The Logic in Computer Science Column

by

Yuri Gurevich

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

Reversify any sequential algorithm

Yuri Gurevich

Abstract

To reversify an arbitrary sequential algorithm A, we gently instrument A with bookkeeping machinery. The result is a step-for-step reversible algorithm that mimics A step-for-step and stops exactly when A does.

Without loss of generality, we presume that algorithm A is presented as an abstract state machine that is behaviorally identical to A. The existence of such representation has been proven theoretically, and the practicality of such representation has been amply demonstrated.

Darn the wheel of the world! Why must it
continually turn over? Where is the reverse gear?
— Jack London

1 Introduction

In 1973, Charles Bennett posited that an"irreversible computer can always be made reversible" [2, p. 525]. To this end, he showed how to transform any one-tape Turing machine M that computes a function $F(x)$, into a reversible three-tape Turing machine M^R computing the function $x \mapsto (x, F(x))$. First, M^R emulates the computation of M on x, saving enough information to ensure step-for-step reversibility. If and when the output is computed, the emulation phase ends, and M^R proceeds to erase all saved information with the exception of the input.

Bennett's construction shows that, in principle, every sequential algorithm, is reversifiable[1]. In practice, you don't want to compile your algorithms to a one-tape Turing machine M and then execute the three-tape Turing machine M^R.

[1]We attempt to give a new useful meaning to the word reversify. To reversify an algorithm means to transform it into a reversible form (rather than to formulate it anew in verse, which is the current dictionary meaning of reversify).

It had been discussed in the programming community, in particular by Edsger Dijkstra [8, pp 351–354] and David Gries [10, pp 265–274], which programs are reversible, but Bennett's reversification idea was either unknown to or neglected by programming experts.

The progress was led by physicists. They reversified Boolean circuits and other computation models. "We have shown", wrote Edward Fredkin and Tommaso Toffoli [9, p 252], "that abstract systems having universal computing capabilities can be constructed from simple primitives which are invertible". The interest in reversible computations and especially in reversible circuit computations soared with the advent of quantum computing. This is related to the fact that pure (involving no measurements) quantum computations are reversible. There are books on reversible computations [1, 6, 17, 18]. The International Conference on Reversible Computation will have its 13th meeting in 2021 [19]

In this paper, we use sequential abstract state machines, in short sequential ASMs, to address the problem of practical reversification of arbitrary sequential algorithms. Why ASMs? Let us explain.

ASMs were introduced to faithfully simulate arbitrary algorithms on their natural abstraction levels [11]. One instructive early result was the formalization of the C programming language [14].

In [12], an ambitious ASM thesis was formulated: For every algorithm A, there is an ASM B that is behaviorally equivalent to A. If A is sequential, then B has the same initial states as A and the same state transition function. In [13], we axiomatized sequential algorithms and proved the ASM thesis for them[2]. Thus, semantically, sequential algorithms are sequential ASMs. In the meantime, substantial evidence has been accumulated to support the practicality of faithful ASM modeling. Some of it is found in the 2003 book [3].

The main result of the present paper is a simple construction, for every sequential ASM A, of a reversible sequential ASM B that step-for-step simulates A and stops when A does. B does exactly what A does plus some bookkeeping. If A uses some input and output variables and computes some function, then B uses the same input and output variables and computes the same function.

[2]Later these results were generalized to other species of algorithms, e.g. to synchronous parallel algorithms [4] and interactive algorithms [5].

2 Preliminaries

The purpose of this section is to make the current paper self-contained.

2.1 Sequential algorithms

By sequential algorithms we mean algorithms as the term was understood before modern computer science generalized the notion of algorithm in various directions, which happened in the final decades of the 20th century. In this connection, sequential algorithms are also called classical.

While the term “sequential algorithm" is short and convenient, it also is too laconic. Some explication is in order. “Algorithms,” said Andrei Kolmogorov in a 1953 talk [16], “compute in steps of bounded complexity.” Let’s look more closely at the two aspects mentioned by Kolmogorov. One aspect is computing in steps, one step after another. Kolmogorov didn’t say “one step after another.” He didn’t have to. That was understood at the time.

The other aspect is a somewhat vague constraint: the bounded complexity of any one step of the algorithm. We prefer a related constraint, arguably a version of Kolmogorov’s constraint: the bounded resources of any one step of the algorithm. The bounded resources constraint, still informal, seems to us clearer and more suitable. It might have been Kolmogorov’s intention all along. We do not know exactly what Kolmogorov said during that talk[3].

To summarize, sequential algorithms can be characterized informally as transition systems that compute in bounded-resources steps, one step after another.

In our axiomatization of sequential algorithms [13], the bounded resources constraint gives rise to the crucial bounded-exploration axiom. It is also used to justify that a sequential algorithm doesn’t hang forever within a step; time is a bounded resource.

In the following subsections, we recall some basic notions of mathematical logic in the form appropriate to our purposes.

[3]Vladimir Uspensky, who chaired the Logic Department of Moscow State University after Kolmogorov’s death, admitted to me that the abstract [16] of Kolmogorov’s talk for the Moscow Mathematical Society was written by him (Uspensky) after many unsuccessful attempts to squeeze an abstract from Kolmogorov.

2.2 Vocabularies

A *vocabulary* is a finite collection of function symbols where each symbol f is endowed with some metadata according to the following clauses (V1)–(V4). We interleave the four clauses with auxiliary definitions and explanations.

(V1) Each symbol f is assigned a natural number, the *arity* of f.

Define *terms* (or *expressions*) by induction. If f is an r-ary symbol in and $t_1, \dots, t_r$ are terms, then $f(t_1, \dots, t_r)$ is a term. (The case $r = 0$ is the basis of induction.)

(V2) Some symbols f are marked as *relational*.

Clauses (V1) and (V2) are standard in logic, except that, traditionally, relations are viewed as separate category, not as a special functions.

(V3) f may be marked as *dynamic*; if not then f is called *static*. Nullary static symbols are called *constants*; nullary dynamic symbols are called *variables*.

