Ordinary Interactive Small-Step Algorithms, III

ANDREAS BLASS
University of Michigan
and
YURI GUREVICH
Microsoft Research

This is the third 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 two papers 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 papers' definitions of such algorithms, of behavioral equivalence, and of abstract state machines (ASMs), we prove the main result: Every ordinary, interactive, small-step algorithm is behaviorally equivalent to an ASM.

We also discuss some possible variations of and additions to the ASM semantics.

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

The main purpose of this paper is to complete the proof that every ordinary, interactive, small-step algorithm is behaviorally equivalent to an abstract state machine (ASM). The algorithms in question are those which do only a bounded amount of work in any single computation step (small-step) but can interact with their environments during a step, by issuing queries and receiving replies (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

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.

answers to their queries (ordinary).

The first two papers of this series [Blass and Gurevich 2006; to appear] laid the groundwork for this proof. The class of ordinary, interactive, small-step algorithms was defined and studied in [Blass and Gurevich 2006] along with a suitable, quite strong notion of behavioral equivalence for such algorithms. In [Blass and Gurevich to appear], we defined the class of ordinary, interactive, small-step ASMs, and we defined a formal semantics for these ASMs whereby they are ordinary, interactive, small-step algorithms in the sense of [Blass and Gurevich 2006]. These papers thus provide all that is needed for stating our main result, the ASM thesis:

THEOREM 1.1. *Every ordinary, interactive, small-step algorithm is equivalent to an ordinary, interactive, small-step ASM.*

Henceforth, we shall omit "ordinary, interactive, small-step" except when needed for emphasis; we shall say simply "algorithm" and "ASM".

In Section 2, we briefly review preliminary material from [Blass and Gurevich 2006] and [Blass and Gurevich to appear] and add a few related observations. In Sections 3 and 4, we set up much of the technical machinery needed for the proof of the theorem. Section 5 presents the construction leading from a given algorithm to an equivalent ASM, and Section 6 establishes the correctness of the construction, thereby completing the proof of the theorem. The final two sections concern possible modifications of the ASM syntax and semantics. Section 7 is about variations in the semantics of let-rules and how these interact with the possible variations, discussed in [Blass and Gurevich to appear, Section 4], in the interpretation of repetitions of queries. Section 8 concerns two additional constructs — sequential composition and conditional terms — that could be added to the ASM syntax and semantics. It is shown that, although these constructs would surely improve programming convenience, they do not enlarge the class of algorithms that can be expressed. Thus, according to our theorem, all ASM programs that involve these constructs can be rewritten in terms of the ASM language as presented in [Blass and Gurevich to appear].

This paper and its predecessors [Blass and Gurevich 2006; to appear] continue the project, begun in [Gurevich 2000] and continued in [Blass and Gurevich 2003], 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].

2. REVIEW OF PARTS I AND II

This section is intended to recapitulate just enough of the preceding two papers in this series to allow for convenient reference later. We shall omit the extensive explanations and justifications given in [Blass and Gurevich 2006; to appear] for the concepts reviewed here. We shall be especially brief in the case of material from [Blass and Gurevich 2006], since it is already available not only in its extensive form in [Blass and Gurevich 2006] but in summary form in [Blass and Gurevich to appear, Section 2].

2.1 Algorithms

Algorithms are defined to be entities that satisfy the States, Interaction, Update, Isomorphism, and Bounded Work Postulates stated below. We intersperse with the postulates the definitions of concepts used in them; for more details and motivation, see [Blass and Gurevich 2006], especially Section 5.

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*.

Definition 2.1. A *potential query* for a state X is a tuple of elements of $X \sqcup \Lambda$. A *potential reply* is an element of X. An *answer function* is a partial function from potential queries to potential replies.

Interaction Postulate: The algorithm determines, for each state X, a relation, called its *causality relation*, $\vdash_X$ or just $\vdash$ when X is clear, between finite answer functions and potential queries.

Definition 2.2. Given an answer function α for a state X, we define the monotone operator $\Gamma_{X,\alpha}$, or just Γ_α when X is understood, on sets of potential queries by

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

Define ${\Gamma_\alpha}^\infty$ to be the least fixed point of Γ_α.

Iteration of Γ_α produces the sequence of sets

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

Thanks to the Bounded Work Postulate, this iteration will continue for only a finite number of steps before reaching the least fixed point ${\Gamma_\alpha}^\infty$; see [Blass and Gurevich 2006, Lemma 5.19].

Definition 2.3. When a state and therefore a causality relation are understood, we write α^n for $\alpha \restriction {\Gamma_\alpha}^n$. Similarly, $\alpha^\infty = \alpha \restriction {\Gamma_\alpha}^\infty$.

Definition 2.4. A *context* for a state X is an answer function α for X such that $\mathrm{Dom}(\alpha) = {\Gamma_\alpha}^\infty$. An answer function α is *well-founded* if $\mathrm{Dom}(\alpha) \subseteq {\Gamma_\alpha}^\infty$. A causality relation $\vdash$ is *clean* if whenever $\alpha \vdash q$ then α is well-founded.

The definition of "context" was different in [Blass and Gurevich 2006], but it follows from [Blass and Gurevich 2006, Lemma 5.7] that the two definitions are equivalent. Since we shall need to invoke that lemma and its corollary again, we state them here (combined) for reference.

LEMMA 2.5. *Let α be an answer function. If ${\Gamma_\alpha}^\infty \subseteq Dom(\alpha)$ then $\alpha \restriction {\Gamma_\alpha}^\infty = \alpha^\infty$ 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 $q \in Dom(\alpha) \cap Dom(\beta)$.*

Definition 2.6. An *update* for a state X is a triple $\langle f, \mathbf{a}, b\rangle$ where f is an n-ary dynamic function symbol (for some n), $\mathbf{a}$ is an n-tuple of elements of X, and b is an element of X.

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

—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 that, if two updates in $\Delta^+_A(X, \alpha)$ *clash*, meaning that they are $\langle f, \mathbf{a}, b\rangle$ and $\langle f, \mathbf{a}, b'\rangle$ with $b \neq b'$, then the algorithm must fail in (X, α), for the next state would be subject to contradictory requirements.

Isomorphisms between states are extended in the natural way to apply to associated entities such as queries, answer functions, and updates.

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

Definition 2.8. Two states X and X' are said to *agree* over a function α with respect to a set W of terms if 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 Range(α).

Bounded Work Postulate:

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

—There is a finite set W of terms, depending only on the algorithm, with the following properties. Assume that states X and X' agree over α with respect to W. 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 2006, 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 2.9. A set W as required by the last part of the Bounded Work Postulate is called a *bounded exploration witness* for the algorithm.

We next summarize some easy consequences of the definitions and postulates.

LEMMA 2.10. *Suppose $i : X \cong Y$ is an isomorphism of states and α is an answer function for X. Then, for each k,*

$$i\left({\Gamma_\alpha}^k\right) = {\Gamma_{i\circ\alpha\circ i^{-1}}}^k,$$

where the Γ on the left side is calculated in X and that on the right in Y.

Proof Immediate from the Isomorphism Postulate. □

Remark 2.11. If W is a bounded exploration witness, then so is any set obtained from W by renaming occurrences of variables, as long as distinct variables remain distinct. Indeed, if two states agree over α with respect to the new set, then they also agree with respect to the original W. It is permitted for two occurrences of the same variable to be renamed as distinct variables. In particular, there is a bounded exploration witness in which no variable occurs more than once.

LEMMA 2.12. *If X and X' agree over α and if α is a context for X, then it is also a context for X'.*

Proof This was proved in the discussion following the Bounded Work Postulate in [Blass and Gurevich 2006, Section 5]. □

LEMMA 2.13. *Every well-founded answer function is a subfunction of some context. Therefore, the bounds, in the Bounded Work Postulate, on the number and length of queries in all contexts apply also to all well-founded answer functions.*

Proof See [Blass and Gurevich to appear, Lemma 2.24 and Corollary 2.25]. □

LEMMA 2.14. *If $\alpha \subseteq \beta$ then ${\Gamma_\alpha}^n \subseteq {\Gamma_\beta}^n$ and $\alpha^n \subseteq \beta^n$.*

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

LEMMA 2.15. *For any answer function α, each ${\alpha^n}$ is well-founded. ${\alpha^\infty}$ is the largest well-founded subfunction of α.*

Proof See [Blass and Gurevich 2006] from Proposition 6.16 to Proposition 6.18. □

LEMMA 2.16. *If α and β are answer functions whose restrictions to ${\Gamma_\alpha}^n$ are equal, then ${\Gamma_\alpha}^k = {\Gamma_\beta}^k$ for all $k \leq n+1$, and ${\alpha^k} = {\beta^k}$ for all $k \leq n$.*

Proof This is Lemma 6.14 in [Blass and Gurevich 2006]. □

COROLLARY 2.17. ${{\Gamma_{\alpha^n}}^k} = {{\Gamma_\alpha}^k}$ *for all $k \leq n+1$, and ${\alpha^k} = {(\alpha^n)^k}$ for all $k \leq n$.*

LEMMA 2.18. *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 See [Blass and Gurevich to appear, Lemma 2.23]. □

2.2 Reachability and Equivalence

Definition 2.19. Fix a causality relation. A query q is *reachable* under an answer function 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 2006] from Definition 6.9 to Proposition 6.11.

Definition 2.20. Two causality relations are *equivalent* if, for every answer function α, they make the same queries reachable under α.

LEMMA 2.21. *Two causality relations are equivalent if and only if, for every answer function α that is well-founded for both of the causality relations, the same queries are caused by subfunctions of α. Moreover, 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.*

Proof This is part of Proposition 6.21 and Corollary 6.22 in [Blass and Gurevich 2006]. □

Definition 2.22. Two algorithms are *(behaviorally) equivalent* if they have

—the same states and initial states,
—the same vocabulary and 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 $\Delta^+(X, \alpha)$.

2.3 Abstract state machines

Ordinary, interactive, small-step ASMs were defined in [Blass and Gurevich to appear, Sections 3–5], to which we refer for the many things omitted from the following brief summary. We adopt, without repeating them, the standard ASM conventions about vocabularies and states; see [Blass and Gurevich 2006, Convention 5.2].

Definition 2.23. An *ordinary ASM* with the finite vocabulary Υ and the finite label set Λ consists of

—an ASM program (see Definition 2.27 below) 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 (see Definition 2.24 below) 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 given by the ASM semantics, and

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

Definition 2.24. A *template* for n-ary functions is a finite tuple whose components are the placeholders $\#1, \ldots, \#n$, occurring exactly once each, and elements of Λ.

Definition 2.25. *Terms* are built just as in traditional first-order logic, using function symbols from $\Upsilon \cup E$ and variables. Boolean terms are defined to be the Boolean variables and those compound terms that begin with relational function symbols.

Definition 2.26. *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.

Definition 2.27. An ASM *program* is a rule with no free variables.

This concludes the syntax of ASMs. The semantics associates to each rule an algorithm (for an appropriate vocabulary, with additional constant symbols as replacements for the rule's free variables), and it associates to each term an entity similar to an algorithm but having, in place of an update set, a possible value. Because of the length of the definition, we refer the reader to [Blass and Gurevich to appear, Section 5] rather than repeating it here.

In Subsection 7.4, we shall propose an addition to the ASM syntax, to bridge the gap between the Lipari convention and the must-vary convention (explained in [Blass and Gurevich to appear, Section 4]) for the interpretation of repeated occurrences of function symbols. Until then, we proceed, as in [Blass and Gurevich to appear], under the Lipari convention.

It is convenient to introduce the following syntactic sugar to abbreviate a couple of commonly used constructions.

Convention 2.28. We use `skip` as an abbreviation for the empty block, i.e., the parallel combination of no subrules (`do in parallel enddo`). We also abbreviate `if` φ `then` R `else skip endif` as `if` φ `then` R `endif`.

3. PHASES AND MATCHING

We now begin working toward the proof of the ASM thesis, Theorem 1.1. So let A be an ordinary algorithm, with vocabulary Υ, label set Λ, set of states $\mathcal{S}$, set of initial states $\mathcal{I}$, causality relations $\vdash_X$ (for all states X), update sets $\Delta^+(X, \alpha)$, transition function τ, bound B on the number and length of queries in any state and context, and bounded exploration witness W. We refrain from introducing a special symbol for failure, but, as the Update Postulate requires, A also determines the states and contexts in which it fails. (Conceptually, the bound on the number of queries and the bound on their length are separate matters, but to simplify notation we take B to be the larger of the two, so that a single B serves both purposes.) All these aspects of A are assumed to satisfy the postulates in Section 2.

In this section, we first make some technical simplifications and observations. Then we introduce an informal picture of the computation performed by the algorithm in one step, starting in the state X and obtaining answers from the environment as given by the well-founded answer function α. After this, and after introducing some convenient notation, we introduce a notion of "matching", intended to capture the idea that two states and associated answer functions behave the same way for at least a certain part of a computation step.

3.1 Preliminaries

We begin by making two simplifications, replacing A by equivalent algorithms. First, we normalize the causality relation for every state, as described in [Blass and Gurevich 2006, Section 6.3]. We recall the definition and essential properties of this construction.[1] The normalization $\tilde{\vdash}$ of a causality relation $\vdash$ is defined by letting $\alpha\tilde{\vdash}q$ mean that α is well-founded and q is reachable under α (with respect to $\vdash$). It was shown in [Blass and Gurevich 2006] that $\tilde{\vdash}$ is equivalent to $\vdash$, and that two causality relations are equivalent if and only if their normalizations are equal. This almost implies that, if we replace, in our algorithm A, all the causality relations $\vdash_X$ by their normalizations $\tilde{\vdash}_X$, then the resulting algorithm is equivalent to A. The point of "almost" here is that we must check that the result of the replacement is an algorithm, i.e., that it still satisfies the postulates. Most of this follows immediately from the fact, proved in [Blass and Gurevich 2006, Corollary 6.22], that equivalent causality relations have the same well-founded answer functions and the same contexts. We must, however, still verify that the bounded exploration witness remains correct.

LEMMA 3.1. *Let W be a bounded exploration witness for the algorithm A. Then W is also a bounded exploration witness when all the causality relations $\vdash_X$ of A are replaced by their normalizations $\tilde{\vdash}_X$.*

Proof It suffices to check that W behaves correctly with respect to causality; its behavior with respect to updates and failures automatically remains correct, because updates, failures, and contexts are unchanged by the normalization. Suppose, therefore, that states X and X' agree over answer function α with respect to W and that $\alpha\tilde{\vdash}_X q$. So, with respect to $\vdash_X$, α is well-founded and q is reachable under α. The latter means (see Definition 2.19) that there is a trace, i.e., a finite sequence $\langle q_1, \ldots, q_n\rangle$ of queries, ending with $q_n = q$, and such that each q_i in the sequence is caused (with respect to $\vdash_X$) by the restriction of α to a subset of $\{q_j : j < i\}$. Because W is a bounded exploration witness for A, the same sequence is a trace with respect to $\vdash_{X'}$. So q is reachable under α with respect to $\tilde{\vdash}_{X'}$. Furthermore, the same argument, applied with an arbitrary member of $\mathrm{Dom}(\alpha)$ in place of q, shows that α is well-founded with respect to $\tilde{\vdash}_{X'}$. Therefore, $\alpha\tilde{\vdash}_{X'}q$, as required. □

Notice that normalization makes a causality relation clean. Indeed, if $\alpha\tilde{\vdash}q$ then α is well-founded with respect to $\vdash$ (by definition of the normalization $\tilde{\vdash}$) and therefore with respect to $\tilde{\vdash}$ (since equivalent causality relations have the same well-founded answer functions).

Our second simplification is that, by adding finitely many more terms to the bounded exploration witness W, we can arrange that it contains a variable and is closed under subterms. That is, it is normalized in the sense of [Blass and Gurevich to appear, Section 5.2] insofar as that notion applies to rules. (The last clause there is specific to terms and thus irrelevant here.) In addition, we arrange that W contains `true` and `false`.

[1] In [Blass and Gurevich 2006], we used the notation $\vdash'$ for the normalization of $\vdash$. Since $\vdash'$ is used for other purposes in [Blass and Gurevich to appear], we change the notation for normalization here to $\tilde{\vdash}$.

Proviso 3.2. Summarizing, we **assume from now on, until the end of the proof of Theorem 1.1** that A has clean (in fact, if we need it, normalized) causality relations $\vdash_X$ and that the bounded exploration witness W contains `true`, `false`, and a variable, and is closed under subterms. We also **fix the notations** Υ, Λ, $\mathcal{S}$, $\mathcal{I}$, $\vdash_X$, Δ^+, B, and W as above.

Recall from Lemma 2.13 that the number and size of the queries in the domain of any well-founded answer function are no larger than B. In particular, the number and length of the queries involved in any cause are no larger than B.

