Ordinary Interactive Small-Step Algorithms, II

ANDREAS BLASS
University of Michigan
and
YURI GUREVICH
Microsoft Research

This is the second in a series of three papers extending the proof of the Abstract State Machine Thesis — that arbitrary algorithms are behaviorally equivalent to abstract state machines — to algorithms that can interact with their environments during a step rather than only between steps. As in the first paper of the series, we are concerned here with ordinary, small-step, interactive algorithms. This means that the algorithms
(1) proceed in discrete, global steps,
(2) perform only a bounded amount of work in each step,
(3) use only such information from the environment as can be regarded as answers to queries, and
(4) never complete a step until all queries from that step have been answered.

After reviewing the previous paper's formal description of such algorithms and the definition of behavioral equivalence, we define ordinary, interactive, small-step abstract state machines (ASM's). Except for very minor modifications, these are the machines commonly used in the ASM literature. We define their semantics in the framework of ordinary algorithms, and we show that they satisfy the postulates for these algorithms.

This material lays the groundwork for the final paper in the series, in which we shall prove the Abstract State Machine Thesis for ordinary, intractive, small-step algorithms: All such algorithms are equivalent to ASMs.

Categories and Subject Descriptors: F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Interactive and Reactive Computation*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Sequential algorithms, Interaction, Postulates, Equivalence of algorithms, Abstract state machines

1. INTRODUCTION

In the first paper [Blass and Gurevich to appear (a)] of this series, we defined, by means of natural postulates, the class of ordinary, interactive, small-step algorithms. These are algorithms that do only a bounded amount of work in any single

Authors' addresses: Andreas Blass, Mathematics Department, University of Michigan, Ann Arbor, MI 48109–1043, U.S.A., ablass@umich.edu; Yuri Gurevich, Microsoft Research, One Microsoft Way, Redmond, WA 98052, U.S.A., gurevich@microsoft.com.

The work of the first author was partially supported by NSF grant DMS–0070723 and by a grant from Microsoft Research. Much of this paper was written during visits to Microsoft Research.

computation step (small-step) but can interact with their environments, by issuing queries and receiving replies, during a step (interactive); furthermore, they complete a step only after all the queries from that step have been answered, and they use no information from the environment except for the answers to their queries (ordinary). We shall briefly review the definition and the associated strong notion of behavioral equivalence in Section 2.

The purpose of the present paper is to introduce the corresponding abstract state machines (ASMs). The syntax of ASM programs is defined in Section 3; it differs very little from the syntax presented in [Gurevich 1995] and widely used in the ASM community (see for example [Huggins] and the literature linked there). Sections 4 and 5 define the semantics of ASMs by showing how to construe them as algorithms of the sort defined in [Blass and Gurevich to appear (a)]. This detailed, formal semantics of ASMs prepares the ground for the final paper [Blass and Gurevich to appear (b)] in this series, in which we shall prove the ASM thesis for ordinary, interactive, small-step algorithms: All such algorithms are equivalent to ASMs.

This series of papers continues the project, begun in [Gurevich 2000] and continued in [Blass and Gurevich 2003b], of analyzing natural classes of algorithms by first defining them precisely, by means of suitable postulates, and then showing that all algorithms in such a class are equivalent, in a strong sense, to ASMs. Further work on this project is under way [Blass, Gurevich, Rosenzweig, and Rossman (a); (b); (c)].

2. ORDINARY ALGORITHMS AND BEHAVIORAL EQUIVALENCE

In this section, we briefly review the definitions, conventions, and postulates from [Blass and Gurevich to appear (a)] that will be needed in the present paper. We do not, however, repeat the extensive explanations, motivations, and commentary given in [Blass and Gurevich to appear (a)]. So readers who wonder about the reasons for our postulates, definitions, and conventions should consult [Blass and Gurevich to appear (a)] for those aspects that deal with interaction and [Gurevich 2000] for those aspects common to all small-step algorithms. We also record some general information, much of it from [Blass and Gurevich to appear (a)], about the interaction mechanism described by our postulates.

2.1 Algorithms

We consider a fixed algorithm A. We may occasionally refer to it explicitly, for example to say that something depends only on A, but usually we leave it implicit.

States Postulate: The algorithm determines

—a nonempty set $\mathcal{S}$ of *states*,
—a nonempty subset $\mathcal{I} \subseteq \mathcal{S}$ of *initial states*,
—a finite vocabulary Υ such that every $X \in \mathcal{S}$ is an Υ-structure, and
—a finite set Λ of *labels*.

If X is a state, or indeed an arbitrary structure, we also write X for its base set. We adopt the following conventions, mostly from [Gurevich 1995], concerning vocabularies and structures.

Convention 2.1. —A vocabulary Υ consists of function symbols with specified arities.

—Some of the symbols in Υ may be marked as *static*, and some may be marked as *relational*. Symbols not marked as static are called *dynamic*.

—Among the symbols in Υ are the logic names: nullary symbols `true`, `false`, and `undef`; unary `Boole`; binary equality; and the usual propositional connectives. All of these are static and all but `undef` are relational.

—In any Υ-structure, the interpretations of `true`, `false`, and `undef` are distinct.

—In any Υ-structure, the interpretations of relational symbols are functions whose values lie in $\{\texttt{true}, \texttt{false}\}$.

—The interpretation of `Boole` maps `true` and `false` to `true` and everything else to `false`.

—The interpretation of equality maps pairs of equal elements to `true` and all other pairs to `false`.

—The propositional connectives are interpreted in the usual way when their arguments are in $\{\texttt{true}, \texttt{false}\}$, and they take the value `false` whenever any argument is not in $\{\texttt{true}, \texttt{false}\}$.

Definition 2.2. A *potential query* in state X is a finite tuple of elements of $X \sqcup \Lambda$. A *potential reply* in X is an element of X. An *answer function* is a partial map from potential queries to potential replies.

The use of the disjoint union $X \sqcup \Lambda$ here means that if X and Λ are not disjoint then they must be replaced by disjoint copies. (One could, of course, leave one of the two sets, say Λ, alone and change only the other, to make them disjoint.) For notational simplicity, we suppress mention of the copies, pretending in effect that X and Λ are always disjoint.

An answer function α represents, from the algorithm's point of view, the replies obtained from the environment. $\alpha(q)$ is the reply to query q. Since we deal only with ordinary algorithms, everything the algorithm does is determined by its program, its state, and an answer function representing the previous interaction with the environment. Thus, for our purposes, answer functions completely model the environment.

Interaction Postulate: The algorithm determines, for each state X, a relation $\vdash_X$, or just $\vdash$ when X is clear, between finite answer functions and potential queries. This relation, which can be quite arbitrary, will be called the *causality relation* of the algorithm.

The intuitive meaning of $\alpha \vdash_X q$ is that, if the algorithm's current state is X and the answers received from the environment, during the current step, include those described by α, then the algorithm issues the query q.

We use the standard notations $\alpha \restriction Z$ and $\beta \subseteq \alpha$ to mean, respectively, the restriction of an answer function α to the set Z and the statement that β is a restriction of α.

Definition 2.3. A *context* for a state X is an answer function that is minimal (with respect to $\subseteq$) among answer functions closed under causality. More explicitly, it is an answer function α with the following properties:

—For all answer functions ξ and all potential queries q, if $\xi \vdash_X q$ and $\xi \subseteq \alpha$, then $q \in \mathrm{Dom}(\alpha)$.

—For any $Z \subseteq \mathrm{Dom}(\alpha)$, if

$$\forall \xi \, \forall q \, [\text{if } \xi \vdash_X q \text{ and } \xi \subseteq \alpha \restriction Z \text{ then } q \in Z],$$

then $Z = \mathrm{Dom}(\alpha)$.

See [Blass and Gurevich to appear (a)], especially the paragraphs following Definition 5.5, for a detailed discussion of the notion of context, including in particular the second clause in the definition.

Given an answer function α for a state X, we define a monotone operator $\Gamma_{X,\alpha}$, or just Γ_α when X is understood, on sets of potential queries by

$$\Gamma_\alpha(Z) = \{q : (\exists \xi \subseteq \alpha \restriction Z)\, \xi \vdash_X q\}.$$

This is the set of queries that the algorithm would issue if it has already issued the queries in Z (and no other queries) and received the answers given by $\alpha \restriction Z$ (and no other answers).

For monotone operators Γ in general, we define the iteration of Γ by

$$\Gamma^0 = \varnothing, \qquad \Gamma^{n+1} = \Gamma(\Gamma^n).$$

In general, this iteration would continue transfinitely, taking unions at limit ordinal stages. It was, however, shown in [Blass and Gurevich to appear (a), Lemma 5.19] that for the operators Γ_α the iteration stabilizes after a finite number of steps. That is, for these operators, there is a finite n such that Γ^n is the least fixed point Γ^∞ of Γ. Recall that the least fixed point of a monotone operator Γ is also the smallest set that is *closed*[1] under Γ, i.e., the smallest Z such that $\Gamma(Z) \subseteq Z$.

Definition 2.4. A *location* in a state X is a pair $\langle f, \mathbf{a}\rangle$ where f is a dynamic function symbol from Υ and $\mathbf{a}$ is a tuple of elements of X, of the right length to serve as an argument for the function f_X interpreting the symbol f in the state X. The *value* of this location in X is $f_X(\mathbf{a})$. An *update* for X is a pair (l, b) consisting of a location l and an element b of X. An update (l, b) is *trivial* (in X) if b is the value of l in X. We often omit parentheses and brackets, writing locations as $\langle f, a_1, \ldots, a_n\rangle$ instead of $\langle f, \langle a_1, \ldots, a_n\rangle\rangle$ and writing updates as $\langle f, \mathbf{a}, b\rangle$ or $\langle f, a_1, \ldots, a_n, b\rangle$ instead of $(\langle f, \mathbf{a}\rangle, b)$ or $(\langle f, \langle a_1, \ldots, a_n\rangle\rangle, b)$.

The intended meaning of an update $\langle f, \mathbf{a}, b\rangle$ is that, at the end of the step, the interpretation of f is to be changed (if necessary, i.e., if the update is not trivial) so that its value at $\mathbf{a}$ is b. This intention is formalized in the following postulate.

Update Postulate: For any state X and any context α for X, either the algorithm provides an *update set* $\Delta^+_A(X, \alpha)$ whose elements are updates or it *fails* (or both). It produces a *next state* $\tau_A(X, \alpha)$ if and only if it doesn't fail. If there is a next state $X' = \tau_A(X, \alpha)$, then it

[1] sometimes called pre-fixed

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

It follows from the Update Postulate that, if $\Delta_A^+(X, \alpha)$ clashes, i.e., if it contains two distinct updates of the same location, then A must fail in state X and context α, because the next state, being subject to contradictory requirements, cannot exist. Similarly, if the structure X' described in the postulate is not a state of the algorithm, then the algorithm must fail.

Definition 2.5. If $i : X \cong Y$ is an isomorphism of states, extend it to act on potential queries by applying i to components from X and leaving components from Λ unchanged. Also extend it to act on locations, by acting componentwise on the tuple of elements of X and leaving the dynamic function symbol unchanged. Finally, extend it to act on updates by acting on both components, the location and the new value. We use the same symbol i for all these extensions, mapping the potential queries, locations, and updates of X bijectively to those of Y.

Notice that any isomorphism $i : X \cong Y$ of states, induces a one-to-one correspondence between answer functions for X and answer functions for Y; the correspondence sends any ξ to $i \circ \xi \circ i^{-1}$ (where, as usual, composition works from right to left).

Isomorphism Postulate:

—Any structure isomorphic to a state is a state.
—Any structure isomorphic to an initial state is an initial state.
—Any isomorphism $i : X \cong Y$ of states preserves causality, i.e., if $\xi \vdash_X q$ then $i \circ \xi \circ i^{-1} \vdash_Y i(q)$.
—If $i : X \cong Y$ is an isomorphism of states and if α is a context for X, then
  —the algorithm fails in (X, α) if and only if it fails in $(Y, i \circ \alpha \circ i^{-1})$, and
  —if the algorithm doesn't fail, then $i[\Delta^+(X, \alpha)] = \Delta^+(Y, i \circ \alpha \circ i^{-1})$

Here and in the rest of the paper, we use the following convention to avoid needless repetition.

Convention 2.6. An equation between possibly undefined entities (like $\Delta^+(X, \alpha)$) means, unless the contrary is explicitly stated, that either both sides are defined and equal, or neither side is defined.

It follows from the last part of the Isomorphism Postulate that, under the assumptions there, if $\tau(X, \alpha)$ is defined, then so is $\tau(Y, i \circ \alpha \circ i^{-1})$, and i is an isomorphism from the former to the latter.

Bounded Work Postulate:

—There is a bound, depending only on the algorithm A, for the lengths of the tuples that serve as queries. That is, the lengths of the tuples in $\mathrm{Dom}(\alpha)$ are uniformly bounded for all contexts α and all states.

—There is a bound, depending only on A, for the cardinalities $|\mathrm{Dom}(\alpha)|$ for all contexts α in all states.

—There is a finite set W of terms, depending only on A, with the following properties. Assume that
  —X and X' are states,
  —α is an answer function for both X and X', and
  —each term in W has the same values in X and in X' when the variables are given the same values in $\mathrm{Range}(\alpha)$.

  If $\alpha \vdash_X q$, then also $\alpha \vdash_{X'} q$. In particular, q is a potential query for X'. If, in addition, α is a context for X (and therefore for X'; see [Blass and Gurevich to appear (a), Section 5]), then
  —if the algorithm fails for either of (X, α) and (X', α), then it also fails for the other, and
  —if it doesn't fail, then $\Delta^+(X, \alpha) = \Delta^+(X', \alpha)$.

Definition 2.7. A set W of terms with the properties required in the last part of the Bounded Work Postulate is called a *bounded exploration witness* for the algorithm A.

We shall often have to deal with the hypotheses considered in the last part of the Bounded Work Postulate, so we introduce the following abbreviated terminology.

Definition 2.8. If

—X and X' are states,

—α is an answer function for both X and X', and

—each term in W has the same values in X and in X' when its variables are given the same values in $\mathrm{Range}(\alpha)$,

then we say that X and X' *agree over* α *with respect to* W.

When we use this terminology, we often omit "with respect to W" because it will be clear what set W is under consideration. Notice that if X and X' agree over α then they also agree over any subfunction of α.

Definition 2.9. An *ordinary, interactive, small-step algorithm* is any entity satisfying the States, Interaction, Update, Isomorphism, and Bounded Work Postulates. We sometimes omit "interactive, small-step" and write simply *ordinary algorithm*; sometimes we even omit "ordinary".

2.2 Reachability and equivalence

For a fixed algorithm A and a fixed state X, and therefore a fixed causality relation $\vdash$, we define reachability of queries with respect to an answer function and well-foundedness of answer functions as follows.