Clause (V3) is related to our use of structures as states of algorithms. The intention is that, during computation, only dynamic functions may be assigned new values. We say that a term is *static* if it involves only static functions.

We presume that every vocabulary contains the following *obligatory* symbols which are all static.

- Constants $\top$ and $\bot$ (read "true" and "false"), unary `Bool`, and the standard propositional connectives. All these symbols are relational.
- Constant 0, and unary `Num`, `increment`, and `decrement`. Of these four symbols, only `Num` is relational.
- Constant `nil` (called `undef` in [13] and other early papers) and the (binary) equality sign =. Of these two symbols, only the equality sign is relational.

(V4) Every dynamic symbol f is assigned a static term, the *default term* of f. If f is relational then so is its default term.

Clauses (V2) and (V4) constitute rudimentary typing which is sufficient for our purposes in this paper. As a rule, the default term for any relational symbol is $\bot$. If a variable v is supposed to take numerical values, then typically the default term for v would be 0, but it could be 1. This concludes the definition of vocabularies.

If Υ and Υ' are vocabularies, we write $\Upsilon \subseteq \Upsilon'$, and we say that Υ is *included* in Υ' and that Υ' *includes* or *extends* Υ, if every Υ symbol belongs to Υ' and has the same metadata in Υ'.

2.3 Structures

A *structure* X of vocabulary Υ is a nonempty set $|X|$, the *universe* or *base set* of X, together with interpretations of the function symbols in Υ. The vocabulary Υ may be denoted Voc(X).

An r-ary function symbol f is interpreted as a function $f : |X|^r \to |X|$ and is called a *basic function* of X. If f is nullary then f is just a name of an element of (the universe of) X. If f is dynamic and d is the default term for f, then the value (denoted by) d is the *default value* of f.

If f is relational, then the elements $\top$ are $\bot$ are the only possible values of f. If $f(\bar{x}) = \top$, we say that f is true (or holds) at $\bar{x}$; otherwise we say that f is false (or fails) at $\bar{x}$. If f, g are relations of the same arity r, then f, g are *equivalent* in X if their values at every r-tuple of elements of X are the same.

Any basic relation f is the characteristic function of the set $\{x : f(x) = \top\}$. It is often convenient to treat f as that set. We will do that in §5.

Remark 2.1 (Names and denotations)**.** Syntactic objects often denote semantical objects. For example, vocabulary symbols denote basic functions. Different conventions may be used for disambiguation, e.g. a basic function may be denoted f_X. We will use no disambiguation convention in this paper. It should be clear from the context whether a symbol means a syntactic or semantic object. $\triangleleft$

The equality sign has its usual meaning. `Bool` comprises (the values of) $\top$, $\bot$ which, together with the propositional connectives, form a two-element Boolean algebra.

Given a structure X, the *value* $\mathcal{V}_X(f(t_1,\dots,t_r))$ of a Voc(X) term $f(t_1,\dots,t_r)$ in X is defined by induction:

$$\mathcal{V}_X(f(t_1,\dots,t_r)) = f(\mathcal{V}_X(t_1),\dots,\mathcal{V}_X(t_r)). \tag{1}$$

Again, the case $r = 0$ is the base of induction.

Instead of `increment`(x), we will write $x + 1$ and $x - 1$. `Num` comprises the values of terms $0, 0 + 1, (0 + 1) + 1, \dots$ which are all distinct. These values are denoted $0, 1, 2, \dots$ respectively; we call them the *natural numbers* of structure X, and we say that these values are *numerical*. `decrement` is interpreted as expected as well. Instead of `decrement`(x), we write $x - 1$. `decrement`$(0) =$ `nil`. The value of `nil` is neither Boolean nor numerical.

Remark 2.2 (Totality)**.** In accordance with §2.1, all basic functions are total. In applications, various error values may arise, in particular *timeout*. But, for our purposes in this paper (as in [13]), an error value is just another value.

A *location* in a structure X is a pair $\ell = (f, \bar{x})$ where f is a dynamic symbol in Voc(X) of some arity r and $\bar{x}$ is an r-tuple of elements of X. The value $f(\bar{x})$ is the *content* of location ℓ.

An *update of location* $\ell = (f, \bar{x})$ is a pair (ℓ, y), also denoted $(\ell \hookleftarrow y)$, where y an element of X; if f is relational then y is Boolean. To *execute* an update $(\ell \hookleftarrow y)$ in X, replace the current content $\mathcal{V}_X(f(\bar{x}))$ of ℓ with y, i.e., set $f_X(\bar{x})$ to y. An update $(\ell \hookleftarrow y)$ of location $\ell = (f, \bar{x})$ is *trivial* if $y = f(\bar{x})$.

An *update of structure* X is an update of any location in X. A set Δ of updates of X is *contradictory* if it contains updates $(\ell \hookleftarrow y_1)$ and $(\ell \hookleftarrow y_2)$ with distinct y_1, y_2; otherwise Δ is *consistent*.

2.4 Sequential abstract state machines

Fix a vocabulary Υ and restrict attention to function symbols in Υ and terms over Υ.

Definition 2.3 (Syntax of rules)**.** *Rules* over vocabulary Υ are defined by induction.

1. An *assignment rule* or simply *assignment* has the form

$$f(t_1, \dots, t_r) := t_0 \tag{2}$$

   where f, the *head* of the assignment, is dynamic, $r = \text{Arity}(f)$, and $t_0, \dots, t_r$ are terms. If f is relational, then the head function of t_0 is relational. The assignment (2) may be called an f assignment.

2. A *conditional rule* has the form

$$\texttt{if } \beta \texttt{ then } R_1 \texttt{ else } R_2 \tag{3}$$

   where β is a Boolean-valued term and R_1, R_2 are Υ rules.

