G and P must be assumed to be calculable. However, since G describes a single step in the calculation, it must be a very simple calculable function; and the same applies to P. We

> can therefore expect, on the basis of other evidence for Church's thesis, that G and P will be recursive. If we assume this, we can prove that F is recursive (Shoenfield, [18, §6.5]). ◁

Shoenfield's idea[1] has been realized. We explain this below. But first let's recall the thesis: Every effective numerical function is (partial) recursive or, equivalently, Turing computable. Here *numerical functions* are partial functions $y = f(x_1, \ldots, x_r)$ where the arguments x_i range over natural numbers and the values y, if defined, are natural numbers. A numerical function is effective if it is computable by an effective algorithm.

The notion of effectivity is famously elusive. In this paper, we propose a more general notion, *means-fit effectivity*. It seems to us useful independently of the Church-Turing thesis, and it elucidates the original notion of effectivity as well.

But let us explain all this in an orderly fashion, starting with an important reservation: This study is restricted to sequential algorithms, a.k.a. classical or traditional, the algorithms of the historical Church-Turing thesis. A sequential algorithm is deterministic. Its computations are finite or infinite sequences of steps. And the computation steps are of bounded complexity. (This last property, observed by Kolmogorov in [15], rules out massive parallelism.)

Thesis-related literature, including Shoenfield's 1967 logic text [18], was virtually restricted to sequential algorithms during the first few decades after the formulation of the thesis. One may be tempted to analyze all algorithms, but this is virtually impossible because the general notion of algorithm has not matured; it is evolving and the evolution may never stop [14].

Proviso 1.1. By default, algorithms are sequential in the rest of this article. ◁

This study is enabled by the axiomatization of sequential algorithms in [13], specifically by the representation theorem in [13, §6], according to which every sequential algorithm is behaviorally identical to a (sequential) abstract state machine (an ASM for short). Since the only aspect of algorithms that we are interested in is their behavior, we work with ASMs and call them algorithms[2].

It is the representation theorem of [13] that realizes Shoenfield's idea, but the realization does not solve the problem of characterizing effectivity. Why not? Well, Shoenfield assumed that initially we have only input. In fact, we have also some basic operations available *and nothing else*. This "nothing else" is in essence the missing ingredient. Adding it to the axiomatization of [13] allows one to derive the Church-Turing thesis in its core setting [12]; we touch on this issue in §8.

[1] Actually we don't know whose idea it is; Shoenfield doesn't quote sources in the textbook.

[2] The axiomatization of [13] is somewhat refined in [2], but the representation theorem remains valid. Much of the analysis in [13] and in this study can be generalized to other species of algorithms which have been axiomatized, e.g., to synchronous parallel algorithms [3] and interactive algorithms [4].

Why did the axiomatization in [13] allow ineffective algorithms? One reason is that we were interested primarily in engineering applications. While tidy initial states are natural in theoretical study, there may be nothing tidy about the initial states of some engineering algorithms. Those initial states might have been prepared — and messed up — by various processes. Besides, abstracting from resource usage, inherent in the notion of effectivity, is incompatible with engineering. But there is something else which is important in applications and is also relevant to our current story.

The realm of not-necessarily-effective algorithms is more natural. You use whatever tools are — in reality or by convenient abstraction — available to you. Turing's idealized human agent uses pen and paper: "Computing is normally done by writing certain symbols on paper" [19, §9]. Ruler and compass don't fit this description but they had been used in antiquity. In real-time engineering applications and in some theoretical computation models, like BSS [5], you work with genuine reals. Hence the interest in tool dependent, or means dependent effectivity.

To formalize the notion of means-fit effectivity, we use the ASM theory. The states of an abstract state machine are first-order structures with static and dynamic basic functions, where the dynamic functions play the role which is normally played by program variables in programming languages. Without loss of generality (as we show in §3), the static functions and input variables carry all the initial information.

The key idea of this study is a classification, in §5, of static functions into intrinsic and extrinsic. As far as an algorithm is concerned, all its static functions look like oracles, but the intrinsic functions are built-in functions provided reliably by the underlying machinery of the algorithm or, more abstractly, by the (in general compound) datastructure of the algorithm. All intrinsic functions are effective relative to the datastructure. On the other hand, extrinsic functions are provided by outside entities with no guarantee of reliability in general. Of course some or all extrinsic functions may be effective as well but such effective functions can be pruned off, as we prove in §7. (Pruning Theorem 7.2 is our main technical result.) Accordingly, we define that an algorithm is *means-fit effective* or *effective relative to its datastructure* if it has no extrinsic functions.

What relevance does this have to the Church-Turing thesis? Well, let's consider how the thesis could possibly be falsified. One way is mathematical. Construct an effective numerical function which is not recursive, as Ackermann (and independently Sudan) constructed a recursive numerical function which is not primitive recursive. Another way may be called physical. Find new means, possibly using new discoveries of physics, which would allow you to effectively compute a numerical function that is not recursive. To us, the physical version makes little sense. The abstraction from resources is incompatible with physics. In any case, predicting the future of physics is beyond the scope of this paper.

The historical Church-Turing thesis was mathematical, and it is the historical thesis that we discuss here and in [12]. One particular datastructure (using our current terminology) was in the center of attention historically. It was the arithmetic of natural numbers; see for example the quotation from Shoenfield's textbook above. Call algorithms with that datastructure *arithmetical*. We believe that the historical thesis can be formulated thus: if a numerical function is computed by an effective numerical algorithm then it is partial recursive. This form of the thesis has been derived in [12] from the axioms of [13] plus an initial-state axiom according to which, initially, the dynamic functions of the algorithm — with the exception of input variables — are uninformative. The current paper provides additional justification for this initial-state axiom. We return to this discussion in §8 where we discuss other related work as well.

Acknowledgments

I am extremely grateful to Andreas Blass who provided useful comments on all aspects of this paper, from high-level ideas to low-level details of exposition including definite/indefinite articles absent in my native Russian. (If you know all rules about the articles, explain this: the flu, a cold, influenza.) I am also very grateful to Udi Boker, Patrick Cegielski, Julien Cervelle, Nachum Dershowitz, Serge Grigorieff and Wolfgang Reisig for most useful comments on short notice.

2 Abstract state machines

To make this paper self-contained and introduce terminology, we recall some basic notions of first-order logic in the form appropriate to our purposes.

2.1 First-order structures

A *vocabulary* is a finite collection of function symbols, each of a fixed arity. Some function symbols may be marked as *relational* and called *relations*. Some function symbols may be marked as *static*; the other function names are *dynamic*.

Two vocabularies are *consistent* if any symbol that belongs to both vocabularies has the same arity and markings in both vocabularies.

Convention 2.1 (Vocabularies)**.**

1. Every vocabulary contains the following *logic symbols:* the equality sign, nullary symbols `true`, `false`, and `nil`, the standard Boolean connectives, and a ternary function symbol `ITE` (read if-then-else)[3].

2. All logic symbols, except `nil` and `ITE`, are relational.

3. All logic symbols are static. Nullary dynamic symbols are *elementary variables*. Relational elementary variables are *Boolean variables*. ◁

A *(first-order) structure* X of vocabulary Υ is a nonempty set $|X|$, the *base set* of X, together with *basic functions* f_X where f ranges over Υ. (The subscript X will be often omitted.) If f is r-ary then f_X is a function, possibly partial, from $|X|^r \to |X|$.

Convention 2.2 (Structures)**.**

1. The *nonlogic vocabulary* of a structure X is the vocabulary of X minus all the logic symbols.

2. By default, basic functions are total. This guideline will save us space. Instead of indicating totality in most cases, we will indicate partiality in just a few cases.

3. In every structure, `true` and `false` and `nil` are defined and distinct.

4. Every (defined) value of every basic relation is either `true` or `false`. The equality sign and the standard Boolean operations have their usual meaning.

