Ordinary Interactive Small-Step Algorithms, II

Andreas Blass[1] Yuri Gurevich[2]

Technical Report
MSR-TR-2004-88

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

[1]Partially supported by NSF grant DMS–0070723 and by a grant from Microsoft Research. Address: Mathematics Department, University of Michigan, Ann Arbor, MI 48109–1109, U.S.A., ablass@umich.edu. Part of this paper was written during visits to Microsoft Research.

[2]Microsoft Research, One Microsoft Way, Redmond, WA 98052, U.S.A. gurevich@microsoft.com

Contents

1 Introduction

The main purpose of this paper is to prove that every ordinary, interactive, small-step algorithm is behaviorally equivalent to an abstract state machine. 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, by issuing queries and receiving replies, during a step (interactive); furthermore, they complete a step only after all the queries from that step have been answered, and they use no information from the environment except for the answers to their queries (ordinary).

The class of ordinary, interactive, small-step algorithms was defined and studied in [3] along with a suitable, quite strong notion of behavioral equivalence for such algorithms. We shall review the relevant definitions in Section 2. In Section 3, we present, in a form suitable for our purposes, the definition of abstract state machines (ASMs), as developed in [6] (see also [9] and the papers linked there, in particular the book [5]). Sections 4 and 5 define the semantics of ASMs by showing how to construe them as algorithms of the sort defined in [3]. In Section 6, we prove our main result, namely that all algorithms of that sort are equivalent to ASMs. In the final Section 8, we discuss some additional ASM constructs, like sequential composition of rules. These constructs are useful in practice though not needed for the simulation of arbitrary ordinary, interactive, small-step algorithms. Since they do not lead outside the class of such algorithms, our main theorem implies that they can be expressed in terms of the constructs from Section 3.

This paper continues the project, begun in [8] and continued in [2], 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 [4].

2 Ordinary Algorithms and Behavioral Equivalence

In this section, we briefly review the definitions, conventions, and postulates from [3] that will be needed in the present paper. We do not, however, repeat the extensive explanations, motivations, and commentary given in [3]. So readers who wonder about the reasons for our postulates, definitions, and conventions should consult [3] for those aspects that deal with interaction and [8] for those aspects common to all small-step algorithms. We also record some general information, much of it from [3], about the interaction mechanism described by our postulates about interaction.

2.1 Algorithms

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

States Postulate: The algorithm determines

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

If X is a state, or indeed an arbitrary structure, we also write X for its base set.

We adopt the following conventions, mostly from [6], concerning vocabularies and structures.

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

- Some of the symbols in Υ may be marked as *static*, and some may be marked as *relational*. Symbols not marked as static are called *dynamic*.
- Among the symbols in Υ are the logic names: nullary symbols `true`, `false`, and `undef`; unary `Boole`; binary equality; and the usual propositional connectives. All of these are static and all but `undef` are relational.
- In any Υ-structure, the interpretations of `true`, `false`, and `undef` are distinct.
- In any Υ-structure, the interpretations of relational symbols are functions whose values lie in $\{\texttt{true}, \texttt{false}\}$.
- The interpretation of `Boole` maps `true` and `false` to `true` and everything else to `false`.
- The interpretation of equality maps pairs of equal elements to `true` and all other pairs to `false`.
- The propositional connectives are interpreted in the usual way when their arguments are in $\{\texttt{true}, \texttt{false}\}$, and they take the value `false` whenever any argument is not in $\{\texttt{true}, \texttt{false}\}$.

□

Definition 2.2 A *potential query* in state X is a finite tuple of elements of $X \sqcup \Lambda$. A *potential reply* in X is an element of X.

Observe that we use a disjoint union $X \sqcup \Lambda$ here, so if X and Λ are not disjoint then they must be replaced by disjoint copies. For notational simplicity, we suppress mention of the copies, pretending in effect that X and Λ are always disjoint.

Definition 2.3 An *answer function* is a partial map from potential queries to potential replies.

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

previous interaction with the environment. Thus, for our purposes, answer functions completely model the environment.

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

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

Definition 2.4 A *context* for a state X is an answer function that is minimal (with respect to $\subseteq$) among answer functions closed under causality. More explicitly, it is an answer function α with the following properties:

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

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

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

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

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

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

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

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

In general, this iteration would continue transfinitely, taking unions at limit ordinal stages. It was, however, shown in [3, Lemma 5.18] that for the operators Γ_α the iteration stabilizes after a finite number of steps. That is, for these operators, there is a finite n such that Γ^n is the least fixed point Γ^∞ of Γ.