3. A *parallel rule* has the form

$$R_1 \ \parallel \ R_2 \ \parallel \ \cdots \ \parallel \ R_k \tag{4}$$

   where k is a natural number and $R_1, \dots, R_k$ are Υ rules. In case $k = 0$, we write `Skip`. $\triangleleft$

Definition 2.4 (Semantics of rules)**.** Fix an Υ structure X. Every Υ rule R generates a finite set Δ of updates in X. R *fails* in X if Δ is contradictory; otherwise R *succeeds* in X. To *fire* (or *execute*) rule R that succeeds in structure X means to execute all Δ updates in X.

1. An assignment $f(t_1, \dots, t_r) := t_0$ generates a single update $(\ell, \mathcal{V}_X(t_0))$ where $\ell = (f, (\mathcal{V}_X(t_1), \dots, \mathcal{V}_X(t_r))$.

2. A conditional rule $\texttt{if}\,\beta\,\texttt{then}\,R_1\,\texttt{else}\,R_2$ works exactly as R_1, if $\beta = \top$ in X, and exactly as R_2 otherwise.

3. A parallel rule $R_1 \ \parallel \ R_2 \ \parallel \ \cdots \ \parallel \ R_k$ generates the union of the update sets generated by rules $R_1, \dots, R_k$ in X. $\triangleleft$

Definition 2.5. A *sequential ASM A* is given by the following three components.

1. A vocabulary Υ, denoted Voc(A).

2. A nonempty collection of Voc(A) structures, closed under isomorphisms. These are the *initial states* of A. $\triangleleft$

3. A Voc(A) rule, called the *program* of A and denoted Prog(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.

In the rest of the paper, by default, all ASMs are sequential.

Consider an ASM A. A Voc(A) structure X is *terminal* for A if Prog(A) produces no updates (not even trivial updates[4]) in X. A *partial computation* of A is a finite sequence $X_0, X_1, \ldots, X_n$ of Υ structures where

- X_0 is an initial state of A,

- every X_{i+1} is obtained by executing Prog(A) in X_i, and

- no structure in the sequence, with a possible exception of X_n, is terminal.

If X_n is terminal, then the partial computation is *terminating*. A *(reachable) state* of A is an Voc(A) structure that occurs in some partial computation of A.

A Boolean expression γ is a *green light* for an ASM A if it holds in the non-terminal states of A and fails in the terminal states.

Lemma 2.6. *Every ASM has a green light.*

Proof. By induction on rule R, we construct a green light γ_R for any ASM with program R. If R is an assignment, set $\gamma_R = \top$. If R is the parallel composition of rules R_i, set $\gamma_R = \bigvee_i \gamma_{R_i}$. If $R = \texttt{if}\ \beta\ \texttt{then}\ R_1\ \texttt{else}\ R_2$, set $\gamma_R = (\beta \wedge \gamma_{R_1}) \vee (\neg\beta \wedge \gamma_{R_2})$. $\square$

In examples and applications, typically, such conditions are easily available. Think of terminal states of finite automata or of halting control states of Turing machines. In a while-loop program, the while condition is a green light.

[4]In applications, trivial updates of A may mean something for its environment.

3 Reducts and expansions

In mathematical logic, a structure X is a *reduct* of a structure Y if $\mathrm{Voc}(X) \subseteq \mathrm{Voc}(Y)$, the two structures have the same universe, and every $\mathrm{Voc}(X)$ symbol f has the same interpretations in X and in Y. If X is a reduct of Y, then Y is an *expansion* of X. For example, the field of real numbers expands the additive group of real numbers.

We say that an expansion Y of a structure X is *uninformative* if the additional basic functions of Y (which are not basic functions of X) are dynamic and take only their default values in Y. (The default values are defined in §2.3.) Clearly, X has a unique uninformative expansion to $\mathrm{Voc}(B)$.

Definition 3.1. An ASM B is a *faithful expansion* of an ASM A if the following conditions hold.

(E1) $\mathrm{Voc}(A) \subseteq \mathrm{Voc}(B)$. The symbols in $\mathrm{Voc}(A)$ are the *principal* symbols of $\mathrm{Voc}(B)$, and their interpretations in $\mathrm{Voc}(B)$ structures are *principal* basic functions; the other $\mathrm{Voc}(B)$ symbols and their interpretations are *ancillary*.

(E2) All ancillary symbols are dynamic, and the initial states of B are the uninformative expansions of the initial states of A.

(E3) If Y is a $\mathrm{Voc}(B)$ structure and X the $\mathrm{Voc}(A)$ reduct of Y, then the principal-function updates (including trivial updates) generated by $\mathrm{Prog}(B)$ in Y coincide with the those generated by $\mathrm{Prog}(A)$ in X, and the ancillary-function updates generated by $\mathrm{Prog}(B)$ in Y are consistent. $\triangleleft$

Corollary 3.2. *Suppose that B is a faithful expansion of an ASM A, then the following claims hold.*

1. *If $X_0, \ldots, X_n$ is a partial computation of A then there is a unique partial computation $Y_0, \ldots, Y_n$ of B such that every X_i is the* $\mathrm{Voc}(A)$ *reduct of the corresponding Y_i.*

2. *If states $Y_0, \ldots, Y_n$ of B form a partial computation of B, then their* $\mathrm{Voc}(A)$ *reducts form a partial computation $X_0, \ldots, X_n$ of A.*

3. *The* $\mathrm{Voc}(A)$ *reduct of a state of B is a state of A.*

If an ASM A computes a function F, one would expect that any faithful expansion of A computes function F as well. To confirm this expectation, we need to formalize what it means to compute a function. In the context of this paper, every ASM state is endowed with a special copy of the set $\mathbb{N}$ of natural numbers. This makes the desired formalization particularly easy for numerical partial functions $F : \mathbb{N}^k \to \mathbb{N}$.

Corollary 3.3. *Suppose that an ASM A computes a partial numerical function $F : \mathbb{N}^k \to \mathbb{N}$ in the following sense:*

1. *A has input variables $\iota_1, \dots, \iota_n$ taking numerical values in the initial states, and A has an output variable o,*
2. *all initial states of A are isomorphic except for the values of the input variables, and*
3. *the computation of A with initial state X eventually terminates if and only if F is defined at tuple $\bar{x} = (\mathcal{V}_X(\iota_1), \dots, \mathcal{V}_X(\iota_n))$, in which case the final value of o is $F(\bar{x})$.*

Then every faithful expansion of A computes F in the same sense. ◁

Corollary 3.3 can be generalized to computing more general functions and to performing other tasks, but this is beyond the scope of this paper.