LEMMA 3.3. *For any well-founded answer function ξ (for any state), ${\Gamma_\xi}^\infty = {\Gamma_\xi}^{B+1}$, and $\xi = \xi^B$.*

Proof Since $\mathrm{Dom}(\xi)$ has cardinality at most B, and since the subfunctions ξ^n form a weakly increasing sequence (with respect to $\subseteq$), there must be some $k \leq B$ such that $\xi^k = \xi^{k+1}$. Then, since

$$ {\Gamma_\xi}^{k+1} = \{q : \zeta \vdash q \text{ for some } \zeta \subseteq \xi^k\}, $$

and since the analogous formula holds with k replaced by $k+1$, we have that ${\Gamma_\xi}^{k+1} = {\Gamma_\xi}^{k+2}$ and therefore $\xi^{k+1} = \xi^{k+2}$. Proceeding by induction, we find that ${\Gamma_\xi}^n = {\Gamma_\xi}^\infty$ for all $n \geq k+1$ and that $\xi^n = \xi \restriction {\Gamma_\xi}^\infty = \xi$ for all $n \geq k$. Since $k \leq B$, the proof is complete. □

This lemma admits the following slight improvement, changing $B+1$ to B in the first part of the conclusion.

LEMMA 3.4. *For any well-founded answer function ξ (for any state), ${\Gamma_\xi}^\infty = {\Gamma_\xi}^B$.*

Proof Let ξ be a well-founded answer function, and suppose, toward a contradiction, that there is a query $q \in {\Gamma_\xi}^{B+1} - {\Gamma_\xi}^B$. We already know that $\xi = \xi^B$, so $\mathrm{Dom}(\xi) \subseteq {\Gamma_\xi}^B$, and in particular $q \notin \mathrm{Dom}(\xi)$. Extend ξ to the properly larger answer function $\xi' = \xi \cup \{(q, r)\}$ for an arbitrarily chosen reply r. Since q is in ${\Gamma_\xi}^{B+1}$, it is reachable under ξ and, a fortiori, under ξ'; thus ξ' is well-founded. By Lemma 3.3 and the definition of $(\xi')^B$, we have that $\mathrm{Dom}(\xi') = \mathrm{Dom}((\xi')^B) \subseteq {\Gamma_{\xi'}}^B$ and therefore $q \in {\Gamma_{\xi'}}^B$. Let k be the smallest integer such that $q \in {\Gamma_{\xi'}}^{k+1}$; so $k < B$. By induction on j, and remembering that q is the only element of $\mathrm{Dom}(\xi') - \mathrm{Dom}(\xi)$, we find that, for all $j \leq k$, ${\Gamma_{\xi'}}^j = {\Gamma_\xi}^j$. In particular, $\xi^k = (\xi')^k$. But $q \in {\Gamma_{\xi'}}^{k+1}$, so $\eta \vdash q$ for some $\eta \subseteq (\xi')^k = \xi^k$. That means $q \in {\Gamma_\xi}^{k+1} \subseteq {\Gamma_\xi}^B$, contrary to our choice of q. □

We next recall and slightly improve some information from [Blass and Gurevich 2006] about critical elements of a state. First, recall the definition.

Definition 3.5. Let ξ be an answer function for a state X. An element of X is *critical* for ξ if it is the value of some term in W for some assignment of values in $\mathrm{Range}(\xi)$ to its variables.

In particular, since W contains a variable, all elements of $\mathrm{Range}(\xi)$ are critical for ξ.

LEMMA 3.6. *Let X be a state and suppose $\xi \vdash_X q$. Then all the components in X of q are critical for ξ.*

Proof It was shown in [Blass and Gurevich 2006, Proposition 5.23] that, if X is a state, α a context, ξ a subfunction of α^n for some n, and $\xi \vdash_X q$, then all components of q are critical for α^n. The proof, however, did not use the assumption that α is a context; it works for any answer function.

In particular, it works when $\alpha = \xi$. In order to use it for this choice of α, we must know that $\xi \subseteq \xi^n$ for some n. But in the situation at hand, since $\xi \vdash_X q$ and $\vdash_X$ is clean, we have $\mathrm{Dom}(\xi) \subseteq {\Gamma_\xi}^\infty = {\Gamma_\xi}^n$ for sufficiently large n. Therefore, for such n, $\xi = \xi^n$. Thus, we get that all components in X of q are critical for $\xi^n = \xi$. □

LEMMA 3.7. *Let α be a context for the state X, and let $\langle f, \mathbf{a}, b\rangle \in \Delta^+(X, \alpha)$. Then b and all components of $\mathbf{a}$ are critical for α.*

Proof This is proved in [Blass and Gurevich 2006, Proposition 5.24]. □

3.2 Phases

The informal picture promised above is as follows. The step begins with state X and with no queries issued yet and thus no replies received yet. That is, the current answer function is $\varnothing = \alpha^0$. All queries caused by this are issued. We call this part of the computation *phase* 0, since it uses only α^0. Notice that the queries issued in this phase of the computation are those in the set

$$\{q : \varnothing \vdash q\} = \{q : (\exists \xi \subseteq \alpha \restriction \varnothing)\, \xi \vdash q\} = \Gamma_\alpha(\varnothing) = {\Gamma_\alpha}^1.$$

Next, the algorithm receives whatever answers to these queries are given in α. It is not guaranteed that all these queries are in the domain of α, so some of them may remain unanswered. The replies received to the queries from phase 0, being in accordance with α, are given exactly by the answer function α^1.

In the next phase, called phase 1, the algorithm issues all the new queries caused by these answers, i.e., all q not already issued such that $\xi \vdash q$ for some $\xi \subseteq \alpha^1$. Inspecting the definitions, we find that the set of queries issued so far is exactly ${\Gamma_\alpha}^2$. (This includes the queries from phase 0 as well as the current phase 1.) Next, the algorithm receives α's replies to (some of) these queries. The replies received so far constitute exactly α^2.

The computation continues in the same fashion for up to B phases. At the beginning of any phase $n < B$, the algorithm has received the answers in α^n. It issues all the new queries caused by these answers, i.e., all q not already issued such that $\xi \vdash q$ for some $\xi \subseteq \alpha^n$. These are the queries in ${\Gamma_\alpha}^{n+1} - {\Gamma_\alpha}^n$, so the set of all queries issued up to this point is ${\Gamma_\alpha}^{n+1}$. Next the algorithm receives α's replies to (some of) these queries, and the replies received up to this point constitute exactly α^{n+1}. Thus, the algorithm is ready to begin phase $n + 1$.

At the end of phase $B - 1$ (i.e., after B phases, because we started counting with 0), the algorithm has issued all the queries in ${\Gamma_\alpha}^B = {\Gamma_\alpha}^\infty \supseteq \mathrm{Dom}(\alpha)$. (The equality here is Lemma 3.4 and the inclusion is the definition of well-foundedness.) When it has received their answers and is ready to begin phase B, it has received

all the information it will ever get from α. If this does not include answers to all the queries issued, i.e., if $\mathrm{Dom}(\alpha) \subsetneq {\Gamma_\alpha}^\infty$, then the computation in this step hangs. (Remember that an ordinary algorithm cannot complete a step unless all its queries from that step are answered.) If, on the other hand, all the queries have been answered, so $\mathrm{Dom}(\alpha) = {\Gamma_\alpha}^\infty$ and α is therefore a context, then the algorithm completes this step by either failing or computing and executing the update set $\Delta^+(X, \alpha)$.

In what follows, we shall use this informal picture to guide our construction of an ASM equivalent to the given algorithm A.

3.3 Uniformity across states and answer functions; matching

We continue to work with a fixed algorithm, but in this subsection we shall consider varying states and answer functions. As before, we let B be the bound on the number and length of queries, and we let W be the bounded exploration witness, provided by the Bounded Work Postulate, and normalized as above to be closed under subterms and to contain `true`, `false`, and at least one variable.

The fact that W is a bounded exploration witness means that, if X and X' agree over α then

—if $\alpha \vdash_X q$ then $\alpha \vdash_{X'} q$ and

—if α is a context for X (and therefore also for X') then the algorithm fails in both or neither of (X, α) and (X', α) and, if it doesn't fail, then $\Delta^+(X, \alpha) = \Delta^+(X', \alpha)$.

Recall also that, for any state X and well-founded answer function α, the iteration of Γ_α leading to ${\Gamma_\alpha}^\infty$ has at most B steps. That is, ${\Gamma_\alpha}^B = {\Gamma_\alpha}^\infty$ and $\alpha^B = \alpha$; see Lemma 3.4. The picture of the computation described in the preceding subsection involves at most $B + 1$ phases, namely at most B phases for sending queries and one final phase, if α is a context, for either failing or computing and executing the transition (if any) to the next state. We now analyze how this computation depends, phase by phase, on the state X and answer function α. To reduce repetition, we adopt the following notational conventions.

Convention 3.8. Unless the contrary is stated explicitly, when we refer to a pair (X, α) (possibly with primes), we intend that X is a state and α a well-founded answer function for X. Furthermore, the notations Γ_α and α^n, which were defined with a fixed state in mind, are assumed to refer to the unique state X such that (X, α) is under consideration. If several X's are under consideration with the same α, then we specify the intended one by writing $\Gamma_{X,\alpha}$ or α^n_X.

Definition 3.9. An *isomorphism* from a pair (X, α) as above to another such pair (Y, β) is an isomorphism of structures, $i : X \cong Y$, such that $\beta \circ i = i \circ \alpha$.

Note that the equation $\beta \circ i = i \circ \alpha$ means that β is the answer function $i \circ \alpha \circ i^{-1}$ to which i sends α.

Definition 3.10. We say that (X', α') *matches* (X, α) *up to phase* n if

—X and X' agree over α^n and

—$\alpha^n = \alpha'^n$.

In accordance with Convention 3.8, it is to be understood here that $\alpha^n = \alpha^n_X$ and $\alpha'^n = \alpha'^n_{X'}$. Notice that "matching up to phase n" is, for each n, an equivalence relation on pairs (X, α) of a state and a well-founded answer function.

Although phases were introduced in the informal picture in the preceding subsection, the present definition of "matching up to a phase" is entirely formal. It is intended to fit with the informal discussion but it does not depend on that discussion.

LEMMA 3.11. *If (X', α') matches (X, α) up to phase n, then*

—$\alpha^k = \alpha'^k$ *for all* $k \leq n$, *and*
—${\Gamma_\alpha}^k = {\Gamma_{\alpha'}}^k$ *for all* $k \leq n+1$.

Proof The first assertion follows immediately from the assumption that $\alpha^n = \alpha'^n$ and Corollary 2.17. For the second, recall that, for each $k > 0$,

$${\Gamma_\alpha}^k = \{q : (\exists \xi \subseteq \alpha^{k-1})\, \xi \vdash_X q\}$$

and similarly,

$${\Gamma_{\alpha'}}^k = \{q : (\exists \xi \subseteq \alpha'^{k-1})\, \xi \vdash_{X'} q\}.$$

Since $k \leq n+1$, the part of the lemma already proved gives $\alpha^{k-1} = \alpha'^{k-1}$. So the only difference between the right sides of the displayed formulas is that one uses $\vdash_X$ and the other uses $\vdash_{X'}$. But this is no real difference; since W is a bounded exploration witness and since X and X' agree over every $\xi \subseteq \alpha^n$, the same q's satisfy $\xi \vdash_{X'} q$ and $\xi \vdash_X q$. □

COROLLARY 3.12. *If (X', α') matches (X, α) up to phase B, then $\alpha = \alpha'$. If, in addition, α is a context for X, then*

—*α is a context for X',*
—*A fails in (X, α) if and only if it fails in (X', α), and*
—*if it doesn't fail then $\Delta^+(X, \alpha) = \Delta^+(X', \alpha)$.*

Proof By the lemma and our choice of B, we have

$$\alpha = \alpha^B = \alpha'^B = \alpha'.$$

If α is a context for X, then we have

$$\mathrm{Dom}(\alpha) = {\Gamma_{X,\alpha}}^B = {\Gamma_{X',\alpha}}^B,$$

so α is also a context for X'. (We have included the subscripts X and X' since, with α and α' equal, we can no longer rely on the notation for them to indicate which state is intended.) Finally, using the fact that W is a bounded exploration witness, we have that failure in either of (X, α) and (X', α) implies failure in the other and that, when there is no failure,

$$\Delta^+(X, \alpha) = \Delta^+(X', \alpha).$$

□

4. MORE UNIFORMITY; SIMILARITY

4.1 Informal picture

The preceding results show that, to tell what an algorithm will do in a particular state X under a (well-founded) answer function α, it suffices to have partial information about (X, α), namely the values of terms in W when their variables get values in Range(α). Our next goal is to show that, in a sense, far less information suffices. Instead of knowing what these critical elements are, we need only know which of them are equal. The reason is that, according to the Isomorphism Postulate, knowledge up to isomorphism is, in a sense, sufficient. The phrase "in a sense" refers to the fact that, if we know the state (and answer function) only up to isomorphism, then of course the queries and updates produced by the algorithm are also determined only up to (the same) isomorphism. This will cause no difficulties, since we already know from Lemmas 3.6 and 3.7 that the elements of the state involved in queries and updates are among the critical values, so we can keep track of the effect of the isomorphism on them.

We now begin to make these remarks precise. The proof will be based on that of [Gurevich 2000, Lemma 6.9], but some additional work is needed to account for the answer functions. So we begin with an informal discussion to motivate that work and the definitions involved in it. The informal discussion will use the picture of the computation proceeding in phases as discussed earlier.

Let (X, α) and (X', α') be two states, each equipped with an answer function, and consider first what happens in phase 0 of the algorithm's execution in these states. So the answer function being used in this phase is $\alpha^0 = \alpha'^0 = \varnothing$. Assume that, for each two closed terms $t_1, t_2 \in W$, their values in X are equal if and only if their values in X' are equal. Then there is a (Y, β) that is isomorphic to (X, α) and matches (X', α') up to phase 0. (Recall that by "isomorphic" we mean that there is an isomorphism $i : X \cong Y$ such that $\beta = i \circ \alpha \circ i^{-1}$.) The existence of such (Y, β) is most easily seen if (the base sets of) X and X' are disjoint, for in this case the required Y can be obtained from X by replacing the denotation in X of each closed term $t \in W$ by the denotation in X' of the same term t. Our assumption about equalities between terms ensures that this replacement is well defined and one-to-one. If X and X' are not disjoint, we can first replace X with an isomorphic copy disjoint from X' and then proceed as before.

As a result, the queries issued by (Y, β) in phase 0 are, on the one hand, the same as those issued by (X', α') (because of the matching, thanks to Lemma 3.11) and, on the other hand, related via the isomorphism to those issued by (X, α). Thus, we find that (X, α) and (X', α') issue, in phase 0, queries that are related by the isomorphism $X \cong Y$. The components of these queries, which are values of closed terms from W by Lemma 3.6, are thus "the same" in the sense that the same closed terms from W produce the corresponding queries in (X, α) and (X', α').

Next, let us consider what happens in phase 1. The main idea is similar, but the situation now differs from that in the preceding paragraphs, because we are dealing with (possibly) nonempty answer functions α^1 and α'^1, and we must pay attention also to β^1 in our construction of the intermediate (Y, β). Where we mentioned closed terms above, we will now have terms in which variables can occur and are to be given values from the ranges of the answer functions.

As before, we can arrange that X and X' are disjoint, and as before, we wish to obtain Y by replacing certain elements of X by the corresponding elements of X'. The elements of X to be replaced are those that are critical for α^1, the answer function used during phase 1, and the corresponding elements of X' are the critical elements there. What needs some care is determining which critical elements of X correspond to which critical elements of X'. Since the critical elements are values of terms from W, when variables are given values in the range of the answer function, corresponding critical elements should be values of the same term from W, with corresponding values for the variables. And what are corresponding values, in Range(α^1) and Range(α'^1) respectively, for the variables? They are $\alpha^1(q)$ and $\alpha'^1(q')$ where q and q' are corresponding queries. Since the queries at this phase are tuples whose components are either labels from Λ or critical elements for $\alpha^0 = \varnothing = \alpha'^0$, there is a clear notion of "corresponding" queries: They have the same length, the Λ components are the same, and the other components are values of the same closed terms from W. Thus, we have, with some effort, determined which elements of X should be replaced by which elements of X' to obtain Y. Obtaining β is then easy, since we must have $\beta = i \circ \alpha \circ i^{-1}$ where i is the isomorphism we constructed from X to Y. Of course, if several terms denote the same critical element in X, then they should, with corresponding values for the variables, denote the same critical element in X' also, and vice versa. This requirement is entirely analogous to what was already assumed for closed terms in phase 0.

Once again, the isomorphism from (X, α) to (Y, β) and the matching up to phase 1 between the latter and (X', α') ensure that the queries issued by (X, α) and (X', α') agree in the sense that corresponding components are values of the same terms in W when the variables are replaced by corresponding values of the functions α and α', where "corresponding values" means that these functions are applied to tuples made from values, in the two structures, of the same closed terms from W.