Definition 2.10. A query q is *reachable* under α if it is a member of ${\Gamma_\alpha}^\infty$. Equivalently, there is a *trace*, a finite sequence $\langle q_1, \ldots, q_n \rangle$ of queries, ending with $q_n = q$, and such that each q_i is caused by some subfunction of $\alpha \restriction \{q_j : j < i\}$.

For the equivalence of the two versions of the definition, see [Blass and Gurevich to appear (a)] from Definition 6.9 to Proposition 6.11.

Definition 2.11. An answer function α is *well-founded* if $\mathrm{Dom}(\alpha) \subseteq {\Gamma_\alpha}^\infty$.

As explained in [Blass and Gurevich to appear (a)], the intuition behind this definition is that α could actually arise at some point during a computation step; it does not answer queries that the algorithm does not ask. But α need not be the complete answer function for a whole step. The complete answer functions in this sense are the contexts, and they satisfy the equality $\mathrm{Dom}(\alpha) = {\Gamma_\alpha}^\infty$ in place of the inclusion that defines well-foundedness. Thus, for example, the empty function is always well-founded, but it is a context only if it causes no queries.

Definition 2.12. Two causality relations are *equivalent* if, for every answer function α, they make the same queries reachable under α.

Proposition 6.21 of [Blass and Gurevich to appear (a)] gives some equivalent characterizations of equivalence of causality relations. We shall need only the following consequence, which is Corollary 6.22 in [Blass and Gurevich to appear (a)].

Corollary 2.13. *If two causality relations are equivalent then they give rise to the same ${\Gamma_\alpha}^n$ for all α and n. In particular, they give rise to the same ${\Gamma_\alpha}^\infty$, the same well-founded answer functions, and the same contexts.*

Definition 2.14. Two algorithms are *(behaviorally) equivalent* if they have

—the same states and the same initial states,
—the same vocabulary Υ and the same set Λ of labels,
—equivalent causality relations in every state,
—failures in exactly the same states and contexts, and,
—for every state X and context α in which they do not fail, the same update set $\Delta^+(X, \alpha)$.

In the tradition of [Gurevich 2000] and [Blass and Gurevich 2003b], this is a very strict notion of equivalence. It implies, in particular, that equivalent algorithms simulate each other step-for-step. Since the main result in [Blass and Gurevich to appear (b)] will be that every ordinary, interactive, small-step algorithm is equivalent to an ASM, the stricter our notion of equivalence, the stronger the theorem.

Remark 2.15. The requirement that equivalent algorithms have the same vocabulary is redundant, because Υ is uniquely determined by any Υ-structure, in particular by any state. This remark is the only use we make of the assumption, in the States Postulate, that $\mathcal{S}$ is not empty.

Remark 2.16. The requirement, in the definition of equivalence, that the two algorithms have the same states could plausibly be weakened to require only that they have the same reachable states. Here, "reachable" means starting with initial states and repeatedly executing the algorithm's transition function with some answer functions. We refrain from adopting this alternative definition for three reasons. First, as indicated above, our stricter notion of equivalence makes our main theorem stronger. Second, in practice it is often easy to check whether something is a state but harder (or impossible) to check whether it is reachable. Third,

we prefer to continue in the tradition of [Gurevich 2000] and [Blass and Gurevich 2003b], where equivalence required the states to be the same.

Remark 2.17. Our definition of equivalence is based on a view of algorithms operating in isolation, except that the outside world answers their queries. Consider, for example, an algorithm A whose causality relation, in any state, consists of just a single instance, $\{(q, r)\} \vdash q'$. (See [Blass and Gurevich to appear (a), Example 6.2].) Despite this instance, the algorithm A, running alone, will issue no queries; the cause, $\{(q, r)\}$ cannot be realized because there is no way for A to issue q.

But imagine A running in parallel with another algorithm B that issues q. Then an answer r to that query might be construed as causing our original algorithm A to issue q'.

Our notion of equivalence makes A equivalent to an algorithm with empty causality relation because, although q' has a cause, it is not reachable. It would seem that, in order to treat algorithms running in combination (parallel or sequential) with other algorithms, we should modify the definition of equivalence to take into account the algorithm's response to "unsolicited" information like the reply r to a query q that the algorithm never asked (and never could ask).

A solution to this problem is implicitly contained in [Blass and Gurevich to appear (a), Section 2]. Unsolicited information can affect the algorithm's computation only if the algorithm pays attention to it, and the act of paying attention can be construed as an implicit query. In general, the algorithm will not "know" what sort of unsolicited information it may receive, but we can imagine a general implicit query q_0 asking for whatever relevant information may be available. ("Relevant" here can be taken to mean "appearing among the causes of the algorithm's causality relation.") Imagine, for example, a person doing a computation (implementing an algorithm), but nevertheless able to react to sufficiently loud, sudden noises (like a knock on the door). An algorithm modeling all the possibilities could represent the person's sensitivity to unexpected noises (and other sensory input) as a query. This is the sort of thing we mean by a general implicit query for relevant information. The reply to such a query could then be of the form (q, r) (if tuple-coding is available in the state), meaning "some agent asked q and the reply is r." (In the absence of tuple-coding, the same effect could be achieved by a longer conversation between the algorithm and the environment.)

In general, given an algorithm with a non-well-founded causality relation, such as the one with $\{(q, r)\} \vdash q'$ discussed above, it may not be obvious whether the presence of the unreachable query is merely a result of sloppy programming or whether the author of the program really intended that query q' be issued if some other process were to issue q and get reply r. The use of a general query can remove this ambiguity. If this instance $\{(q, r)\} \vdash q'$ was really intended, then the causality relation should include, in place of this instance, the instances $\{(q_0, (q, r))\} \vdash q'$ and $\varnothing \vdash q_0$ to make the intention clear. Similarly, one can modify other non-well-founded causality relations to make them well-founded and express the intention that the originally non-well-founded parts should become active if, as a result of another process's queries, their causes are realized. If, on the other hand, the non-well-founded part of the original causality relation was just junk, it should be

deleted. In general, whenever an algorithm is to be used as a component of a larger system, it should be augmented by the necessary general queries and ways of handling the replies, so that it behaves as intended in the presence of other components. The intention may be suggested by non-well-founded parts of a causality relation, but for actual use it should be made explicit, not merely suggested.

The use of general queries presupposes some integrity constraints on the environment. First, when it provides an answer r to a query q, the environment must also provide the answer (q, r) to any general query seeking this information. Second, if the answer to the general query "what relevant information is out there" is "nothing," i.e., if the environment has not provided any answers of the sort sought by q_0, then q_0 should get a reply saying "nothing." This is needed because an ordinary algorithm cannot complete its step until all its queries have been answered.

We do not explore further the details of general queries (e.g., how many should be issued, when should a reply to one trigger a new one, etc.). For our present purposes, they are to be treated no differently than any other queries. An algorithm can specify them and their use just as it does for more traditional queries. Nothing we do requires a distinction between general and traditional queries (and in fact one can imagine fuzzy borderline cases).

In the case of interactive components, it makes good sense to study the phenomenon of non-well-founded causality relations in greater depth. This is done in [Blass, Gurevich, Rosenzweig, and Rossman (b)].

2.3 Answer functions and their approximations

In this subsection, we collect for later use some facts about answer functions. Though they seem fairly technical and will be used for technical purposes, we hope that they may also help to guide intuition. Much of this material is from [Blass and Gurevich to appear (a)], and we do not repeat proofs from there. Unless otherwise specified, we deal with a fixed causality relation $\vdash$, meaning $\vdash_X$ for a fixed state X.

LEMMA 2.18. *Let α be an answer function. If ${\Gamma_\alpha}^\infty \subseteq Dom(\alpha)$ then $\alpha \restriction {\Gamma_\alpha}^\infty$ is the unique context that is $\subseteq \alpha$. If ${\Gamma_\alpha}^\infty \not\subseteq Dom(\alpha)$ then there is no context $\subseteq \alpha$. In particular, α has at most one subfunction that is a context. Thus, if α and β are two distinct contexts for the same state, then $\alpha(q) \neq \beta(q)$ for some $q \in Dom(\alpha) \cap Dom(\beta)$.*

Proof See Lemma 5.7 and Corollary 5.8 in [Blass and Gurevich to appear (a)]. □

Definition 2.19. For any answer function α and natural number n, we write α^n for $\alpha \restriction {\Gamma_\alpha}^n$. Similarly, $\alpha^\infty = \alpha \restriction {\Gamma_\alpha}^\infty$.

Remark 2.20. In [Blass and Gurevich to appear (a)], we used the notation α_n instead, but we shall need that notation for other purposes here, so we have lifted the n to be a superscript, matching the n in ${\Gamma_\alpha}^n$.

LEMMA 2.21. *If $\alpha \subseteq \beta$ then ${\Gamma_\alpha}^n \subseteq {\Gamma_\beta}^n$ and $\alpha^n \subseteq \beta^n$ for all natural numbers n.*

Proof The first assertion follows from the definition of the Γ operators, and the second is an immediate consequence of the first. □

LEMMA 2.22. *For any answer function α, each α^n is well-founded. α^∞ is the largest well-founded subfunction of α.*

Proof See [Blass and Gurevich to appear (a)] from Proposition 6.16 to Proposition 6.18. □

LEMMA 2.23. *Suppose that an answer function α includes both a context β with respect to $\vdash$ and an answer function η that is well-founded with respect to $\vdash$. Then $\eta \subseteq \beta$*

Proof Since both η and β are restrictions of α, it suffices to prove that $\mathrm{Dom}(\eta) \subseteq \mathrm{Dom}(\beta)$. Since η is well-founded, it suffices to prove ${\Gamma_\eta}^\infty \subseteq \mathrm{Dom}(\beta)$, and for this purpose we prove, by induction on k, that ${\Gamma_\eta}^k \subseteq \mathrm{Dom}(\beta)$.

This inclusion is vacuously true for $k = 0$. Suppose it is true for a certain k, and suppose $q \in {\Gamma_\eta}^{k+1} = \Gamma_\eta({\Gamma_\eta}^k)$. This means that there is $\delta \subseteq \eta \restriction {\Gamma_\eta}^k$ such that $\delta \vdash q$. By induction hypothesis, $\delta \subseteq \beta$. Therefore, $q \in \Gamma_\beta(\mathrm{Dom}(\beta)) = \mathrm{Dom}(\beta)$, where the last equality comes from the assumption that β is a context. □

LEMMA 2.24. *Every well-founded answer function is a subfunction of some context.*

Proof Let ξ be any well-founded answer function for a state X. Let β be an arbitrary extension of ξ to an answer function whose domain contains all the potential queries for X. Then, since ${\Gamma_\beta}^\infty$ consists of queries, it is included in $\mathrm{Dom}(\beta)$. By Lemma 2.18, there is a context $\alpha \subseteq \beta$. By Lemma 2.23, $\xi \subseteq \alpha$, as desired. □

COROLLARY 2.25. *The Bounded Exploration Postulate implies that there is a bound, depending only on the algorithm, for the number and length of the queries in any well-founded answer function for any state.*

3. ABSTRACT STATE MACHINES — SYNTAX

3.1 Informal description of ASMs

In this section, we describe the syntax of abstract state machines. This syntax is nearly the same as in [Gurevich 2000, Section 6]; the only differences are that we include

—outputs,

—a `let` construction for remembering values, and

—an explicit failure instruction.

There will also be a critical difference in the semantics, namely that interaction with the environment (especially via external functions) can take place within steps, not only between steps as in [Gurevich 2000], but we postpone discussion of semantics to Sections 4 and 5.

Except for the explicit failure instruction (on which we shall comment further below), the differences from [Gurevich 2000, Section 6] are not new here. Sequential ASMs with outputs were defined in [Blass and Gurevich 2003b]. We slightly extend that definition by allowing several output channels. The `let` construction also occurs briefly in a remark in [Gurevich 1995] at the end of Subsection 3.1; a fuller

presentation is in [Gurevich 2000, Subsection 7.3]. We extend it by allowing several variables to be bound by a single use of `let`, rather than requiring nesting of several `let` rules.

To distinguish the ASMs defined here from other classes of ASMs, it is natural to call them ordinary, interactive, small-step ASMs, but, as they are the only ASMs considered in this paper, we just call them ASMs here.

As is standard in the recent ASM literature (see for example [Gurevich 1997]) and for the most part also in older ASM literature (like [Gurevich 1995]), we take the vocabulary Υ and the states of an ASM to be subject to Convention 2.1. Thus, static logic names are available and interpreted correctly in all Υ-structures, in particular in all states.

Our ASMs interact with the environment in two ways, outputs and external functions, both of which involve additional "symbols" (beyond those in the vocabulary Υ). For outputs, an ASM determines a finite set of *output labels*, which we think of as corresponding to different output channels, such as a computer screen or a printer or an e-mail server. The program of the ASM can contain commands of the form $\texttt{Output}_l(t)$ where l is an output label and t a term; the meaning of this command is to output the value (in the current state) of t on channel l. This extension of basic ASMs to allow output was introduced in [Blass and Gurevich 2003b, Section 2]. The official definition there omitted output labels, in effect allowing only one output channel. This involved no real loss of generality, since the structures used in [Blass and Gurevich 2003b] always included enough set-theoretic background to permit any desired labels to be coded as part of the term t. In the present paper, we do not need so much background for other purposes, so, to avoid an artificial restriction of generality, it is better to allow several output channels.

The *external function symbols* of an ASM constitute a second vocabulary E, disjoint from the ASM's own ("internal") vocabulary Υ. The symbols of E can be used along with those of Υ in forming terms, but their interpretations are not part of the ASM's state. Rather, the meaning of the external function symbols is given by the environment. Thus, if $f \in E$, if $f(t_1, \ldots, t_n)$ is to be evaluated in the course of execution of the ASM's program, and if the t_i have already been evaluated as elements a_i of the state, then the required value of $f(a_1, \ldots, a_n)$ is obtained from the environment.

We note that this use of external functions goes beyond that in [Gurevich 1995] by allowing nesting of external function symbols. Such nesting is useful in applications, and it also reflects our purpose in this paper, namely to study environmental interactions that occur during a step, rather than between steps, of a computation.

3.2 Syntax of ASMs

The preceding discussion introduces the three main classes of symbols used by our ASMs: the function symbols of the state vocabulary Υ, the external function symbols, and the output labels. In addition, ASMs use a few keywords and symbols (see the definitions of terms and rules below) and variables of two kinds, general variables and Boolean variables.

The two categories of meaningful expressions in ASM programs are terms and rules.

Definition 3.1. *Terms* are built just as in traditional first-order logic, using function symbols from $\Upsilon \cup E$ and variables.