5. `ITE`(x, y, z) is y if $x =$ `true`, is z if $x =$ `false`, and is `nil` otherwise. ◁

Remark 2.3. In constructive mathematics, (constructive) real numbers are represented by algorithms. As a result, the equality of (thus represented) real numbers becomes partial. We took a similar position in [2]. But one may want to distinguish between genuine reals and their representations and to keep the equality of genuine reals total.

Here `nil` is an error value of sorts; the first argument of `ITE` is normally Boolean. `nil` replaces `undef` of [13] to emphasize the difference between default/error values and the absence of any value.

Terms of vocabulary Υ are defined by induction: If f is an r-ary symbol and $t_1, \dots, t_r$ are terms then $f(t_1, \dots, t_r)$ is a term. (Here the case $r = 0$ is the basis of induction.) A term $f(\bar{t})$ is *Boolean* if f is relational.

The value of an Υ term in a Υ structure X is defined by induction:

$$\mathcal{V}_X f(t_1, \dots, t_r) = f_X(\mathcal{V}_X t_1, \dots, \mathcal{V}_X t_r).$$

[3] `ITE` is a novelty introduced here, though we intended to do that already for a while. It is used below in §3.

By default, to evaluate $f(t_1, \ldots, t_r)$, evaluate the terms t_i first. But $\texttt{ITE}\,(t_1, t_2, t_3)$ is an exception: Evaluate t_1 first. f the result is `true` then evaluate t_2 but not t_3, and if the result is `false` then evaluate t_3 but not t_2; otherwise evaluate neither t_2 nor t_3.

If X_1, X_2 are structures of vocabularies Υ_1, Υ_2 respectively, and t_1, t_2 are terms of vocabularies Υ_1, Υ_2 respectively, then $\mathcal{V}_{X_1}(t_1) = \mathcal{V}_{X_2}(t_2)$ means that either both sides are defined and have the same value (so that the two structures have common elements) or else neither side is defined.

The rest of this subsection is devoted to introduction of the union of structures. Call two structures are *consistent* if their vocabularies are consistent and the following condition holds for every joint function symbol f. If r is the arity of f and elements $x_1, \ldots, x_r$ belong to both structures then $f(x_1, \ldots, x_r)$ is the same in both structures.

Definition 2.4 (Union)**.** Let $X_1, \ldots, X_N$ be pairwise consistent structures with the same logic elements. The *union* $X_1 \cup X_1 \cup \cdots \cup X_N$ of the structures X_i is the structure X such that

1. the vocabulary of X is the union of the vocabularies of $X_1, \ldots, X_N$,
2. the base set of X is the union of the base sets of $X_1, \ldots, X_N$, and
3. if f is an r-ary basic function of X_i then $\mathcal{V}_X(f(x_1, \ldots, x_r))$ is the default value unless all r elements $x_1, \ldots, x_r$ belong to X_i, in which case $\mathcal{V}_X(f(x_1, \ldots, x_r)) = \mathcal{V}_{X_i}(f(x_1, \ldots, x_r))$. ◁

Notice that different structures X_i may share nonlogic elements, in which case the union is not a disjoint union modulo the logic.

2.2 Abstract state machines: Definition

Traditionally, in logic, structures are static, but we will use structures as states of algorithms. Let X be a structure of vocabulary Υ. A *location* ℓ of X is a pair $(f, \bar{x})$ where $f \in \Upsilon$, f is dynamic, $\bar{x} \in |X|^r$ and r is the arity of f. The *content* of location ℓ is $f_X(\bar{x})$. An *(atomic) update* of X is a pair (ℓ, y) where ℓ is a location $(f, \bar{x})$ and $y \in |X|$. To execute the update (ℓ, y) means to replace the current content of ℓ with y, that is to set $f_X(\bar{x})$ to y. This produces a new structure. Updates (ℓ_1, y_1) and (ℓ_2, y_2) are *contradictory* if $\ell_1 = \ell_2$ but $y_1 \neq y_2$; otherwise the updates are *consistent*.

Definition 2.5 (Rules)**.** *Rules* of vocabulary Υ are defined by induction.

1. An *assignment* has the form $f(t_1, \ldots, t_r) := t_0$ where
   $f \in \Upsilon$, f is dynamic, r is the arity of f, and $t_0, \ldots, t_r$ are Υ terms.

2. A *conditional rule* has the form $\texttt{if}\ \beta\ \texttt{then}\ R_1\ \texttt{else}\ R_2$ where β is a Boolean Υ term and R_1, R_2 are Υ rules.

3. A *parallel rule* has the form $R_1 \parallel R_2$ where R_1, R_2 are Υ rules. ◁

Semantics of rules. A successful execution of an Υ rule R at an Υ structure X produces a pairwise consistent finite set $\{(\ell_1, y_1), \ldots, (\ell_n, y_n)\}$ of updates and thus results in a new state X', obtained from X by executing these updates. An assignment $f(t_1, \ldots, t_r) := t_0$ produces a single update $(\ell, \mathcal{V}_X(t_0))$ where $\ell = \big(f, (\mathcal{V}_X(t_1), \ldots, \mathcal{V}_X(t_r))\big)$. A conditional rule $\texttt{if}\ \beta\ \texttt{then}\ R_1\ \texttt{else}\ R_2$ produces the update set of R_1 if $\beta = \texttt{true}$ in X and the update set of R_2 if $\beta = \texttt{false}$. A parallel rule $R_1 \parallel R_2$ produces the union of the update sets of R_1 and R_2.

Notice that the execution of the assignment in a given state X does not require the evaluation of the term $f(t_1, \ldots, t_r)$. Let $x_i = \mathcal{V}_X(t_i)$ for $i = 0, \ldots, r$. If $f(x_1, \ldots, x_r)$ is undefined but x_0 is defined than $f(x_1, \ldots, x_r) = x_0$ after the assignment.

Definition 2.6. A (sequential) ASM A of vocabulary Υ is given by a *program* and *initial states*. The program is a rule of vocabulary Υ. Initial states are Υ structures. The set of initial states is nonempty and closed under isomorphisms. ◁

A *computation* of A is a finite or infinite sequence $X_0, X_1, X_2, \ldots$ of Υ structures where X_0 is an initial state of A and where every X_{i+1} is obtained by executing Π at X_i. A *(reachable) state* of A is an Υ structure that occurs in some computation of A.

As we mentioned in §1, every (sequential) algorithm A is behaviorally identical to some (sequential) ASM B; they have the same initial states and the same state-transition function.

Proviso 2.7. By default, algorithms are abstract state machines in the rest of this article.

For future use, we formulate the following obvious observation.

Observation 2.8 (Failure)**.** There are two scenarios that an algorithm A fails at a given state X. One is that the algorithm attempts to evaluate a basic partial function f at an input where f is undefined. The other reason is that the program of A produces contradictory updates of some basic function f.

3 Separating static and dynamic

3.1 The task of an algorithm

In logic, traditionally, algorithms compute functions. But in the real world, algorithms perform many other tasks and may be intentionally non-terminating. Here,

for example and future reference, is a very simple task where a variable signals some undesirable condition.

Task 3.1. Keep watching a Boolean variable b. Whenever it becomes true, issue an error message, set b to `false`, and resume watching it. ◁

To simplify the exposition, we impose the following proviso.

Proviso 3.2. By default, in the rest of this article, the task of an algorithm is to compute a function.

The function computed by an algorithm will be called its *objective function*.

Convention 3.3 (Input and output)**.** If an algorithm A computes an r-ary objective function F, then it has *r input variables* and one *output variable*. These are elementary variables designated to hold the input and output values of F. All of the initial states of A are isomorphic except for the values of the input variables. The initial value of the output variable is the default value. ◁

3.2 Making dynamic functions initially uninformative

Every dynamic function f has a default value. The generic default value is `nil` but, if f is relational, then the default value of f is `false`.