An ASM may be faithfully expanded by instrumenting its program for monitoring purposes. For example, if you are interested how often a particular assignment σ fires, replace σ with a parallel composition

$$\sigma \quad \| \quad \kappa := \kappa + 1$$

where a fresh variable κ, initially zero, is used as a counter. A similar counter is used in our Reversibility Theorem below.

4 Reversibility

Definition 4.1. An ASM B is *reversible (as is)* if there is an ASM C which reverses all B's computations in the following sense. If $Y_0, Y_1, \dots, Y_n$ is a partial computation of B, then $Y_n, Y_{n-1}, \dots, Y_0$ is a terminating computation of C.

Theorem 4.2 (Reversification Theorem)**.** *Every ASM A has a faithful reversible expansion.*

Proof. Enumerate the (occurrences of the) assignments in Prog(A) in the order they occur:

$$\sigma_1, \sigma_2, \ldots, \sigma_N$$

It is possible that σ_i, σ_j are identical even though $i \neq j$. The metavariable n will range over numbers $1, 2, \ldots, N$. For each n, let f^n be the head of σ_n, $r_n =$ Arity(f^n), and t_0^n, t_1^n, $\ldots$, $t_{r_n}^n$ the terms such that

$$\sigma_n = \left(f^n(t_1^n, \ldots, t_{r_n}^n) := t_0^n\right).$$

We construct an expansion B of A. The ancillary symbols of B are as follows.

1. A variable κ.
2. For every n, a unary relation symbol $\mathtt{Fire}^n$.
3. For every n, unary function symbols f_0^n, f_1^n, $\ldots$, $f_{r_n}^n$.

The default term for κ is 0. The default term for all relations $\mathtt{Fire}^n$ is $\bot$. The default term for all functions f_0^n, f_1^n, $\ldots$, $f_{r_n}^n$ is `nil`. Accordingly, the initial states of B are obtained from the initial states of A by setting $\kappa = 0$, every $\mathtt{Fire}^n(x) = \bot$, and every $f_i^n(x) = \mathtt{nil}$,

The intention is this. If $X_0, X_1, \ldots, X_l$ is a partial computation of B, then for each $k = 0, \ldots, l$ we have the following.

1. The value of κ in X_k is k, so that κ counts the number of steps performed until now; we call it a *step counter*.
2. $\mathtt{Fire}^n(\kappa)$ holds in X_{k+1} if and only if σ_n fires in X_k.
3. The values of $f_1^n(\kappa), \ldots, f_{r_n}^n(\kappa)$ in X_{k+1} record the values of the terms $t_1^n, \ldots, t_{r_n}^n$ in X_k respectively, and the value of $f_0^n(\kappa)$ in X_{k+1} records the value of the term $f^n(t_1^n, \ldots, t_{r_n}^n)$ in X_k.

The program of B is obtained from Prog(A) by replacing every assignment σ_n with Instr(n) (an allusion to "instrumentation") where

```
Instr($n$) =
        $\sigma_n$   ||  $\kappa := \kappa + 1$  ||  $\mathtt{Fire}^n(\kappa+1) := \top$  ||
        $f^n_0(\kappa+1) := f^n(t^n_1, \dots, t^n_{r_n})$  ||
        $f^n_1(\kappa+1) := t^n_1$  ||  ...  ||  $f^n_{r_n}(\kappa+1) := t^n_{r_n}$
```

It is easy to check that the conditions (E1)–(E3) of Definition 3.1 hold, and B is indeed a faithful expansion of A. In particular, if Y and X are as in (E3) and X is terminal, then no assignment σ_n fires in X, and therefore no Instr(n) fires in Y, so that Y is terminal as well.

Lemma 4.3. *If $Y_0, \dots, Y_k$ is a partial computation of B, then*

- *$\kappa = k$ in Y_k and*
- *if $j > k$ then $\mathtt{Fire}^n(j), f^n_0(j), \dots, f^n_{r_n}(j)$ have their default values in Y_k.*

Proof of lemma. Induction on k. □

Now, we will construct an ASM C which reverses B's computations. The vocabulary of C is that of B, and any Voc(C) structure is an initial state of C. The program of C is

$$\mathtt{if}\ \kappa > 0\ \mathtt{then}\ \Big(\kappa := \kappa - 1 \ \ \|\ \ \underset{n}{\mathbf{PAR}}\, \mathrm{Undo}(n)\Big)$$

where **PAR** is parallel composition, n ranges over $\{1, 2, \dots, N\}$, and

```
Undo($n$) =
      if $\mathtt{Fire}^n(\kappa) = \top$ then
          $\mathtt{Fire}^n(\kappa) := \bot$  ||
          $f^n\big(f^n_1(\kappa), \dots, f^n_{r_n}(\kappa)\big) := f^n_0(\kappa)$  ||  $f^n_0(\kappa) := \mathtt{nil}$  ||
          $f^n_1(\kappa) := \mathtt{nil}$  ||  ...   ||  $f^n_{r_n}(\kappa) := \mathtt{nil}$
```

Lemma 4.4. *Let Y be an arbitrary nonterminal* Voc(B) *structure such that all functions $\mathtt{Fire}^n$ and f^n_i have their default values at argument $k = \mathcal{V}_Y(\kappa)$ in Y. If* Prog(B) *transforms Y to Y', then* Prog(C) *transforms Y' back to Y, i.e.,* Prog(C) *undoes the updates generated by* Prog(B) *and does nothing else.*

Proof of lemma. The updates generated by Prog(B) in Y are the updates generated by the rules Instr(n) such that σ_n fires in Y. Since $k = \mathcal{V}_Y(\kappa)$, we have $\mathcal{V}_{Y'}(\kappa) = k + 1 > 0$, and therefore Prog($C$) decrements κ. It also undoes the other updates generated by the rules Instr(n). Indeed, suppose that σ_n fires in Y.