Note that external function symbols are allowed alongside the function symbols of the states' vocabulary with no restrictions on nesting. The *Boolean terms* are the Boolean variables and the compound terms whose outermost function symbol is relational.

Definition 3.2. *Rules* are defined by the following recursion.

—If f is a dynamic n-ary function symbol in Υ, $t_1, \ldots t_n$ are terms, and t_0 is a term that is Boolean if f is relational, then

$$f(t_1, \ldots, t_n) := t_0$$

is a rule, called an *update rule*.

—If l is an output label and t is a term, then

$$\texttt{Output}_l(t)$$

is a rule, called an *output rule*.

—If k is a natural number (possibly zero) and $R_1, \ldots, R_k$ are rules, then

$$\texttt{do in parallel } R_1, \ldots, R_k \texttt{ enddo}$$

is a rule, called a *parallel combination* or a *block*. The subrules R_i are called its *components*.

—If φ is a Boolean term and if R_0 and R_1 are rules, then

$$\texttt{if } \varphi \texttt{ then } R_0 \texttt{ else } R_1 \texttt{ endif}$$

is a rule, called a *conditional rule*. We call φ its *guard* and R_0 and R_1 its *branches*.

—If $x_1, \ldots, x_k$ are variables, if $t_1, \ldots, t_k$ are terms with each t_i Boolean if x_i is, and if R_0 is a rule, then

$$\texttt{let } x_1 = t_1, \ldots, x_k = t_k \texttt{ in } R_0 \texttt{ endlet}$$

is a rule, called a *let rule*. We call $x_1, \ldots, x_k$ its *variables*, $t_1, \ldots, t_k$ its *bindings*, and R_0 its *body*.

—`Fail` is a rule.

Free and bound variables are defined as usual, with `let` as the only variable-binding operator. Specifically, in $\texttt{let } x_1 = t_1, \ldots, x_k = t_k \texttt{ in } R_0 \texttt{ endlet}$, the exhibited occurrences of the x_i's and all their occurrences in R_0 are bound. An ASM *program* is a rule with no free variables.

Remark 3.3. If a variable x_i of a let rule $\texttt{let } x_1 = t_1, \ldots, x_k = t_k \texttt{ in } R_0 \texttt{ endlet}$ occurs in a binding t_j, then (whether or not $i = j$) these occurrences are free in the rule. This situation would be excluded if we adopted the convention that no variable can have both free and bound occurrences in a rule and that no variable can be bound twice in a rule. This convention, analogous to a convention often made in first-order logic, is sometimes convenient, but we shall have no need for it here.

Remark 3.4. Other versions of ASM notation, for example in [AsmL] and [Gurevich 1995], avoid the need for end-markers like `endif` and `enddo` by using conventions about indentation of lines in programs. We do not impose such conventions here, so some markers are needed to ensure unique parsing. Readers are free to adopt other marking systems; outside this remark, we shall pay no attention to this issue.

Remark 3.5. Previous definitions of ASMs have not included the rule `Fail`. We have added it in order to be able to simulate, with ASMs, algorithms that may fail, i.e., that may have $\tau(X, \alpha)$ undefined for certain states X and contexts α. In almost all situations, we can do without `Fail`. Indeed, the semantics for ASMs, defined below, makes an algorithm fail if it attempts two conflicting updates. Thus, if the vocabulary Υ contains a nullary, dynamic symbol d, then

$$\texttt{do in parallel } d := \texttt{true}, \; d := \texttt{false enddo}$$

is equivalent to `Fail`. If there is no such d but there is a dynamic f of higher arity, then there is a similar replacement for `Fail`, using, say, $f(\texttt{true}, \ldots, \texttt{true})$ instead of d. So the only time we really need `Fail` is when we are dealing with a vocabulary having no dynamic symbols. Such a situation may seem strange, but it is not entirely silly. An algorithm without dynamic symbols cannot update its state, but it can still interact with its environment by asking queries.

4. QUERIES AND TEMPLATES

4.1 Interaction via external functions

In this section, we prepare for the definition of the semantics of ASMs by discussing the main aspect not already treated in [Gurevich 1995; 2000], namely the interaction with the environment. Our ASMs interact with the environment by means of external functions and output rules. We treat the case of external functions first and afterward deal with outputs.

The basic idea here is simply that when the execution of an ASM program needs the value of a term $f(t_1, \ldots, t_n)$ that begins with an external function symbol f, it first evaluates the subterms t_i, obtaining values a_i, and then sends a query asking the environment for the value of $f(a_1, \ldots, a_n)$.

An important question that is not resolved by this basic idea is what to do if the same external function f occurs several times in the program and its arguments get, during the execution of the program, the same values at several of these occurrences. Much of this section is devoted to this question. Another question that we must address is what the queries actually are. Recall from our definition of algorithms that a query is a tuple of elements from the disjoint union of the (base set of the) state X and the fixed set Λ of labels. How, exactly, is a query of this sort to be assigned to the request for the value of $f(a_1, \ldots, a_n)$?

The simplest answer to this last question would be that the query is $\langle f, a_1, \ldots, a_n \rangle$, where we have included all the external function symbols f among the labels (i.e., $E \subseteq \Lambda$) so that this is a legitimate query.

Unfortunately, there are two problems with this answer. The first is that it pre-judges the question of how to treat repeated occurrences of the same external function symbol with the same values for its arguments. In such a situation, the

query $\langle f, a_1, \ldots, a_n \rangle$ would be the same for all the occurrences. Any reply to this query, given by an answer function, would be used as the value of f at all these occurrences. In effect, this means that, no matter how often f occurs with arguments $\mathbf{a}$, only one query is issued; the answer to this query is remembered and used at all occurrences. This is a reasonable convention, and in fact we shall adopt it, but the adoption should and will be based on a serious consideration of alternatives, not merely on an arbitrary decision about the representation of queries.

A second and even more serious problem with the $\langle f, a_1, \ldots, a_n \rangle$ representation of queries is that, under this convention, not all ordinary algorithms (in the sense defined in Section 2) are equivalent to ASMs. An ASM operating under this convention would issue only queries of the special form $\langle f, a_1, \ldots, a_n \rangle$ in which a label from Λ occurs in the first position and elements of the state occur in all subsequent positions (and similar queries resulting from output rules). Any algorithm issuing queries of a more general form, say with several components from Λ, would not be equivalent to an ASM, because equivalence of algorithms requires them to issue the same queries under the same circumstances. Thus, our main goal in defining ASMs, namely to model all ordinary, small-step algorithms, cannot be achieved under the $\langle f, a_1, \ldots, a_n \rangle$ convention.

Most of this section will be devoted to the solution of the problems just indicated. A final subsection will extend the discussion to output rules.

Remark 4.1. The decision in [Blass and Gurevich to appear (a)] to allow queries of a rather general form, rather than the special form $\langle f, a_1, \ldots, a_n \rangle$, has motivation in actual practice. Queries frequently look like

$$\langle \texttt{print file } x \texttt{ on printer } y \texttt{ at resolution } z \rangle$$

or

$$\langle \texttt{insert } x \texttt{ at position } y \texttt{ in file } z \rangle,$$

where the labels (everything but x, y, z) are spread throughout the query, not confined to the first position.

4.2 Templates

As indicated above, in order to simulate all ordinary algorithms, our ASMs must be able to ask queries of the same general form allowed in these algorithms, arbitrary tuples of elements from $X \sqcup \Lambda$; see Definition 2.2. These queries must, for the most part, result from the evaluation of external functions, since the other sort of interaction in our ASMs, via output rules, produces only queries of the trivial sort for which no reply is used by the algorithm. (As discussed in [Blass and Gurevich to appear (a), Section 2], such a query is regarded as having an automatic, uninformative "OK" as its reply.)

A minimal modification of the simple $\langle f, a_1, \ldots, a_n \rangle$ proposal above is to assign to each n-ary external function symbol f (or to each occurrence of such a symbol) what we shall call a template, a tuple consisting of labels from Λ and place-holders $\#1, \ldots, \#n$ for the arguments of f. The query asking for the value of $f(a_1, \ldots, a_n)$ is then obtained by replacing each $\#i$ in the template by a_i. This approach provides the necessary flexibility in the format of the queries issued by an ASM. Furthermore, by attaching templates to occurrences of external function symbols rather

than to the symbols themselves, we can accommodate the possibility that different occurrences of the same f with the same arguments $\mathbf{a}$ produce different queries.

Remark 4.2. Our use of templates may seem too simplistic in that all the Λ components of a query are determined by the (occurrence of the) function symbol and all the X components of the query are the arguments of the function. Why couldn't some Λ components be determined by the arguments, and why couldn't some of the X components be determined by the function symbol?

An easy answer is that this extra complexity is not needed. Our approach is, as we shall prove in [Blass and Gurevich to appear (b)], adequate to provide ASMs equivalent to all ordinary algorithms.

Furthermore, deviations from our approach lead to intuitively unnatural situations. For example, suppose a Λ component of the query is to be determined by the arguments. That would involve a (possibly small but non-vacuous) computation, to determine which element of Λ corresponds to a given argument tuple. If such a computation is to be done, it would be better to include it explicitly in the program, rather than hiding it in the conversion of external function calls to queries. Our approach makes this conversion trivial: take the template and plug in the arguments.

A similar comment applies to the idea of having an X component of the query determined by the function symbol rather than by the arguments. If the component in question is specified as the value of some term, then the production of the query hides the task of evaluating that term.

Finally, if one tries to specify a component $a \in X$ of the query directly, rather than as the value of a term, then in order to satisfy the Isomorphism Postulate, there would have to be such a specification for every state (or at least every state isomorphic to X). That would be, in effect, interpreting (as a) a hidden constant symbol, not present in Υ but nevertheless available in the computation.

Definition 4.3. For a fixed label set Λ, a *template* for n-ary function symbols is any tuple in which certain positions are filled with labels from Λ while the rest are filled with the *placeholders* $\#1, \ldots, \#n$, occurring once[2] each. We assume that these placeholders are distinct from all the other symbols under discussion ($\Upsilon \cup E \cup \Lambda$). If t is a template for n-ary functions, then we write $t[a_1, \ldots, a_n]$ for the result of replacing each placeholder $\#i$ in t by the corresponding a_i.

The intended meaning of a template is a format for queries about n-ary external functions. The query for $f(a_1, \ldots, a_n)$ using template t is obtained from t by replacing each of the placeholders $\#i$ with the corresponding a_i. Thus, the template tells, for each position in the resulting query, whether it should contain a label or an element of the state; if it should contain a label, then the template specifies the label; if it should contain an element of the state, then the template specifies (using one of the placeholders) which argument of the external function should be there.

In the following subsections, we shall discuss various possible conventions governing whether the same query can result from several occurrences of an external

[2]In principle, we could allow templates in which the same placeholder occurs several times. That would complicate the definition of "collide" below (if it is to retain its intuitive meaning), and it would not increase the expressive power of ASMs.

function symbol in an ASM program. In this discussion, it will be useful to have a short way to say that two template assignments could, given appropriate values for the arguments, produce the same query. That is the reason for the following terminology.

Definition 4.4. Two templates *collide* if they have the same length and have the same labels from Λ in the same positions. That is, they differ at most by a permutation of the placeholders $\#i$.

It is clear that two templates collide if and only if it is possible to produce the same query by replacing their placeholders $\#i$ by elements of a state (possibly different replacements for the two templates).

We now turn to the discussion of various conventions for the variability of queries associated to the same external function symbol with the same values for its arguments.

4.3 No variation — the Lipari convention

We consider first the convention used in the Lipari guide, [Gurevich 1995, Section 3.3.2], where it was formulated as follows: "[T]he oracle should be consistent during the execution of any one step of the program. In an implementation, this may be achieved by not reiterating the same question during a one-step execution. Ask the question once and, if necessary, save the result and reuse it."

In our present context, this means the following. Suppose an ASM program contains several occurrences of terms $f(\mathbf{t})$ that begin with the same external function symbol f (but may involve different tuples $\mathbf{t}$ of argument terms). Suppose further that, when the ASM program is executed in a particular state X with a particular answer function, the values of these argument terms are, at all the occurrences under consideration, the same tuple $\mathbf{a}$ of elements of X. Then only a single query is produced by the evaluation of all these occurrences. The environment's reply to this query is used as the value of all these subterms $f(\mathbf{t})$.

Another way to express this *Lipari convention* is that the same template is used at all occurrences of any one external function symbol. That is, we assign templates to external function symbols, not to their occurrences.

Notice that, for an ASM program to describe an algorithm, it must be accompanied by a template assignment, telling how to produce the queries that correspond to external function calls. The Lipari convention facilitates the syntactic description of the template assignment. One can simply append to an ASM program a table of its external function symbols and their associated templates.

The Lipari convention has, however, a serious disadvantage: In practice, one needs external functions whose value can be different for different occurrences within the same step of a computation. A typical example is the nullary external function symbol `new`, whose intended interpretation is an element chosen from the reserve (to be imported as a new element of the active part of the state; see [Gurevich 1995, Section 3.2] for information about reserve and importing, although `new` is not used there). The essential property of `new` is that it produces a different value each time it is evaluated. This directly contradicts the Lipari convention.

Also for other external function symbols, it frequently happens in practice that the value of such a function at a certain argument tuple changes during a step

of the computation. For example, in a distributed computation, each agent can be regarded as an algorithm whose environment includes all the other agents. If an agent reads a value written by another agent, then this amounts to obtaining the value of an external function of the reading agent. Because the agents work asynchronously, such a value may well change in the middle of the reading agent's step. It may even change several times during one step, if the writing agent works faster than the reading agent. (For simplicity we assume here that reads and writes are atomic and thus non-interfering operations.)

Such behavior can, however, be modeled within the Lipari convention, namely by replacing the various occurrences of such a "varying" external function symbol with different function symbols, perhaps by attaching subscripts or otherwise tagging them. Then one has a separate template for each of the new external function symbols. For the sake of human readability, it may be useful to precede such a modified ASM program with a preamble telling which of the external function symbols in the program correspond to which of the "intended" (untagged) symbols.

Remark 4.5. In fact, for the purpose of writing actual programs under the Lipari convention, it seems useful to automate the subscripting process by introducing syntactic sugar of the following sort. Allow the preamble to say that certain external function symbols (like `new`) are "vary-query," which means that the query to be issued varies every time the function symbol is evaluated. That is, each occurrence of such a symbol in the program is to be regarded as a different function symbol. The compiler should automatically attach different subscripts to all these occurrences. There would have to also be a convention for assigning templates to these subscripted symbols. For example, one could specify, for each vary-query function symbol, a template with one component left blank; then the subscripts could be automatically filled in for the blank components to form the templates for the subscripted function symbols.