Definition 3.4. A dynamic function f of an algorithm A is *uninformative* in a state X of A if all its values in X are the default value of f. Function f is *initially uninformative* (for algorithm A) if it is uninformative in every initial state of A. ◁

The definition of ASMs allows a dynamic function to differ from the default at infinitely many arguments in an initial state and even to be partial there. One might reasonably stipulate that, initially, every dynamic function f is (total and) uninformative, unless it is an input variable. Rather than stipulating, however, we can arrive at this desirable situation by a simple transformation of any given algorithm. If the initial configuration of f is preserved as a static function s, then the changes made to f can be tracked by a dynamic function d with the help of a dynamic relation δ indicating the arguments where f has been updated.

Lemma 3.5. *Let A be an algorithm of vocabulary $\Upsilon \cup \{f\}$ where f is dynamic, not in Υ, and different from the input and output variables. Let s, d, δ be fresh function symbols of the arity of f where s is static, d and δ are dynamic, and δ is relational. There is an algorithm B of vocabulary $\Upsilon \cup \{s, d, \delta\}$ satisfying the following requirements.*

1. *The initial states of B are obtained from those of A by renaming f to s and introducing uninformative d, δ.*

2. *Every computation $Y_0, Y_1, \ldots, Y_j$ of B is obtained from a computation $X_0, X_1, \ldots, X_j$ of A in such a way that the following claims hold where for brevity $X = X_j$ and $Y = Y_j$.*

   (a) $\mathtt{ITE}\,(\delta_Y(\bar{x}), d_Y(\bar{x}), s_Y(\bar{x})) = f_X(\bar{x})$.

   (b) Every $g \in \Upsilon$ has the same interpretation and is evaluated at exactly the same arguments in X and in Y.

   (c) For every term t in the program of A, we have $\mathcal{V}_X(t) = \mathcal{V}_Y(\tilde{t})$ where $\tilde{t}$ is the result of replacing[4] the subterms $f(t')$ of t with terms $\mathtt{ITE}\,(\delta_Y(t'), d_Y(t'), s_Y(t'))$.

   (d) The arguments where s_Y is evaluated are exactly the arguments where f_X is evaluated and where f has not been updated yet.

   (e) B fails at Y if and only if A fails at X.

3. *B computes the objective function of A.* ◁

Proof. To simplify notation, we assume that f is unary and we write $f(x)$ rather than $f(\bar{x})$. The generalization to the case of arbitrary arity will be obvious. Let Π be the program of A. The desired algorithm B is obtained from A in a simple and effective way. The initial states of B are defined by requirement 1, and the program Σ of B is obtained from Π in two stages. Recall how the terms t of Π are transformed into terms $\tilde{t}$ in claim 2(c) of the lemma.

Stage 1 For every term $f(t)$ in Π, substitute $\mathtt{ITE}\,(\delta(t), d(t), s(t))$ for every occurrence of $f(t)$ where $f(t)$ isn't the left side of an assignment. Let $\tilde{\Pi}$ be the resulting program.

Stage 2 Replace every assignment $f(\tilde{t}) := \tilde{\tau}$ in $\tilde{\Pi}$ with parallel rule
$d(\tilde{t}) := \tilde{\tau} \parallel \delta(\tilde{t}) := \mathtt{true}$.

It remains to prove that B works as intended. Since requirement 1 holds by construction and requirement 3 follows from requirement 2, it suffices to prove requirement 2.

By induction on j, we prove claims (a)–(c). Assume that the claims have been proved for all $i < j$. Notice that the induction hypothesis and (a) imply (b) and (c). To prove (a), we consider two cases.

Case 1: $\delta_Y(x) = \mathtt{true}$. Then there is a positive integer $i < j$ such that, at step $i+1$, Σ executes a rule ($d(\tilde{t}) := \tilde{\tau} \parallel \delta(\tilde{t}) := \mathtt{true}$) for some $\tilde{t}$ and $\tilde{\tau}$ with $\mathcal{V}_{Y_i}(\tilde{t}) = x$.

[4] We have not specified the order in which replacements are done. It is more efficient to use a bottom-up strategy, so that if $f(t_1)$, $f(t_2)$ are subterms of t and $f(t_1)$ is a subterm of t_2 then deal with $f(t_1)$ before $f(t_2)$. But the result does not depend on the order of replacements.

There may be several triples $(i, \tilde{t}, \tilde{t}')$ fitting the bill; in such a case, fix such a triple with i as big as possible.

By the construction of Σ, the rule $(d(\tilde{t}) := \tilde{\tau} \parallel \delta(\tilde{t}) := \texttt{true})$ replaces an assignment $f(\tilde{t}) := \tilde{\tau}$ in $\tilde{\Pi}$ which, in its turn, replaces an assignment $f(t) := \tau$ in Π. By the induction hypothesis, $\mathcal{V}_{X_i}(t) = \mathcal{V}_{Y_i}(\tilde{t}) = x$ and $\mathcal{V}_{X_i}(\tau) = \mathcal{V}_{Y_i}(\tilde{\tau})$. By the choice of i, we have

$$\begin{aligned}&\texttt{ITE}\,(\delta_Y(x), d_Y(x), s_Y(x)) = d_Y(x) = d_{Y_{i+1}}(x) = \mathcal{V}_{Y_{i+1}}(d(\tilde{t})) = \\ &\mathcal{V}_{Y_i}(\tilde{\tau}) = \mathcal{V}_{X_i}(\tau) = \mathcal{V}_{X_{i+1}}(f(t)) = f_{X_{i+1}}(x) = f_X(x).\end{aligned}$$

Case 2: $\delta_Y(x) = \texttt{false}$. It suffices to prove that program Π does not update f_X at x during the j-step computation, because then we have

$$\texttt{ITE}\,(\delta_Y(x), d_Y(x), s_Y(x)) = s_Y(x) = s_{Y_0}(x) = f_{X_0}(x) = f_X(x).$$

Suppose toward a contradiction that an assignment subprogram $f(t) := \tau$ of Π is executed at step $i \leq j$. But then a subprogram $(d(\tilde{t}) := \tilde{\tau} \parallel \delta(\tilde{t}) := \texttt{true})$ of Σ is executed at stage i. By the induction hypothesis, $\mathcal{V}_{Y_i}(\tilde{t}) = \mathcal{V}_{X_i}(t) = x$ and therefore $\texttt{true} = \delta_{Y_{i+1}}(x) = \delta_Y(x)$ which contradicts the case hypothesis.

This concludes the proof of claims (a)–(c). They imply the following auxiliary claim.

(c') f has been updated at x if and only if $\delta_Y(x) = \texttt{true}$.

Claim (d). Taking claims (a)–(c') into account, we have

$$\begin{aligned}&f(x) \text{ is evaluated in } X \text{ and } f(x) \text{ has not been updated}\\ \iff\ &A \text{ evaluates } f_X(t) \text{ for some } t \text{ in } \Pi \text{ with } \mathcal{V}_X(t) = x\\ &\quad\text{and } f(x) \text{ has not been updated}\\ \iff\ &B \text{ evaluates } f_X(\tilde{t}) \text{ for some } t \text{ in } \Pi \text{ with } \mathcal{V}_X(t) = x\\ &\quad\text{and } \delta(x) = \texttt{false}\\ \iff\ &s(x) \text{ is evaluated in } Y\end{aligned}$$

Claim (e). There are two failure scenarios, and we consider them in turn.

Scenario 1: Evaluating a basic function where it is undefined. For some term t in Π, we have

$$\begin{aligned}&A \text{ fails at } X\\ \iff\ &\Pi \text{ attempts to evaluate undefined } \mathcal{V}_X(t)\\ \iff\ &\Sigma \text{ attempts to evaluate undefined } \mathcal{V}_Y(\tilde{t})\\ \iff\ &B \text{ fails at } Y.\end{aligned}$$

Scenario 2: Producing contradicting updates for some basic function.