To undo the update $\texttt{Fire}^n(k+1) \hookleftarrow \top$, Prog($C$) sets $\texttt{Fire}^n(k+1)$ back to $\bot$.

To undo the update $f^n(t_1^n, \dots, t_{r_n}^n) \hookleftarrow t_0^n$, generated by σ_n itself, Prog(C) sets $f^n\big(f_1^n(k+1), \dots, f_{r_n}^n(k+1)\big)$ to $f_0^n(k+1)$. Recall that $f_1^n(k+1), \dots, f_{r_n}^n(k+1)$ record $t_1^n, \dots, t_{r_n}^n$ in Y and $f_0^n(k+1)$ records the value of $f^n(t_1^n, \dots, t_{r_n}^n)$ in Y. Thus, Prog(C) sets $f^n(t_1^n, \dots, t_{r_n}^n)$ back to its value in Y.

To undo the updates of $f_0^n(k+1), f_1^n(k+1), \dots, f_{r_n}^n(k+1)$, Prog($C$) sets $f_0^n(k+1), f_1^n(k+1), \dots, f_{r_n}^n(k+1)$ back to $\texttt{nil}$.

Thus, being executed in Y', Prog(C) undoes all updates generated by the rules Instr(n) in Y. A simple inspection of Prog(C) shows that it does nothing else. Thus, Prog(C) transforms Y' to Y. □

Now suppose that $Y_0, \dots, Y_n$ is a computation of B, $k < n$, $Y = Y_k$, and $Y' = Y_{k+1}$. Then Y is nonterminal and, by Lemma 4.3, all $\texttt{Fire}^n(\kappa)$ and $f_i^n(\kappa)$ have their default values in Y. By Lemma 4.4, Prog(C) transforms Y_{k+1} to Y_k. The Y_0 is a terminal state of C. Thus, C reverses all B's computations. □

The proof of Reversification Theorem uses notation and the form of Prog(B) which is convenient for the proof. In examples and applications, notation and Prog(B) can be simplified.

Remark 4.5 (Notation)**.** Let σ_n be an assignment $(g(t_1^n, \dots, t_r^n) := t_0^n)$ so that f^n is g. If σ_n is the only g assignment in Prog(A) or if every other g assignment σ_m in Prog(A) is just another occurrence of σ_n, then the ancillary functions f_i^n may be denoted g_i; no confusion arises. ◁

Recall that a green light for an ASM A is a Boolean-valued expression that holds in the nonterminal states and fails in the terminal states.

Remark 4.6 (Green light and step counter)**.** In Prog(B), every Instr(n) has an occurrence of the assignment $\kappa := \kappa + 1$. A green light for A provides an efficient way to deal with this excess. Notice that B increments the step counter exactly when the green light is on.

Case 1: Prog(A) has the form $\;\texttt{if}\,\gamma\,\texttt{then}\;(k := k+1 \parallel \Pi)$.

In this case, γ is a green light for A, and A has already a step counter, namely k. Without loss of generality, k is the step counter κ used by Prog(B); if not, rename one of the two variables. Notice that the assignment $\sigma_1 = (k := k+1)$ needs no instrumentation. There is no need to signal firings of σ_1 because σ_1 fires at every step. And, when a step is completed, we know the previous value of the step counter; there is no need to record it.

Let Instr$^-$(Π) be the rule obtained from Π by first replacing every assignment σ_n with the rule Instr(n) defined in the proof of the program, and then removing all occurrences of $k := k+1$. Then the program

$$\texttt{if}\,\gamma\,\texttt{then}\,(k := k+1 \parallel \mathrm{Instr}^-(\Pi))$$

has only one occurrence of $k := k+1$ and is equivalent to Prog(B).

Case 2: Prog(A) = $(\texttt{if}\,\gamma\,\texttt{then}\,\Pi)$ where γ is a green light for A and the step counter κ of Prog(B) does not occur in Prog(A).

The modified program $\;\texttt{if}\,\gamma\,\texttt{then}\,(\kappa := \kappa+1 \parallel \Pi)$,
where κ is the step counter of B, is a faithful expansion of Prog(A), and thus Case 2 reduces to Case 1.

Case 3 is the general case.

By Lemma 2.6, every ASM program has a green light. If γ is a green light for A and Π = Prog(A), then the program $\;\texttt{if}\,\gamma\,\texttt{then}\,\Pi\;$ is equivalent to Prog(A), and thus Case 3 reduces to Case 2. ◁

The rules Instr(n) and Undo(n), described in the proof of the theorem, are the simplest in the case when $r_n = 0$. In such a case, σ_n has the form $v := t$ where v is a variable, so that $f^n = v$ and $t_0^n = t$. Then

```
Instr(n) =
    $\sigma_n$  ||  $\kappa := \kappa + 1$  ||  $\texttt{Fire}^n(\kappa+1) := \top$  ||  $v_0(\kappa+1) := v$,
Undo(n) =
            if $\texttt{Fire}^n(\kappa) = \top$ then
                $\texttt{Fire}^n := \bot$  ||  $v := v_0(\kappa)$  ||  $v_0(\kappa) := \texttt{nil}$.
```

Lemma 4.7. *Suppose that an assignment σ_n to a variable v can fire only at the last step of A and that the update generated by σ_n is never trivial. Then,* Instr(n) *and* Undo(n) *can be simplified to*

$$\begin{aligned}\text{Instr}(n) &= \qquad \sigma_n \;\; \| \;\; \kappa := \kappa + 1\\ \text{Undo}(n) &= \qquad \texttt{if}\ v \neq d\ \texttt{then}\ v := d\end{aligned}$$

where d is the default term for v in Voc(A).

We do not assume that σ_n fires at the last step of every computation of A, and so the expression $v \neq d$ is not necessarily a green light for A.