As another practical matter, it may be useful to adopt a default convention for template assignments. In the case of a (non-vary-query) external, n-ary function symbol f, a reasonable default would be the template $\langle f, \#1, \ldots, \#n\rangle$. For an output label l the default could be $\langle l, \#1\rangle$. Then only deviations from these defaults would have to be explicitly indicated along with the program.

Alternatively, the traditional way of writing function symbols before their arguments could be augmented by notations that make the query explicit, as mentioned in Remark 4.1. That is, instead of function symbols, one would use the templates themselves, and the arguments would be written in place of the placeholders $\#i$.

4.4 Mandatory variation — the must-vary convention

At the opposite extreme from the Lipari convention is the *must-vary convention*. Under this convention, reflecting the treatment of procedure calls and new object creation in current standard programming practice, all occurrences of external function symbols are required to produce distinct queries. This means that templates must be assigned not to the external function symbols but to their occurrences in the program. Furthermore, the assignment must be such that there is no possibility of issuing the same query from different places in the program. This means that, if two of the assigned templates collide, then either at most one of them is actually

used in the execution of the program (for example if the relevant occurrences are on different branches of a conditional rule) or the values of the arguments must be such that the resulting queries differ. In general, this requirement is an undecidable, run-time property of the program, but one can ensure it with fairly simple syntactic requirements. For example, one can require that, if two occurrences have colliding templates and are not in different branches of any conditional rule, then they must occur within the scope of guards that guarantee enough distinctness of the arguments to make the queries distinct. Nevertheless, the requirements on the templates are fairly complicated.

It is also a bit complicated to describe the template assignment syntactically. If we wanted to append it as a table, just as we did for the Lipari convention, then we would have to indicate, in each row of the table, not just an external function symbol but a specific occurrence of such a symbol, for example by indicating its location in the program. Alternatively, we could write each template into the program, immediately after the relevant occurrence of an external function symbol. Both approaches work, but neither is as simple as the table available under the Lipari convention.

The main advantage of the must-vary convention is that it easily and automatically handles external function symbols like `new` whose values must change, and other external function symbols whose values may change, in mid-step. In view of the relevance of such behavior to distributed computation, one may expect that the must-vary convention will be useful for modeling distributed systems.

The question arises whether this convention can also handle the situation where only one query should be issued despite repeated occurrences in the program. That is, can it simulate the Lipari convention? In many cases, it can, by using `let`. For a simple example, consider the program

```
do in parallel
   a := e, b := f(e, e)
enddo
```

in which e is external and the other function symbols are internal. Under the must-vary convention, this would issue three queries about e, possibly getting three different values. But the Lipari convention's interpretation of this program, where all three e's necessarily get the same value, can be simulated under the must-vary convention by the program

```
let x := e in
   do in parallel
      a := x, b := f(x, x)
   enddo
endlet
```

Thanks to `let`, this program mentions e only once, so there is just one query about e.

Unfortunately, `let` does not suffice to handle all patterns of repeated use of the result of a single query. Consider, for example, the following scenario.

—An algorithm begins by issuing two queries, say q and q'.

—After receiving any reply r to q, it computes (without waiting for a reply to q') two things that depend on r, say $f_1(r)$ and $f_2(r)$ and issues new queries that depend on these.

—Similarly, after receiving any reply r' to q', it computes (without waiting for a reply to q) $f_1(r')$ and $f_2(r')$ and issues new queries that depend on these.

—Finally, if it gets answers to both q and q', then it uses them to compute some $g(r, r')$ and issues a query that depends on this.

We can simulate the first part of this, the part that uses only r, with a rule that begins `let` $x = e$ `in` ..., where e is an external function symbol producing the query q. Similarly, the part involving only r' can be handled by `let` $x' = e'$ `in` .... But the computation of $g(r, r')$ would require the program to have $g(x, x')$ in the scope of both occurrences of `let`. That could be achieved only by nesting the two occurrences or by combining them into `let` $x = e, x' = e'$ `in` .... Either way, at least one of the two earlier parts, either the r part or the r' part, will not be executed until after both q and q' are answered. So the causality relation of the ASM is inequivalent to that of the given algorithm and thus the ASM is not behaviorally equivalent to the given algorithm. (See, however, the discussion of "let by name" in [Blass and Gurevich to appear (b), Section 7].)

Because of such difficulties, we do not adopt the must-vary convention in this paper.

4.5 Flexible variation

Having treated the two extremes, the Lipari and must-vary conventions, we now consider a compromise, namely to put no a priori constraints on the interpretation of multiple occurrences of an external function symbol with the same argument values. This *flexible convention* allows the programmer to decide, in any ASM program, which occurrences of an external function symbol should be allowed to produce the same query (if the arguments agree) and which occurrences should be required to produce distinct queries. More precisely, we allow an ASM program to be accompanied by any assignment of templates to the occurrences of external function symbols. There are no restrictions concerning equality or collisions of the templates assigned to different occurrences.

This convention has the advantage that it can clearly simulate anything that either of the previous conventions can produce.

Its main disadvantage is that, like the must-vary convention, it makes it awkward to write the template assignment syntactically. Probably the cleanest syntax is to put the desired template immediately after any occurrence of an external function symbol. Notice that in this situation the external function symbols have very little significance; one could erase them, leaving only the templates, and the execution of the program would be unaffected. In fact, this syntax practically brings us back to the Lipari convention, with the templates here playing the role of the external function symbols there.

4.6 Reduction to the Lipari convention

In view of this consideration, we shall use in this paper the Lipari convention. A reader who prefers the flexible convention can pretend that our external function symbols are his templates and our template assignments are simply the identity map on templates. What, in our picture, corresponds to such a reader's external function symbols? Nothing. And this causes no problem, because his external function symbols play no role in the execution of a program; only the templates are relevant.

4.7 Different external functions

In the preceding subsections, we discussed conventions for interpreting multiple occurrences of the same external function symbol. We did not discuss occurrences of different function symbols. Could they produce the same query? That is, could they be assigned the same template or colliding templates? Intuitively, it seems that the answer should be no, but fortunately it is not necessary to decide this issue.

On the one hand, we shall define the semantics of ASMs in enough generality to cover even the "strange" template assignments where different external function symbols are assigned colliding templates. In fact, the semantics of such an ASM will be the same as for the ASM obtained by (1) replacing any symbols with colliding templates by a single symbol and (2) permuting the arguments of these symbols to match the permutations of the $\#i$'s involved in the definition of colliding templates. For instance, if f is assigned the template $(l_1, \#1, l_2, \#2, \#3)$ and g is assigned the template $(l_1, \#2, l_2, \#3, \#1)$, then $g(x, y, z)$ can be replaced with $f(y, z, x)$.[3]

On the other hand, when we prove that every ordinary algorithm is equivalent to an ASM, we shall not use such strange template assignments.

Thus, all our work will make sense whether or not one allows these strange template assignments.

4.8 Outputs

Most of the preceding discussion of external function symbols also applies, with trivial changes, to output rules. The execution of $\texttt{Output}_l(t)$ should produce a query that contains, as one of its components, the value of t. The other components should be labels, determined by l (Lipari convention) or by the particular occurrence of $\texttt{Output}_l$ (flexible convention), in the latter case possibly subject to a non-collision constraint (must-vary convention). The advantages and disadvantages of the various conventions are analogous to those already discussed for external functions.

One difference between the output situation and the external function situation is that we could, if we wanted, insist on the simplest form for the output queries, namely $\langle l, a\rangle$ where a is the value of t. This is a very strong form of the Lipari

[3]Had we allowed repetitions of placeholders in our templates, eliminating collisions would involve increasing the arities of function symbols. For instance, if f is assigned the template $(l_1, \#1, l_2, \#2, \#1)$ and g is assigned the template $(l_1, \#2, l_2, \#2, \#1)$, then we would want to introduce an external function symbol h with template $(l_1, \#1, l_2, \#2, \#3)$, so that $f(x, y)$ can be replaced with $h(x, y, x)$ and $g(x, y)$ can be replaced with $h(y, y, x)$.

convention. In the case of external functions, we could not insist on the analogous form $\langle f, \mathbf{a}\rangle$, because we needed ASMs to be able to produce queries of the general form permitted by Definition 2.2. We don't need to match these general queries with output rules, simply because our construction of an ASM equivalent to any given ordinary algorithm will not use output rules at all.

We shall not avail ourselves of the option of insisting on the $\langle l, a\rangle$ form for the queries produced by output rules. Instead we shall, for the sake of uniformity, adopt for output rules the same Lipari convention already adopted for external function symbols. Each of the output channels is to be assigned a template for unary functions; the query produced by $\mathtt{Output}_l(t)$ will be the result of substituting the value a of t for the (unique) placeholder #1 in the template associated to l.

As before, we do not insist that the templates assigned to output channels are distinct from each other or from those assigned to unary external function symbols, even though such a constraint seems intuitively justified. Our results in this paper apply equally well with or without this constraint. (Notice, by the way, that two templates for unary functions collide if and only if they are equal.)

5. ABSTRACT STATE MACHINES — SEMANTICS

5.1 General features of ASM semantics

The semantics of ASMs will be defined by associating to each ASM an ordinary algorithm. As a first step, we must say exactly what an (ordinary) ASM is; it is more than just a program.

Definition 5.1. An *ordinary ASM* with the finite vocabulary Υ and the finite label set Λ consists of

—an ASM program using vocabulary Υ together with some vocabulary E of external function symbols and some set of output labels,

—a template assignment, i.e, a function assigning
  —to each n-ary external function symbol f a template $\hat{f}$ for n-ary functions and
  —to each output label l a template $\hat{l}$ for unary functions,
  where the templates use labels from Λ,

—a nonempty set $\mathcal{S}$ of Υ-structures called the *states* of the ASM, such that $\mathcal{S}$ is closed under isomorphisms and under the transition functions to be defined below, and

—a nonempty isomorphism-closed subset $\mathcal{I} \subseteq \mathcal{S}$ of states called the *initial states* of the ASM.

This definition incorporates the Lipari convention discussed in Section 4, because templates are assigned to external function symbols and to output labels, not to their occurrences in the program.

According to this definition, an ASM trivially satisfies the States Postulate and the first two parts (dealing with states and initial states) of the Isomorphism Postulate. We shall define, for any ASM, the causality relations $\vdash_X$, the update function Δ^+, and the failure conditions in such a way as to satisfy the remaining postulates for algorithms. Once the definitions are given, the Interaction, Update, and Isomorphism Postulates will clearly be satisfied, but the Bounded Work Postulate will require verification.

Our definitions of $\vdash_X$, Δ^+, and failure will be formulated as though all Υ-structures were states. To get the definitions for the actual family $\mathcal{S}$ of states, we need only restrict $\vdash_X$, $\Delta^+(X, \alpha)$, and the failure criterion to the situation where $X \in \mathcal{S}$.

Our definition of the algorithm associated to an ASM will proceed by recursion on the syntactic structure of the program. This has two consequences that should be observed before we begin the definition.

First, although a program has, by definition, no free variables, the subrules from which it is constructed may well have free variables. So we need to associate algorithms not only to programs but to arbitrary rules. If a rule R has free variables among $v_1, \ldots, v_k$, then its interpretation involves not only an Υ-structure X but also values for the variables v_i. It will be convenient to accommodate this situation by regarding these n values as the values of n new constant (i.e. static nullary) symbols, one for each v_i. We shall use $\dot{v}$ for the constant symbol associated to a variable v. Thus, we expand the vocabulary Υ by adding the constant symbols $\dot{v}_1, \ldots, \dot{v}_n$, and we interpret R in any structure for this expanded vocabulary by using the value of $\dot{v}_i$ as the value of v_i.

Remark 5.2. It is tempting to ignore the distinction between the variable v_i and the constant symbol $\dot{v}_i$. In fact, we chose the dot notation because dots are easier to ignore than most other symbols. Nevertheless, one should be somewhat careful if one wants to ignore the distinction entirely. At the very least, one should then insist that no variable occur both free and bound and that no variable be bound twice in a rule.

The second consequence of building algorithms by recursion on the syntactic structure of an ASM is that, to start the recursion, before even getting to rules, we must define the semantics of terms. Since terms can contain external function symbols, their evaluation can involve the issuance of queries. We shall describe this by associating to each term and each state a causality relation, just as in the Interaction Postulate. Notions of context, reachability, and well-foundedness can be derived from these causality relations exactly as from the causality relations attached to algorithms. The semantics of terms will thus be quite analogous to that of rules; the difference is that in place of an update set and possible failure, what is associated to a term in a state and context is a value, an element of the state.

5.2 Semantics of terms

In view of the preceding discussion, we intend to define the following for any term t with free variables among $\mathbf{v} = v_1, \ldots, v_n$ and any $\Upsilon \cup \dot{\mathbf{v}}$-structure X:

—a causality relation $\vdash^t_X$ between finite answer functions and potential queries,

—for any context α, a value $\mathrm{Val}(t, X, \alpha)$.

If we think of the semantics of a term as being analogous to an algorithm, but with Val replacing Δ^+ (and with no mention of failures), then what we have said so far amounts to a promise to satisfy the analogs of the States, Interaction, and Update Postulates. (In the States Postulate, we have $\Upsilon \cup \dot{\mathbf{v}}$ in place of Υ, and all structures

count as initial states.) In fact, we shall also satisfy the analogs of the remaining postulates. For the Isomorphism Postulate, this means:

Isomorphism Postulate for Terms:

—Any isomorphism $i : X \cong Y$ of states preserves causality, i.e., if $\xi \vdash^t_X q$ then $i \circ \xi \circ i^{-1} \vdash^t_Y i(q)$.

—If $i : X \cong Y$ is an isomorphism of states and if α is a context for X, then $\mathrm{Val}(t, X, \alpha) = \mathrm{Val}(t, Y, i \circ \alpha \circ i^{-1})$.

For the Bounded Work Postulate, it means:

Bounded Work Postulate for Terms: There are uniform bounds for the cardinalities of the contexts and the lengths of the queries that occur in them. Furthermore, there is a bounded exploration witness, i.e., a finite set W of terms with the following properties. Assume X and X' are states that agree for W over an answer function α. If $\alpha \vdash^t_X q$, then also $\alpha \vdash^t_{X'} q$. If α is a context for X, then $\mathrm{Val}(t, X, \alpha) = \mathrm{Val}(t, X', \alpha)$.

It will be convenient to normalize our bounded exploration witnesses to have the following additional properties.

—W contains a variable.

—W is closed under subterms.

—The bounded exploration witness for a term t contains the variant of t obtained by replacing its variables v by the associated constants $\dot{v}$ and replacing subterms that begin with external function symbols with new, distinct variables.

We shall also arrange for our causality relations — both for terms and for rules — to be clean in the following sense.

Definition 5.3. A causality relation $\vdash$ is *clean* if its domain consists only of well-founded functions.