Case 1: The basic function in question is an Υ function g. To simplify notation we assume that g is unary. For some terms t_1, t_2, τ_1, τ_2 in Π, we have

$$
\begin{aligned}
& A \text{ fails at } X \\
\Longleftrightarrow\ & \Pi \text{ attempts to execute } g(t_1) := \tau_1 \text{ and } g(t_2) := \tau_2 \text{ in X,} \\
& \quad \text{where } \mathcal{V}_X(t_1) = \mathcal{V}_X(t_2) \text{ but } \mathcal{V}_X(\tau_1) \neq \mathcal{V}_X(\tau_2), \\
\Longleftrightarrow\ & \Sigma \text{ attempts to execute } g(\tilde{t}_1) := \tilde{\tau}_1 \text{ and } g(\tilde{t}_2) := \tilde{\tau}_2 \text{ in X,} \\
& \quad \text{where } \mathcal{V}_Y(\tilde{t}_1) = \mathcal{V}_X(t_1) = \mathcal{V}_X(t_2) = \mathcal{V}_Y(\tilde{t}_2) \\
& \quad \text{but } \mathcal{V}_Y(\tilde{\tau}_1) = V_X(\tau_1) \neq \mathcal{V}_X(\tau_2) = \mathcal{V}_Y(\tilde{\tau}_2), \\
\Longleftrightarrow\ & B \text{ fails at } Y
\end{aligned}
$$

Case 2: The basic function in question is not an Υ function. Then, it must be f in the case of A, and it must be d in the case of Σ. While δ is also dynamic, the only value assigned to δ is `true`. The rest is as in case 1 except that, in the equivalence chain, g is replaced with f in the second item and with d in the third. □

Theorem 3.6. *Let $f_1, f_2, \ldots$ be all of the dynamic functions of an algorithm A excluding the input and output variables. There is an algorithm B, satisfying the following requirements.*

1. *The vocabulary of B is obtained from that of A by replacing dynamic symbols f_m with fresh static symbols s_m of the same arity and by adding some dynamic symbols.*

2. *The initial states of B are obtained from those of A by renaming every f_m to s_m and making all the new dynamic functions uninformative.*

3. *Every computation $Y_0, Y_1, \ldots$ of B is obtained from a computation $X_0, X_1, \ldots$ of A in such a way that*

   (a) *s_m is evaluated at $\bar{x}$ in Y_j if and only if f_m is evaluated at $\bar{x}$ in X_j and f_m has not been updated at $\bar{x}$,*

   (b) *for any static function g of A, $g_{Y_j} = g_{X_j}$ and g is evaluated at exactly the same arguments in Y_j and in X_j, and*

   (c) *B fails at Y_j if and only if A fails at X_j.*

4. *B computes the objective function of A.*

Proof. Use Lemma 3.5 to replace every f_m with s_m and dynamic bookkeeping functions d_m and δ_m. □

The theorem allows us to impose the following proviso without loss of generality.

Proviso 3.7. Below, by default, the non-input dynamic functions of any algorithm are initially uninformative. ◁

Remark 3.8. One generalization of the theorem is related to the task performed by the given algorithm A. It does not have to be computing a function. It could be any other reasonable task, e.g. Task 3.1. ◁

4 Effectivity: Intuition and tool dependence

Effective algorithms are also known by names like effective procedures and mechanical methods.

4.1 Intuition

The notion of effective algorithm has been informal and intuitive. Here is an explanation of it from an influential book on recursive functions and effective computability:

> "Several features of the informal notion of algorithm appear to be essential. We describe them in approximate and intuitive terms.
>
> *1. An [effective] algorithm is given as a set of instructions of finite size. (Any classical mathematical algorithm, for example, can be described in a finite number of English words.)
>
> *2. There is a computing agent, usually human, which can react to the instructions and carry out the computations.
>
> *3. There are facilities for making, storing, and retrieving steps in a computation.
>
> *4. Let P be a set of instructions as in *1 and L be a computing agent as in *2. Then L reacts to P in such a way that, for any given input, the computation is carried out in a discrete stepwise fashion, without use of continuous methods or analogue devices.
>
> *5. L reacts to P in such a way that a computation is carried forward deterministically, without resort to random methods or devices, e.g., dice.
>
> Virtually all mathematicians would agree that features *1 to *5, although inexactly stated, are inherent in the idea of algorithm" (Hartley Rogers [17, §1.1]). ◁

A numerical function is effective if there is an effective algorithm that computes the function. Everybody agrees that partial recursive functions on natural numbers are effectively computable.

4.2 Relevance

There is an important property of effective algorithms that Rogers did not emphasize: There are no restrictions on resources. The computing agent does not run out of time, out of paper, etc. This makes effectivity incompatible with engineering or physics. How is it relevant today?

It often happens in mathematics that it is easier to prove a stronger statement, especially if the stronger statement is cleaner. It is indeed often easier to prove a computational problem ineffective (if we accept the Church-Turing thesis) than to prove that it is not solvable given such and such resources.

There is also, for what it's worth, historical interest in effectivity. But there is something else. Notice that the notion of effectivity readily generalizes to effectivity relative to a given oracle or oracles. There is a good reason for that. Let's have a closer look at item *2 in the quote above. There is something implicit there which we want to make explicit. The computing agent should be able to "react to the instructions." But that ability of the agent depends on the available tools, doesn't it? Working with ruler and compass is different from working with pen and paper. A personal computer may make you more productive than pen and paper.

This leads us to the notion of effectivity relative to the available tools. This more general notion seems to us more interesting and more relevant today.

5 Means-fit effectivity

Any static function of an algorithm is essentially an oracle as far as the algorithm is concerned. Upon invoking/querying a static function on some input, the algorithm waits until, if ever, the oracle provides a reply. If the oracle does not reply then the algorithm is stuck forever. But not all static functions are equally oracular.

5.1 Intrinsic and extrinsic

In the real world, some static functions are provided by the underlying machinery of the algorithm. They are part of the normal functionality of the computer system of the algorithm; in that sense they are built-in. These functions are, or at least are supposed to be, provided in a reliable and prompt way. When the algorithm

invokes such a built-in function, it gets a value. It may be an error message, if for example the algorithm attempts to divide by 0, but it is a value nevertheless.

Of course, in the real world, things may get more complicated. Much of the functionality of your computer system may be provided via the Internet, and you may lose connection to the Internet. Even if you work offline, your computer system may malfunction.

Here we abstract away from such engineering concerns but we retain the important distinction between the built-in static functions, which we call *intrinsic*, and the other static functions which we call *extrinsic*. To this end, we stipulate that the vocabulary of an algorithm indicates which static symbols are intrinsic. As far as the underlying machinery is concerned, the important part for our purposes is what functions are provided rather than how they are computed. Accordingly, we abstract the underlying machinery of an algorithm to the (in general compound) *datastructure* of the algorithm; see §5.2 for details.

As far as a given algorithm is concerned, extrinsic functions are provided by the unknown world which is not guaranteed to be prompt or reliable. They are genuine oracles. (In the real world, the extrinsic functions may be also supported by reliable computer systems; we address this in §7).) Let us see some examples.

Example 5.1 (Ruler and compass)**.** In the historically important realm of ruler-and-compass algorithms, the underlying machinery includes (unmarked) ruler and compass. These algorithms are not effective in the Church-Turing sense because of the analog, continuous nature of their basic operations, but they had been practical in antiquity. (They also admit limited nondeterminism but they can be made deterministic by means of some simple choice functions.) ◁

Example 5.2 (Idealized human)**.** An algorithm can be executed by a human being. Viewing an (idealized) human as the underlying machinery of an algorithm is a key idea in Turing's celebrated analysis [19]. ◁

The datastructure of natural numbers $0, 1, 2, \ldots$, with the standard operations will be called *natural-numbers arithmetic* or simply *arithmetic*. The exact set of standard operations does not matter as long as we have (directly or via programming) 0 and the successor operation. We will assume here that the standard operations are zero, successor operation and predecessor operation.