Proof. Suppose that σ_n fires in state Y of B. Then Y is nonterminal, $v = d$ in Y, the next state Y' is terminal, and $v \neq d$ in Y'. It is easy to see the simplified version of Undo(n) indeed undoes the updates generated by the simplified version of Instr(n) in Y. □

5 Examples

To illustrate the reversification procedure of §4, we consider three simple examples. By the reversification procedure we mean not only the constructions in the proof of Reversification Theorem, but also Remarks 4.5 and 4.6 and Lemma 4.7. Unsurprisingly, in each case, the faithful reversible expansion produced by the general-purpose procedure can be simplified.

5.1 Bisection algorithm

The well-known bisection algorithm solves the following problem where $\mathbb{R}$ is the field of real numbers. Given a continuous function $F : \mathbb{R} \to \mathbb{R}$ and reals a, b, ε such that $F(a) < 0 < F(b)$ and $\varepsilon > 0$, find a real c such that $|F(c)| < \varepsilon$. Here is a draft program for the algorithm:

```
if  $|F((a+b)/2)| \geq \varepsilon$    then
    if      $F((a+b)/2) < 0$   then $a := (a+b)/2$
    elseif  $F((a+b)/2) > 0$   then $b := (a+b)/2$
elseif $c = $ nil then $c := (a+b)/2$
```

The condition $c = \texttt{nil}$ in the last line ensures that computation stops when c is assigned a real number for the first time.

The Boolean expression $|F((a+b)/2)| \geq \varepsilon$ is not quite a green light for the algorithm. When it is violated for the first time, c is still equal to nil. But the equality $c = \texttt{nil}$ is a green light. With an eye on using Remark 4.6, we modify the draft program to the following program, our "official" program of an ASM A representing the bisection algorithm.

```
if  $c = \texttt{nil}$  then
    if      $F((a+b)/2) < -\varepsilon$    then $a := (a+b)/2$
    elseif  $F((a+b)/2) > \varepsilon$     then $b := (a+b)/2$
    else    $c := (a+b)/2$
```

Voc(A) consists of the obligatory symbols, the symbols in Prog(A), and the unary relation symbol Real. In every initial state of A, Real is (a copy of) the set of real numbers, the static functions of Prog(A) have their standard meaning, and $c = \texttt{nil}$.

Notice that Lemma 4.7 applies to Prog(A) with σ_n being $c := (a+b)/2$. Taking this into account, the reversification procedure of §4 gives us a reversible expansion B of A with the following program.

```
if $c = \texttt{nil}$ then
    $\kappa := \kappa + 1 \quad \|$
    if $F((a+b)/2) < -\varepsilon$ then
        $a := (a+b)/2 \quad \| \quad \texttt{Fire}^1(\kappa+1) := \top \quad \| \quad a_0(\kappa+1) := a$
    elseif $F((a+b)/2) > \varepsilon$ then
        $b := (a+b)/2 \quad \| \quad \texttt{Fire}^2(\kappa+1) := \top \quad \| \quad b_0(\kappa+1) := b$
    else $c := (a+b)/2$
```

This program can be simplified (and remain reversible). Notice that

- if $\texttt{Fire}^1(k+1) = \top$, then $\texttt{Fire}^2(k+1) = \bot$, the previous value of b is the current value of b, and the previous value of a is $2a - b$ where a, b are the current values; and
- if $\texttt{Fire}^1(k+1) = \bot$, then $\texttt{Fire}^2(k+1) = \top$, the previous value of a is the current value of a, and the previous value of b is $2b - a$ where a, b are the current values.

Thus, there is no need for functions a_0, b_0, recording the previous values of variables a, b, and there is no need for `Fire`^{2}. We get:

```
if $c = \texttt{nil}$ then
    $\kappa := \kappa + 1$  ||
    if $F((a+b)/2) < -\varepsilon$ then
        $a := (a+b)/2$  ||  $\texttt{Fire}^1(\kappa+1) := \top$
    elseif $F((a+b)/2) > \varepsilon$ then $b := (a+b)/2$
    else $c := (a+b)/2$
```

The corresponding inverse algorithm may have this program:

```
if $\kappa > 0$ then
  $\kappa := \kappa - 1$  ||  if $c \neq \texttt{nil}$ then $c := \texttt{nil}$
  || if $\texttt{Fire}^1(\kappa) = \top$ then $(\texttt{Fire}^1(\kappa) := \bot \parallel a := 2a - b)$
     else $b := 2b - a$
```

5.2 Linear-time sorting

The information needed to reverse each step of the bisection algorithm is rather obvious; you don't have to use our reversification procedure for that. Such information is slightly less obvious in the case of the following sorting algorithm.

For any natural number n, the algorithm sorts an arbitrary array f of distinct natural numbers $< n$ in time $\leq 2n$. Let m be the length of an input array f, so that $m \leq n$. The algorithm uses an auxiliary array g of length n which is initially composed of zeroes.

Here is a simple illustration of the sorting procedure where $n = 7$ and $f = \langle 3, 6, 0 \rangle$. Traverse array f setting entries $g[f[i]]$ of g to 1 for each index i of f, i.e., setting $g[3], g[6]$ and $g[0]$ to 1, so that g becomes $\langle 1, 0, 0, 1, 0, 0, 1 \rangle$. Each index j of g with $g[j] = 1$ is an entry of the input array f. Next, traverse array g putting the indices j with $g[j] = 1$ — in the order that they occur — back into array f, so that f becomes $\langle 0, 3, 6 \rangle$. Voila, f has been sorted in $m + n$ steps.

We describe an ASM A representing the sorting algorithm. Arrays will be viewed as functions on finite initial segments of natural numbers. The nonobligatory function symbols in Voc(A) are as follows.

0. Constants m, n and variables k, l.

1. Unary dynamic symbols f and g.

2. Binary static symbols $\quad <, +, - \quad$ where $<$ is relational.

In every initial state of A,

0. m and n are natural numbers such that $m \leq n$, and $k = l = 0$,

1. f, g are arrays of lengths m, n respectively, the entries of f are distinct natural numbers $< n$, and all entries of g are zero,

2. the arithmetical operations $+, -$ and relation $<$ work as expected on natural numbers.