Remark 5.4. The technical value of working with clean causality relations will become clear in the proofs given below, but the intuitive value can be easily explained by considering the simplest example of an unclean causality relation. This relation $\vdash$ has only a single instance, $\{(q, r)\} \vdash q'$. It says that q' is to be issued if the answer r has been received for the query q, but it provides no way for q to be issued in the first place. Thus, the unique context for this $\vdash$ is the empty function; an algorithm or term having this causality relation (in some state) will issue no queries.

Suppose, however, that such a term or algorithm is being used in parallel with another (for instance if the term is one of the t_i in $f(t_1, \ldots, t_n)$ and is therefore being evaluated in parallel with the other t_j's). And suppose one of those other computations produces the query q and the environment answers with r. Then the causality relation $\vdash$ would result in the issuance of q'. That is, our term or algorithm is not operating independently of the others but is producing a query q' on the basis of the answer r to their query q.

Notice that the problem arises from the uncleanness of $\vdash$. The cause $\{(q, r)\}$ is not well-founded because it uses q without providing a cause for q to be issued. (More formally, this cause ξ has $q \in \mathrm{Dom}(\xi)$ but $q \notin \Gamma_\xi{}^\infty = \varnothing$.) For a clean causality relation, the only causes that could lead to a query (like q') would also explain how the term or algorithm itself (rather than some other parallel process) came to issue all the queries involved in the cause.

This completes our discussion of the requirements to be satisfied by our semantics of terms. We now present the semantics itself. As already mentioned, we proceed by recursion on the structure of the terms.

For a variable v, the causality relation $\vdash^v_X$ is empty and the value $\mathrm{Val}(v, X, \alpha)$ is simply the value assigned by the structure X to $\dot{v}$. The Isomorphism and Bounded Work Postulates are clearly satisfied, with $\varnothing$ being the only context and with $\{\dot{v}\}$ serving as the bounded exploration witness. The requirement of cleanness is vacuously satisfied.

Consider next a term t of the form $f(t_1, \ldots, t_n)$ where $f \in \Upsilon$. As always, we write $\mathbf{v}$ for a list of variables that includes all the variables of t and $\dot{\mathbf{v}}$ for the associated constants. Since $\mathbf{v}$ includes all the variables of each t_i, the corresponding causality relations and values can be taken as already defined and satisfying all our requirements. Let us write $\vdash^i$ to abbreviate $\vdash^{t_i}$, and let us write $\vdash$ for the causality relation $\vdash^t$ that we must define for t. (Here and in much of the following, we suppress mention of X when only a single state is under consideration.) The definition is very simple; we just take the union of the $\vdash^i$. That is, $\xi \vdash q$ if and only if $\xi \vdash^i q$ for some i.

This clearly satisfies the part of the Isomorphism Postulate referring to causality. As for the Bounded Work Postulate, we take as our bounded exploration witness W the union of the witnesses for the t_i augmented by the variant of t itself described in the last part of our normalization of bounded exploration witnesses. (The augmentation is needed only for the sake of the normalization.) It is clear that this choice of W behaves as required with respect to causality. The part of the Bounded Work Postulate concerning the number and length of queries is also satisfied, as will become clear once we determine, in Lemma 5.6 below, what the contexts are.

LEMMA 5.5. *If all the $\vdash^i$ are clean, then so is their union $\vdash$.*

Proof Let us write Γ_ξ and $(\Gamma_i)_\xi$ for the Γ operators associated, as in Section 2, to the answer function ξ and the causality relations $\vdash$ and $\vdash^i$, respectively. Notice that $\Gamma_\xi(Z) = \bigcup_i (\Gamma_i)_\xi(Z)$ for any set Z. In particular, for each i, we have $(\Gamma_i)_\xi(Z) \subseteq \Gamma_\xi(Z)$ and therefore $(\Gamma_i)_\xi{}^\infty \subseteq \Gamma_\xi{}^\infty$.

Now suppose $\xi \vdash q$. So for some i, $\xi \vdash^i q$. As $\vdash^i$ is clean, ξ is well-founded with respect to $\vdash^i$. So we have $\mathrm{Dom}(\xi) \subseteq (\Gamma_i)_\xi{}^\infty \subseteq \Gamma_\xi{}^\infty$, as required. □

Before defining the values $\mathrm{Val}(t, X, \alpha)$, we need to know what the contexts α are. The following lemma gives the answer; its proof is the main reason for requiring our causality relations to be clean. The reader may want to check that the unclean example in Remark 5.4 would violate the conclusion of this lemma.

LEMMA 5.6. *Let $\vdash$ be the union of the clean causality relations $\vdash^i$. Then an answer function α is a context for $\vdash$ if and only if it is the union of subfunctions*

α_i that are contexts for the respective $\vdash^i$'s. Furthermore, these subfunctions are uniquely determined.

Proof Suppose first that $\alpha = \bigcup_i \alpha_i$, where each α_i is a context for the corresponding $\vdash^i$. This means that, in the notation of the preceding proof, $\mathrm{Dom}(\alpha_i) = (\Gamma_i)_{\alpha_i}{}^{\infty}$. As a result, we have

$$\mathrm{Dom}(\alpha) = \bigcup_i \mathrm{Dom}(\alpha_i) = \bigcup_i (\Gamma_i)_{\alpha_i}{}^{\infty} \subseteq \bigcup_i (\Gamma_i)_{\alpha}{}^{\infty} \subseteq \Gamma_{\alpha}{}^{\infty}.$$

Here, the first inclusion comes from the fact that each $\alpha_i \subseteq \alpha$ and the second was established in the preceding proof. To finish the proof that α is a context, it suffices, by Lemma 2.18, to prove the reverse inclusion,

$$\Gamma_{\alpha}{}^{\infty} \subseteq \bigcup_i (\Gamma_i)_{\alpha_i}{}^{\infty} = \mathrm{Dom}(\alpha).$$

Suppose, toward a contradiction, that there are elements $q \in \Gamma_{\alpha}{}^{\infty}$ that are not in $(\Gamma_i)_{\alpha_i}{}^{\infty}$ for any i. Choose such a q that is in $\Gamma_{\alpha}{}^{n+1}$ for as small an n as possible. (Remember that $\Gamma^0 = \varnothing$ always, so it is correct to write the exponent as $n+1$.) So $q \in \Gamma_\alpha(\Gamma_{\alpha}{}^{n})$, which means that there is $\xi \subseteq \alpha \restriction \Gamma_{\alpha}{}^{n}$ such that $\xi \vdash q$. Fix such a ξ and, in view of the definition of $\vdash$, fix i such that $\xi \vdash^i q$. Because $\vdash^i$ is clean, ξ is well-founded with respect to it, and so we have

$$\mathrm{Dom}(\xi) \subseteq (\Gamma_i)_{\xi}{}^{\infty} \subseteq (\Gamma_i)_{\alpha}{}^{\infty}.$$

But then

$$q \in (\Gamma_i)_\alpha((\Gamma_i)_{\alpha}{}^{\infty}) = (\Gamma_i)_{\alpha}{}^{\infty},$$

contrary to our choice of q. This completes the proof that $\bigcup_i \alpha_i$ is a context for $\vdash$. It remains to prove that every context for $\vdash$ is of this form and that the relevant α_i are unique.

Let α be any context for $\vdash$. Then for each i we have

$$(\Gamma_i)_{\alpha}{}^{\infty} \subseteq \Gamma_{\alpha}{}^{\infty} = \mathrm{Dom}(\alpha).$$

According to Lemma 2.18, the subfunction α_i defined by restricting α to $(\Gamma_i)_{\alpha}{}^{\infty}$ is a context for $\vdash^i$. Since $\alpha \supseteq \bigcup_i \alpha_i$, it remains only to prove that this inclusion is in fact an equality. But both sides of the inclusion are contexts for $\vdash$, the left side by assumption and the right side by the part of the lemma already proved. By Lemma 2.18, these two contexts cannot be distinct.

Finally, the uniqueness of the α_i follows immediately from Lemma 2.18. □

Observe that, as promised earlier, the lemma's characterization of contexts immediately implies that the number and length of the queries in a context are bounded, as required by the Bounded Work Postulate, provided this was so for the $\vdash^i$'s.

Lemma 5.6 also enables us to complete the definition of the semantics for terms t of the form $f(t_1, \ldots, t_n)$ with $f \in \Upsilon$ by defining $\mathrm{Val}(t, X, \alpha)$ whenever α is a context for $\vdash$ in the state X. By the lemma, α is the union of uniquely determined contexts α_i for the causality relations $\vdash^i$ associated in state X with the terms t_i. So we can define

$$\mathrm{Val}(t, X, \alpha) = f_X(\mathrm{Val}(t_1, X, \alpha_1), \ldots \mathrm{Val}(t_n, X, \alpha_n)).$$

It is clear that this definition satisfies the Isomorphism Postulate and the Bounded Work Postulate, using the same bounded exploration witness W that we used for the causality relation.

It remains to define the semantics of terms t of the form $f(t_1, \ldots, t_n)$ where f is an external function symbol. We begin with the causality relation. The idea here is to proceed first just as in the case where $f \in \Upsilon$, obtaining a causality relation whose contexts suffice to provide values a_i for all the terms t_i. Then, in contrast to the previous case, one further query is needed to obtain the value of $f(a_1, \ldots, a_n)$ (previously provided, in f_X, by the state X).

Given the causality relations $\vdash^i$ associated to the subterms t_i in state X, let $\vdash'$ be their union. Our discussion in the preceding case, where the term t began with a function symbol from Υ and where this union was the desired causality relation for t, shows that $\vdash'$ is clean and satisfies the relevant parts of the Isomorphism and Bounded Work Postulates. Since our present t begins with an external function symbol f, the required causality relation $\vdash$ is a bit larger than $\vdash'$, for it must also produce the final query, asking for the appropriate value of f. Specifically, we define

$$\vdash \;=\; \vdash' \cup \vdash'',$$

where $\xi \vdash'' q$ means the following: First ξ is required to be a context for $\vdash'$. According to Lemma 5.6, $\xi = \bigcup_i \xi_i$ for uniquely determined contexts ξ_i for $\vdash^i$. Let $a_i = \mathrm{Val}(t_i, X, \xi_i)$. Then q is required to be $\hat{f}[a_1, \ldots, a_n]$. (Recall that the square-bracket notation here means to substitute the elements a_i for the placeholders $\#i$ in the template $\hat{f}$ that the ASM assigns to the symbol f.)

The causality part of the Isomorphism Postulate is clearly satisfied by $\vdash$. Using the same bounded exploration witness W as for $\vdash'$, we see that it behaves properly also with respect to $\vdash$. This uses the fact that W contains a variable so that the environment's reply to the new query $\hat{f}[a_1, \ldots, a_n]$ will be the same in the two states X and X' considered in the Bounded Work Postulate. The rest of the Bounded Work Postulate will be verified after we describe the contexts and define Val.

First, however, we check that $\vdash$ is clean. Note that if $\xi \vdash q$ then either $\xi \vdash' q$ in which case ξ is well-founded for $\vdash'$ since $\vdash'$ is clean, or else $\xi \vdash'' q$ in which case ξ is a context for $\vdash'$. Since contexts are always well-founded (as is clear by inspection of the definitions), we have, in either case, that ξ is well-founded with respect to $\vdash'$. As $\vdash' \subseteq \vdash$, it follows that (with Γ' and Γ associated to $\vdash'$ and $\vdash$)

$$\mathrm{Dom}(\xi) \subseteq {\Gamma'_\xi}^\infty \subseteq {\Gamma_\xi}^\infty,$$

and so ξ is well-founded also with respect to $\vdash$.

As before, we need to know what the contexts for $\vdash$ look like before we can reasonably discuss the value assigned to t in a context. Recall that we already know, from Lemma 5.6, what the contexts ξ for $\vdash'$ look like; they are the unions of contexts ξ_i, one for each $\vdash^i$.

Consider now an arbitrary context α for $\vdash$. We have (with notation as above)

$$\mathrm{Dom}(\alpha) = {\Gamma_\alpha}^\infty \supseteq {\Gamma'_\alpha}^\infty,$$

and so, by Lemma 2.18, there is a unique subfunction of α that is a context for $\vdash'$, namely $\xi = \alpha \restriction {\Gamma'_\alpha}^\infty$. So $\xi = \bigcup_i \xi_i$ for unique contexts ξ_i for $\vdash^i$. Let $a_i =$

$\mathrm{Val}(t_i, X, \xi_i)$ and let q be the query $\hat{f}[a_1, \ldots, a_n]$. So $\xi \vdash q$ and therefore

$$q \in \Gamma_\alpha({\Gamma'_\alpha}^\infty) \subseteq \Gamma_\alpha({\Gamma_\alpha}^\infty) = {\Gamma_\alpha}^\infty = \mathrm{Dom}(\alpha).$$

So α includes $\xi \cup \{(q, r)\}$ for some reply r. We shall show that $\mathrm{Dom}(\xi \cup \{(q, r)\})$ is closed under Γ_α. Then it follows that

$$\mathrm{Dom}(\alpha) = {\Gamma_\alpha}^\infty \subseteq \mathrm{Dom}(\xi \cup \{(q, r)\}) \subseteq \mathrm{Dom}(\alpha)$$

and therefore $\alpha = \xi \cup \{(q, r)\}$.

To verify the claim about being closed, suppose

$$q' \in \Gamma_\alpha(\mathrm{Dom}(\xi \cup \{(q, r)\})) = \Gamma_\alpha(\mathrm{Dom}(\xi) \cup \{q\}).$$

So there is $\beta \subseteq \xi \cup \{(q, r)\}$ such that $\beta \vdash q'$. There are now two cases to consider.

Suppose first that $\beta \vdash'' q'$. Then, by definition of $\vdash''$, β must be a context for $\vdash'$. But ξ is the unique such context that is a subfunction of α, so $\beta = \xi$. Then it follows, by inspection of the definitions, that $q' = q$ and in particular $q' \in \mathrm{Dom}(\xi \cup \{(q, r)\})$, as desired.

The other possibility is that $\beta \vdash' q'$. If $\beta \subseteq \xi$ then, since ξ is a context for $\vdash'$, we have $q' \in \Gamma'_\xi(\mathrm{Dom}(\xi)) = \mathrm{Dom}(\xi) \subseteq \mathrm{Dom}(\xi \cup \{(q, r)\})$ as required. So we may assume that $q \in \mathrm{Dom}(\beta)$. Since $\vdash'$ is clean, β is well-founded for $\vdash'$, and so $q \in {\Gamma'_\beta}^\infty$. Let n be the smallest integer such that $q \in {\Gamma'_\beta}^{n+1}$. So $\beta \restriction {\Gamma'_\beta}^n \subseteq \xi$, and some subfunction δ of this has $\delta \vdash' q'$. But then

$$q \in \Gamma'_\xi({\Gamma'_\beta}^n) \subseteq \Gamma'_\xi({\Gamma'_\xi}^\infty) = {\Gamma'_\xi}^\infty = \mathrm{Dom}(\xi).$$

So $\mathrm{Dom}(\beta) \subseteq \mathrm{Dom}(\xi) \cup \{q\} \subseteq \mathrm{Dom}(\xi)$ and we are back in the case treated at the beginning of this paragraph.