Example 5.3 (Recursion theory)**.** Natural-numbers arithmetic is the datastructure of traditional recursion theory, studying partial recursive functions [17]. ◁

Example 5.4 (Turing machines)**.** Consider Turing machines with a single tape which is one-way infinite and has only finitely many non-blank symbols. The datastructure of such machines is composed of three finite datatypes — control states, tape symbols and movement directions — and one infinite datatype of tape

cells. All by itself, the infinite datatype is isomorphic to the natural-numbers arithmetic. ◁

Example 5.5 (Programming language)**.** A modern programming language may involve a number of datastructures. We see them all as parts of one compound datastructure which is the datastructure of any algorithm written in the programming language. The algorithm may also query some additional functions online. These outside functions would be extrinsic. ◁

Example 5.6 (Random access machines)**.** Real-world computers are too messy for many theoretical purposes. This led to an abstract computation model called Random Access Machine, in short RAM [10], whose datastructure is more involved than arithmetic but can be encoded in arithmetic. ◁

Example 5.7 (BSS machines)**.** The datastructure of Blum-Shub-Smale machines, also known as BSS machines [5], involves natural numbers and genuine reals. BSS machines are able to compute functions over the reals which, for the obvious reason, cannot be computed by Turing machines. But the numerical functions computed by BSS machines are partial recursive. ◁

5.2 Datastructures

In logic terms, a datastructure is a many-sorted first-order structure. Since our algorithms are abstract state machines, the datastructure of an algorithm always includes the logic sort comprising elements `true`, `false`, and `nil`. Another sort could be that of natural numbers. In the case of BSS machines, we have also the sort of genuine reals. In the case of ruler-and-compass algorithms, we have three nonlogic sorts: points, straight lines, and circles of the real plane [1].

For our purposes, it will be convenient to see datastructures as ordinary structures, with just one base set, where the sorts are represented by static unary relations, a.k.a. characteristic functions. If A is an algorithm with datastructure D then any initial state X of A is an extension of D with (i) dynamic functions which, with the exception of input variables, are uninformative, and (ii) extrinsic functions, if any.

5.3 Definition

In a somewhat anticlimactic way, the intrinsic/extrinsic dichotomy allows us to characterize algorithms which are effective relative to their datastructures and are means bound in that sense. We formulate this characterization as a definition.

Definition 5.8 (Means-fit effectivity)**.** An algorithm A is *means-fit effective* or *effective relative to its datastructure* if A has no extrinsic functions.

The relative-to-datastructure character of effectivity is natural. The underlying datastructure matters. The effectivity of arithmetic-based algorithms is different from the effectivity of ruler-and-compass algorithms and from the effectivity of BSS algorithms.

Definition 5.8 is especially natural if the extrinsic functions of the algorithm in question are genuine oracles. But what if some of those oracles are computable, that is, computable by algorithms effective relative to their respective datastructures? In the real world, new software is rarely written from scratch. Typically, it reuses pieces of software which reuse other pieces of software, and so on. And this is not a strict hierarchy in general; the dependencies between various pieces may be more complicated.

Can we expand a given algorithm A so that it incorporates the auxiliary algorithms behind A's computable extrinsic functions and, for each of these auxiliary algorithms B, the algorithms behind B's computable extrinsic functions, etc.? It turns out that the answer is positive if only finitely many algorithms are involved altogether. The following two sections are devoted to proving this result. After that, we will return to the discussion of effectivity.

6 Query serialization

An algorithm may produce, during one step, many extrinsic queries, that is queries to extrinsic functions. This is problematic for the purpose of the next section. Fortunately query production can be serialized as the following Theorem 6.3 shows. The theorem is of independent interest but, in this paper, it is just an auxiliary result to be used in the following section.

We say that an ASM program Π is a *compound conditional composition* or a *compound conditional* if it has the form

$$\begin{aligned} &\texttt{if } g_1 \texttt{ then } P_1 \\ &\texttt{elseif } g_2 \texttt{ then } P_2 \\ &\qquad\vdots \\ &\texttt{elseif } g_n \texttt{ then } P_n \end{aligned}$$

If each P_i is a parallel composition of assignments, then Π is a compound conditional of parallel assignments.

Lemma 6.1. *For every ASM A there is a behaviourally equivalent ASM A' such that the program of A' is a compound conditional of parallel assignments and for every state X of A (and thus state of A' as well), A and A' generate exactly the same extrinsic queries at X.* ◁

Proof. It suffices to prove the following claim. Let $\Pi = (P \parallel Q)$ where P, Q are compound conditionals of parallel assignments. There is a compound conditional of parallel assignments Π' of the vocabulary of Π such that for every state X of Π (and of Π'), we have

(i) Π and Π' generate the same updates at X, and

(ii) Π and Π' generate the same extrinsic queries at X.

We illustrate the proof of the claim on an example where Π is

$$\begin{array}{rcl} \texttt{if } g_1 \texttt{ then } P_1 & \parallel & \texttt{if } h_1 \texttt{ then } Q_1 \\ \texttt{elseif } g_2 \texttt{ then } P_2 & & \end{array}$$

respectively. The desired Π' is

$$\begin{array}{rcl} \texttt{if } g_1 \wedge h_1 & \texttt{then } P_1 \parallel Q_1 \\ \texttt{elseif } g_1 & \texttt{then } P_1 \\ \texttt{elseif } g_2 \wedge h_1 & \texttt{then } P_2 \parallel Q_1 \\ \texttt{elseif } g_2 & \texttt{then } P_2 \\ \texttt{elseif } h_1 & \texttt{then } Q_1 \end{array}$$

All states X of Π and Π' split into six categories depending on which, if any, of the 5 guards holds in X. It is easy to check, for each of the six categories, that Π and Π' generate the same extrinsic queries. Consider for example, a state X satisfying $(\neg g_1 \wedge g_2) \wedge h_1$. Both Π and Π' evaluate the same guards g_1, g_2 and h_1 and execute the same rule $P_2 \parallel Q_1$ at X, and therefore the requirements (i) and (ii) are satisfied. □

Definition 6.2. An algorithm A' *tightly elaborates* an algorithm A if the vocabulary of A' is that of A plus some auxiliary elementary variables and if the following two conditions are satisfied where the *default expansion* X' of a state X of A is obtained by setting the auxiliary variables to their default values.

1. Every single step X, Y of A (where Y is the result of executing A at X) gives rise to a unique computation $X', \ldots, Y'$ of A' called a *mega-step*. The *mega-step* $X', \ldots, Y'$ is composed from a bounded number of regular steps of A'. The extrinsic queries issued by A' during the mega-step $X', \ldots, Y'$ are exactly the extrinsic queries issued by A during the step X, Y.

2. Every finite (resp. infinite) computation of A' has the form

$$X'_0, \ldots, X'_1, \ldots, X'_2, \ldots, X'_N \quad (\text{resp. } X'_0, \ldots, X'_1, \ldots, X'_2, \ldots)$$

$$\text{where} \quad X_0,\ X_1,\ X_2, \ldots, X_N \quad (\text{resp. } X_0,\ X_1,\ X_2,\ X_3, \ldots)$$

is a finite (resp. infinite) computation of A. ◁

Notice that A is function-computing if and only if A' is, and then they compute the same objective function.

Theorem 6.3. *For any algorithm A, there is an algorithm A' which tightly elaborates A and produces at most one extrinsic query per regular step.* ◁

Proof. In virtue of Lemma 6.1, we may assume without loss of generality that the program Π of A is a compound conditional of parallel assignments.

The program of the desired algorithm A' has the form

$$\texttt{if } \neg\text{Done } \texttt{then } \Pi' \texttt{ else Done} := \texttt{false}$$

where Π' is the *meaningful part* of the program and Done is an auxiliary Boolean variable. We construct Π' by induction on Π.