In phase 2, we can proceed analogously until we arrive at the question of when two queries should be considered as corresponding. As before, they should have the same length and the same components from Λ, but the requirement for the components that are critical elements becomes a bit more complicated since these elements are now values of the answer functions at queries made from values of terms that are not necessarily closed (as they were before) but can have variables assigned corresponding values in the ranges of α^1 and α'^1. So instead of requiring components of q and q' to be built (by means of terms in W) from values of answer functions at values of the same closed term, we now require them to be built from values of the answer functions at the values of the same term with corresponding values of the free variables. Here "corresponding" refers to the concept for phase 1 as developed in the preceding paragraph.

For later phases, the same pattern continues, but expressing it becomes more and more convoluted. We therefore organize it into an induction that avoids the need for the most convoluted phrasings. Specifically, "corresponding" is defined by an induction on phases; we have seen, for example, that the definition of this notion at phase 2 depends on its availability for phase 1. For the construction of Y to succeed, we must require that equality relations between terms from W, with corresponding values for variables, be the same in X as in X'. In the following, we shall give precise

definitions of these concepts and carry out the constructions without depending on the informal picture of phases used in the preceding discussion.

4.2 Tags and their values

An important ingredient in a precise formulation of the preceding ideas is to define the extent to which two (state, answer-function) pairs must resemble each other in order to get a suitable correspondence between what they do up to phase n. Thus, in our discussion of phase 0, we needed that the states should satisfy the same equations between closed terms from W. In phase 1, we needed the analogous hypothesis not only for closed terms but also for terms with variables, provided the variables are given values that are obtained by applying the answer functions to queries made from the corresponding values of some closed terms. In phase 2, we needed it also when the variables are assigned values of the answer functions at queries made from the corresponding values of some terms with variables wherein these (second level) variables are assigned values of the answer functions at queries made of values of closed terms. And so forth.

We begin with some definitions designed to handle this sort of bookkeeping. The symbol ρ used here is intended to suggest "reply"; think of $\rho(q)$ as meaning the reply to the query q. The "element-tags" introduced in these definitions denote, at phase n, those elements that must have the same equality relations in (X, α) as in (X', α') in order that the algorithms' actions in phase n correspond properly.

Definition 4.1. An *element-tag* or *e-tag* is the result of taking any term in W and replacing its variables by expressions of the form $\rho(p)$ where the p's are query-tags. A *query-tag* or *q-tag* is a tuple, of length at most B, of elements of Λ and element-tags.

Notice that this definition by simultaneous recursion has as its basis the e-tags that are closed terms from W (needing no q-tags to substitute for variables) and the q-tags that are tuples of elements of Λ (needing no e-tags for non-Λ components). Notice also that the definition does not depend on any particular state or answer function; tags are determined by the algorithm (and the specified bounded exploration witness W).

There is an obvious notion of subtag; the following definition formalizes it and adds terminology for keeping track of how many occurrences of ρ have a given subtag in their scopes.

Definition 4.2. Any tag has itself as a subtag of depth 0. In addition:

—An e-tag of the form $\rho(p)$ has, as subtags of depth d, all the subtags of p of depth $d-1$.

—An e-tag of the form $f(t_1, \ldots, t_n)$ has, as subtags of depth d, all subtags of depth d in the e-tags t_i.

—A q-tag has, as subtags of depth d, all the subtags of depth d in its e-tag components.

The *nesting level* of a tag is the maximum depth of any subtag.

Notice that in the second clause all the t_i are e-tags because W is closed under subterms.

LEMMA 4.3. *For any n, there are only finitely many tags of nesting level n.*

Proof This follows immediately, by induction on n, from the finiteness of W, Λ, and B. □

Except in degenerate situations, the total number of tags of all nesting levels is countably infinite, but we shall never have any real need for tags of nesting level greater than B, so there are only finitely many tags relevant for our purposes. In fact, we shall occasionally say "all tags" or "arbitrary tags" when we really mean only those of nesting level $\leq B$.

Definition 4.4. Let α be an answer function for a state X. We specify the *value*, in (X, α), of a tag *at phase* n by the following recursion on the natural number n. These values are sometimes undefined. When defined, the values of e-tags will be elements of X; the values of q-tags will be potential queries for X.

—The value at phase n of an e-tag of the form $\rho(p)$ is obtained by applying α^n to the value at phase $n-1$ of the q-tag p. It is undefined if the value of p at phase $n-1$ is either undefined or outside the domain of α^n.

—The value at phase n of an e-tag of the form $f(t_1, \ldots, t_n)$ is obtained by applying the interpretation f_X of f in X to the values at phase n of the t_i's. It is undefined if any of the t_i have no value at phase n.

—The value of a q-tag at phase n is obtained by replacing those components that are e-tags (i.e., that are not in Λ) by their values at phase n. It is undefined if any of these components have no value at phase n.

We sometimes abbreviate "value at phase n" as *n-value*

Here (in the first clause, when $n = 0$) and below, values at phase -1 are to be understood as always undefined. Thus, the only e-tags that have values at phase 0 are those that contain no ρ, i.e., those of nesting level 0. The following corollary gives the analogous fact for higher phases.

COROLLARY 4.5. *If a tag has a value at phase n then its nesting level is at most n, and any subtag of depth k has a value at phase $n-k$.*

Proof The second conclusion is proved by a routine induction on tags. The first follows, because, if the nesting level were greater than n, i.e., if there were a subtag of depth $> n$, then, by the second conclusion, this subtag would have a value at a negative phase, which is absurd. □

Remark 4.6. Observe that the answer function α enters the definition of the n-values of tags only via the well-founded subfunction α^n. In particular, when dealing with n-values of tags, we may consider α, α^n, α^m for any $m \geq n$, and the well-founded part α^∞ interchangeably; see Corollary 2.17. In particular, we can safely confine our attention to well-founded answer functions.

The following lemma makes precise the intuitively evident observation that, if a tag has no value at a certain phase, then this undefinedness is ultimately caused by some q-tag having a value that is outside the domain of the appropriate restriction of α.

LEMMA 4.7. *If, in (X, α), a certain tag has no n-value then there is a sub-q-tag of some depth k that has an $(n-k)$-value outside $Dom(\alpha^{n-k+1})$.*

Proof In the given tag, consider a minimal subtag that has no value at the phase appropriate for its depth, i.e., for some j it has depth j and has no value at phase $n-j$. Such a subtag exists, because the given tag is a candidate with $j = 0$. What can this minimal subtag be?

It cannot be a q-tag, for then all its e-tag components would have the same depth j, so by minimality they would have $(n-j)$-values, but then the q-tag itself would have an $(n-j)$-value.

It cannot be an e-tag of the form $f(t_1, \ldots, t_k)$ for then all the t_i would have the same depth j, so by minimality they would have $(n-j)$-values, but then the e-tag itself would have an $(n-j)$-value.

So it must be an e-tag of the form $\rho(p)$. The subtag p has depth $j+1$, so minimality requires it to have an $(n-j-1)$-value, say q. As $\rho(p)$ has no $(n-j)$-value, it follows that $q \notin \mathrm{Dom}(\alpha^{n-j})$. So we have the conclusion of the lemma, with $k = j+1$. □

We shall also need the rather evident fact that values of tags are preserved by isomorphisms. Recall that we extended isomorphisms of states to act on potential queries by acting on all the non-Λ components of a query. Recall also that i is an isomorphism from (X, α) to (Y, β) if it is an isomorphism from X to Y and $\beta \circ i = i \circ \alpha$.

LEMMA 4.8. *Suppose that i is an isomorphism from (X, α) to (Y, β). Then any tag that has an n-value v in (X, α) has n-value $i(v)$ in (Y, β).*

LEMMA 4.9. *If, in (X, α), a tag has n-value v, then it also has m-value v for all $m \geq n$.*

LEMMA 4.10. *If (X, α) matches (X', α') up to phase n, then any tag that has an n-value in one of (X, α) and (X', α') has the same n-value in the other.*

All three of the preceding lemmas are proved by a routine induction on tags.

4.3 Tags suffice for queries and updates

In this subsection, we prove two lemmas saying that all of the queries issued and all elements of updates performed, in a given state X under an answer function α, are named by tags.

LEMMA 4.11. *For any (X, α), every member of ${\Gamma_\alpha}^{k+1}$ is the k-value of some q-tag.*

Proof We proceed by induction on k, using as basis the vacuous case $k = -1$. (There are no (-1)-values, but ${\Gamma_\alpha}^0$ is empty.)

Consider an arbitrary $q \in {\Gamma_\alpha}^{k+1}$. By definition of ${\Gamma_\alpha}^{k+1} = \Gamma_\alpha({\Gamma_\alpha}^k)$, we have some $\xi \subseteq \alpha^k$ with $\xi \vdash_X q$. By Lemma 3.6, all components in X of q are critical for α^k. That is, they are obtainable by evaluating terms in W with the variables assigned values from $\mathrm{Range}(\alpha^k)$.

Look first at these values that are assigned to variables. They have the form $\alpha^k(q')$ for certain queries $q' \in \mathrm{Dom}(\alpha^k) \subseteq {\Gamma_\alpha}^k$. By induction hypothesis, these q'

are the $(k-1)$-values of some q-tags p'. So the values $\alpha^k(q')$ that are assigned to variables are the k-values of the e-tags $\rho(p')$.

Next, look at the critical values obtained, from terms t in W, by giving the variables these values. These critical values are the k-values of the e-tags obtained from the terms t by replacing each variable with the corresponding $\rho(p')$. So the critical values that serve as components of q are k-values of some e-tags.

Finally, look at q. It is a tuple, of length at most B, whose components are either members of Λ or these critical values. Replacing each of the critical values v by an e-tag whose k-value is v, we obtain a q-tag whose k-value is q. □

We have the following analogous lemma for updates in place of queries.

LEMMA 4.12. *Suppose α is a context for X, and suppose $\langle f, \mathbf{a}, b\rangle \in \Delta^+(X, \alpha)$. Then b and all components of $\mathbf{a}$ are B-values of e-tags in (X, α).*

Proof By Lemma 3.7, each of the elements x under consideration is critical, i.e., it is the value of some term $t \in W$ when the variables are given certain values $\alpha(q) \in \mathrm{Range}(\alpha)$. Since α is a context, we have

$$\mathrm{Dom}(\alpha) = {\Gamma_\alpha}^\infty = {\Gamma_\alpha}^B.$$

Thus, by the preceding lemma, each of the q's involved here is the $(B-1)$-value of some q-tag p. Let us replace, in the term t, each of its variables by the corresponding $\rho(p)$. ("Corresponding" means that the $(B-1)$-value of p is a q such that $\alpha(q)$ is the value given to that variable in obtaining x from t.) The result is an e-tag whose B-value is x. □

4.4 Similarity

The following definition is intended to capture the amount of resemblance needed between two state-answer-function pairs in order to ensure that they behave the same way up to phase n, as in the informal discussion at the beginning of this section.

Definition 4.13. (X, α) and (X', α') are *n-similar* if, for each $k \le n$ and each pair of e-tags t_1, t_2, if t_1 and t_2 have k-values that are the same[2] in one of (X, α) and (X', α'), then they also have k-values that are the same in the other.

In particular, taking t_1 and t_2 in the definition to be the same e-tag t, we see that similarity requires t to have a k value in both or neither of (X, α) and (X', α').

LEMMA 4.14. *n-similarity is an equivalence relation with finitely many equivalence classes.*

Proof That n-similarity is an equivalence relation is clear from the definition.

According to Corollary 4.5, the only tags that need to be checked in the definition of n-similarity are those of nesting level $\le n$; tags of higher nesting levels won't have k-values, for $k \le n$, in any state and any answer function. Since, by Lemma 4.3,

[2]Note that the meaning of "t_1 and t_2 have values that are the same in (X, α)" differs from the meaning of $\mathrm{Val}(t_1, X, \alpha) = \mathrm{Val}(t_2, X, \alpha)$, in that the latter would be true if both values are undefined, while the former requires the values to be defined.

there are only finitely many tags of any single nesting level, there are only finitely many relevant tags and thus only finitely many similarity classes. □

The next two propositions will play a key role in the construction of an ASM simulating the algorithm A. Intuitively, they tell us that what the algorithm does in phase n — issuing queries or executing updates or failing — is determined by the n-similarity class of the state and answer function. In view of Lemma 4.14, this means that these actions of the algorithm are determined by finitely much information about the current state and answer function. Of course, the actions in question must, for this purpose, be described in a way that remains invariant when (X, α) is replaced by an isomorphic copy; such a replacement won't change the n-similarity class, so it won't change our description of the actions. Tags provide the required, invariant way to describe the algorithm's actions.

PROPOSITION 4.15. *Assume that (X, α) and (X', α') are n-similar. Then for all $k \leq n$ and all q-tags p, if p has a k-value in (X, α) that is in ${\Gamma_\alpha}^{k+1}$, then it also has a k-value in (X', α') that is in ${\Gamma_{\alpha'}}^{k+1}$.*

Proof We proceed by induction on n. Since n-similarity trivially implies $(n-1)$-similarity, the induction hypothesis gives the required conclusion for all $k < n$, so we consider only the case $k = n$. Let p be a q-tag with n-value q in (X, α) such that $q \in {\Gamma_\alpha}^{n+1}$. The assumption of n-similarity ensures that p also has an n-value q' in (X', α'). It remains to prove that $q' \in {\Gamma_{\alpha'}}^{n+1}$. It is here that we need the idea from [Gurevich 2000, Lemma 6.9].

We begin by constructing an isomorphic copy (Y, β) of (X, α) as follows. Assume, without loss of generality, that X is disjoint from X'. (If this is not the case, replace X with an isomorphic copy disjoint from X', replace α with the corresponding context for this isomorphic copy, and work with the copy instead of X in the following.) Obtain Y by replacing the n-value of each e-tag in (X, α) by the n-value of the same tag in (X', α'). The assumption of n-similarity ensures that this replacement is well-defined and one-to-one, so we obtain a state Y with an isomorphism $i : X \cong Y$. Define β to be $i \circ \alpha \circ i^{-1}$, so that $i : (X, \alpha) \cong (Y, \beta)$.

We claim that (Y, β) matches (X', α') up to phase n. Once this claim is proved, the rest of the argument is as follows. By Lemma 2.10, from $q \in {\Gamma_\alpha}^{n+1}$ we get $i(q) \in {\Gamma_\beta}^{n+1}$. By Lemma 3.11 we get that ${\Gamma_\beta}^{n+1} = {\Gamma_{\alpha'}}^{n+1}$. By Lemma 4.8 we get $i(q) = q'$. Putting these facts together, we have $q' \in {\Gamma_{\alpha'}}^{n+1}$, as required.

So it remains to prove the claim, namely that

(1) Y and X' agree over ${\alpha'}^n$, i.e., they give the same value to any term from W when its variables are assigned values in $\mathrm{Range}({\alpha'}^n)$, and
(2) ${\alpha'}^n = \beta^n$.

Let us therefore begin by considering ${\alpha'}^n$. Its domain ${\Gamma_{\alpha'}}^n$ consists, by Lemma 4.11, of $(n-1)$-values v in (X', α') of certain q-tags p. In view of the definition of Y and i, which replaced n-values in (X, α) of e-tags by their n-values in (X', α'), and in view of the fact that any $(n-1)$-value is also an n-value, we have that these elements v can also be described as the i-images of the $(n-1)$-values in (X, α) of the same tags p. But, since i is an isomorphism, these are also the $(n-1)$-values in (Y, β) of the same tags p. Thus, each element $v \in \mathrm{Dom}({\alpha'}^n)$ is the $(n-1)$-value in (Y, β) of the corresponding q-tag p.

Repeating the argument with $\rho(p)$ in place of p, we find that ${\alpha'}^n(v)$, the n-value of $\rho(p)$ in (X', α'), is also the n-value of $\rho(p)$ in (Y, β), i.e., it is $\beta^n(v)$. This establishes that ${\alpha'}^n \subseteq \beta^n$. Symmetrically, we have $\beta^n \subseteq {\alpha'}^n$. This completes the proof of part (2) of the claim.

For part (1), consider any term $t \in W$ and any assignment of values from $\mathrm{Range}({\alpha'}^n)$ to its variables. Obtain an e-tag u by replacing in t each variable by an e-tag $\rho(p)$ as above corresponding to the value assigned to that variable. So the value that t gets, with these values for the variables, is the n-value of u, whether in (X', α') or in (Y, β). Repeating again the argument from two paragraphs ago, this time with u in place of p, we find that the values in (X', α') and (Y, β) are the same. This proves (1), hence the claim, and hence the proposition. □

COROLLARY 4.16. *Suppose α and α' are well-founded answer functions for X and X' respectively, and suppose that (X, α) and (X', α') are B-similar. If α is a context for X, then α' is a context for X'.*

Proof We must show that ${\Gamma_{\alpha'}}^\infty \subseteq \mathrm{Dom}(\alpha')$. Consider, therefore, an arbitrary $q' \in {\Gamma_{\alpha'}}^\infty = {\Gamma_{\alpha'}}^B$. By Lemma 4.11, q is the $(B-1)$-value of some q-tag p. By Proposition 4.15, p also has a $(B-1)$-value q in (X, α), and $q \in {\Gamma_\alpha}^B$. As α is a context, $q \in \mathrm{Dom}(\alpha)$. Thus, the e-tag $\rho(p)$ has a B-value in (X, α). Our assumption of B-similarity implies that $\rho(p)$ also has a B-value in (X', α'). This means that $q' \in \mathrm{Dom}(\alpha')$, as required. □

The following proposition does for updates and failures what the previous one did for queries.