This completes the proof that every context for $\vdash$ has the form $\xi \cup \{(q, r)\}$ where ξ is a (uniquely determined) context for $\vdash'$ and $q = \hat{f}[a_1, \ldots, a_n]$, the a_i being the values of the t_i with respect to the (unique) contexts ξ_i for $\vdash^i$ whose union is ξ.

Conversely, every such $\xi \cup \{(q, r)\}$ is a context for $\vdash$. To see this, notice first that, by the argument given above, $\mathrm{Dom}(\xi \cup \{(q, r)\})$ is closed under Γ_γ for any γ that includes $\xi \cup \{(q, r)\}$; we choose in particular $\gamma = \xi \cup \{(q, r)\}$. Since ${\Gamma_\gamma}^\infty$ is the smallest set closed under Γ_γ it is included in $\mathrm{Dom}(\gamma)$. By Lemma 2.18, there is a (unique) $\alpha \subseteq \gamma$ that is a context for $\vdash$. By what we already proved, this context includes a context for $\vdash'$, which can only be ξ because a single answer function γ cannot include two different contexts for the same causality relation $\vdash'$ (see Lemma 2.18). The rest of our analysis of what a context for $\vdash$ must look like shows that $q \in \mathrm{Dom}(\alpha)$. Thus, $\mathrm{Dom}(\gamma) = \mathrm{Dom}(\xi) \cup \{q\} \subseteq \mathrm{Dom}(\alpha)$. Since $\alpha \subseteq \gamma$, we conclude that γ is equal to α, which is a context for $\vdash$. This completes the proof that $\xi \cup \{(q, r)\}$ is a context for $\vdash$ and thus completes our description of all contexts for $\vdash$.

Thus, contexts for $\vdash$ are larger than contexts for $\vdash'$ by at most one element (q, r), where q is obtained by instantiating the template $\hat{f}$. Since, by induction hypothesis, the number and length of the queries in a context for $\vdash'$ are bounded, the same holds for $\vdash$, as required by the Bounded Work Postulate.

With the description of contexts available, we are ready to define $\mathrm{Val}(t, X, \alpha)$ when α is a context for the causality relation $\vdash$ attached to state X and term $t = f(t_1, \ldots, t_n)$. Since α includes a unique context ξ for $\vdash'$, which is in turn a

union of unique contexts ξ_i for the $\vdash^i$ (associated to the t_i), there are well-defined elements $a_i = \mathrm{Val}(t_i, X, \xi_i)$. Furthermore, the query $q = \hat{f}[a_1, \ldots, a_n]$ is in $\mathrm{Dom}(\alpha)$. We define $\mathrm{Val}(t, X, \alpha)$ to be $\alpha(q)$.

This definition clearly satisfies the Isomorphism Postulate and the Bounded Work Postulate, using the same bounded exploration witness that we used above for the causality relation $\vdash$.

This completes the definition of the semantics of terms, i.e., the associated causality relations and Val functions, along with the verification of the postulates for terms and the cleanness of the causality relations.

Remark 5.7. The intention behind the notions of relational symbols and Boolean variables and terms is that their values (when they exist) should always be `true` or `false`. Unfortunately, the environment may not cooperate with this intention; it may provide a non-Boolean reply to a query $\hat{f}(\mathbf{t})$ when f is a relational external function symbol. As indicated in the discussion of failure in [Blass and Gurevich to appear (a), Section 5], this situation can cause the algorithm to fail. Our definitions of the semantics of rules will produce failures whenever non-Boolean replies to Boolean queries cause a problem that prevents the computation from continuing. One such situation is an attempt to update a relational function to take a non-Boolean value. This would result in a "next state" that is not really a state because it is not even a structure; Convention 2.1 requires the values of relational function symbols of Υ to be `true` or `false`. The other such situation is a non-Boolean value for a guard in a conditional. Here the conditional would no longer make sense.

Our semantic definitions could be modified (at the cost of some additional complexity) to make the algorithm fail whenever the environment gives a non-Boolean reply to a Boolean query, whether or not it causes such a serious problem that the computation cannot continue. Alternatively, without changing our semantic definitions, one can incorporate "fail on all bad replies" behavior into programs, by using conditionals of the form `if` $\neg$`Boole`(t) `then fail else` ....

Note that this whole discussion of non-Boolean replies to Boolean queries would be irrelevant if no function symbols of the external vocabulary were declared to be relational. In fact, the proof in [Blass and Gurevich to appear (b)] that all algorithms are equivalent to ASMs will not require the use of external, relational function symbols. So the reader will not lose anything essential by (1) pretending that such symbols are prohibited and (2) therefore ignoring all the clauses, in the semantics of rules below, that refer to the pathological situation of non-Boolean replies to Boolean queries.

5.3 Semantics of rules

In this section, we complete the definition of the semantics of ASMs. Given a rule R, written with state vocabulary Υ and some external vocabulary and output labels, having free variables among $\mathbf{v}$, and given a template assignment using labels in Λ, we shall define an algorithm A_R with vocabulary $\Upsilon \cup \dot{\mathbf{v}}$, with label set Λ, and with all Υ-structures as initial states. As indicated earlier, we can then get the semantics of an ASM, in whose program, by definition, no free variables occur, as an algorithm with Υ and Λ, and with the desired sets of states and initial states, simply by restricting the algorithm to the given states.

Our definition of A_R will be by recursion on the structure of R. Although the algorithm A_R depends not only on R but also on the template assignment, we do not indicate this explicitly in the notation, since the template assignment will remain fixed during the recursion. As before, we write $\hat{f}$ for the template assigned to an n-ary external function symbol f and we write $\hat{f}[a_1, \ldots, a_n]$ for the query obtained by replacing the placeholders $\#i$ in $\hat{f}$ by the elements a_i of a state. Similarly, $\hat{l}$ is the unary template assigned to an output label l, and $\hat{l}[a]$ is the result of replacing $\#1$ with a.

By recursion on rules R, we shall define the causality relations $\vdash_X$, the failures, and Δ^+. As already mentioned, all structures for $\Upsilon \cup \dot{\mathbf{v}}$ will serve as states and as initial states. Here $\mathbf{v}$ is a list of variables that includes all the free variables of R and $\dot{\mathbf{v}}$ is the corresponding list of constant symbols. (Technically, we associate an algorithm not just to R but rather to R together with a template assignment and a choice of the list $\mathbf{v}$ of variables. Most of the time, these additional parameters can safely be suppressed.) In each case, the States, Interaction, Update, and Isomorphism Postulates will be obviously satisfied. To be more precise about the Update Postulate, although we shall define only failures and Δ^+ explicitly, the transition function τ required in the Update Postulate can simply be defined by the requirements in the postulate itself. Notice that this presupposes the following connection between updates and failures: If $\Delta^+(X, \alpha)$ contains two conflicting updates (i.e., distinct updates of the same location) then the algorithm must fail in (X, α). Our definitions of updates and failures will be such that this connection obviously holds.

In each case of our recursion, we shall verify that the causality relation is clean. Furthermore, we shall establish an explicit description of the contexts, from which the first two items in the Bounded Work Postulate — bounding the numbers and lengths of queries in any context — will follow. Finally, we shall present, in each case, a bounded exploration witness verifying the remaining parts of the Bounded Work Postulate

Update rules. The intuition here is that an update rule issues just the queries needed to evaluate the terms that occur in it. Given enough answers from the environment (a context) to evaluate these terms, it produces the single specified update. The following definition formalizes this idea.

Let R be an update rule $f(t_1, \ldots, t_n) := t_0$. We define its causality relation, in any state X, to be the union of the causality relations $\vdash^i$ of all the t_i $(0 \leq i \leq n)$ in state X. We already considered such unions in our discussion of the semantics of terms of the form $f(\mathbf{t})$ with $f \in \Upsilon$. The discussion there carries over to the present situation and establishes that $\vdash$ is clean and that its contexts are just the unions of (uniquely determined) contexts for the $\vdash^i$. As before, the bounds on the number and length of queries in any context follow immediately, since such bounds hold for the $\vdash^i$.

It remains to define failures and Δ^+ for $f(t_1 \ldots, t_n) := t_0$, and this is easy. An update rule $f(t_1, \ldots, t_n) := t_0$ fails if and only if f is relational but the value of t_0 is not Boolean. (This can happen only if the environment gave a non-Boolean answer to a query for a relational function symbol.) If it does not fail, then its update set $\Delta^+(X, \alpha)$, for a state X and context α, consists of a single update $\langle f, \langle a_1, \ldots, a_n\rangle, a_0\rangle$, where each a_i is the value $\mathrm{Val}(t_i, X, \alpha_i)$ and the functions α_i

are the unique contexts for the respective $\vdash^i$ whose union is α.

For the bounded exploration witness, it suffices to take the union of the bounded exploration witnesses assigned to the terms t_i. It is then easy to check that the Bounded Work Postulate holds.

Output rules. The intuition is that an output rule first issues enough queries to evaluate the term occurring as its argument. Once it has enough information from the environment to evaluate this term, it issues one more query, namely the output itself. Recall from [Blass and Gurevich to appear (a), Section 2] that outputs are regarded as queries that receive an automatic and uninformative answer OK. Here is the formal definition.

Let R be $\texttt{Output}_l(t)$. The causality relation for R is the union of the causality relation $\vdash'$ of t and the relation $\vdash''$, where $\xi \vdash' q$ means the following. First ξ is required to be a context for $\vdash'$. Let $a = \mathrm{Val}(t, X, \xi)$. Then q is required to be $\hat{l}[a]$. (Recall that the square-bracket notation here means to substitute a for the unique placeholder #1 in the template $\hat{l}$ that the ASM assigns to the output label l. Recall also that we feel free to omit mention of the state X when it is fixed in a particular discussion.)

This causality relation is clean, and its contexts are exactly the answer functions of the form $\xi \cup \{(q, r)\}$ where ξ is a context for $\vdash'$ and $q = \hat{l}[\mathrm{Val}(t, X, \xi)]$. The proof of this is a slightly simplified version of what we did for the causality function of a term $f(\mathbf{t})$ when f is an external function symbol. The notations $\vdash'$ and $\vdash''$ are used here just as they were there, so it is easy to transcribe the proof. The only difference is that in the present situation we can work with ξ directly, while the previous argument required us to consider the pieces ξ_i. Thus, the present argument is a bit simpler, just because the n of the earlier argument is now 1.

Finally, we specify, in agreement with intuition, that an output rule never fails and that its update set, for any state and context, is empty.

Parallel blocks. A parallel block should issue all the queries and perform all the updates produced by any of its components. It should fail if either one of its components fails or two of its components produce conflicting updates. Here is the formal definition.

Let R be the rule `do in parallel` $R_1, \ldots, R_n$ `enddo`. For each component R_i, let $\vdash^i$ be the associated causality relation and Δ_i^+ the associated update function. The causality relation $\vdash$ for R is defined to be the union of the causality relations $\vdash^i$ of its components. We have already seen, in discussing the semantics of terms $f(\mathbf{t})$ with $f \in \Upsilon$, that such a union is clean and that its contexts are simply the unions of contexts for the $\vdash^i$'s. If we take, as in previous such situations, the bounded exploration witness W to be the union of the bounded exploration witnesses for the components R_i, then all parts of the Bounded Work Postulate that concern causality are verified.

To define Δ^+ for R, let α be a context, with respect to $\vdash$, for the state X. So α is the union of uniquely determined contexts α_i for the $\vdash^i$. Define

$$\Delta^+(X, \alpha) = \bigcup_{i=1}^{n} \Delta_i^+(X, \alpha_i).$$

This satisfies the Δ^+ part of the Bounded Work Postulate with the same W as above.

Finally, define that R fails in state X and context α if either some R_i fails in X and α_i or $\Delta^+(X, \alpha)$ contains two distinct updates of the same location.

Remark 5.8. The parallel block with no components is often denoted by `skip`. Taking $n = 0$ in the previous definition, we find that `skip` has the expected semantics. Its causality relation is empty; its update set in any state (and with the unique context $\varnothing$) is empty; and it doesn't fail.

A parallel block R consisting of a single rule R_1 is equivalent to R_1. The verification of this fact is by inspection of the definition of parallel block semantics, with $n = 1$, keeping in mind that an algorithm that produces conflicting updates for some (X, α) must fail there. (The issue here is that if R_1 produced conflicting updates without failing, then R would rectify this error by failing and would therefore differ from R_1.)

Conditional rules. A conditional rule should first issue whatever queries are needed for the evaluation of its guard. When enough answers have been received for this evaluation, the algorithm should continue by executing the appropriate branch. If, because of absurd answers from hte environment, the guard has a non-Boolean value, the conditional rule should fail. Here is the formal definition.

Let R be the rule `if` φ `then` R_0 `else` R_1 `endif`. Write $\vdash'$ for the causality relation associated (in a tacitly understood state X) to the Boolean term φ, and write $\vdash^i$ for the causality relations associated to the branches R_i. Then the causality relation $\vdash$ associated to R is the union of $\vdash'$ and a second causality relation $\vdash''$ defined by letting $\xi \vdash'' q$ mean that ξ is the union of a context ξ' with respect to $\vdash'$ and an answer function η such that either $\mathrm{Val}(\varphi, X, \xi') =$ `true` and $\eta \vdash^0 q$ or $\mathrm{Val}(\varphi, X, \xi') =$ `false` and $\eta \vdash^1 q$. This construction is rather similar to what we did for terms beginning with an external function symbol and for output rules, but it is a bit more complicated in that the second part, $\vdash''$, involves causes ξ that are not simply contexts for $\vdash'$. We therefore verify the necessary properties of $\vdash$ here.

Lemma 5.9. $\vdash$ *is clean.*

Proof Suppose $\xi \vdash q$; we must show that ξ is well-founded with respect to $\vdash$. If $\xi \vdash' q$ then, by induction hypothesis, ξ is well-founded with respect to $\vdash'$ and therefore with respect to the larger relation $\vdash$.