`Basis of induction:` Π is a parallel composition of assignments. Form a list

$$t_1,\ t_2,\ t_3,\ \ldots,\ t_{n-1},\ t_n$$

of all extrinsic-head terms in Π such that if t_j is a subterm of t_k then $j < k$. Let $d_1, d_2, \ldots, d_n$ be fresh elementary variables. The plan is to evaluate $t_1, \ldots, t_n$ one by one and store the results in $d_1, \ldots, d_n$ respectively, and then to complete the job of the original assignment without any extrinsic calls. This plan requires us to be careful with substitutions. Construct a matrix

$$\begin{array}{l}
t_1^0,\ t_2^0,\ t_3^0,\ \ldots,\ t_{n-1}^0,\ t_n^0 \\
d_1,\ t_2^1,\ t_3^1,\ \ldots,\ t_{n-1}^1,\ t_n^1 \\
d_1,\ d_2,\ t_3^2,\ \ldots,\ t_{n-1}^2,\ t_n^2 \\
\vdots \\
d_1,\ d_2,\ d_3,\ \ldots,\ d_{n-1},\ t_n^{n-1} \\
d_1,\ d_2,\ d_3,\ \ldots,\ d_{n-1},\ d_n
\end{array}$$

where $t_j^0 = t_j$ and if $i > 0$ then $t_j^i = t_j^{i-1}\{d_i \mapsto t_i^{i-1}\}$, so that every instance of t_i^{i-1} in t_j^i is replaced with d_i. Similarly, construct programs

$$\Pi^0,\ \Pi^1,\ \Pi^2,\ \Pi^3,\ \ldots,\ \Pi^{n-1},\ \Pi^n$$

where $\Pi^0 = \Pi$, and if $i > 0$ then $\Pi^i = \Pi^{i-1}\{d_i \mapsto t_i^{i-1}\}$. Notice that Π^n has no extrinsic functions.

Let $b_1, b_2, \ldots, b_n$ be fresh Boolean variables. Recall that every b_i is initially false and therefore every $\neg b_i$ is initially true. The desired program Π' is

$$\begin{aligned}
&\texttt{if } \neg b_1 \texttt{ then } (d_1 := t_1^0 \;\; \| \;\; b_1 := \texttt{true})\\
&\texttt{elseif } \neg b_2 \texttt{ then } (d_2 := t_2^1 \;\; \| \;\; b_2 := \texttt{true})\\
&\qquad\vdots\\
&\texttt{elseif } \neg b_n \texttt{ then } (d_n := t_n^{n-1} \;\; \| \;\; b_n := \texttt{true})\\
&\quad\texttt{else} \qquad (\Pi^n \;\; \| \;\; \text{Done} := \texttt{true} \;\; \|\\
&\qquad\qquad b_1 := \texttt{false} \;\; \| \;\; \ldots \;\; \| \;\; b_n := \texttt{false})
\end{aligned}$$

This completes the basis of our induction.

`Induction step:` Π has the form $\texttt{if } \beta \texttt{ then } P \texttt{ else } Q$ where P, Q are compound conditionals of parallel assignments. By the induction hypothesis, there are programs

$$\begin{aligned}
&\texttt{if } \neg\text{Done}^P \texttt{ then } P' \texttt{ else } \text{Done}^P := \texttt{false},\\
&\texttt{if } \neg\text{Done}^Q \texttt{ then } Q' \texttt{ else } \text{Done}^Q := \texttt{false}
\end{aligned}$$

which tightly elaborate P, Q respectively and produce at most one extrinsic query per regular step.

The desired algorithm A' starts with evaluating β. This part is exactly like the induction basis except that we are given a term β rather than an assignment. In particular, now $t_1, t_2, \ldots, t_n$ are all of the extrinsic-head terms in β, and instead of programs Π^0, Π^1, Π^2, Π^3, $\ldots$, Π^{n-1}, Π^n we have terms

$$\beta^0,\ \beta^1,\ \beta^2,\ \beta^3,\ \ldots,\ \beta^{n-1},\ \beta^n$$

where $\beta^0 = \beta$ and if $i > 0$ then $\beta^i = \beta^{i-1}\{d_i \mapsto t_i^{i-1}\}$. The desired program Π' uses additional auxiliary Boolean variables a and b.

$$
\begin{array}{rl}
\texttt{if} & \neg a \wedge \neg b_1 \ \texttt{then}\ (d_1 := t_1^0 \ \parallel \ b_1 := \texttt{true}) \\
\texttt{elseif} & \neg a \wedge \neg b_2 \ \texttt{then}\ (d_2 := t_2^1 \ \parallel \ b_2 := \texttt{true}) \\
& \vdots \\
\texttt{elseif} & \neg a \wedge \neg b_n \ \texttt{then}\ (d_n := t_n^{n-1} \ \parallel \ b_n := \texttt{true}) \\
\texttt{elseif} & \neg a \ \texttt{then}\ (b := \beta^n \ \parallel \ a := \texttt{true} \ \parallel \\
& b_1 := \texttt{false} \ \parallel \ \ldots \ \parallel \ b_n := \texttt{false}) \\
\texttt{elseif} & a \wedge b \wedge (\neg \mathrm{Done}^P) \ \texttt{then}\ P' \\
\texttt{elseif} & a \wedge \neg b \wedge (\neg \mathrm{Done}^Q) \ \texttt{then}\ Q' \\
\texttt{else} & (\mathrm{Done} := \texttt{true} \ \parallel \ a := \texttt{false} \ \parallel \\
& \mathrm{Done}^P := \texttt{false} \ \parallel \ \mathrm{Done}^Q := \texttt{false})
\end{array}
$$ □

Remark 6.4. For those interested in details, let us illustrate two subtleties which complicate the proof of the theorem. First, consider the program Π' of the induction step. One may be tempted to eliminate the else clause and replace P', Q' with $P' \parallel \mathrm{Done} := \texttt{true}$ and $Q' \parallel \mathrm{Done} := \texttt{true}$ respectively. But this does not work because Done will be set to `true` on the very first regular step of P', Q' and thus will prevent them from completing their mega-step.

Second, why did we bother with proving that lemma? It seems that we could consider another case of the induction step where $\Pi = P \parallel Q$. The desired Π' could be

$$
\begin{array}{rl}
\texttt{if} & \neg \mathrm{Done}^P \ \texttt{then}\ P' \\
\texttt{elseif} & \neg \mathrm{Done}^Q \ \texttt{then}\ Q' \\
\texttt{else} & \mathrm{Done} := \texttt{true}
\end{array}
$$

But it is risky to serialize parallel composition. Just consider the case where P, Q are $a := b$ and $b := a$ respectively. ◁

For future use we record the following corollary.

Corollary 6.5. *The program of A' has the form*

$$\texttt{if}\ \neg \mathrm{Done}\ \texttt{then}\ \Pi'\ \texttt{else}\ \mathrm{Done} := \texttt{false}$$

where Π' has the form

$$
\begin{array}{rl}
\texttt{if} & g_1 \ \texttt{then}\ R_1 \\
\texttt{elseif} & g_2 \ \texttt{then}\ R_2 \\
& \vdots \\
\texttt{elseif} & g_n \ \texttt{then}\ R_n
\end{array}
$$

and it is last rule R_n that sets Done *to* `true`*; the rules $R_1, \ldots, R_{n-1}$ do not mention* Done*. Furthermore, the n clauses split into two categories:*

`Pure` *clauses with no occurrences of extrinsic functions, and*

`Tainted` *clauses of the form* `[else]if` g_k `then` $d_k := t_k \parallel Q_k$
where d_k is an elementary variable, the head function of t_k is extrinsic and there are no other occurrences of extrinsic functions. ◁

7 Pruning off effective oracles

Definition 7.1 (Numeric algorithms)**.**

1. An algorithm (that is an ASM) A is *numeric* if, up to isomorphism, every state of A incorporates natural-numbers arithmetic.