PROPOSITION 4.17. *Assume that (X, α) and (X', α') are B-similar, and assume that α and (therefore) α' are contexts.*

—*Suppose that $\langle f, \mathbf{a}, b\rangle \in \Delta^+(X, \alpha)$, where the a_j and b are the B-values in (X, α) of e-tags v_j and w. Then in (X', α') these tags have B-values a'_j and b' such that $\langle f, \mathbf{a}', b'\rangle \in \Delta^+(X', \alpha')$.*

—*If the algorithm fails in (X, α), then it also fails in (X', α').*

Proof We prove the first assertion; the proof of the second is similar but easier as one can omit all considerations of a_j, a'_j, b, and b'.

The existence of the B-values a'_j and b' is given by the hypothesis of B-similarity. What must be proved is that $\langle f, \mathbf{a}', b'\rangle \in \Delta^+(X', \alpha')$.

Exactly as in the proof of Proposition 4.15, with B in place of n, produce an isomorphism $i : (X, \alpha) \cong (Y, \beta)$ such that (Y, β) matches (X', α') up to phase B. Define $a'_j = i(a_j)$ and $b' = i(b)$. By Lemma 4.8, each a'_j is the B-value of u_j and b is the B-value of v in (Y, β). By Lemma 4.10, the same holds in (X', α'). By the Isomorphism Postulate, $\langle f, \mathbf{a}', b'\rangle \in \Delta^+(Y, \beta)$. By Corollary 3.12, $\langle f, \mathbf{a}', b'\rangle \in \Delta^+(X', \alpha')$. □

5. CONSTRUCTING THE EQUIVALENT ASM

In this section, we describe the ASM that will be equivalent to the given algorithm A. We first give an informal description of how the ASM is intended to function. Then we set up the necessary notation and finally describe the actual ASM program.

5.1 Informal discussion

Proposition 4.15 tells us that the collection of q-tags p whose n-values are in ${\Gamma_\alpha}^{n+1}$ depends not on all the details of the state X and answer function α but only on the n-similarity class of (X, α). Similarly, Proposition 4.17 tells us that failures and updates are determined by the B-similarity class of (X, α) when α is a context. Lemma 4.14 tells us that there are only finitely many n-similarity classes for $n \leq B$. These facts provide the following, fairly simple description of the operation of the algorithm. In the description, we shall occasionally refer to the state X and the answer function α, but the reader should observe that, in the discussion up to and including phase n, we use only the information about (X, α) that is provided by its n-similarity class.

In phase 0, no information from the environment is yet available. So the only e-tags that can be evaluated are those without any ρ, i.e., those of nesting level 0. The algorithm evaluates these finitely many e-tags and checks which of the values are equal. The equalities and inequalities so found are clearly determined by the 0-similarity class of (X, α).

Furthermore, they determine the 0-similarity class. Recall that a 0-similarity class specifies not only which equalities between 0-values of e-tags hold but also which e-tags have 0-values at all. But this additional information, which e-tags have 0-values, is easily available, since tags of nesting level 0 always have values and tags of higher nesting level cannot have 0-values.

Having computed (the information needed to determine) the 0-similarity class of (X, α), the algorithm knows, via Proposition 4.15, what queries to issue. More precisely, it knows which q-tags p have 0-values that are in ${\Gamma_\alpha}^1$. Exactly how it finds these tags is irrelevant to our discussion since it will not affect the queries issued, the updates performed, or failures. But, since there are only finitely many 0-similarity classes and each produces only finitely many p's, we may imagine that the algorithm has a table in which it can look up the p's after it has calculated the 0-similarity class.

Having obtained the appropriate q-tags p and having computed the 0-values of all e-tags of nesting level 0, the algorithm can produce the 0-values of these q-tags p. This is because each p is also at nesting level 0 (because it has a 0-value) and is therefore a tuple of members of Λ and e-tags of nesting level 0. So the algorithm can compute and issue the queries in ${\Gamma_\alpha}^1$. This completes phase 0.

The answer function α provides answers for some subset of these queries, in the form of $\alpha \restriction {\Gamma_\alpha}^1 = \alpha^1$. Given these answers, the algorithm can proceed to phase 1 of its computation. It knows which e-tags have 1-values, namely precisely those such that, in every subtag of the form $\rho(p)$, p has the same 0-value as one of the q-tags used as queries in phase 0 and answered in α^1. (Remember that the algorithm has already computed which e-tags have the same 0-value, so it can easily check which q-tags have the same 0-value.) So it can evaluate these e-tags and determine which of them have equal 1-values. This, together with the information that the other e-tags have no 1-values, suffices to determine the 1-similarity class of (X, α). Just as in phase 0, the algorithm now has enough information to determine, perhaps by table look-up, the q-tags whose 1-values are in ${\Gamma_\alpha}^2$. As before, it computes the 1-values of these q-tags and issues the resulting queries, except for those that were

already issued at phase 0.

Again, the answer function provides, in the form of α^2, the answers to some subset of these queries. With this information, the algorithm begins phase 2. Again, it knows which e-tags have 2-values, namely those whose q-subtags of depth 1 have the same 1-values as the q-tags used as queries in phase 1 or phase 0 and answered by α^2. From here on, the pattern of phase 1 simply repeats for phase 2 and the later phases, until phase B is reached, where there are no new queries to issue and it is time to either perform the updates leading to the next state or fail.

In this final phase, the algorithm again knows already which e-tags will have B-values. It computes and compares these values to determine the B-similarity class of (X, α). According to Proposition 4.17, this information determines whether there is a failure or an update set and, in the latter case, which e-tags u_j and v have B-values used in the updates to be performed. The algorithm finds those tags (again perhaps by table-look-up), evaluates them, and performs the resulting updates.

The results of the preceding section show that this description matches the given algorithm, with any state and answer function, insofar as issuing queries and failing or executing updates are concerned. In the remainder of this section, we convert this description into an ASM, which will be equivalent to the given algorithm.

5.2 The ASM: easy part

To prove Theorem 1.1, we must exhibit an ASM equivalent to a given algorithm. This means that we must describe a set of states, a set of initial states, a vocabulary, a set of labels, a template assignment, and a program.

Part of the description is trivial, as we must define the ASM to have the same set $\mathcal{S}$ of states, the same set $\mathcal{I}$ of initial states, (therefore) the same vocabulary Υ, and the same set Λ of labels as the given A. It remains to describe the template assignment and (most importantly) the program Π of the ASM.

The template assignment is fairly easy to describe. Consider all the q-tags p whose nesting level is at most B; recall that there are only finitely many of these. Convert each one into a template by leaving its components from Λ unchanged but replacing its e-tag components by the placeholders $\#i$ in order, from left to right. It is possible that several p's yield the same template, but we ignore such multiplicities and consider simply the set of templates so obtained. Fix a one-to-one correspondence between this set of templates and some set of symbols E, disjoint from $\Upsilon \sqcup \Lambda$, which will serve as the external function symbols for our ASM. If a template is for k-ary functions (i.e., if the placeholders in it are $\#1, \ldots, \#k$), then the corresponding external function symbol in E is to be k-ary. This one-to-one correspondence will serve as our template assignment. That is, for any $f \in E$, the template $\hat{f}$ will be the template that contributed f to E.

All that remains is to describe the program Π, but of course this is the hard part of our task, and it will occupy the remainder of this section.

5.3 Terms for tags

The program Π will involve terms whose purpose is to denote the values of e-tags or the replies to the potential queries that are values of q-tags. We introduce a convenient notation for these terms, by recursion on tags.

Definition 5.1. We define, for each tag t of nesting level $\leq B$, a term t^* as follows.

—If t is an e-tag of the form $\rho(p)$, then $t^* = p^*$.

—If t is an e-tag of the form $f(t_1, \ldots, t_n)$, then $t^* = f(t_1^*, \ldots, t_n^*)$.

—If p is a q-tag, then let f be the external function symbol associated (in our description of the template assignment) to the template arising from p, and let the e-tag components of p be (in left-to-right order) $t_1, \ldots, t_k$. Then $p^* = f(t_1^*, \ldots, t_k^*)$.

Like the definition of tags, this recursion has as its basis the cases of e-tags that contain no ρ, i.e., e-tags of nesting level 0, and q-tags consisting entirely of labels from Λ. Note that, for e-tags t of nesting level 0, we have $t^* = t$.

The following lemma says that tags and their associated terms have the same semantics.

LEMMA 5.2. *Let X be a state, α an answer function for it, and k a natural number.*

—*For e-tags t of nesting level $\leq k$, the k-value of t in (X, α) equals the value of t^* in (X, α).*

—*If p is a q-tag of nesting level $\leq k$, let f be the external function symbol corresponding to the template obtained from p (as in the definition of our template assignment) — so p is $\hat{f}$ with the placeholders $\#i$ replaced by some tags t_i. Then the k-value of p in (X, α) is the query $\hat{f}[\mathbf{a}]$, where the a_i are the values of the t_i^* in (X, α). The reply to this query in α is the value of p^* in (X, α).*

Moreover, the values of terms in (X, α) mentioned here are the same as the values of those terms in (X, α^k).

Here, as usual, the assertion that two things are equal is to be understood as implying, in particular, that if either of them is defined then so is the other.

Proof We proceed by induction on tags, for all k simultaneously. There are three cases, depending on the type of tag under consideration. The state X and answer function α remain fixed throughout the proof, so we sometimes omit mention of them.

Suppose first that t is an e-tag of the form $\rho(p)$, where p is a q-tag of nesting level $\leq k-1$ (since t has nesting level $\leq k$). Then the k-value of t, in either (X, α) or (X, α^k), is obtained by applying α^k to the $(k-1)$-value of p. By induction hypothesis, this is the value of p^*. Since $t^* = p^*$ in this situation, the proof is complete in this first case.

Suppose next that t is an e-tag of the form $f(\mathbf{t}')$ for some $f \in \Upsilon$ and some tuple $\mathbf{t}'$ of e-tags. The desired conclusion follows from the induction hypothesis and the observation that the recursion clause defining values for e-tags of this form exactly matches the recursion clause defining values of terms of the form $f(\mathbf{t}')$.

Finally, suppose p is a q-tag, and use the notation f, $\mathbf{t}$, and $\mathbf{a}$ from the statement of the lemma. Since p is the template $\hat{f}$ with the placeholders $\#i$ replaced by t_i, the k-value of p is, by definition, the template $\hat{f}$ with each $\#i$ replaced by the value of t_i, namely a_i. The induction hypothesis lets us use t_i^* in place of t_i here. That confirms the first of the lemma's assertions for this case. Since p^* is $f(\mathbf{t}^*)$,

the second assertion follows by the definition of values of terms that begin with external functions. □

5.4 Partial equivalence relations and their descriptions

The following (moderately standard) terminology will be convenient for dealing with n-similarity.

Definition 5.3. A *partial equivalence relation* or *per* on a set S is an equivalence relation $\sim$ on a subset of S, called the *field* of $\sim$. One such per, $\sim$ with field F, is a *restriction* of another, $\sim'$ with field F', and $\sim'$ is an *extension* of $\sim$, if $F \subseteq F'$ and, for all $x, y \in F$, $x \sim y \iff x \sim' y$.

Definition 5.4. For any state X, answer function α, and natural number n, define $t \sim_{X,\alpha,n} t'$ to mean "t and t' are e-tags having n-values that are equal".

LEMMA 5.5. *The relation $\sim_{X,\alpha,n}$ is a per on the set of e-tags. Its field is the set of e-tags that have n-values. If $n \leq m$, then $\sim_{X,\alpha,n}$ is a restriction of $\sim_{X,\alpha,m}$. Two pairs, (X,α) and (X',α'), are n-similar if and only if the associated pers $\sim_{X,\alpha,k}$ and $\sim_{X',\alpha',k}$ coincide for all $k \leq n$.*

Proof The first two sentences are obvious, the third follows from Lemma 4.9, and the fourth just restates the definition of n-similarity. □

According to this lemma, the n-similarity class of any (X,α) can be completely described by a tower of pers, $\langle \sim_{X,\alpha,k} \rangle_{k \leq n}$, on the set of e-tags. We shall be interested in n-similarity only for $n \leq B$, so we may regard these pers as being on the finite set of e-tags of nesting level $\leq B$, because tags of higher nesting level will not have n-values when $n \leq B$.

Definition 5.6. A per on e-tags of nesting level $\leq B$ is *n-realizable* if it is $\sim_{X,\alpha,n}$ for some state X and answer function α.

Remark 5.7. In view of Remark 4.6, replacing an answer function α by its well-founded part α^∞ or even by the smaller function α^n will not change the per $\sim_{X,\alpha,n}$. In particular, in the definition of n-realizability, we can safely require α to be well-founded.

We need a technical lemma, allowing us to adjust the realizers of pers in certain circumstances.

LEMMA 5.8. *Suppose $\sim_{X,\alpha,n}$ is a restriction of $\sim_{X',\alpha',n}$. Then there is a sub-function β of α' such that $\sim_{X,\alpha,n}$ coincides with $\sim_{X',\beta,n}$.*

Proof We simply shrink α' to β by removing from its domain those queries which, according to $\sim_{X,\alpha,n}$, ought not to have replies. More precisely, whenever a q-tag p has $(n-1)$-values in both (X,α) and (X',α'), say q and q' respectively, and $\rho(p)$ has an n-value in (X',α') but not in (X,α) (so $q' \in \mathrm{Dom}(\alpha')$ but $q \notin \mathrm{Dom}(\alpha)$), then delete q' from the domain of α'.

What e-tags t have n-values in (X',α') but lose those values in (X',β) as a result of these deletions? According to Lemma 4.7, t must have a sub-q-tag p whose value q' (at an appropriate phase) was removed from the domain of α' in forming β. Then

$\rho(p)$ is a sub-e-tag of t. Our definition of the deletions used in producing β implies that $\rho(p)$ and therefore t had no n-value in (X, α). So t's loss of its value in passing from α' to β is correct; it results in agreement with $\sim_{X,\alpha,n}$.

A similar argument, again using Lemma 4.7, shows that, conversely, all e-tags t that have values in (X', α') but not in (X, α) lose those values as a result of the deletions leading to β. □

Using the terms associated to tags, we can write formulas describing, to some extent, pers on the set of e-tags (of nesting level $\leq B$, as usual).

Definition 5.9. Let $\sim$ be a per on the set of e-tags. Its *description* $\delta(\sim)$ is the conjunction of all the Boolean terms $t_1^* = t_2^*$ for e-tags such that $t_1 \sim t_2$ and all the Boolean terms $\neg(t_1^* = t_2^*)$ for e-tags in the field of $\sim$ such that $t_1 \not\sim t_2$.

Notice that tags not in the field of $\sim$ don't contribute to $\delta(\sim)$.

LEMMA 5.10. *Let X be a state and α an answer function for it. Let $\sim$ be a per on the set of e-tags. Then the truth value of $\delta(\sim)$ in (X, α) is:*

—`true` *if $\sim$ is a restriction of $\sim_{X,\alpha,B}$,*
—`false` *if the field of $\sim$ is included in that of $\sim_{X,\alpha,B}$ but $\sim$ is not a restriction of $\sim_{X,\alpha,B}$,*
—*undefined if the field of $\sim$ is not included in that of $\sim_{X,\alpha,B}$.*

Proof We prove the last part first. Suppose t is a tag that is in the field of $\sim$ but not in that of $\sim_{X,\alpha,B}$. The latter means that t has no B-value. By Lemma 5.2, t^* has no value in (X, α). But $\delta(\sim)$ includes a conjunct $t^* = t^*$; so $\delta(\sim)$ also has no value.

From now on, we assume that the field of $\sim$ is included in that of $\sim_{X,\alpha,B}$. That is, every t in the field of $\sim$ has a B-value in (X, α), and so, by Lemma 5.2, t^* has a value in (X, α). Since $\delta(\sim)$ is a conjunction of equations and negated equations built from just these terms t^*, it has a value in (X, α).

It remains to determine when this value, necessarily Boolean, is `true`. There are two requirements for that. First, whenever $t_1 \sim t_2$, we must have that $t_1^* = t_2^*$ is true in (X, α). By Lemma 5.2, this means that the B-values of t_1 and t_2 agree, i.e., that $t_1 \sim_{X,\alpha,B} t_2$. The second requirement is that whenever t_1 and t_2 are in the field of $\sim$ but $t_1 \not\sim t_2$, then $\neg(t_1^* = t_2^*)$ is true. Such t_1 and t_2 are in the field of $\sim_{X,\alpha,B}$, and, arguing as above using Lemma 5.2, we need that their B-values are distinct, i.e., that $t_1 \not\sim_{X,\alpha,B} t_2$. The two requirements together say exactly that $\sim$ is a restriction of $\sim_{X,\alpha,B}$, as desired. □

5.5 The ASM program

We are now ready to describe the program Π, in a top-down manner.