Assume, therefore, that $\xi = \xi' \cup \eta$ where ξ' is a context for $\vdash'$, where $\mathrm{Val}(\varphi, X, \xi') =$ `true`, and $\eta \vdash^0 q$. (The other possibility, that $\mathrm{Val}(\varphi, X, \xi') =$ `false`, and $\eta \vdash^1 q$, is handled analogously.) Since ξ' is a context for $\vdash$, we have

$$\mathrm{Dom}(\xi') = {\Gamma'_{\xi'}}^{\infty} \subseteq {\Gamma_{\xi'}}^{\infty} \subseteq {\Gamma_{\xi}}^{\infty}.$$

(As before, Γ' is the operator induced by the causality relation $\vdash'$, while Γ is induced by $\vdash$.) So it remains to prove that $\mathrm{Dom}(\eta) \subseteq {\Gamma_{\xi}}^{\infty}$.

Since $\eta \vdash^0 q$ and since $\vdash^0$ is clean by induction hypothesis, we have $\mathrm{Dom}(\eta) \subseteq {(\Gamma_0)_{\eta}}^{\infty}$, where we have, as before, written Γ_0 for the operator associated to the causality relation $\vdash^0$. So the proof of the lemma will be complete if we show that ${(\Gamma_0)_{\eta}}^{\infty} \subseteq {\Gamma_{\xi}}^{\infty}$. For this purpose, it suffices to show that ${\Gamma_{\xi}}^{\infty}$ is closed under $(\Gamma_0)_{\eta}$, since this operator's smallest closed set is ${(\Gamma_0)_{\eta}}^{\infty}$.

Consider, therefore, an arbitrary $q' \in (\Gamma_0)_\eta({\Gamma_\xi}^\infty)$. This means that $\delta \vdash^0 q'$ for some $\delta \subseteq \eta \restriction {\Gamma_\xi}^\infty$. Then we have $\xi' \cup \delta \vdash q'$ by the definition of $\vdash$, and we have $\xi' \cup \delta \subseteq \xi \restriction {\Gamma_\xi}^\infty$ because $\eta \subseteq \xi$ and because we already checked that $\mathrm{Dom}(\xi') \subseteq {\Gamma_\xi}^\infty$. Therefore, $q \in \Gamma_\xi({\Gamma_\xi}^\infty) = {\Gamma_\xi}^\infty$, as required. □

As in previous cases, we intend to characterize the contexts for $\vdash$. We begin with a consequence of Lemma 2.23, making use of the cleanness of our causality relations.

COROLLARY 5.10. *Suppose $\vdash$ is a clean causality relation and suppose that a certain answer function α includes both a context β with respect to $\vdash$ and an answer function η such that $\eta \vdash q$ for a certain q. Then $\eta \subseteq \beta$ and $q \in Dom(\beta)$.*

Proof Since $\vdash$ is clean, η is well-founded. So Lemma 2.23 gives us that $\eta \subseteq \beta$. Then from $\eta \vdash q$ we infer that $q \in \Gamma_\beta(\mathrm{Dom}(\beta)) = \mathrm{Dom}(\beta)$. □

With this corollary available, we are ready to characterize contexts for the causality relation $\vdash$ associated to a conditional rule. We use the same notation as in the definition of this $\vdash$ above.

LEMMA 5.11. *The contexts for $\vdash$ are of three sorts:*

—*Unions $\xi \cup \beta$ where ξ is a context for $\vdash'$, $Val(\varphi, X, \xi) =$ `true`, and β is a context for $\vdash^0$,*

—*Unions $\xi \cup \beta$ where ξ is a context for $\vdash'$, $Val(\varphi, X, \xi) =$ `false`, and β is a context for $\vdash^1$, and*

—*Contexts ξ for $\vdash'$ such that $Val(\varphi, X, \xi)$ is not Boolean.*

The third case in the lemma arises only in the pathological case that the environment gives a non-Boolean reply to a Boolean query.

Proof Assume first that α is a context for $\vdash$. We shall produce the ξ and (in the non-pathological cases) β required by the lemma. As before, we use the notations Γ, Γ', and Γ_i $(i = 0, 1)$ for the operators associated to the causality relations $\vdash$, $\vdash'$, and $\vdash^i$.

Since $\mathrm{Dom}(\alpha) = {\Gamma_\alpha}^\infty \supseteq {\Gamma'_\alpha}^\infty$, we know by Lemma 2.18 that α includes a unique context ξ for $\vdash'$.

Let us first dispose of the pathological case, where $\mathrm{Val}(\varphi, X, \xi)$ is not Boolean. In this case, we shall show that ξ itself is a context for $\vdash$ and therefore, by Lemma 2.18, $\alpha = \xi$. By the same lemma, it suffices to check that $\mathrm{Dom}(\xi)$ is closed under Γ_ξ, so consider any $q \in \Gamma_\xi(\mathrm{Dom}(\xi))$. So there is $\zeta \subseteq \xi$ with $\zeta \vdash q$. If $\zeta \vdash' q$, then $q \in \Gamma'_\xi(\mathrm{Dom}(\xi)) = \mathrm{Dom}(\xi)$ because ξ is a context for $\vdash'$. It remains to consider the possibility that $\zeta \vdash'' q$. By definition of $\vdash''$, this requires ζ to include a context for $\vdash'$; as ξ is a context for $\vdash'$ and cannot properly include another, we must have $\zeta = \xi$. But then the case hypothesis, that $\mathrm{Val}(\varphi, X, \xi)$ is not Boolean, implies that we cannot have $\zeta \vdash'' q$. This completes the argument in the pathological case.

We now assume that $\mathrm{Val}(\varphi, X, \xi) =$ `true`; the case of `false` is handled in the same way.

We show next that $\mathrm{Dom}(\alpha)$ is closed under $(\Gamma_0)_\alpha$. Suppose $q \in (\Gamma_0)_\alpha(\mathrm{Dom}(\alpha))$. Thus, $\delta \vdash^0 q$ for some $\delta \subseteq \alpha$. Then $\xi \cup \delta \vdash q$. Since $\xi \cup \delta \subseteq \alpha$, we conclude that

$q \in \Gamma_\alpha(\mathrm{Dom}(\alpha)) = \mathrm{Dom}(\alpha)$, where the last equality comes from the assumption that α is a context for $\vdash$.

Since ${(\Gamma_0)_\alpha}^\infty$ is the smallest set closed under $(\Gamma_0)_\alpha$, we have ${(\Gamma_0)_\alpha}^\infty \subseteq \mathrm{Dom}(\alpha)$. Again invoking Lemma 2.18, we infer that α includes a unique context β for $\vdash^0$. Notice that, at this point, we have established the uniqueness assertion in the lemma.

So we have contexts ξ and β for $\vdash'$ and $\vdash^0$, such that $\alpha \supseteq \xi \cup \beta$. To show that this inclusion is in fact an equality, it suffices to show that $\mathrm{Dom}(\xi \cup \beta)$ is closed under Γ_α, because $\mathrm{Dom}(\alpha)$ is the smallest such closed set. Suppose, therefore, that we have $\delta \vdash q$ and $\delta \subseteq \alpha \restriction \mathrm{Dom}(\xi \cup \beta) = \xi \cup \beta$. We must show that $q \in \mathrm{Dom}(\xi \cup \beta)$.

Consider first the case that $\delta \vdash' q$. Apply Corollary 5.10 to the clean causality relation $\vdash'$, the context ξ for $\vdash'$, and the fact that $\delta \vdash' q$. The corollary gives that $q \in \mathrm{Dom}(\xi) \subseteq \mathrm{Dom}(\xi \cup \beta)$ as required.

There remains the case that $\delta \vdash'' q$. This means that, first, δ includes a context for $\vdash'$, which can only be ξ since at most one context for $\vdash'$ can be a subfunction of α. Then, since $\mathrm{Val}(\varphi, X, \xi) = \texttt{true}$, there must be some η such that $\eta \vdash^0 q$ and $\delta = \xi \cup \eta$. Apply Corollary 5.10 to the clean causality relation $\vdash^0$, the context β for this causality relation, and the fact that $\eta \vdash^0 q$. The corollary gives that $q \in \mathrm{Dom}(\beta) \subseteq \mathrm{Dom}(\xi \cup \beta)$ as required.

This completes the proof that $\mathrm{Dom}(\xi \cup \beta)$ is closed under Γ_α and therefore $\alpha = \xi \cup \beta$. Thus, every context for $\vdash$ has the form specified in the lemma. It remains to prove that every answer function of the specified form is a context for $\vdash$.

The argument for the pathological case is contained in the argument already given. We showed there that any context ξ for $\vdash'$ that gives φ a non-Boolean value is also a context for $\vdash$. So we may now confine our attention to the normal situation, where φ gets a Boolean value.

Suppose, therefore, that $\alpha = \xi \cup \beta$ where ξ is a context for $\vdash'$, where $\mathrm{Val}(\varphi, X, \xi) = \texttt{true}$, and where β is a context for $\vdash^0$. (The case where $\mathrm{Val}(\varphi, X, \xi) = \texttt{false}$ and β is a context for $\vdash^1$ is handled analogously.) The argument given in the preceding paragraphs shows that $\mathrm{Dom}(\alpha)$ is closed under Γ_α. Thus, by Lemma 2.18, α includes a context α' for $\vdash$. That context must, by what we have already proved, have the form $\xi' \cup \beta'$ where ξ' and β' are contexts for $\vdash'$ and the appropriate $\vdash^i$, respectively. Since α can include at most one context for $\vdash'$, we have $\xi' = \xi$. In particular, the appropriate $\vdash^i$ is $\vdash^0$. Since α can include at most one context for $\vdash^0$, we have $\beta' = \beta$ and therefore $\alpha' = \alpha$. Thus, α is a context for $\vdash$. □

From this characterization of the contexts for $\vdash$, it is clear that the number and length of queries in any context are bounded, provided the same is true of $\vdash'$, $\vdash^0$, and $\vdash^1$. Taking W to be the union of bounded exploration witnesses for φ, R_0, and R_1, we find that the parts of the Bounded Work Postulate that refer to causality are satisfied for the conditional rule R.

We define failures and updates for R in the natural way. Let a state X and a context α for it be given, and apply Lemma 5.11 to α. In the pathological third case, where α is a context for $\vdash'$ giving φ a non-Boolean value, let R fail and produce no updates. In the other cases, write $\alpha = \xi \cup \beta$ where ξ and β are as in Lemma 5.11. Also, let i be 0 or 1 according to whether $\mathrm{Val}(\varphi, X, \xi)$ is $\texttt{true}$ or $\texttt{false}$. So β is

a context for the causality relation $\vdash^i$ associated to R_i. Then R fails in X and α if and only if R_i fails in X and β. The update set $\Delta^+(X, \alpha)$ is defined to be the update set of R_i in state X and context β.

It is now easy to verify that the bounded exploration witness described above in connection with causality also works with respect to updates. Thus, all the postulates hold for conditional rules.

Let rules. Consider a let rule R, say

$$\texttt{let } x_1 = t_1, \ldots, x_k = t_k \texttt{ in } R_0 \texttt{ endlet}.$$

The intended execution of R in a state X consists of two phases. In the first phase, the terms t_i are evaluated in X. In the second phase, R_0 is executed in the state X^* that differs from X only in that each $\dot{x}_i$ has the value that was obtained, in the first phase, for t_i.

Before formalizing this, it is useful to consider the vocabularies involved. R is to be evaluated in a state X for the vocabulary $\Upsilon \cup \dot{\mathbf{v}}$. Here $\mathbf{v}$ is a list of variables that includes all the free variables of R. Since the x_i are not free in R, they need not be among the v's, but some (or all) of them may be. We write $\mathbf{v} \cup \mathbf{x}$ for the union, without repetitions, of the lists $\mathbf{v}$ and $\mathbf{x} = x_1, \ldots, x_k$. This list includes all the free variables of R_0, so we know, by induction hypothesis, that the semantics of R_0 is already defined for any $\Upsilon \cup \dot{\mathbf{v}} \cup \dot{\mathbf{x}}$-structure.

Given an $\Upsilon \cup \dot{\mathbf{v}}$-structure X and given k elements $a_1, \ldots, a_k$ of X, we write (X but $\mathbf{x} \mapsto \mathbf{a}$) for the structure with the same base set as X and the same interpretations of all function symbols except that each $\dot{x}_i$ is interpreted as the corresponding a_i, whether or not $\dot{x}_i$ had a value in X (i.e., whether or not $\dot{x}_i \in \Upsilon \cup \dot{\mathbf{v}}$)[4].

We define the causality relation $\vdash_X$ associated to R in an $\Upsilon \cup \dot{\mathbf{v}}$-structure X as follows. (In contrast to previous cases, we do not suppress X from the notation, because we shall also have to consider another structure X^*.) We set $\vdash_X = \vdash' \cup \vdash''$, where $\vdash'$ is the union of the causality relations $\vdash^i_X$ associated, in X, to the bindings t_i. The other part, $\vdash''$ is defined by letting $\xi \vdash'' q$ if, first, ξ is the union of a context ξ' for $\vdash'$ and another answer function η, and, second, ξ' and η are related as follows. Let ξ_i be contexts for the $\vdash^i_X$ such that $\xi' = \bigcup_i \xi_i$; such ξ_i exist and are unique by Lemma 5.6. For $i = 1, \ldots, k$, let $a_i = \mathrm{Val}(t_i, X, \xi_i)$. Let $X^* = (X \text{ but } \mathbf{x} \mapsto \mathbf{a})$. Let $\vdash^0$ be the causality relation associated in state X^* to the rule R_0. Then we require $\eta \vdash^0 q$.

The proof that $\vdash_X$ is clean is essentially the same as in the case of conditional rules. Only the following minor differences need to be taken into account. First, in the case of conditional rules, $\vdash'$ was clean by induction hypothesis. In the present situation, the induction hypothesis tells us that each $\vdash^i_X$ is clean, so we must invoke Lemma 5.5 to infer that $\vdash'$ is clean. Second, in the case of conditional rules, the second part of the causality definition used the causality relation $\vdash^0$ or $\vdash^1$ associated, in state X, to R_0 or R_1 according to the truth value $\mathrm{Val}(\varphi, X, \xi)$. In the present situation, the causality relation used in the second part is always obtained from the same rule R_0 but in different states X^* according to the values

[4]It is usually safe to assume that no x_i occurs in $\mathbf{v}$. Then (X but $\mathbf{x} \mapsto \mathbf{a}$) simply adjoins values for the x_i's. The assumption becomes unsafe only if a variable occurs both free and bound in the same rule.

$\mathrm{Val}(t_1, X, \xi)$. Neither of these differences affects the structure of the proof, so we do not repeat the details.

Similarly, the following lemma is obtained by essentially the same argument as the analog for conditionals.

LEMMA 5.12. *The contexts for $\vdash_X$ are the unions $\xi \cup \beta$ of a context ξ for $\vdash'$ and a context β for $\vdash^0$, where $\vdash'$ and $\vdash^0$ are as in the definition of $\vdash_X$. Furthermore, for each context α, the associated ξ and β are uniquely determined.*

As usual, this lemma immediately implies that the number and length of queries in any context are uniformly bounded — just take the sum of the bounds for the t_i and R_0.