Definition 2.5 A *location* in a state X is a pair $\langle f, \mathbf{a}\rangle$ where f is a dynamic function symbol from Υ and $\mathbf{a}$ is a tuple of elements of X, of the right length to serve as an argument for the function f_X interpreting the symbol f in the state X. The *value* of this location in X is $f_X(\mathbf{a})$. An *update* for X is a pair (l, b) consisting of a location l and an element b of X. An update (l, b) is *trivial* (in X) if b is the value of l in X. We often omit parentheses and brackets, writing locations as $\langle f, a_1, \ldots, a_n\rangle$ instead of $\langle f, \langle a_1, \ldots, a_n\rangle\rangle$ and writing updates as $\langle f, \mathbf{a}, b\rangle$ or $\langle f, a_1, \ldots, a_n, b\rangle$ instead of $(\langle f, \mathbf{a}\rangle, b)$ or $(\langle f, \langle a_1, \ldots, a_n\rangle\rangle, b)$.

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

Update Postulate: For any state X and any context α for X, the algorithm either *fails* or provides an *update set* $\Delta^+_A(X, \alpha)$ whose elements are updates (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 from the Update Postulate that, if $\Delta^+_A(X, \alpha)$ clashes, i.e., if it contains two distinct updates of the same location, then A must fail in state X and context α, because the next state, being subject to contradictory requirements, cannot exist. Similarly, if the structure X' described in the postulate is not a state of the algorithm, then the algorithm must fail.

Definition 2.6 If $i : X \cong Y$ is an isomorphism of states, extend it to act on potential queries by applying i to components from X and leaving components from Λ unchanged. Also extend it to act on locations, by acting componentwise on the tuple of elements of X and leaving the dynamic function symbol unchanged. Finally, extend it to act on updates by acting on both components, the location and the new value. We use the same symbol i for all these extensions, mapping the potential queries, locations, and updates of X bijectively to those of Y.

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

Isomorphism Postulate:

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

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

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

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

We record for reference another immediate consequence of the Isomorphism Postulate.

Lemma 2.8 *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.

Bounded Work Postulate:

- There is a bound, depending only on the algorithm A, for the lengths of the tuples that serve as queries. That is, the lengths of the tuples in $\mathrm{Dom}(\alpha)$ are uniformly bounded for all contexts α and all states.
- There is a bound, depending only on A, for the cardinalities $|\mathrm{Dom}(\alpha)|$ for all contexts α in all states.
- There is a finite set W of terms, depending only on A, with the following properties. Assume
  - X and X' are states,
  - α is an answer function for both X and X', and
  - each term in W has the same values in X and in X' when the variables are given the same values in $\mathrm{Range}(\alpha)$.

  If $\alpha \vdash_X q$, then also $\alpha \vdash_{X'} q$. In particular, q is a potential query for X'. If α is a context for X, 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 of terms with the properties required in the last part of the Bounded Work Postulate is called a *bounded exploration witness* for the algorithm A.

Remark 2.10 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. 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. □

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

Definition 2.11 If

- X and X' are states,
- α is an answer function for both X and X', and
- each term in W has the same values in X and in X' when its variables are given the same values in Range(α),

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

When we use this terminology, we often omit "with respect to W" because it will be clear what set W is under consideration.

Notice that if X and X' agree over α then they also agree over any subfunction of α.

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 [3, Section 5]. □

Definition 2.13 An *ordinary, interactive, small-step algorithm* is any entity satisfying the States, Interaction, Update, Isomorphism, and Bounded Work Postulates. We sometimes omit "interactive, small-step" and write simply *ordinary algorithm*; sometimes we even omit "ordinary".

2.2 Reachability and equivalence

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

Definition 2.14 A query q is *reachable* under α if it is a member of ${\Gamma_\alpha}^\infty$. Equivalently, there is a *trace*, a finite sequence $\langle q_1, \ldots, q_n \rangle$ of queries, ending with $q_n = q$, and such that each q_i is caused by some subfunction of $\alpha \restriction \{q_j : j < i\}$.

For the equivalence of the two versions of the definition, see [3] from Definition 6.9 to Proposition 6.11.

Definition 2.15 An answer function α is *well-founded* if $\mathrm{Dom}(\alpha) \subseteq {\Gamma_\alpha}^\infty$.

Definition 2.16 Two causality relations are *equivalent* if, for every answer function α, they make the same queries reachable under α.