To begin, consider the set T_0 of e-tags t of nesting level 0. Since they don't involve ρ and q-tags, they have 0-values in all (X, α), and these values depend only on X, not on α. Consider all the equivalence relations $\sim$ on this set T_0 of tags; these can be regarded as pers on the set of all e-tags. Every state X will make exactly one of their descriptions $\delta(\sim)$ true, namely the one whose $\sim$ agrees with the equivalence relation $\sim_{X,\alpha,0}$ on e-tags defined by equality of their 0-values in

X. This follows from Lemma 5.10, since there is no danger of undefined values for tags of nesting level 0. Notice that two states satisfy the same $\delta(\sim)$ if and only if they are 0-similar. (Here and below, we slightly abuse notation by saying X and X' are 0-similar to mean that (X, α) and (X', α') are 0-similar for any answer functions α and α'. The point is that for 0-similarity, in contrast to k-similarity for larger k, the answer functions are irrelevant.) We say that a 0-realizable $\sim$ and also its description $\delta(\sim)$ *describe* the 0-similarity class consisting of those X for which $\sim = \sim_{X,\alpha,0}$ (which is independent of α).

We construct the desired program Π as the parallel combination of components, which we call the components for *phase* 0. Each of these components is a conditional rule of the form

$$\texttt{if } \delta(\sim) \texttt{ then } R_\sim \texttt{ endif},$$

and there is one such rule for each equivalence relation $\sim$ on the e-tags of nesting level 0 that is, as a per, 0-realizable. We shall refer to the pers $\sim$, the descriptions $\delta(\sim)$, and the rules $R_\sim$ as the pers, guards, and true branches at phase 0. (Recall that conditional rules with no explicit `else` clause were defined as syntactic sugar for rules containing `else skip`; so we could speak also of "false branches" at phase 0, which would be simply `skip`.) To complete the definition of Π, it remains to say what the true branches $R_\sim$ are that go with these equivalence relations. **Until further notice, we fix one of these pers**, $\sim_0$, and we work toward describing the corresponding rule $R_{\sim_0}$.

Because of the definition of $\delta(\sim_0)$, the rule $R_{\sim_0}$ that we intend to define will be executed only in those states X that belong to one specific 0-similarity class, namely that described by $\sim_0$. By Proposition 4.15, all these states agree as to which q-tags p will have 0-values in ${\Gamma_\alpha}^1$. (In terms of our informal descriptions, they agree as to what queries are issued in phase 0. Note that α, though needed in the notation ${\Gamma_\alpha}^1$, is irrelevant at this phase, since only $\alpha^0 = \varnothing$ plays a role, so far.) Call such p *relevant* (for the fixed $\sim_0$ under consideration).

Let T_1 be the set of all e-tags of nesting level ≤ 1 in which all subtags of the form $\rho(p)$ have p relevant. These are the e-tags that would have 1-values if α provided answers to all the queries in ${\Gamma_\alpha}^1$. In particular, these tags will have 1-values if α is a context, but not necessarily when α is an arbitrary answer function, not even when it is well-founded.

We define $R_{\sim_0}$ to be a parallel block, with one component for each extension of $\sim_0$ to a 1-realizable per $\sim$ whose field is a subset of T_1. We call these the components for *phase 1*. The component associated to such a per $\sim$ has the form

$$\texttt{if } \delta(\sim) \texttt{ then } R_\sim \texttt{ endif},$$

where we must still specify the rule $R_\sim$. We refer to these $\sim$, $\delta(\sim)$, and $R_\sim$ as pers, guards, and true branches at phase 1. **Until further notice, we fix one of these pers**, $\sim_1$, and we work toward describing the corresponding rule $R_{\sim_1}$.

Our present task, defining the true branches at phase 1, is analogous to our task (still pending) of describing the true branches at phase 0. There are, however, differences that we must be careful about. When we dealt with pers $\sim$ whose field consisted only of the e-tags of nesting level 0, there was no possibility of their 0-values being undefined. In particular, $\delta(\sim)$ always had a truth value. Furthermore,

this value was `true` in exactly one 0-similarity class of states (or, more precisely, of pairs (X, α)). Now, however, when we deal with pers $\sim$ whose field also contains some tags of nesting level 1, their 1-values may be undefined in some (X, α) (if the answer function α is not a context for the state X). As a result, the value of $\delta(\sim)$ may be undefined. In such a situation, the presence of the guard $\delta(\sim)$ in our program Π will cause the execution of Π to hang. Fortunately, this is the correct behavior, because this situation arises exactly when the algorithm A has issued a query that α doesn't answer; thus α is not a context and the execution of A also hangs.

Furthermore, the same (X, α) may satisfy several of our guards at phase 1, not just a single guard as at phase 0. Lemma 5.10 tells us which guards $\delta(\sim)$ are satisfied by which (X, α). Thus, the true branch $R_{\sim_1}$ currently under consideration will be executed in several 1-similarity classes of (X, α)'s, not only the class described by $\sim_1$ but also the classes described by its realizable extensions. Nevertheless, we shall design $R_{\sim_1}$ to work properly in the 1-similarity class described by $\sim_1$. That it causes no trouble when executed in the other 1-similarity classes described by extensions of $\sim_1$ will have to be verified as part of the proof that our Π is equivalent to A.

Except for the differences just outlined, what we are about to do for $\sim_1$ will be just like what we did above for $\sim_0$.

Let us, therefore, consider pairs (X, α) in the 1-similarity class described by $\sim_1$. By Proposition 4.15, all these (X, α) agree as to which q-tags p will have 1-values in ${\Gamma_\alpha}^2$. (In terms of our informal descriptions, they agree as to what queries are issued in phase 1.) Call such p *relevant* (for the fixed $\sim_1$ under consideration).

Let T_2 be the set of all e-tags of nesting level ≤ 2 in which all subtags of the form $\rho(p)$ have p relevant. These are the e-tags that would have 2-values if α provided answers to all the queries in ${\Gamma_\alpha}^2$. In particular, these tags will have 2-values if α is a context.

We define $R_{\sim_1}$ to be a parallel block, with one component for each extension of $\sim_1$ to a 2-realizable per $\sim$ whose field is a subset of T_2. The component (a component at *phase 2*) associated to such a per $\sim$ has the form

$$\texttt{if } \delta(\sim) \texttt{ then } R_\sim \texttt{ endif},$$

where we must still specify the true branch $R_\sim$.

This specification follows the pattern begun above. For any particular $\sim_2$ (a 2-realizable extension of $\sim_1$), we let T_3 be the set of e-tags of nesting level ≤ 3 all of whose subtags of the form $\rho(p)$ have p among the q-tags with 2-values in ${\Gamma_\alpha}^3$ whenever (X, α) is in the 2-similarity class described by $\sim_2$. Then we let $R_{\sim_2}$ be a parallel block of conditional rules (components at *phase* 3), with one block guarded by each $\delta(\sim)$, where $\sim$ ranges over 2-realizable extensions of $\sim_2$ with field included in T_3.

Continue in this manner for B steps. At this point, there are no new relevant q-tags. We need to define the true branches $R_{\sim_{B-1}}$ at phase $B-1$, where $\sim_{B-1}$ describes a certain $(B-1)$-similarity class of pairs (X, α). (In terms of our informal discussion earlier, we are considering phase B, where each (X, α) has finished issuing queries and receiving answers, and is ready to either fail or update its state.)

As before, $R_{\sim_{B-1}}$ is a parallel block of conditional rules

$$\texttt{if } \delta(\sim) \texttt{ then } R_\sim \texttt{ endif},$$

where $\sim$ ranges over extensions of $\sim_{B-1}$ to B-realizable pers whose field is a subset of T_B. Here, just as before, T_B consists of the e-tags of nesting level $\leq B$ all of whose subtags of the form $\rho(p)$ have p among the q-tags with $(B-1)$-values in ${\Gamma_\alpha}^B$ whenever (X,α) is in the $(B-1)$-similarity class described by $\sim_{B-1}$. But this time the corresponding subrules, $R_\sim$, are different and indeed correspond to a different aspect of the algorithm. Instead of issuing queries while evaluating guards $\delta(\sim)$, they will either fail or perform updates.

Consider (X,α) in the B-similarity class described by $\sim$. If α is not a context for X, then, by Corollary 4.16, there is no pair (X',α') in the same B-similarity class where α' is a context for X'. In this situation, define $R_\sim$ to be `skip`.

Now consider the case that α is a context for X for (one and therefore all) (X,α) in the B-similarity class described by $\sim$. In this case, Proposition 4.17 applies. In particular, if the algorithm A fails in one such (X,α) then it fails in them all. In this situation, we let $R_\sim$ be `Fail`.

Finally, when A doesn't fail, Proposition 4.17 also tells us that, for any dynamic function symbol f, all (X,α) in our B-similarity class agree about the e-tags $\mathbf{v}$ and w for whose B-values $\mathbf{a}$ and b the update $\langle f,\mathbf{a},b\rangle$ belongs to $\Delta^+(X,\alpha)$. (The elements $\mathbf{a}$ and b will be different in different states, but the tags $\mathbf{v}$ and w will be the same.) In this situation, we define $R_\sim$ to be a parallel block whose components, the components at *phase* B, are the update rules $f(\mathbf{v}^*) := w^*$. This completes the definition of the ASM program Π.

6. PROOF OF EQUIVALENCE

In the preceding section, we have constructed from an arbitrary algorithm A an ASM which we shall call simply Π (even though technically it consists of the program Π together with the template assignment, the set of labels, the set of states, and the set of initial states). The present subsection is devoted to the proof that Π is equivalent to the given algorithm A. Referring to Definition 2.22 of equivalence, we see that the first two requirements, namely the agreement of states, initial states, vocabulary, and labels, are immediate consequences of the definition of the ASM. It remains therefore to verify the last three requirements, agreement of causality relations (up to equivalence), failures, and updates.

We need a preliminary lemma, saying roughly that the tree-like structure of Π is rich enough to contain branches executed in any state with any answer function.

LEMMA 6.1. *Let X be a state, α an answer function for it, and, for each k, $\sim_k = \sim_{X,\alpha,k}$ the per describing the k-similarity class of (X,α). The program Π contains a nested sequence of (occurrences of) components, one for each phase, such that, for each k, the phase k component in the sequence has guard $\delta(\sim_k)$.*

Proof Since every equivalence relation on the e-tags of nesting level 0 that is 0-realizable as a per gives rise to a phase 0 component, and since $\sim_0$ is such an equivalence relation, we have the required component at phase 0. Proceeding inductively, for the step from k to $k+1$, suppose we have obtained the desired sequence

of components through phase k. It ends with (an occurrence of)

$$\texttt{if } \delta(\sim_k) \texttt{ then } R_{\sim_k} \texttt{ endif}.$$

We need only check that $\sim_{k+1}$ is one of the pers that provide the components (at phase $k+1$) in the parallel block $R_{\sim_k}$, for then the component it provides will serve as the next term of the required sequence. Looking at the definition of Π, we find that we must check the following.

(1) $\sim_{k+1}$ is an extension of $\sim_k$.
(2) $\sim_{k+1}$ is $(k+1)$-realizable.
(3) The field of $\sim_{k+1}$ is a subset of T_{k+1}.

The first two of these are obvious in view of the definition of $\sim_k$ in the statement of the lemma. To prove the third, recall that, by definition of T_{k+1}, what we must show is that each element in the field of $\sim_{k+1}$ is an e-tag of nesting level $\leq k+1$ such that all subtags of the form $\rho(p)$ have p relevant for $\sim_k$. Relevance of p was defined in terms of any state and answer function in the k-similarity class described by $\sim_k$. In our present situation, there is an obvious representative of this k-similarity class, namely the given (X, α). So relevance of p means that its k-value is in ${\Gamma_\alpha}^{k+1}$. Now if t is in the field of $\sim_{k+1}$, then, by definition of this per as $\sim_{X,\alpha,k+1}$, t must be an e-tag with a $(k+1)$-value in (X, α), so in particular its nesting level must be $\leq k+1$ (see Corollary 4.5). Furthermore, each subtag of the form $\rho(p)$ must have a $(k+1)$-value, and so p must have a k-value in $\mathrm{Dom}(\alpha^{k+1}) \subseteq {\Gamma_\alpha}^{k+1}$. This completes the verification of item (3) and thus the proof of the lemma. □

To prove the equivalence of A and Π, we begin by considering their respective causality relations $\vdash_X^A$ and $\vdash_X^\Pi$. According to Lemma 2.21, we must show that, for every state X and every answer function α that is well-founded for both $\vdash_X^A$ and $\vdash_X^\Pi$, the same queries are caused, under these two causality relations, by subfunctions of α. For this purpose, it will be useful to invoke the well-foundedness of α and [Blass and Gurevich 2006, Proposition 6.19] to replace, in the case of $\vdash_X^A$, the phrase "caused by a subfunction of α" with "reachable under α." (The replacement would be equally legitimate in the case of $\vdash_X^\Pi$, but it will not be useful there.) Thus, our immediate goal is to prove the following.

LEMMA 6.2. *Let X be a state (for A and Π) and let α be an answer function for X that is well-founded with respect to both A and Π. Then, for any potential query q for X, the following two statements are equivalent.*

—*There is a subfunction ξ of α such that $\xi \vdash_X^\Pi q$.*
—$q \in {\Gamma_{A,\alpha}}^\infty$.

We have used the notation $\Gamma_{A,\alpha}$ to distinguish it from $\Gamma_{\Pi,\alpha}$, but from now on we shall write simply Γ_α because there will be no need to refer to $\Gamma_{\Pi,\alpha}$.

Proof of Lemma Fix X, α, and q as in the hypotheses of the lemma. We consider how an answer function $\xi \subseteq \alpha$ could cause q with respect to Π. Inspection of the definitions of the causality relations for ASMs in [Blass and Gurevich to appear, Section 5] reveals that any instance of causality, like $\xi \vdash_X^\Pi q$, originates in either an output rule or a term that begins with an external function symbol. For all other

rules and terms, the queries that ξ causes are simply copied from the causality relations of subterms or subrules. Since our Π contains no output rules, all the causality instances that we must analyze originate from subterms that begin with external function symbols.

Such terms occur in two places in Π, namely the guards of the components at various phases and the update rules at the final phase. Let us consider first the terms occurring in the update rules. Recall that the update subrules of Π occur only in the true branches at phase B, whose guards $\delta(\sim)$ describe B-similarity classes (Y, β) where β is a context for Y. (We changed the notation to (Y, β) here because we have already fixed particular X and α.) The update rules are of the form $f(\mathbf{v}^*) := w^*$ where e-tags $\mathbf{v}$ and w have B-values $\mathbf{a}$ and b such that $\langle f, \mathbf{a}, b\rangle \in \Delta^+(Y, \beta)$. For these B-values to exist at all, it is necessary that the q-tags p whose $\rho(p)$ occur in $\mathbf{v}$ and w have $(B-1)$-values in $\mathrm{Dom}(\beta) = {\Gamma_\beta}^B$. Thus, $\mathbf{v}$ and w are members of T_B, eligible to be in the field of a per at phase B. The per $\sim$ that describes (Y, β) will have these tags in its field, precisely because the tags have B-values in (Y, β). Thus, for each component v_i of the tuple $\mathbf{v}$, the equation $v_i = v_i$ is among the conjuncts in $\delta(\sim)$, and so is $w = w$. As a result, any queries originating in the update rules in $R_\sim$ also originated in the guard $\delta(\sim)$. This means that, in analyzing the instances $\xi \vdash^\Pi_X q$, we may confine our attention to instances originating in the guards at various phases.

How could the causality instance $\xi \vdash^\Pi_X q$ originate from a particular guard, say $\delta(\sim)$ at some phase k? In view of the definition of the description $\delta(\sim)$, our instance would have to originate in a term t^* where t is an e-tag in the field of the phase k per $\sim$. That field is, by definition, included in T_k, so the subtags of t of the form $\rho(p)$ all have p relevant with respect to the per $\sim_{k-1}$ within whose true branch our $\delta(\sim)$ is located. Inspection of the definition of t^* shows that the queries originating there in fact originate from subterms $\rho(p)$ (as these are the only source of external function symbols in t^*) and have the form described in the second part of Lemma 5.2 (with k changed to $k-1$). Thus, q is the $(k-1)$-value of one of these p's.

Furthermore, for this guard $\delta(\sim)$ to contribute any queries at all in (X, ξ), the governing guard from the previous phase, $\delta(\sim_{k-1})$, had to get the value `true` in (X, ξ) and therefore in (X, α). According to Lemma 5.10, $\sim_{k-1}$ must be a restriction of $\sim_{X,\alpha,B}$. Since $\sim_{k-1}$ is $(k-1)$-realizable (as only realizable pers were used in our construction of Π), Lemma 5.8 shows that α has a subfunction γ such that $\sim_{k-1} = \sim_{X,\gamma,k-1}$. Since the q-tag p is relevant with respect to $\sim_{k-1}$, its $(k-1)$-value q is in ${\Gamma_\gamma}^k \subseteq {\Gamma_\alpha}^k \subseteq {\Gamma_\alpha}^\infty$. This establishes the implication from the first to the second of the allegedly equivalent statements in the lemma.