In the following program of A, k is the step counter, and l indicates the current position in array f to be filled in.

```
if $k < m + n$ then
    $k := k + 1 \quad ||$
    if $k < m$ then $g(f(k)) := 1$
    elseif $g(k - m) = 1$ then $(f(l) := k - m \,||\, l := l + 1)$
```

The reversification procedure of §4 plus some simplifications described below give us a faithful reversible expansion B of A with a program

```
if $k < m + n$ then
    $k := k + 1 \quad ||$
    if $k < m$ then $(g(f(k)) := 1 \,||\, g_1(k + 1) := f(k))$
    elseif $g(k - m) = 1$ then
        $f(l) := k - m \quad || \quad l := l + 1 \quad || \quad f_0(k + 1) := f(l)$
```

We made some simplifications of Prog(B) by discarding obviously unnecessary ancillary functions.

- It is unnecessary to record the firings of assignment $\sigma_2 = (g(f(k)) := 1)$ because, in the states of B, the condition $\texttt{Fire}^2(k) = \top$ is expressed by the inequality $k \leq m$.

- The ancillary function g_0 recording the previous values of g is unnecessary because those values are all zeroes.
- The final two assignments in Prog(A) fire simultaneously, so that one fire-recording function, say $\mathtt{Fire}^3$, suffices. But even that one ancillary function is unnecessary because, in the states of B, the condition $\mathtt{Fire}^3(k) = \top$ is expressed by $m < k \wedge g(k - m - 1) = 1$.
- The ancillary functions f_1 and l_0 recording the previous value of l are unnecessary because we know that value, it is $l - 1$.

The desired inverse algorithm C may be given by this program:

```
if $k > 0$ then
    $k := k - 1$
    $\|$ if $k \le m$ then $(g(g_1(k)) := 0 \| g_1(k) :=$ nil)
    $\|$ if $m < k$ and $g(k - m - 1) = 1$ then
        $f(l) := f_0(k)$  $\|$  $l := l - 1$  $\|$  $f_0(k) :=$ nil
```

Obviously, A is not reversible as is; its final state doesn't have information for reconstructing the initial f. But do we need both remaining ancillary functions? Since f_0 is obliterated after the first n steps of C, f_0 seems unlikely on its own to ensure reversibility. But it does. The reason is that, after the first n steps of C, the original array f is restored. Recall that $g_1(k)$ records the value $f(k-1)$ of the original f for each positive $k \le m$, but we can discard g_1 and modify Prog(C) to

```
if $k > 0$ then $k := k - 1$
    $\|$ if $k \le m$ then $g(f(k - 1)) := 0$
    $\|$ if $m < k$ and $g(k - m - 1) = 1$ then
        $f(l) := f_0(k) \| l := l - 1 \| f_0(k) :=$ nil
```

Alternatively, we can discard f_0 but keep g_1. Indeed, the purpose of the assignment $f(l) := f_0(k)$ in Prog(C) is to restore $f(l)$ to its original value. But recall that every $f(l)$ is recorded as $g_1(l + 1)$. So we can modify Prog(C) to

```
if $k > 0$ then $k := k - 1$
    $\|$ if $k \le m$ then $(g(g_1(k)) := 0 \| g_1(k) :=$ nil)
    $\|$ if $m < k$ and $g(k - m - 1) = 1$ then
        $f(l) := g_1(l + 1)$  $\|$  $l := l - 1$
```

5.3 External functions and Karger's algorithm

Until now, for simplicity, we restricted attention to algorithms that are isolated in the sense that their computations are not influenced by the environment. Actually, the analysis of sequential algorithms generalizes naturally and easily to the case when the environment can influence the computation of an algorithm [13, §8]. To this end, so-called external functions are used.

Syntactically, the item (V3) in §2.2 should be refined to say that a function symbol f may be dynamic, or static, or external. Semantically, external functions are treated as oracles[5] When an algorithm evaluates an external function f at some input $\bar{x}$, it is the environment (and typically the operating system) that supplies the value $f(\bar{x})$. The value is well defined at any given step of the algorithm; if f is called several times, during the same step, on the same input $\bar{x}$, the same value is given each time. But, at a different step, a different value $f(\bar{x})$ may be given.

To illustrate reversification involving an external function, we turn attention to Karger's algorithm [15]. In graph theory, a minimum cut of a graph is a cut (splitting the vertices into two disjoint subsets) that minimizes the number of edges crossing the cut. Using randomization, Karger's algorithm constructs a cut which is a minimum cut with a certain probability. That probability is small but only polynomially (in the number of vertices) small. Here we are not interested in the minimum cut problem, only in the algorithm itself.

Terminology 5.1. Let $G = (V, E)$ be a graph and consider a partition P of the vertex set V into disjoint subsets which we call *cells*; formally P is the set of the cells. The *P-ends* of an edge $\{x, y\}$ are the cells containing the vertices x and y. An edge is *inter-cell* (relative to P) if its P-ends are distinct. ◁

Now we describe a version of Karger's algorithm that we call KA. Given a finite connected graph (V, E), KA works with partitions of the vertex set V, one partition at a time, and KA keeps track of the set Inter of the inter-cell edges. KA starts with the finest partition $P = \{\{v\} : v \in V\}$ and Inter $= E$. If the current partition P has > 2 cells, then Inter is nonempty because the graph (V, E) is connected. In this case, KA selects a random inter-cell edge e, merges the P-ends

[5]In that sense, our generalization is similar to the oracle generalization of Turing machines.

p, q of e into one cell, and removes from Inter the edges in $\{\{x, y\} : x \in p \wedge y \in q\}$. The result is a coarser partition and smaller Inter. When the current partition has at most two cells, the algorithm stops.