2. Some function symbols in the vocabulary of a numeric algorithm A are marked *numerical*. A basic numerical function takes only nonnegative integer values, and its default value is 0. ◁

In §2, we defined the notions of consistency and union of structures. Since we view datastructures as first-order structures (see §5), we have the notions of consistency and union of datastructures.

Theorem 7.2 (Pruning Theorem)**.** *Consider N algorithms $A_0, A_1, \ldots, A_{N-1}$ whose datastructures are pairwise consistent, and suppose that, for every A_i, every extrinsic function of A_i is the objective function of some A_j. There is a numerical algorithm B, with no extrinsic functions, such that*

1. *the datastructure of B is the union of the N datastructures of the algorithms A_i plus natural-numbers arithmetic.*

2. *B computes the objective function of A_0.* ◁

Remark 7.3.

1. If the algorithms A_i are numerical and have the same datastructure D, then D is also the datastructure of B.

2. The algorithm B, constructed in the proof, works exactly like A_0 except that, instead of waiting for extrinsic queries to be answered, B computes the answers. It is therefore no wonder that B executes the task of A_0 which happens to be computing the objective function. But it could be virtually any other reasonable task, e.g. Task 3.1. ◁

Proof of theorem. Without loss of generality, different algorithms A_i compute different objective functions. Indeed, if $m < n$ and A_m, A_n compute the same objective function, remove A_n. The remaining set is still closed in the sense that, for every A_i, every extrinsic function of A_i is computed by some A_j.

In essence, we have a mutually recursive system of algorithms. Whenever one algorithm A_i poses a query to an extrinsic function powered by algorithm A_j (which may be A_i itself), it implicitly calls A_j. The caller suspends its execution and transfers the execution control to the callee. If and when the execution control returns from the callee, the caller resumes executing from the point where it put itself on hold, and it needs the exact same data it was working on except that the query is answered.

Following the standard practice, we use a call stack to implement recursion. To this end, we introduce a numerical variable $\top$ (read "top," not "true"), indicating the current position of the top of the stack, and a unary dynamic function Active such that Active($\top$) indicates the index i of the currently active algorithm.

However we have two non-standard difficulties which make our work harder. One of them is that an algorithm may generate many extrinsic queries in one step. That difficulty was addressed in the previous section. To illustrate the other difficulty, consider this scenario. A_1 calls A_2 and increases $\top$, say from 7 to 8. When A_2 is done and $\top$ is decreased to 7, it would be useful to defaultify A_2, that is to make all its dynamic functions uninformative. But defaultification, a form of garbage collection, is a problem in our abstract setting. Fortunately — and ironically — effectivity does not have to be efficient. Instead of cleaning up a used copy of A_2, we abandon it and use instead another, fresh copy of A_2. To this end we need an ample supply of copies of every A_i. This will be achieved by means of a special parameter `n`.

In accordance with Corollary 6.5, we may assume that the program of algorithm A_i has the form

$$\texttt{if } \neg \text{Done}_i \texttt{ then } \Pi_i \texttt{ else } \text{Done}_i := \texttt{false}$$

where the meaningful part Π_i has the form

$$\begin{aligned} &\texttt{if } g_{i1} \texttt{ then } R_{i1} \\ &\texttt{elseif } g_{i2} \texttt{ then } R_{i2} \\ &\vdots \\ &\texttt{elseif } g_{in_i} \texttt{ then } R_{in_i} \end{aligned}$$

described in Corollary 6.5. In particular, tainted clauses have the form

$$\texttt{[else]if } g_{ik} \texttt{ then } d_{ik} := t_{ik} \parallel Q_{ik} \tag{1}$$

To construct the program of B, we modify programs Π_i in three stages.

`Stage 1.` Π'_i is obtained from Π_i by expanding every dynamic function $d(x_1, \ldots, x_r)$ to $d(\mathtt{n}, x_1, \ldots, x_r)$, so that d acquires a new argument position, ahead of all the old argument positions, and this new position is occupied by a numerical variable `n`. The expansion transforms any term t to a term t' and any rule R to a rule R'. In particular, the tainted clauses of Π_i acquire this form:

$$\texttt{[else]if } g'_{ik} \texttt{ then } d_{ik}(\mathtt{n}) := t'_{ik} \parallel Q'_k \tag{2}$$

As a result we have infinite supply of copies of Π_i.

Notice that the same parameter `n` is used for all programs Π'_i. We will have one consecutive numbering of copies of all these programs. We will speak about sessions. Each value of `n` corresponds to one session, and a particular A_i executes during that session. Each extrinsic call will create a new session. A special numerical variable `Max` will keep track of the maximal session number so far. For example, suppose that A_1 executes session 10, $\top = 7$, $\mathtt{Max} = 20$, and A_1 calls A_2. This will start session 21, increment $\top$ to 8 and set `Max` to 21.

The input and output variables of A_i will be denoted Input_{i1}, Input_{i2}, …, Output_i. The input and output variables of the desired algorithm B will be denoted Input_1, Input_2, …, Output.

`Stage 2.` Π''_i is obtained from Π'_i by modifying every tainted clause (2). If $t'_{ik} = e(\tau'_1, \ldots, \tau'_r)$ and e is computed by A_j, then replace the assignment $d_{ik}(\mathtt{n}) := t'_{ik}$ with the following administrative program which we display first and then explain.

$$\begin{aligned}
&\texttt{if } \neg b_{ik} \texttt{ then } \top := \top + 1 \parallel \mathrm{Active}(\top + 1) := j \parallel \\
&\quad \mathtt{n} := \mathtt{Max} + 1 \parallel \mathtt{Max} := \mathtt{Max} + 1 \parallel \mathtt{Top}(\mathtt{Max} + 1) := \top \parallel \\
&\quad \mathrm{Input}_{j1}(\mathtt{Max} + 1) := \tau'_1 \parallel \ldots \parallel \mathrm{Input}_{jr}(\mathtt{Max} + 1) := \tau'_r \parallel \\
&\quad \mathtt{Ret}(\mathtt{Max} + 1) := \mathtt{n} \parallel b_{ik} := \mathtt{true} \\
&\texttt{elseif } \mathtt{Top}(\mathtt{n}) = \top \texttt{ then} \\
&\quad d_{ik}(\mathtt{n}) := \mathtt{To}(\mathtt{n}) \parallel Q'_{ik} \parallel b_{ik} := \mathtt{false}
\end{aligned}$$

Thanks to b_{ik}, the administrative program works in two steps. On the first step it increments $\top$ and passes control to A_j which involves a number of details. A new copy of A_j is being engaged and given number $\mathtt{Max} + 1$ where `Max` holds the maximal value of `n` used so far. Accordingly `Max` is incremented. A special dynamic function `Top`(`n`) records the value of $\top$ corresponding to the $\mathtt{n}^{th}$ program copy in the consecutive numbering of all copies of all programs Π_i. Further, the proper input information for A_j is supplied. Finally, using a special unary function `Ret` (alluding to “return address"), A_j is notified that, upon termination, it should return control to the current copy of A_i.

The administrative program executes the second step only when (and if) A_j computes the desired output and passes the control and the output to A_i. The output is passed by means of a special function To. But how did A_j know who to pass the output to? For this we need to see how Π_j'' was modified at Stage 3.

Stage 3. Π_i^+ is the program

$$\begin{aligned}&\texttt{if } \neg\mathrm{Done}_i(\mathtt{n}) \texttt{ then } \Pi_i''\\ &\texttt{else } \top := \top - 1 \ \parallel\ \mathtt{n} := \mathtt{Ret}(\mathtt{n}) \ \parallel\\ &\qquad \mathtt{To}(\mathtt{Ret}(\mathtt{n})) := \mathrm{Output}_i(\mathtt{n})\end{aligned}$$

To explain what goes on, let us return to the scenario where A_1 executed session 10, $\top$ was 7, Max was 20, and A_1 passed control to A_2 which started session 21, incremented $\top$ to 8 and set Max to 21. On the same occasion, A_1 set Ret(21) to 10. If and when A_2 finishes session 21, it will decrement $\top$ to 7 and will pass control to the program of session 10 which happens to be A_1. On the same occasion, A_2 will assign its output value to To(10).