For the converse, consider any query $q \in {\Gamma_\alpha}^\infty$; fix some k such that $q \in {\Gamma_{A,\alpha}}^k$. By Lemma 4.11, q is the $(k-1)$-value in (X, α) of some q-tag p. So $\rho(p) \in T_k$. Consider the nested sequence of components given by Lemma 6.1 for the state X and answer function α. The component at phase k in this sequence has a guard $\delta(\sim_k)$ that includes the conjunct $\rho(p)^* = \rho(p)^*$. Furthermore, the components from earlier phases, in whose true branches this $\delta(\sim_k)$ lies, all have guards that are true in (X, α) (by Lemma 5.10). Thus, α contains enough replies to evaluate, in X, all these earlier guards and all the proper sub-e-tags of p that begin with external function symbols. Let ξ be the subfunction of α consisting of just what is used in

these evaluations. Then, by definition of the semantics of conditional and parallel rules, $\xi \vdash^A_X q$. □

To complete the proof of equivalence between A and Π, it remains to consider failures and updates. We assume, from now on, that α is a context for X, since otherwise neither failures nor updates can occur in (X, α). (Note that, by Lemmas 6.2 and 2.21, A and Π have the same contexts in each state, so there is no ambiguity in saying that α is a context for X.) The only sources of updates and failures in Π are the update rules and `Fail` that are components at phase B. These occur in the true branches of components guarded by $\delta(\sim)$, where $\sim$ describes a B-similarity class of state-context pairs.

Which such guards can get value `true` in (X, α)? By Lemma 5.10, $\sim$ would have to be a restriction of $\sim_{X,\alpha,B}$. Being B-realizable (by construction of Π), $\sim$ would have to be $\sim_{X,\beta,B}$ for some subfunction β of α. But $\sim$ describes a B-similarity class of pairs whose second component is a context. So β is, on the one hand, a subfunction of α and, on the other hand, a context for X. According to Lemma 2.5, this requires that $\beta = \alpha$ and therefore $\sim = \sim_{X,\alpha,B}$. So in (X, α) only one of the phase B guards is true, namely $\delta(\sim)$ for $\sim = \sim_{X,\alpha,B}$.

Therefore, each of the following is equivalent to the next.

—Π fails in (X, α).

—The true branch $R_\sim$, for $\sim = \sim_{X,\alpha,B}$, is `Fail`.

—A fails in any member of the B-similarity class described by $\sim$.

—A fails in (X, α).

This completes the verification that A and Π agree as to failures.

The argument for updates is similar. Suppose A doesn't fail in (X, α). The updates produced by Π in (X, α) are those produced by the phase B components of $R_\sim$ where, as before $\sim = \sim_{X,\alpha,B}$. These components were defined as update rules $f(\mathbf{v}^*) := w^*$, where $\mathbf{v}$ and w are e-tags whose B-values $\mathbf{a}$ and b (respectively) participate in an update $\langle f, \mathbf{a}, b\rangle$ produced by A in (X, α) (because (X, α) is in the B-similarity class described by $\sim$). Since the values of $\mathbf{v}^*$ and w^* in (X, α) are also $\mathbf{a}$ and b, by Lemma 5.2, we have that Π produces the same updates as A.

This completes the verification that A and Π are equivalent, so Theorem 1.1 is proved.

Remark 6.3. We emphasize that the construction of the program Π in the preceding proof was intended only for the purpose of proving the theorem. We do not advocate writing ASM programs that look like this Π. In practice, there will almost surely be simpler ASMs equivalent to a given algorithm. In fact, algorithms are usually described by being written in some programming language (or pseudocode, or something similar), and then it is advisable to convert this written form of the algorithm into an ASM directly, without going through explicit definitions of causality relations, updates, and failures.

Remark 6.4. Our proof of the main theorem did not use output or let rules. We avoided output rules by simulating them with external functions. Specifically, if an ASM uses output rules with a label l and associated template $\hat{l}$, then our proof provides a simulation in which, in place of l, there is a new, unary, external

function symbol f, assigned the same template. The role of $\texttt{Output}_l(t)$ is now played by `if` $f(t) = f(t)$ `then skip endif`. The semantics is the same: if t gets value a, then the query $\hat{l}[a]$ is caused.

The avoidability of let rules is a more complicated issue, which we explore in Section 7.

7. LET RULES, REPEATED QUERIES, AND BINDING

7.1 Eliminating let

The `let` construct was never used in the ASM program constructed in the proof of Theorem 1.1. So the theorem would remain true if we omitted let rules from the definition of ASMs. It follows, in particular, that any ASM that uses `let` is equivalent to one that doesn't.

This situation may not surprise readers familiar with [Gurevich 2000, Section 7.3], where `let` is introduced as syntactic sugar. Specifically, `let` $x = t$ `in` $R(x)$ `endlet` is defined there as simply $R(t)$. Here and below, we use the traditional notations $R(x)$ and $R(t)$ to mean that the latter is the result of substituting the term t for all free occurrences of the variable x in the former, renaming bound variables if necessary to avoid clashes. Thus, the `let` construct in [Gurevich 2000] amounts to a notation for substitution.

In the present paper, where algorithms interact with the environment within a step, the interpretation of `let` is more subtle than mere substitution. Consider, for example, the ASM

$$\texttt{let } x = p \texttt{ in if } q = x \texttt{ then skip endif endlet},$$

where p and q are nullary, external function symbols, and compare it with the result of substitution,

$$\texttt{if } q = p \texttt{ then skip endif}.$$

These are not equivalent. The first one begins by issuing the query $\hat{p}$ (where the hat refers to the template assignment); if and when an answer is provided, then it assigns that answer as the value of x and proceeds to evaluate the body of the let rule. So it issues the query $\hat{q}$ and, if and when it gets a reply, ends the step (with the empty set of updates). The second ASM immediately issues both of the queries $\hat{p}$ and $\hat{q}$; if and when it gets both replies, it ends the step. The difference is that the first ASM will issue $\hat{q}$ only after getting a reply to $\hat{p}$ but the second will issue $\hat{q}$ immediately. We can express the same observation more formally, in terms of the definitions 2.20 and 2.22 of equivalence, by considering the empty answer function. The query $\hat{q}$ is reachable under $\varnothing$ for the second ASM but not for the first.

Our semantics for `let` $x = t$ `in` $R(x)$ `endlet` builds in some sequentiality; t must be evaluated first, and only afterward does R become relevant. In contrast, $R(t)$ could evaluate t in parallel with other parts of R.

Even though our `let` is not eliminable by simple substitution, it is eliminable with a little more work; we just have to ensure the appropriate sequentiality. A suitable tool for this is provided by conditional rules, because their semantics also involves sequentiality — the guard must be fully evaluated before the branches become relevant. Thus, `let` can be simulated by means of conditional rules, in

which the binding of the let rule is put into the guard of a conditional to ensure that it is evaluated first. Specifically,

$$\texttt{let } x_1 = t_1, \ldots, x_k = t_k \texttt{ in } R(x_1, \ldots, x_k) \texttt{ endlet}$$

is equivalent to

$$\texttt{if } \bigwedge_{i=1}^{k} (t_i = t_i) \texttt{ then } R(t_1, \ldots, t_k) \texttt{ endif}.$$

Of course, there is nothing to prevent us from introducing a different construct into our ASM syntax as syntactic sugar for simple substitution in the style of [Gurevich 2000]. We call this construct *let-by-name* and use the notation `n-let` for it.

Remark 7.1. The corresponding terminology and notation for our standard `let` would be "let-by-value" and `v-let`, because its semantics insists on having a value for the terms t_i and thus for the variables x_i before proceeding. Let-by-name, in contrast, is content to simply use the names t_i in place of the variables.

If the answer function is insufficient for the evaluation of the bindings, an `n-let` rule will execute as much of the body as it can with the partial information provided. In contrast, an ordinary `let` rule (which means let-by-value) will not even look at the body unless the bindings have been fully evaluated. Usually an `n-let` rule's attempt to execute its body in the absence of values for its bindings will not lead to updates or failure, because an ordinary algorithm cannot finish a step until all its queries have been answered, but it may well lead to additional queries that would not have been produced under the standard interpretation of `let`. The word "usually" in the preceding sentence cannot be replaced by "always." In

$$\texttt{n-let } x_1 = t_1, \ldots, x_k = t_k \texttt{ in } R(x_1, \ldots, x_k),$$

it could happen that the execution of $R(t_1, \ldots, t_n)$ does not require the evaluation of all the terms t_i; for example, they could be in the unexecuted branches of conditional rules, or they might not occur at all. In such a case, the let-by-name computation could finish the step even without enough answers to evaluate all the t_i. With `v-let` in place of `n-let`, the execution of the rule would evaluate all the t_i, whether or not they are needed.

7.2 Uneliminability of let

The preceding discussion about eliminating `let` depended crucially on the fact that we adopted the Lipari convention of [Blass and Gurevich to appear, Subsection 4.3] as our official understanding of repeated calls of an external function with the same argument values. Had we adopted either of the alternative conventions, from [Blass and Gurevich to appear, Subsections 4.4 and 4.5], the picture would be very different. In this subsection, we briefly discuss the difference and the role of `let` under these alternative conventions.

The key point is that x may occur many times in a rule $R(x)$. Then the single occurrence of t as the binding in $\texttt{let } x = t \texttt{ in } R(x)$ becomes many occurrences of t in $R(t)$ (and even more if we adopt the technique outlined above for enforcing sequentiality by means of conditionals). Any occurrence of an external function symbol in

t can thus become many occurrences in $R(t)$. Under the flexible convention ([Blass and Gurevich to appear, Subsection 4.5]), these many occurrences might produce different queries; under the must-vary convention ([Blass and Gurevich to appear, Subsection 4.4]) they definitely produce different queries. On the other hand, each occurrence of an external function symbol in t produces only one query in `let` $x = t$ `in` $R(x)$; the reply to that query is then used in a single evaluation of t, whose result is used for all the occurrences of x in $R(x)$.

In the case of the flexible convention, this discrepancy can be removed by suitably defining the template assignment. (Recall that, under both the flexible and the must-vary conventions, templates are assigned not to external function symbols but to their occurrences.) When one occurrence of an external function symbol becomes many occurrences as a result of substitution, then whatever template was originally assigned to the one occurrence must be assigned to all of the occurrences it produces. With this additional explanation attached to the notion of substitution, the elimination of `let` that we gave for the Lipari convention works also for the flexible convention. Let-by-name would be subject to the same additional explanation.

Under the must-vary convention, different occurrences of a function symbol must always produce different queries. So the additional explanation that we used under the flexible convention is not available. In fact, `let` is not eliminable under the must-vary convention. For a specific example, let f be a binary, Boolean function symbol in the vocabulary Υ, let p be an external, nullary function symbol, and consider the ASM program

$$\texttt{let } x = p \texttt{ in if } f(x, x) \texttt{ then skip else Fail endlet}.$$

Without `let` there would be no way, under the must-vary convention, to issue the query $\hat{p}$ just once and use the answer for both arguments of f.

The program just exhibited certainly describes an algorithm (in fact, the same algorithm under all three conventions, since no external function symbol is repeated), but expressing it under the must-vary convention requires the use of `let`. Thus, our proof of Theorem 1.1 would not work under the must-vary convention. To repair it, one would have to use let rules, rather than the guards $\delta(\sim)$, to produce the queries. The guards would still have to be present, to control the flow of the computation, but instead of containing external function symbols, they would use variables bound by `let` to terms that begin with those external function symbols.

7.3 Let rules under the must-vary convention

In fact, the situation is yet more complicated under the must-vary convention. In this subsection, we indicate, mostly by means of examples, the difficulties that arise in attempting to adapt the proof of Theorem 1.1 to the must-vary convention.

Not only is `let` essential rather than eliminable, but ordinary let rules by themselves are not sufficient; let-by-name rules are needed. The following example shows the problem.

Example 7.2. Consider an algorithm with the following causality behavior (in all states). The empty answer function causes two queries, q_1 and q_2. Any answer function that provides a reply r_1 to q_1 causes the query $\langle r_1, r_1 \rangle$. Similarly, any

answer function that provides a reply r_2 to q_2 causes the query $\langle r_2, r_2\rangle$. Finally, any answer function that provides replies r_i to both q_i's causes the query $\langle r_1, r_2\rangle$. We can set up appropriate external function symbols, say nullary p_1 and p_2 and binary f, with templates $\hat{p}_1 = q_1$, $\hat{p}_2 = q_2$, and $\hat{f} = \langle \#1, \#2\rangle$. Under the Lipari convention, the following program would have the specified causality relation. The curious-looking conditionals of the form `if` $t = t$ `then skip endif` are just a way of issuing all the queries involved in the evaluation of t.

```
do in parallel
   if f(p1,p1) = f(p1,p1) then skip endif
   if f(p2,p2) = f(p2,p2) then skip endif
   if f(p1,p2) = f(p1,p2) then skip endif
enddo
```

Under the must-vary convention, this would not work, since the many occurrences of p_1 would all produce different queries (and likewise for p_2). The natural way to prevent this multiplicity of queries would be to use let rules, binding some variables, say x_1 and x_2, to the terms p_1 and p_2, and then using these variables to specify the later queries as $f(x_1, x_1)$, $f(x_2, x_2)$, and $f(x_1, x_2)$. But there are problems with this approach. In the first place, this will have to be modified if the replies r_1 and r_2 are equal, for then these later queries may be different when they ought to be the same. This problem can be solved by testing for equality before issuing the later queries. But even if $r_1 \neq r_2$, there is a more serious problem: How should the let-bindings be arranged?

A single let rule binding both variables won't do. That is,

```
let x1 = p1, x2 = p2 in
   do in parallel
      if f(x1,x1) = f(x1,x1) then skip endif
      if f(x2,x2) = f(x2,x2) then skip endif
      if f(x1,x2) = f(x1,x2) then skip endif
   enddo
endlet
```

does not have the right causality relation. If an answer function provides a reply r_1 to q_1 but provides no reply to q_2, then this program will never execute its body, whereas it ought to issue the query $\langle r_1, r_1\rangle$.

Suppose, therefore, that we use let twice, once to bind each variable. Nesting the two occurrences of `let`, as in

```
let x1 = p1 in
   let x2 = p2 in
      do in parallel
         if f(x1, x1) = f(x1, x1) then skip endif
         if f(x2, x2) = f(x2, x2) then skip endif
         if f(x1, x2) = f(x1, x2) then skip endif
      enddo
   endlet
endlet
```

still doesn't work correctly if the answer function contains a reply to one q_i but not to the other. The variation

```
let x1 = p1 in
   do in parallel
      if f(x1, x1) = f(x1, x1) then skip endif
      let x2 = p2 in
         do in parallel
            if f(x2, x2) = f(x2, x2) then skip endif
            if f(x1, x2) = f(x1, x2) then skip endif
         enddo
      endlet
   enddo
endlet
```

is only slightly better. It works if q_1 is answered, whether or not q_2 is. But if q_2 is answered and q_1 isn't then it fails to issue $\langle r_2, r_2 \rangle$.

Finally, if we try unnested occurrences of `let`, then the bodies of the two let rules will be disjoint. But $f(x_1, x_2)$ has to be in both bodies, since it involves both of the bound variables.

One can make the ASM syntax sufficiently expressive to circumvent this counterexample by introducing let-by-name. Indeed, the program written in the example with a single let rule binding both x_1 and x_2 would become correct if we changed `let` to `n-let`.

Actually, this assertion is vague, because let-by-name has been defined, as simple substitution, only under the Lipari convention. Under the must-vary convention, this definition is not what we want; multiple occurrences of the substituted terms ruin the intended semantics, which should evaluate those terms only once. Indeed, under the must-vary convention, let-by name would have to be defined as a primitive construct in its own right, not as syntactic sugar. In this situation, one might want to introduce another name for the construct, since let-by-name strongly suggests simple substitution. Let us suppose, for the rest of this discussion, that we have a semantics for let-by-name that captures, under the must-vary convention, the desired intuitive meaning: The binding is evaluated (once) and its value is used as the value of the corresponding variable in the body. If the answer function is insufficient to evaluate all the bindings, proceed nevertheless to execute as much of

the body as possible, i.e., as much as does not involve the variables that have been given no values.

The difference between let-by-name and traditional let (i.e., let-by-value) is that the latter imposes sequentiality, by requiring the binding to be evaluated before work on the body can begin, whereas `n-let` allows them to proceed in parallel. It follows that `v-let` can be defined in terms of `n-let`, using conditionals to enforce sequentiality just as we did when we eliminated `let` under the Lipari convention. That is,

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

is equivalent to

$$\texttt{n-let } x_1 = t_1, \ldots, x_k = t_k \texttt{ in if } \bigwedge_{i=1}^{k} (x_i = x_i) \texttt{ then } R \texttt{ endlet}.$$

So it seems reasonable, if one wants to use the must-vary convention, to replace `let` with `n-let` in the list of primitive ASM constructs. Unfortunately, as the following example shows, the resulting ASMs are still insufficient to express all ordinary algorithms (up to equivalence) as in Theorem 1.1.