To define failures and updates for the let-rule R in state A and context α, begin by writing $\alpha = \xi \cup \beta$ with ξ and β as in the preceding lemma. By Lemma 5.6, ξ admits a unique representation as a union of contexts ξ_i for the t_i in state X. Let $a_i = \mathrm{Val}(t_i, X, \xi_i)$ and let $X^* = (X \text{ but } \mathbf{x} \mapsto \mathbf{a})$. By the preceding lemma and the definition of $\vdash^0$, β is a context for R_0 in X^*. We define that R fails in X and α if and only if R_0 fails in X^* and β. The update set for R in X and α is defined to be the update set of R_0 in X^* and β.

To complete the verification of the postulates, we must produce a bounded exploration witness W. This will be the union $W' \cup W''$ of two parts. The first part, W', is the union of bounded exploration witnesses for the terms t_i. Recall that we saw, when discussing the semantics of terms of the form $f(t_1, \ldots, t_k)$, that if states X and Y agree as to the values of terms from W' when the variables are given values from $\mathrm{Range}(\alpha)$, then the q's such that $\alpha \vdash' q$ will be the same in both states and, if α is a context with respect to $\vdash'$ then the t_i will have the same values a_i in both states. To form W'', start with a bounded exploration witness W_0 for R_0 and, in each of its terms, replace all occurrences of any constant $\dot{x}_i$ with the following variant $\bar{t}_i$ of t_i. To get $\bar{t}_i$, make the following two substitutions in t_i. Replace all occurrences of variables v from $\mathbf{v}$ with the corresponding constants $\dot{v}$. Replace all subterms that begin with external function symbols by new, distinct variables.

To see that this W does what the Bounded Work Postulate requires, suppose X, Y and α are as in the postulate, i.e., α is an answer function for both of the states X and Y, and each term in W has the same values in X and Y when the variables are given the same values in $\mathrm{Range}(\alpha)$. We must show, first, that α causes the same queries in both states and, second, that if α is a context then it produces the same failures (if any) and the same updates in both states.

Suppose first that $\alpha \vdash_X q$. There are two cases to consider, according to whether $\alpha \vdash' q$ or $\alpha \vdash'' q$. In the first case, we have $\alpha \vdash^i_X q$ for some i. Since W includes a bounded exploration witness for $\vdash^i_X$, we can apply the Bounded Work Postulate for the term t_i to conclude that $\alpha \vdash^i_Y q$ and therefore $\alpha \vdash_Y q$, as required.

Suppose, therefore, that $\alpha \vdash'' q$. According to the definition of $\vdash''$, we have $\alpha = \xi' \cup \eta$, where ξ' is the union of contexts ξ_i for the $\vdash^i_X$, where $\eta \vdash^0 q$, where $\vdash^0$ is the causality relation associated to R_0 in the state $X^* = (X \text{ but } \mathbf{x} \mapsto \mathbf{a})$, and where $a_i = \mathrm{Val}(t_i, X, \xi_i)$. Because W includes a bounded exploration witness for each $\vdash^i_X$, the Bounded Work Postulate for t_i implies that ξ_i is a context for t_i in Y and $\mathrm{Val}(t_i, Y, \xi_i) = a_i$. (The fact that it is a context was deduced from the Bounded Work Postulate shortly after the statement of the postulate in [Blass and Gurevich

to appear (a), Section 5].) Let $Y^* = (Y \text{ but } \mathbf{x} \mapsto \mathbf{a})$. So we have that $\eta \vdash^0 q$ in X^*, and we must show that the same is true in Y^*. Since W_0 is a bounded exploration witness for R_0, it suffices to show that each term in W_0 has the same values in X^* and Y^* when the variables are given values in Range(η).

Consider any term $s \in W_0$. According to the definition of W'', there is a term $s^* \in W'' \subseteq W$ that is obtained from s by replacing each $\dot{x}_i$ with the $\bar{t}_i$ described in the definition. Compare the evaluation of s in X^*, using some values in Range(η) for the variables, and the evaluation of s^* in X, using the same values for those variables and values (to be specified later) from Range(α) for any additional variables that may occur in s. In the first evaluation, any subterm $\dot{x}_i$ gets value a_i, as given by the structure X^*. In the second, such a subterm $\dot{x}_i$ has been replaced by $\bar{t}_i$. Our intention is to get $\bar{t}_i$ to have value a_i in X. If we can achieve this, then it will follow that the two evaluations agree, since the rest of s (i.e., all but the $\dot{x}_i$) is unchanged in s^* and its evaluation proceeds the same way in X^* as in X. The same argument works with Y and Y^* in place of X and X^*. Thus, we shall have that the values of x in X^* and Y^* agree because they are the same as the values of s^* (which is in W) in X and Y, respectively.

We therefore try to obtain that the value in X of $\bar{t}_i$ is a_i, which was defined as the value of t_i in X with answer function ξ_i. There are, according to the definition of $\bar{t}_i$, two differences between t_i and $\bar{t}_i$. First, each v_j in t has been replaced by $\dot{v}_j$. Second, the subterms of t_i that begin with external function symbols have been replaced by new variables. The first of these modifications causes no problem, since the definition of evaluation of terms says to use, for any variable v_j, the value assigned by the structure to the constant $\dot{v}_j$. The second also causes no problem, since the variables introduced here can be assigned any values from Range(α). (Here it is important that they were new variables, not already assigned values in the evaluation of s.) So we simply assign to each of these variables the same value that the corresponding subterm of t_i had in X with ξ_i. Notice that this value is, by the definition of values of terms that begin with external function symbols, in Range(ξ_i) $\subseteq$ Range(α), so it is a permissible value for a variable here.

This completes the verification that, for suitable values of the new variables, the value of s in X^* agrees with the value of s^* in X. It therefore also completes the verification that $\eta \vdash^0 q$ in Y^* and thus $\alpha \vdash_X q$, as required.

We must still verify that our bounded exploration witness behaves properly with respect to failures and updates, but most of the work for this has already been done in the preceding treatment of causality. Suppose, in addition to the preceding assumptions on α, X, and Y, that α is a context for $\vdash_X$. So it is $\xi' \cup \eta$ where ξ' and η are as in the preceding discussion and, in addition, η is a context for $\vdash^0$. The argument above shows that terms in W_0 get the same values in X^* and Y^* when the variables are given the same values in Range(η). Since W_0 is a bounded exploration witness for R_0, we conclude that R_0 fails in X^* with η if and only if it fails in Y^* with η and that, if it doesn't fail, then it produces the same updates in these two states. But these failures and updates of R_0 are exactly the failures and updates of R in states X and Y, with α. So these also agree, and the verification of the Bounded Work Postulate is complete.

Fail. The causality relation and the update sets for `Fail` are empty, and it fails in all states and (necessarily empty) contexts. The postulates and cleanness are trivial in this case.

Remark 5.13. In Definition 5.1 of ordinary ASMs, we required that the set of states be closed under the transition function, which had not yet been defined. The preceding construction of the semantics of ASMs (under the temporary assumption that all structures of the appropriate vocabulary are states) determined the transition functions, via the specification contained in the Update Postulate. Thus, this construction completes Definition 5.1.

We close with two examples; additional examples are in Sections 2 and 3 of [Blass and Gurevich to appear (a)].

Example 5.14. We give a small but otherwise realistic example of an ordinary interactive algorithm and show how to represent it with an ASM. The algorithm's vocabulary contains nullary dynamic symbols x and y whose values in all states are numbers, which we think of as the coordinates of a point in the plane. The algorithm accepts as input from a user two numbers δx and δy, to be used as increments of x and y. (We write δx rather than the customary Δx to avoid any possible confusion with the notation for update functions.) So at the end of its step, the algorithm will update x and y to $x + \delta x$ and $y + \delta y$. But first, it wants to draw the new point (x, y) on the computer screen, and for this purpose, it must invoke a drawing routine provided by the operating system. So this drawing routine is part of the algorithm's environment. After it is invoked, the drawing routine calls back to our algorithm, asking for the coordinates of the point to be plotted. When it gets these coordinates, the drawing routine draws the point and confirms to the algorithm that this job has been done. At this point, the algorithm can complete its step.

In terms of queries and replies, the interaction here is as follows. The user is, of course, part of the environment, and the input he provides, δx and δy, is regarded as the reply to queries. These queries may represent explicit prompts issued by the algorithm, or they may be implicit queries indicating willingness to pay attention to input. The replies to these queries, i.e., the numbers provided by the user, cause the algorithm to call the drawing routine. This call is another query, whose reply will be the drawing routine's confirmation that it has done the job. (If the scenario didn't involve confirmation, then this call would be an output, i.e., a query for which only the vacuous and automatic reply "OK" is expected). The callbacks from the environment (i.e., from the drawing routine) are replies to implicit queries of the form "I'm willing to pay attention to input," and the algorithm's responses are outputs.

To write this as an ASM, we must decide on a vocabulary of external function symbols and output channels to correspond to the interaction described above. We use δx as a nullary external function symbol, whose associated query (associated by the template assignment) is the query asking for the user's first input. So the number supplied by the user will be the value of the term δx (in agreement with the notation used earlier). Of course, δy is handled analogously. The algorithm's initial call to the drawing routine will be the query assigned to an external function symbol

Draw, whose value will be the element sent by the drawing routine to confirm that its job is done. Next, we have the two implicit queries by which the algorithm looks for the drawing routine's callbacks. We use external function symbols *ReqX* (for "request x") and *ReqY* for these. So the values of these symbols will be whatever signals the drawing routine sends as its callbacks. The algorithm's outputs, in response to the callbacks, are of course the new values of x and y; we use X and Y as names for the output channels on which these numbers are sent.

For several of the queries used here — *Draw*, *ReqX*, and *ReqY* — the value of the reply doesn't matter, but the existence of the reply is important because it initiates further actions on the algorithm's part. In ASM syntax, one can express the existence of a reply, without saying anything more about that reply, by writing $Draw = Draw$ and similarly for the other queries. To make the intention behind such equations more evident, we employ the syntactic sugar $t!$ for $t = t$.

With these preparations, we can formalize our algorithm as the following ASM.

```
let u = x + δx, v = y + δy in
   do in parallel
      if ReqX! then Output_X(u) endif
      if ReqY! then Output_Y(v) endif
      if Draw! then
         do in parallel x := u, y := v enddo
      endif
   enddo
endlet
```

The initial let-bindings cause the queries for δx and δy to be issued; the computation proceeds only when replies have been received giving these values. They are added to x and y to give what will become, at the end of the current step, the new values of x and y. But during the current step, these values are temporarily assigned to u and v. Then the outer parallel block begins by issuing the three queries *ReqX*, *ReqY*, and *Draw*. If the drawing routine works as expected, the first two of these queries will get replies promptly (the callbacks), so the guards of the first two conditionals in our program will be `true` and the algorithm will produce the two outputs (the answers to the callbacks). Then the drawing routine will do its job and send confirmation, the reply to *Draw*. That makes the guard of the third conditional `true`, so the algorithm will update x and y and finish this step.

It is important to remember that the relative order of the components of a parallel block has no semantical effect. We have chosen to write the three conditional rules in an order that reflects when the replies are expected — *ReqX* and *ReqY* before *Draw*. But it would make no difference if we wrote the last of these conditionals first, to reflect the order in which the queries are issued.

Example 5.15. The following example was suggested by Dean Rosenzweig. Consider an algorithm that includes, among the functions of its state, an encryption mechanism and a function for producing e-mail messages with specified content and addressee. It could then, during a single step, encrypt a file and e-mail it to someone, as described by the ASM program

```
Output(Message(addressee, Encrypt(file))).
```

Now suppose the situation changes so that the encryption is no longer done locally within the algorithm but rather is done by an outside server called by our algorithm. Making the encryption operation external does not change the algorithm in an essential way. It is thus natural to continue to view the whole process as being done within a step. What was previously done by the function `Encrypt` is now the result of a conversation between our algorithm and the server, which is part of the environment. First the algorithm sends its username (formally, a query), and the server replies with a prompt for a password. When the algorithm sends the password (another query), the server replies with a prompt for the file to be encrypted. Finally, the algorithm sends the file and the server replies with the encrypted form of it. The process involves several external functions:

—`F`, whose associated template $\hat{\texttt{F}}$ attaches labels to a username to produce a query whose intuitive meaning is "this user wants to encrypt a file" and whose reply is a prompt for a password.

—`G`, whose $\hat{\texttt{G}}$ adds labels to a password prompt[5] and a password, producing a query with the intuitive meaning "here's my password"; the reply to this will be a prompt for the file to be encrypted.

—`E`, whose $\hat{\texttt{E}}$ adds labels to a file prompt and a file, producing a query that means "here's the file to encrypt"; the reply will be the encrypted file.

In terms of these functions, the change in the ASM program above will be to replace the term `Encrypt(file)` with the term

```
E(G(F(username), password), file).
```

(We have, for the sake of simplicity, omitted from the ASM program any instructions for what to do if something goes wrong, e.g., if the password is not accepted.)

REFERENCES

The AsmL webpage, `http://research.microsoft.com/foundations/AsmL/`.

Andreas Blass and Yuri Gurevich 2003. Abstract state machines capture parallel algorithms. *ACM Trans. Computational Logic 4:4*, 578–651.

Andreas Blass and Yuri Gurevich Ordinary Interactive Small-Step Algorithms, I. *ACM Trans. Computational Logic*, to appear (a).

Andreas Blass and Yuri Gurevich Ordinary Interactive Small-Step Algorithms, III. *ACM Trans. Computational Logic*, to appear (b).

Andreas Blass, Yuri Gurevich, Dean Rosenzweig and Benjamin Rossman. General interactive small-step algorithms. In preparation (a).

Andreas Blass, Yuri Gurevich, Dean Rosenzweig and Benjamin Rossman. Composite interactive algorithms (tentative title) In preparation (b).

Andreas Blass, Yuri Gurevich, Dean Rosenzweig and Benjamin Rossman. Interactive wide-step algorithms (tentative title) In preparation (c).

Yuri Gurevich 1995. Evolving algebra 1993: Lipari guide. In *Specification and Validation Methods*, E. Börger, Ed. Oxford Univ. Press, 9–36.

[5]The presence of the prompt in the query serves to make the ASM formalization simpler than it otherwise would be.

Yuri Gurevich 1997. ASM guide. Univ. of Michigan Technical Report CSE-TR-336-97. See [Huggins].

Yuri Gurevich 2000. Sequential abstract state machines capture sequential algorithms. *ACM Trans. Computational logic, 1:1*, 77—111.

James K. Huggins. ASM Michigan webpage. `http://www.eecs.umich.edu/gasm.`

Received August 2004; revised July 2005; accepted July 2005