Next we describe an ASM A representing KA. There are many ways to represent KA as an ASM. Thinking of the convenience of description rather than implementation of KA, we chose to be close to naive set theory. Let U be a set that includes V and all subsets of V and all sets of subsets of V (which is much more than needed but never mind). The relation $\in$ on U has its standard meaning; the vertices are treated as atoms (or urelements), not sets.

The nonobligatory function symbols of Voc(A) are as follows.

0. Nullary variables P and Inter.

1. Unary static symbols $V, E, |.|$, and a unary external symbol R.

2. Binary static symbols $>, -$, Merge, and Intra, where $>$ is relational.

In every initial state of A,

- V, U and $\in$ are as described above (up to isomorphism). $|s|$ is the cardinality of a set s, and $-$ is the set-theoretic difference. The relation $>$ is the standard ordering of natural numbers

- E is a set of unordered pairs $\{x, y\}$ with $x, y \in V$ such that the graph (V, E) is connected. P is the finest partition $\{\{v\} : v \in V\}$ of V. Inter $= E$.

- If $e \in E$, S is a partition of V, and p, q are the S-ends of e, then

  - Merge$(e, S) = (S - \{p, q\}) \cup \{p \cup q\}$, and
  - Intra$(e, S) = \{\{x, y\} : x \in p \wedge y \in q\}$.

The external function R takes a nonempty set and returns a member of it. The program of A can be this:

```
if $|P| > 2$ then
    $P :=$ Merge($R$(Inter), $P$)
    $||$ Inter := Inter $-$ Intra($R$(Inter), $P$)
```

Now we apply the reversification procedure of Theorem 4.2, taking Remark 4.6 into account. We also take into account that both assignments fire at every step of the algorithm and so there is no need to record the firings. This gives us a faithful reversible expansion B of A with a program

```
if $|P| > 2$ then
    $\kappa := \kappa + 1 \quad \|$
    $P := \text{Merge}(R(\text{Inter}), P) \quad \| \quad P_0(\kappa + 1) := P \quad \|$
    $\text{Inter} := \text{Inter} - \text{Intra}(R(\text{Inter}), P) \quad \| \quad \text{Inter}_0(\kappa + 1) := \text{Inter}$
```

The corresponding inverse ASM C may be given by the program

```
if $\kappa > 0$ then
    $\kappa := \kappa - 1 \quad \|$
    $P := P_0(\kappa) \quad \| \quad P_0(\kappa) :=$ nil $\quad \|$
    $\text{Inter} := \text{Inter}_0(\kappa) \quad \| \quad \text{Inter}_0(\kappa) :=$ nil
```

Remark 5.2. A custom crafted faithful expansion may be more efficient in various ways. For example, instead of recording the whole P, it may record just one of the two P-ends of R(Inter). This would require a richer vocabulary.

6 Conclusion

We have shown how to reversify an arbitrary sequential algorithm A by gently instrumenting A with bookkeeping machinery. The result is a step-for-step reversible algorithm B whose behavior, as far as the vocabulary of A is concerned, is identical to that of A.

We work with an ASM (abstract state machine) representation of the given algorithm which is behaviorally identical to it. The theory of such representation is developed in [13], and the practicality of it has been amply demonstrated.

Acknowledgment

Many thanks to Andreas Blass for generous sanity check.

References

[1] Anas N. Al-Rabadi, “Reversible logic synthesis: From fundamentals to quantum computing,” Springer Berlin, 2014

[2] Charles H. Bennett, “Logical reversibility of computation," *IBM Journal of Research and Development* 17:6 (1973), 525–532

[3] Egon Börger and Robert Stärk, “Abstract state machines: A method for high-level system design and analysis," Springer Berlin, 2003

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

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

[6] Alexis De Vos, “Reversible computing: Fundamentals, quantum computing, and applications,” Wiley-VCH Weinheim, 2010

[7] Nachum Dershowitz and Yuri Gurevich, “A natural axiomatization of computability and proof of Church’s thesis,” *Bulletin of Symbolic Logic* 14:3 (2008), 299–350

[8] Edsger W. Dijkstra, “Selected writings on computing: A personal perspective,” Springer New York, 1982

[9] Edward Fredkin and Tommaso Toffoli , “Concervative logic,” International Journal of Theoretical Physics 21:3/4 (1982), 219–253

[10] David Gries, “The science of programming,” Springer New York, 1981

[11] Yuri Gurevich, “Evolving algebras: An introductory tutorial,” The Bulletin of the EATCS 43 (1991), 264–284, and (slightly revised) in “Current Trends in Theoretical Computer Science: Essays and Tutorials” (eds. G. Rozenberg and A. Salomaa), World Scientific, 1993, 266–292. (Abstract state machines used to be called evolving algebras.)

[12] “Evolving Algebra 1993: Lipari Guide,” in Specification and Validation Methods (ed. E. Börger), Oxford University Press 1995, 9–36. Reprinted at arXiv, `https://arxiv.org/abs/1808.06255`. (Abstract state machines used to be called evolving algebras.)

[13] Yuri Gurevich, “Sequential abstract state machines capture sequential algorithms,” ACM Transactions on Computational Logic 1:1 (2000), 77–111

[14] Yuri Gurevich and Jim Huggins, “The semantics of the C programming language,” in Proc. CSL’92, Computer Science Logic (eds. E. Börger et al.), Springer Lecture Notes in Computer Science 702 (1993), 274–308

[15] David Karger, “Global min-cuts in RNC and other ramifications of a simple min-cut algorithm,” Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993

[16] Andrei N. Kolmogorov, “On the concept of algorithm,” Uspekhi Matematicheskikh Nauk 8:4 (1953), 175–176 (Russian). English version in Vladimir Uspensky and Alexei Semenov, “Algorithms: Main ideas and applications,” Kluwer 1993, 18—19

[17] Kenichi Morita, “Theory of reversible computing,” Springer Japan, 2017

[18] Kalyan S. Perumala, “Introduction to reversible computing,” CRC Press Boca Raton, 2014

[19] RC2021, 13th International Conference on Reversible Computation, `https://reversible-computation-2021.github.io/`