Example 7.3. Consider an algorithm that begins (in any state) by issuing two queries, a and b. When it gets a reply to a, it issues c (whether or not it got a reply to b); when it gets a reply to b, it issues d (whether or not it got a reply to a). When it gets replies r and s to both c and d, it sets $F(r, s)$ and $G(r, s)$ to `true`, where F and G are dynamic, binary symbols. Formally, the algorithm is given, under the Lipari convention, by

```
do in parallel
   if a = a then if c = c then skip endif endif
   if b = b then if d = d then skip endif endif
   if a = a ∧ b = b then
      if c = c ∧ d = d then
         do in parallel F(c, d) :=true, G(c, d) :=true enddo
      endif
   endif
enddo,
```

where we have simplified notation by using the same symbols like a for an external nullary function symbols and the associated query (which is technically $\hat{a}$). An attempt to express this under the must-vary convention using `n-let` exactly as in the previous example leads to

```
n-let x = a, y = b, z = c, w = d in
   do in parallel
      if x = x then if z = z then skip endif endif
      if y = y then if w = w then skip endif endif
      if x = x ∧ y = y then
         if z = z ∧ w = w then
            do in parallel F(z, w) :=true, G(z, w) :=true enddo
         endif
      endif
   enddo
endlet.
```

This contains a lot of irrelevant code, but, more importantly, it is too quick to proceed to the body of let-rules. It will issue c and d and update F and G without waiting for replies to a and b. The same error occurs if we arrange the uses of `n-let` differently, for example by nesting them. Using ordinary `let` or, equivalently as explained above, using conditionals to enforce sequentiality doesn't work either. The scopes of these let rules or conditionals would have to overlap, since the updates of F and G must be in both of them. But then these scopes would be nested or identical. As a result, it would not be possible for the evaluations of c and d to wait independently for answers to a and b respectively.

A variation of this `n-let` attempt would work if, in place of nullary external function symbols producing the queries c and d, we had unary function symbols having the replies to a and b, respectively as their arguments. That is, if we had $c(x)$ in place of c and $d(y)$ in place of d, then the following program would be correct, because, despite the use of let-by-name, it could not begin to evaluate $c(x)$ until it had a value for x.

```
n-let x = a, y = b in
   n-let z = c(x), w = d(x) in
      do in parallel F(z, w) :=true, G(z, w) :=true enddo
   endlet
endlet
```

We close this subsection with one additional example. Some ASM constructs introduce parallelism into the computation: `do in parallel`, multiple bindings in `let`, update rules, and even evaluation of terms. Others introduce sequentiality: `let`, `if − then − else`, and again evaluation of terms. No other arrangements, beyond parallelism and sequentiality are explicitly built in to the semantics. So it may be useful to show how an ASM program can produce an arrangement of queries that looks like the Wheatstone bridge, the simplest electrical circuit not obtainable by parallel and series composition of elementary pieces. In the following example, we describe this arrangement and exhibit two programs that express it, one under the Lipari convention and one, using let-by-name, under the must-vary convention.

Example 7.4. Consider an algorithm that works as follows. First, it issues queries a and b. After it gets a reply to a, it issues c and d. After it gets replies to

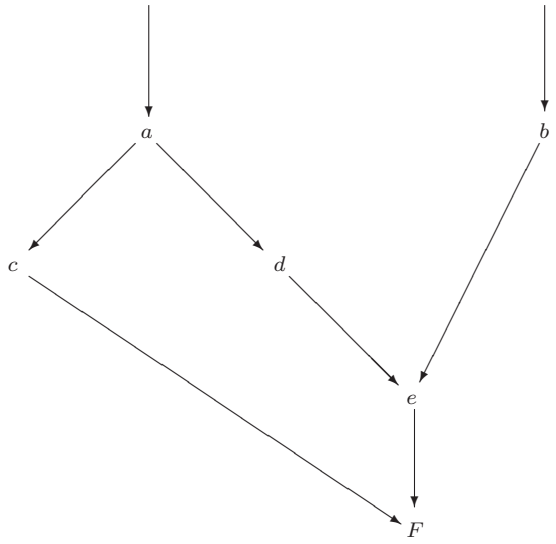

Fig. 1. Control flow of the algorithm of Example 7.4

both b and d, it issues e. Finally, after it gets replies to both c and e it sets F to `true`. See Figure 1.

Here is an ASM expressing this algorithm under the Lipari convention. Again, we economize on notation by using a as an external function symbol assigned to the template a; in other words, we identify a and $\hat{a}$; similarly for the other queries. We also economize by omitting the end-markers `endif` and `enddo`, relying, as many programming languages do, on indentation to provide the right parsing.

```
do in parallel
   if a = a ∧ b = b then
      do in parallel
         if c = c ∧ d = d then
            if e = e then F :=true
         if d = d then
            if e = e then skip
   if a = a then
      if c = c ∧ d = d then skip
```

This ASM was obtained by going through the construction of Π in the proof of Theorem 1.1 and omitting a great deal of extraneous code.

Here is an ASM expressing the same algorithm under the must-vary convention. More precisely, here is an ASM that would do this job once let-by-name is properly defined. We use the same economizing conventions as before, also omitting `endlet`. (The correspondence between the external symbols like a and the variables like x_a

is intended only as a memory aid; it has no formal significance.)

```
n-let x_a = a,  x_b = b in
   if x_a = x_a then
      n-let x_c = c,  x_d = d in
         if x_b = x_b ∧ x_d = x_d then
            n-let x_e = e in
               if x_c = x_c ∧ x_e = x_e then F :=true
```

7.4 Binding external function symbols

In this subsection, we introduce a way to avoid the problems, indicated above, with the must-vary convention. The essential idea is to use must-vary as a default but to allow the program to explicitly say that certain occurrences of an external function symbol are to be assigned the same template. We present two ways to build this idea into the syntax of ASMs; the first way is kept as simple as possible for theoretical purposes, while the second is designed to be easier to use in programming.

The first way adds to the ASM syntax a new operator `bind` with the formation rule saying that, if R is a rule and f is an external function symbol, then

`bind` f `in` R `endbind`

is a rule (in which we would omit `endbind` in situations where the intended scope is made clear by other means, such as indentation). The intended interpretation of this new rule is the same as that of R, but it imposes a constraint on the template assignment accompanying the program, namely that all occurrences of f in R are to be assigned the same template.

In more detail, consider an ASM program Π written in the enlarged syntax with `bind`, and consider an external function symbol f occurring in Π. Call an occurrence of f a *virgin* occurrence if it is either immediately after (an occurrence of) `bind` or not in the scope of any `bind`. The must-vary convention applies to virgin occurrences: they must be assigned templates different enough to ensure that they will not issue the same query. (See [Blass and Gurevich to appear, Subsection 4.4] for details about this.) Any occurrence of f in the scope of a `bind` is to be given the same template as the virgin occurrence at the start of that `bind` construct.

Under this interpretation, a traditional ASM program, not using `bind`, would be interpreted exactly as it would be under the must-vary convention.

At the other extreme, binding all the external functions at the beginning of a program, with the matching `endbind`s at the end, would have the effect of interpreting the program according to the Lipari convention. Given this way of simulating the Lipari convention, and given Theorem 1.1, we see that the ASM thesis holds for the "must-vary with `bind`" interpretation.

Between the two extremes, "must-vary with `bind`" allows much of the freedom of the flexible convention described in [Blass and Gurevich to appear, Subsection 4.5]. Admittedly, it doesn't allow quite as much freedom as the flexible convention — for example, it would not allow some but not all of the occurrences of f in a single term to have different templates — but such situations seem rather exotic. It seems that `bind` allows the sort of flexibility that one often wants in programming, for

example the ability to issue a call to an external library just once, no matter how often it is used, while requests for new elements, to be imported from the reserve, are kept distinct.

For programming purposes, it seems useful to amplify the syntax of `bind` by allowing an explicit specification, in the program, of the template to be used. That is, a `bind` rule would have the form

$$\texttt{bind } f \texttt{ to } t \texttt{ in } R,$$

where t is a template (with the right number of placeholders), and where f and R are, as before, an external function symbol and a rule.

If this is to reflect the must-vary convention, then one would, of course, have to be careful that the specified templates prohibit collisions of external function calls. But one could relax this requirement and use the `bind − to` construction to indicate some of the templates to be used under the flexible convention.

Remark 7.5. While working on an implementation of some of the ideas of [Blass and Gurevich 2006; to appear], for use in AsmL, Davor Runje found it useful to think of queries as objects and therefore to separate the tasks of creating queries and of issuing them. His suggestion of that separation was instrumental in leading us to the `bind` construct.

8. ADDITIONAL ASM CONSTRUCTS

In this section, we present two constructs, which were omitted from the definition of ASMs in [Blass and Gurevich to appear] because they do not enlarge the class of expressible algorithms, but which are very useful in actual programming. We provide, for each of these constructs, a semantics in the style of [Blass and Gurevich to appear, Section 5]. If, therefore, we added these constructs to our definition of ASMs, the resulting ASMs would still define ordinary algorithms and would, according to Theorem 1.1, be equivalent to ASMs that do not use the new constructs.

8.1 Sequential composition

We add to the ASM syntax the construction rule that, whenever R_1 and R_2 are rules (for vocabulary Υ), then so is

$$R_1 \texttt{ seq } R_2 \texttt{ endseq},$$

which is called the *sequential composition* of its *first stage* R_1 and its *second stage* R_2. A variable is free in the sequential composition if and only if it is free in at least one of the stages.

The intended meaning of this sequential composition is that one first executes R_1 and then, in the new state resulting from this execution, executes R_2.

We now present the formal semantics expressing this intention, and we verify that it produces a clean algorithm. Let R be the sequential composition R_1 `seq` R_2 `endseq`. As in [Blass and Gurevich to appear, Section 5], we assume as an induction hypothesis that R_1 and R_2 have already been interpreted as clean algorithms, with all structures (of the right vocabulary) as states. Let $\mathbf{v}$ be a list that includes all the free variables of R, and let X be an $\Upsilon \cup \dot{\mathbf{v}}$-structure.

We begin by defining the causality relation $\vdash_X$ attached to state X by the rule R. It is the union $\vdash^1 \cup \vdash''$ where $\vdash^1$ is the causality relation attached to X by R_1 and where $\vdash''$ is defined by letting $\xi \vdash'' q$ under the following circumstances. First, ξ must be the union of a context ξ' for $\vdash^1$ and another answer function η. Second, R_1 must not fail in (X, ξ'); so there is a well-defined state $X^* = \tau_{R_1}(X, \xi')$, the state obtained from X by executing R_1 with context ξ'. Third, $\eta \vdash^* q$, where $\vdash^*$ is the causality relation attached to X^* by R_2.

We omit the proof that $\vdash_X$ is clean, because it is entirely analogous to the proof for conditional and let rules. We also get a characterization of contexts almost as we did for conditional and let rules. The main difference from the earlier situation is that R_1 could fail and then R_2 would not be executed; that accounts for item 1 in the following description of contexts.

LEMMA 8.1. *The contexts α for $\vdash_X$ are of two sorts:*

(1) α is a context for $\vdash^1$ and R_1 fails in (X, α).

(2) α is the union $\xi \cup \beta$ of a context ξ for $\vdash^1$, such that R_1 doesn't fail in (X, ξ), and a context β for $\vdash^*$.

Here $\vdash^1$ and $\vdash^*$ are as in the definition of $\vdash_X$. Furthermore, in Case 2, both ξ and β are uniquely determined by α.

Proof We use the notations Γ, (Γ_1), and (Γ_*) for the operators associated to the causality relations $\vdash_X$, $\vdash^1$, and $\vdash^*$, respectively.

Assume first that α is a context for $\vdash_X$. Since $\mathrm{Dom}(\alpha) = {\Gamma_\alpha}^\infty \supseteq {(\Gamma_1)_\alpha}^\infty$, Lemma 2.5 tells us that α includes a unique context ξ for $\vdash^1$.

Case 1: R_1 fails in (X, ξ). In this case, we shall show that $\alpha = \xi$ and so we have Condition 1 of the lemma. Suppose, toward a contradiction, that there is some $q \in \mathrm{Dom}(\alpha) - \mathrm{Dom}(\xi) = {\Gamma_\alpha}^\infty - \mathrm{Dom}(\xi)$. Choose such a q that is in ${\Gamma_\alpha}^{k+1}$ for the smallest possible k. By definition of ${\Gamma_\alpha}^{k+1}$, there is some $\beta \subseteq \alpha \restriction {\Gamma_\alpha}^k$ such that $\beta \vdash_X q$. Because we chose $k+1$ as small as possible, ${\Gamma_\alpha}^k \subseteq \mathrm{Dom}(\xi)$. Thus, $\beta \subseteq \xi$. Since $\beta \vdash_X q$, we have two subcases according to the definition of $\vdash_X$.

In the first subcase, $\beta \vdash^1 q$. This means that $q \in (\Gamma_1)_\xi({(\Gamma_1)_\xi}^\infty) = {(\Gamma_1)_\xi}^\infty = \mathrm{Dom}(\xi)$, contrary to our choice of q.

In the second subcase, $\beta = \xi' \cup \eta$ where ξ' is a context for $\vdash^1$, the rule R_1 does not fail in (X, ξ'), and several more conditions hold (which we won't need). Since $\xi' \subseteq \beta \subseteq \xi$ and since both ξ and ξ' are contexts for $\vdash^1$, we conclude by Lemma 2.5 that $\xi = \xi'$. But our case hypothesis in Case 1 is that R_1 does fail in (X, ξ), and so we have reached a contradiction. This establishes Condition 1 of the lemma in Case 1.

Case 2: R_1 does not fail in (X, ξ). In this case, we get Condition 2 of the lemma. The argument is exactly parallel to the proof of [Blass and Gurevich to appear, Lemma 5.11], so we do not repeat it here.

For the converse, we must show that any α as in Conditions 1 and 2 is a context for $\vdash_X$. Under Condition 2, the argument is again just like the proof of [Blass and Gurevich to appear, Lemma 5.11], so we omit it. The argument under Condition 1 is easier, but we give it for the sake of completeness.

Suppose, therefore that α is a context for $\vdash^1$ and that R_1 fails in (X, α). Then $\mathrm{Dom}(\alpha) = {(\Gamma_1)_\alpha}^\infty \subseteq {\Gamma_\alpha}^\infty$. All that remains is to prove the converse inclusion.

Suppose, toward a contradiction, that there are queries in ${\Gamma_\alpha}^\infty - \mathrm{Dom}(\alpha)$, and let q be such a query that is in ${\Gamma_\alpha}^{k+1}$ for the smallest possible $k+1$. By definition of ${\Gamma_\alpha}^{k+1}$, we have $\delta \vdash_X q$ for some $\delta \subseteq \alpha \restriction {\Gamma_\alpha}^k$. By definition of $\vdash$, there are two subcases.

In the first subcase, $\delta \vdash^1 q$. Then $q \in (\Gamma_1)_\alpha(\mathrm{Dom}(\alpha))$. But, since α is a context for $\vdash^1$, its domain is fixed by $(\Gamma_1)_\alpha$. So $q \in \mathrm{Dom}(\alpha)$, contrary to our choice of q.

In the second subcase, $\delta = \xi \cup \eta$ where ξ is a context for $\vdash^1$, R_1 doesn't fail in (X, ξ), and some additional conditions hold (which we won't need). Since $\xi \subseteq \delta \subseteq \alpha$ and since both ξ and α are contexts for $\vdash^1$, Lemma 2.5 gives that $\xi = \alpha$. But R_1 fails in (X, α) and not in (X, ξ), so the second subcase has also produced a contradiction.

Finally, we observe that the uniqueness of ξ and β in Case 2 also follows just as in [Blass and Gurevich to appear, Lemma 5.11]. □

As in [Blass and Gurevich to appear, Section 5], this lemma immediately implies that number and length of the queries in any context for $\vdash_X$ are uniformly bounded, namely by the sum of the bounds for R_1 and R_2.

To complete the definition of the semantics of sequential composition, we must define updates and failures for states X and contexts α. We consider separately the two types of contexts described in Lemma 8.1.

If α is a context for $\vdash^1$ and R_1 fails in (X, α), then $R = R_1$ `seq` R_2 `endseq` also fails in (X, α), and we leave $\Delta^+_R(X, \alpha)$ undefined. (One might prefer to set $\Delta^+_R(X, \alpha) = \Delta^+_{R_1}(X, \alpha)$. Up to equivalence of algorithms, it doesn't matter, since Δ^+ of failing state-context pairs is irrelevant.)