Finally, the program of the desired algorithm B is

$$\begin{aligned}\texttt{if } \neg\mathrm{initialized} \quad &\texttt{then Initialize}\\ \texttt{elseif } \mathrm{Output}_1(0) = \texttt{nil} \quad &\texttt{then Toil}\\ \texttt{else} \quad &\texttt{Finish}\end{aligned}$$

where initialized is a fresh Boolean variable and the three constituent programs are as follows:

Initialize passes the inputs of B to the 0^{th} copy of A_0 and sets initialized to true:

$$\begin{aligned}&\mathrm{Input}_{01}(0) := \mathrm{Input}_1 \ \parallel\ \ldots\ \parallel\ \mathrm{Input}_{0r}(0) := \mathrm{Input}_r\\ &\mathrm{initialized} := \texttt{true}\end{aligned}$$

where r is the arity of the objective function of A_0.

Toil is the parallel composition of rules

$$\texttt{if } \mathrm{Active}(i) \texttt{ then } \Pi_i^+$$

where $i = 0, \ldots, N-1$.

Finish just passes the output from the 0^{th} copy of A_0 to B:

$$\mathrm{Output} := \mathrm{Output}_0(0)$$

This completes the proof of Theorem 7.2. □

In the obvious way, Theorem 7.2 relativizes in the standard sense of computation theory. More explicitly, the theorem remains true if the algorithms A_i have access to some genuine oracles $O_{i1}, O_{i2}, \ldots$ and if B has access to all these oracles.

8 Absolute effectivity, and related work

As we saw above, different algorithms may use different datastructures. But, in logic, one datastructure has been playing the central role historically. This is arithmetic of course. Call an algorithm *arithmetical* if its datastructure is arithmetic.

Definition 8.1. A numerical function F is *effective relative to artihmetic* if it computable by an effective arithmetical algorithm. ◁

As we mentioned in the introduction, it is the historical, mathematical Church-Turing thesis that we are discussing here. The minimum task needed to verify the thesis is to prove the claim that every numerical function effective relative to arithmetic is partial recursive. In [12], this claim has been derived from the axioms of [13] plus an initial-state axiom asserting that, in the initial states, the dynamic functions — with the exception of input variables — are uninformative. The current paper provides additional justification for this axiom.

A more ambitious task would be to show that, for any reasonable datastructure D, the numerical functions computed by the D based algorithms are partial recursive. The obvious problem is: Which datastructures are reasonable? This allows us to segue into our quick review of related work.

The idea to define effectivity by expanding the axiomatization of [13] with an initial-state axiom was enunciated by Udi Boker and Nachum Dershowitz in [6] and elaborated in [7]. Their Initial Data Axiom asserts that an initial state comprises a Herbrand universe, some finite data, and "effective oracles" in addition to input. They shifted focus from single algorithms to computation models. "The resultant class ... includes all known Turing-complete state-transition models, operating over any countable domain." Any computation model satisfying the axioms of [13] plus the Initial Data Axiom is "of equivalent computational power to, or weaker than, Turing machines."

In [12], where the minimum task was accomplished, that task was extended to include any algorithms over a countable domain equipped with an injective mapping from that domain to the natural numbers, subject to some recursivity requirement.

A very different approach was taken by Wolfgang Reisig in [16]. Every first-order structure S of vocabulary Υ imposes an equivalence relation on Υ terms: $t_1 \sim_S t_2 \iff \mathcal{V}_S(t_1) = \mathcal{V}_S(t_2)$. "The successor state $\mathbf{M}(S)$ of a state S is fully specified by the equivalence $\sim_S$ ... Consequently, $\mathbf{M}(S)$ is computable in case $\sim_S$ is decidable. Furthermore, this result implies a notion of computability for general structures, e.g. for algorithms operating on real numbers."

In [8], Boker and Dershowitz observe that Reisig's approach gives rise to a novel definition of effectivity, though they restrict attention to the case where ev-

ery element of any initial state is nameable by a term. They show that the three definitions of equivalence — in [12], in [7], and implicitly in [16] — coincide.

In [11], Dershowitz and Falkovich "describe axiomatizations of several aspects of effectiveness: effectiveness of transitions; effectiveness relative to oracles; and absolute effectiveness, as posited by the Church-Turing Thesis."

This concludes our quick review of related literature.

References

[1] Michael Beeson, "Logic of ruler and compass constructions," *Springer Lecture Notes in Computer Science* 7318 46–55 2012

[2] Andreas Blass, Nachum Dershowitz and Yuri Gurevich, "Exact exploration and hanging algorithms," *Springer Lecture Notes in Computer Science* 6247 140–154 2010

[3] Andreas Blass and Yuri Gurevich, "Abstract state machines capture parallel algorithms," *ACM Transactions on Computation Logic* 4:4 578–651 2003. Correction and extension, ibid. 9:3 Article 19 2008

[4] Andreas Blass, Yuri Gurevich and Benjamin Rossman, "Interactive small-step algorithms," Parts I, II; *Logical Methods in Computer Science* 3:4 Articles 3, 4 2007

[5] Lenore Blum, Mike Shub and Steve Smale, "On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines," *Bulletin of the American Mathematical Society* 21:1 1—46 1989

[6] Udi Boker and Nachum Dershowitz, "Abstract effective models," *Electronic Notes in Theoretical Computer Science* 135 15–26 2006

[7] Udi Boker and Nachum Dershowitz, "The Church-Turing Thesis over Arbitrary Domains," in Festschrift for B. Trakhtenbrot (eds. A. Avron, N. Dershowitz and A. Rabinovich), *Lecture Notes in Computer Science* 4800 199–229 2008

[8] Udi Boker and Nachum Dershowits, "Three paths to effectiveness," in Festschrift for Y. Gurevich (eds. A. Blass, N. Dershowitz and W. Reisig), *Lecture Notes in Computer Science* 6300 135–146 2010

[9] Alonzo Church, "An unsolvable problem of elementary number theory," *American Journal of Mathematics* 58:2 345–363 1936

[10] Stephen A. Cook and Robert A. Reckhow, "Time-bounded random access machines," *Journal of Computer Systems Science* 7:4 354–375 1973

[11] Nachum Dershowitz and Evgenia Falkovich, "Effectiveness," in "A computable universe" (ed. H. Zenil) 77–97, World Scientific 2013

[12] Nachum Dershowitz and Yuri Gurevich, "A natural axiomatization of computability and proof of Church's thesis," Bulletin of Symbolic Logic 14:3 299–350 2008

[13] Yuri Gurevich, “Sequential Abstract State Machines capture Sequential Algoritheorems,” *ACM Transactions on Computational Logic* 1:1 77–111, 2000

[14] Yuri Gurevich, “What is an Algorithm?” *Springer Lecture Notes in Computer Science* 7147 31–42 2012. A slightly revised version in “Church’s Thesis: Logic, Mind and Nature" (eds. A. Olszewski et al.) Copernicus Center Press 2014

[15] Andrei N. Kolmogorov, “On the concept of algorithm”, *Uspekhi Mat. Nauk* 8:4 (1953) 175–176, Russian

[16] Wolfgang Reisig, “The computable kernel of abstract state machines," *Theoretical Computer Science* 409 126–136 2008

[17] Hartley Rogers, Jr. “Theory of recursive functions and effective computability," McGraw-Hill 1967.

[18] Joseph R. Shoenfield, “Mathematical logic,” Addison-Wesley 1967

[19] Alan M. Turing, “On computable numbers, with an application to the Entscheidungsproblem”, *Proceedings of London Mathematical Society*, 42 230–265 1936-1937, Corrections *ibid.* 43 544–546 1937