Suppose now that $\alpha = \xi \cup \beta$ as in Condition 2 of Lemma 8.1. So R_1 does not fail in (X, ξ) and produces a transition to X^*, and β is a context for the causality relation $\vdash^*$ of R_2 in X^*. We define that R fails in (X, α) if and only if R_2 fails in (X^*, β). The update set $\Delta^+_R(X, \alpha)$ is the union of

—$\Delta^+_{R_2}(X^*, \beta)$ and

—the subset of $\Delta^+_{R_1}(X, \xi)$ consisting of those updates that don't clash with any updates in $\Delta^+_{R_2}(X^*, \beta)$.

Intuitively, this definition says that, to execute R in (X, α), first one executes R_1 in (X, ξ), producing X^* unless it fails, and then one executes R_2 in (X^*, β). The execution of R fails if either of the two stages fails. If it doesn't fail, then the resulting transition is to the state that the execution of R_2 in (X^*, β) produces. Thus, the update set for R consists of the updates performed by R_1 (leading to X^*) and the updates performed by R_2 (leading to the final result), except that if both algorithms update the same location, then the last update, the one done by R_2, prevails.

To complete the verification that the semantics of sequential composition produces an algorithm, we must produce a bounded exploration witness W. Let W_1 and W_2 be bounded exploration witnesses for R_1 and R_2, respectively. As usual, we assume that these W_i are closed under subterms and contain `true`, `false`, and at least one variable. W will be the union of W_1, W_2, and an additional set $\bar{W}$ of terms associated to the terms in W_2 in the following way.

Each term $t \in W_2$ may have numerous (but finitely many) associated terms in $\bar{W}$. We require:

(1) Every variable is an associate of itself.
(2) If $f(t_1, \ldots, t_n) \in W_2$ and if $\bar{t}_i$ is an associate of t_i for $1 \leq i \leq n$, then there are terms $\tilde{t}_i$, differing from the $\bar{t}_i$ only in that their variables have been renamed so that no variable occurs in two of them, such that $f(\tilde{t}_1, \ldots, \tilde{t}_n)$ is an associate of $f(t_1, \ldots, t_n)$.
(3) If $t \in W_2$ begins with a dynamic function symbol and if $s \in W_1$, then s is an associate of t.

In item 2, it is intended that, for each choice of $f(t_1, \ldots, t_n)$ and $\bar{t}_i$'s as there, one particular renaming of variables is chosen, to produce $\tilde{t}_i$ with disjoint sets of variables, for which $f(\tilde{t}_1, \ldots, \tilde{t}_n)$ is made an associate of $f(t_1, \ldots, t_n)$. Thus, item 2 contributes only finitely many terms to $\bar{W}$. When we refer below to the recursive definition of associates, we mean the description above, made into an unambiguous definition by some specification of which variables to use for the renaming at each step.

Remark 8.2. The intention behind the notion of associates is the following; it will be made precise in Lemma 8.8 below. In verifying bounded exploration, we shall need to deal with the values in X^* of terms $t \in W_2$ when the variables are given values in the range of an answer function α. These are not the same as the corresponding values in X, because the dynamic functions can be different in X and X^*. Nevertheless, the value of t in X^* can be obtained as the value in X of a suitably chosen term $\bar{t}$, again with the variables getting values in Range(α). The idea is to modify t by replacing subterms that begin with dynamic function symbols, the subterms where the difference between X and X^* makes itself felt, by terms that describe in X the values in X^* of those subterms. These modified terms $\bar{t}$ are the associates of t.

Remark 8.3. The idea behind the renaming of variables in item 2 is this. Suppose we have, in some state X, certain elements a_i that are the values of the terms $\bar{t}_i$ under certain assignments of values to the variables. It may happen that $f_X(a_1, \ldots, a_n)$ is not the value of $f(\bar{t}_1, \ldots, \bar{t}_n)$ under any assignment of values to variables. The problem is that, if a variable occurs in several of the terms t_i, then the assignments that produced the a_i might disagree as to this variable's value. There might be no single assignment that simultaneously gives all the t_i the corresponding values a_i. By renaming the variables in each term t_i so that no variable is used in both t_i and t_j with $i \neq j$, we prevent this problem from arising. That is, although $f_X(a_1, \ldots, a_n)$ may not be the value of $f(\bar{t}_1, \ldots, \bar{t}_n)$ under any assignment of values to variables, it will clearly be the value of $f(\tilde{t}_1, \ldots, \tilde{t}_n)$ under some such assignment.

Remark 8.4. According to Remark 2.11, we can arrange that no variable is repeated in any term from W_1 (or even in the whole set W_1, if we are willing to weaken the convention that W is closed under subterms by allowing renaming of variables). If we assume that this has been done, then no variable will be repeated in any associate, thanks to the renaming. In this situation, the recursive definition of associates is equivalent to the following construction. To produce an associate of a term $t \in W_2$, first choose some (occurrences of) subterms r_i that begin with dynamic function symbols and that are unnested (i.e., no r_i is a subterm of another).

Second, replace each r_i by a term $s_i \in W_1$. Finally, if t wasn't just a variable, then rename (occurrences of) variables so that no variable occurs more than once. More precisely, for each choice of renamings in either this construction or the recursive definition, there is a corresponding choice of renamings in the other, producing the same associate for t.

Remark 8.5. In our definition of W as $W_1 \cup W_2 \cup \bar{W}$, we could have omitted W_2, because every term $t \in W_2$ is (up to renaming variables — possibly renaming different occurrences of a variable as distinct variables) an associate of itself. The presence in W of a variant of t with changed variables makes the presence of t itself irrelevant to the notion of agreement with respect to W and thus to the question whether W is a bounded exploration witness.

To see why $W = W_1 \cup W_2 \cup \bar{W}$ serves as a bounded exploration witness for the sequential composition R, consider two states X and X' that agree, with respect to W, over an answer function α. Recall that this means that each term in W gets the same values in both states whenever the variables are given the same values from Range(α).

LEMMA 8.6. *For any subfunction ξ of α, we have the following agreements between X and X'.*

(1) ξ causes the same queries under $\vdash^1_X$ and $\vdash^1_{X'}$.
(2) ξ is a context for both or neither of $\vdash^1_X$ and $\vdash^1_{X'}$.
(3) If ξ is a context, then R_1 fails in both or neither of (X, ξ) and (X', ξ).
(4) If R_1 doesn't fail in these pairs, then $\Delta^+_{R_1}(X, \xi) = \Delta^+_{R_1}(X', \xi)$.

Proof Since $W_1 \subseteq W$ and $\xi \subseteq \alpha$, our two states agree with respect to W_1 over ξ. Now parts 1, 3, and 4 of the lemma follow from the fact that W_1 is a bounded exploration witness for R_1. Part 2 follows by Lemma 2.12. □

Let us consider first the case that α does not include a context for $\vdash^1_X$. Then, by Part 2 of the lemma, it doesn't include a context for $\vdash^1_{X'}$ either. Therefore, by Lemma 8.1, α is not a context for $\vdash_X$ or $\vdash_{X'}$, so we need only check that α causes the same queries q with respect to $\vdash_X$ and $\vdash_{X'}$. But these are, by definition, the queries caused by α with respect to $\vdash^1_X$ and $\vdash^1_{X'}$. These agree, by Part 1 of Lemma 8.6, so the proof is complete in this case.

From now on, suppose instead that α includes a context ξ for $\vdash_X$. Then ξ is uniquely determined as $\alpha \restriction {\Gamma_{X,\alpha}}^\infty$ (by Lemma 2.5) and is also a context with respect to $\vdash_{X'}$ (by Part 2 of Lemma 8.6). The queries caused by α with respect to $\vdash_X$ are, according to the definition of this causality relation, of two sorts, those caused by α under $\vdash^1_X$ and those caused by some η under $\vdash^*$, where $\vdash^*$ is, as before, the causality relation of R_2 in the state X^* obtained by executing R_1 in (X, ξ), and where $\alpha = \xi \cup \eta$. The same description applies with X' in place of X. The first sort of causality is, as in the preceding paragraph, the same for X and X', because W_1 is a bounded exploration witness for R_1. It remains to consider causality of the second sort. Of course, such causality occurs only if R_1 doesn't fail (in one and hence by Part 3 of Lemma 8.6 in both of the states). So we assume from now on that R_1 doesn't fail in (X, ξ) and (X', ξ).

To complete the proof, we shall need the following result.

LEMMA 8.7. *The states X^* and $(X')^*$, obtained by executing R_1 in (X, ξ) and (X', ξ), agree with respect to W_2 over α (and therefore over any η as in the second sort of causality).*

Once this lemma is established, the rest of the proof is easy, because W_2 is a bounded exploration witness for R_2. Specifically, what is caused, with respect to R_2, by any $\eta \subseteq \alpha$ is the same in X^* and $(X')^*$; so causality of the second sort is the same in X as in X'. In particular, the contexts with respect to R_2 are the same for X^* and $(X')^*$, and so, by Lemma 8.1, the contexts with respect to R are the same for X and X'. If α is such a context, then R fails in both or neither of (X, α) and (X', α) because R_2 fails in both or neither of (X^*, η) and $((X')^*, \eta)$. Similarly, the updates contributed by R_2 are the same in both situations. Since the updates contributed by R_1 are also the same (Part 4 of Lemma 8.6), we obtain that R produces the same update set in (X, α) and in (X', α). This completes the proof that W is a bounded exploration witness for R provided Lemma 8.7 holds.

So it remains to prove this lemma. We shall obtain it as a consequence of the following stronger result, in which we use the notation introduced above and also the notation $\mathrm{Val}(t, X, \sigma)$ for the value of a term t in a state X when the variables are assigned values by σ.

LEMMA 8.8. *Let $t \in W_2$ and let the variables in t be assigned values in Range(α); call the assignment σ. There is an associate $\bar{t}$ of t and there is an assignment $\bar{\sigma}$ of values in Range(α) to its variables such that Val(t, X^*, σ) = Val$(\bar{t}, X, \bar{\sigma})$ and Val$(t, (X')^*, \sigma)$ = Val$(\bar{t}, X', \bar{\sigma})$*

Proof We proceed by induction on t. This is legitimate because W_2 is closed under subterms.

If t is a variable, then we can take $\bar{t} = t$ and $\bar{\sigma} = \sigma$.

Suppose next that t is $f(t_1, \ldots, t_n)$ where f is a static function symbol. By induction hypothesis, we have associates $\bar{t}_i$ for t_i and we have assignments $\bar{\sigma}_i$ (possibly different assignments for different i's) such that $\mathrm{Val}(t_i, X^*, \sigma) = \mathrm{Val}(\bar{t}_i, X, \bar{\sigma}_i)$ for each i and analogously with X' in place of X. (The $\bar{t}_i$ and $\bar{\sigma}_i$ are the same for X and X'.) By definition of associates, t has an associate $\bar{t} = f(\tilde{t}_1, \ldots, \tilde{t}_n)$, where the $\tilde{t}_i$ are obtained from the $\bar{t}_i$ by renaming the variables to be distinct. Because of the renaming, we can find a single assignment $\bar{\sigma}$ giving each $\tilde{t}_i$ the same value that σ_i gave $\bar{t}_i$, both in X and in X'. (Formally, if v is the variable in some (unique) $\tilde{t}_i$ that replaced w in $\bar{t}_i$, then $\bar{\sigma}(v)$ is defined to be $\bar{\sigma}_i(w)$.) With this choice of $\bar{t}$ and $\bar{\sigma}$, the conclusion of the lemma is clearly satisfied, since $f_X = f_{X^*}$ and $f_{X'} = f_{(X')^*}$.

Finally, suppose that t is $f(t_1, \ldots, t_n)$ where f is a dynamic function symbol. Choose $\bar{t}_i$ and $\bar{\sigma}_i$ as in the case of static f. Let $a_i = \mathrm{Val}(t_i, X^*, \sigma) = \mathrm{Val}(\bar{t}_i, X, \bar{\sigma}_i)$, and, as usual, let $\mathbf{a}$ denote the n-tuple $\langle a_1, \ldots, a_n \rangle$.

Notice that we also have $a_i = \mathrm{Val}(\bar{t}_i, X', \bar{\sigma}_i) = \mathrm{Val}(t_i, (X')^*, \sigma)$. The first equality here comes from the facts that $\bar{t}_i \in \bar{W} \subseteq W$ and that X and X' agree with respect to W over α. The second comes from our choice of $\bar{t}_i$ and $\bar{\sigma}_i$, which works for X' as well as for X.

We consider two cases.

Case 1: The update set $\Delta^+_{R_1}(X, \xi)$ contains no update of the form $\langle f, \mathbf{a}, b \rangle$ for any b. Then $\Delta^+_{R_1}(X', \xi)$ also contains no update of this form, by Part 4 of

Lemma 8.6. Thus, the relevant values of f_X and $f_{X'}$ are unchanged, and we can proceed exactly as we did in the case of a static function symbol.

Case 2: For some b, we have $\langle f, \mathbf{a}, b\rangle \in \Delta^+_{R_1}(X, \xi)$. Of course b is unique, as otherwise R_1 would have failed in (X, ξ). By Part 4 of Lemma 8.6, we also have $\langle f, \mathbf{a}, b\rangle \in \Delta^+_{R_1}(X', \xi)$. By Lemma 3.7, b is critical, with respect to R_1, for ξ in X. That is, it is $\mathrm{Val}(s, X, \bar\sigma)$ for some term $s \in W_1$ and some assignment $\bar\sigma$ of values in $\mathrm{Range}(\xi)$ to the variables. Because $W_1 \subseteq W$ and $\xi \subseteq \alpha$, the agreement of X and X' over α with respect to W implies that b is also $\mathrm{Val}(s, X', \bar\sigma)$. Since s is an associate of t, we can use it as our $\bar t$; together with $\bar\sigma$, it clearly satisfies the conclusion of the lemma. □

Proof of Lemma 8.7 We must show that, for any $t \in W_2$ and any assignment σ of values in $\mathrm{Range}(\alpha)$ to the variables, $\mathrm{Val}(t, X^*, \sigma) = \mathrm{Val}(t, (X')^*, \sigma)$. By Lemma 8.8, this amounts to showing that $\mathrm{Val}(\bar t, X, \bar\sigma) = \mathrm{Val}(\bar t, X', \bar\sigma)$. But this follows from the fact that $\bar t \in \bar W \subseteq W$ and our assumption that X and X' agree with respect to W over α. □

As we already showed, this concludes the verification that W is a bounded exploration witness for R and thus the proof that the semantics of R defines an ordinary algorithm.

8.2 Conditional terms

We introduce a conditional construction for terms,

$$\texttt{if } \varphi \texttt{ then } t_0 \texttt{ else } t_1 \texttt{ endif},$$

where φ is a Boolean term and t_0 and t_1 are arbitrary terms. The intended semantics is entirely analogous to that of conditional rules.

Remark 8.9. It may seem that this conditional construction could be handled simply by including, in all our vocabularies, a static ternary function symbol C, to be interpreted in all structures by $C(\texttt{true}, x, y) = x$ and $C(\texttt{false}, x, y) = y$. (The value when the first argument isn't Boolean is irrelevant.) Then $\texttt{if } \varphi \texttt{ then } t_0 \texttt{ else } t_1 \texttt{ endif}$ could be represented by $C(\varphi, t_0, t_1)$. This gives the right values for conditional terms, but not the right causality relations. The evaluation of $C(\varphi, t_0, t_1)$ would begin by evaluating, in parallel, all three of φ, t_0, and t_1; then it would apply C to the results. The intended interpretation of $\texttt{if } \varphi \texttt{ then } t_0 \texttt{ else } t_1 \texttt{ endif}$, on the other hand, would first evaluate only φ; then, depending on the value obtained, it would evaluate just one, not both, of t_0 and t_1.

The formal semantics of $\texttt{if } \varphi \texttt{ then } t_0 \texttt{ else } t_1 \texttt{ endif}$ is exactly like that of conditional rules, as far as causality is concerned, so we do not repeat it here. The definition of Val is merely analogous, not identical, to the definitions of failures and updates for conditional rules, so we write it out explicitly. Let t be the term $\texttt{if } \varphi \texttt{ then } t_0 \texttt{ else } t_1 \texttt{ endif}$, let X be a state, and let α be a context for t and X. Then, by [Blass and Gurevich to appear, Lemma 5.11], α can be uniquely expressed as $\xi \cup \beta$ with ξ a context for the causality relation $\vdash'$ of φ and η a context for the causality relation for t_0 or t_1 according to whether $\mathrm{Val}(\varphi, X, \xi)$ is $\texttt{true}$ or $\texttt{false}$. If $\mathrm{Val}(\varphi, X, \xi) = \texttt{true}$ then define $\mathrm{Val}(t, X, \alpha) = \mathrm{Val}(t_0, X, \eta)$; if, on the other hand, $\mathrm{Val}(\varphi, X, \xi) = \texttt{false}$ then define $\mathrm{Val}(t, X, \alpha) = \mathrm{Val}(t_1, X, \eta)$.

The verification that the postulates are satisfied, including the construction of the bounded exploration witness, is just as for conditional rules.

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*, vol. 7, no. 2 (April 2006).

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

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

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

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 April 2006; accepted April, 2006