Lemma 2.17 *For any two causality relations, the following three statements are equivalent.*

- *The two causality relations are equivalent.*
- *The two causality relations have the same well-founded answer functions α, and, for each such α, the same queries are caused by subfunctions of α.*
- *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 [3]. □

Definition 2.18 Two algorithms are *(behaviorally) equivalent* if they have

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

In the tradition of [8] and [2], this is a very strict notion of equivalence. It implies, in particular, that equivalent algorithms simulate each other step-for-step. Since our main result is that every ordinary, interactive, small-step algorithm is equivalent to an ASM, the stricter our notion of equivalence, the stronger the theorem.

Remark 2.19 The requirement that equivalent algorithms have the same vocabulary is redundant, because Υ is uniquely determined by any Υ-structure, in particular by any state. This remark is the only use we make of the assumption, in the States Postulate, that $\mathcal{S}$ is not empty. □

Remark 2.20 The requirement, in the definition of equivalence, that the two algorithms have the same states could plausibly be weakened to require only that they have the same reachable states. Here, "reachable" means starting with initial states and repeatedly executing the algorithm's transition function with some answer functions. We refrain from adopting this alternative definition for three reasons. First, as indicated above, our stricter notion of equivalence makes our main theorem stronger. Second, in practice it is often easy to check whether something is a state but harder (or impossible) to check whether it is reachable. Third, we prefer to continue in the tradition of [8] and [2], where equivalence required the states to be the same. □

Remark 2.21 Our definition of equivalence is based on a view of algorithms operating in isolation, except that the outside world answers their queries. Consider, for example, an algorithm A whose causality relation, in any state, consists of just a single instance, $\{(q, r)\} \vdash q'$. (See [3, Example 6.2].) Despite this instance, the algorithm A, running alone, will issue no queries; the cause, $\{(q, r)\}$ cannot be realized because there is no way for A to issue q.

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

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

A solution to this problem is implicitly contained in [3, Section 2]. Unsolicited information can affect the algorithm's computation only if the algorithm pays attention to it, and the act of paying attention can be construed as an implicit query. In general, the algorithm will not "know" what sort of unsolicited information it may receive, but we can imagine a general implicit query q_0 asking for whatever relevant information may be available. ("Relevant" here can be taken to mean "appearing among the causes of the algorithm's causality relation.") Imagine, for example, a person doing a computation (implementing an algorithm), but nevertheless able to react to sufficiently loud, sudden noises (like a knock on the door). An algorithm modeling all the possibilities could represent the person's sensitivity to unexpected noises (and other sensory input) as a query. This is the sort of thing we mean by a general implicit query for relevant information. The reply to such a query could then be of the form (q, r) (if tuple-coding is available in the state), meaning "some agent asked q and the reply is r." (In the absence of tuple-coding, the same effect could be achieved by a longer conversation between the algorithm and the environment.)

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

The use of general queries presupposes some integrity constraints on the environment. First, when it provides an answer r to a query q, the envi-

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

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

2.3 Answer functions and their approximations

In this subsection, we collect for later use some facts about answer functions. Though they seem fairly technical and will be used for technical purposes, we hope that they may also help to guide intuition. Much of this material is from [3], and we do not repeat proofs from there. Unless otherwise specified, we deal with a fixed causality relation $\vdash$, meaning $\vdash_X$ for a fixed state X.

Lemma 2.22 *Let α be an answer function. If ${\Gamma_\alpha}^\infty \subseteq Dom(\alpha)$ then $\alpha \restriction {\Gamma_\alpha}^\infty$ is the unique context that is $\subseteq \alpha$. If ${\Gamma_\alpha}^\infty \not\subseteq Dom(\alpha)$ then there is no context $\subseteq \alpha$. In particular, α has at most one subfunction that is a context. Thus, if α and β are two distinct contexts for the same state, then $\alpha(q) \neq \beta(q)$ for some $q \in Dom(\alpha) \cap Dom(\beta)$.*

Proof See Lemma 5.7 and Corollary 5.8 in [3]. □

Definition 2.23 For any answer function α and natural number n, we write α^n for $\alpha \restriction {\Gamma_\alpha}^n$.

Remark 2.24 In [3], we used the notation α_n instead, but we shall need that notation for other purposes here, so we have lifted the n to be a superscript, matching the n in ${\Gamma_\alpha}^n$. □

Lemma 2.25 *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 2.26 *For any answer function α, each α^n is well-founded. α^∞ is the largest well-founded subfunction of α.*

Proof See [3] from Proposition 6.16 to Proposition 6.18. □

Lemma 2.27 *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 [3]. □

Corollary 2.28 ${\Gamma_{\alpha^n}}^k = {\Gamma_\alpha}^k$ *for all $k \leq n+1$, and $\alpha^k = (\alpha^n)^k$ for all $k \leq n$.*

Lemma 2.29 *Suppose that an answer function α includes both a context β with respect to $\vdash$ and an answer function η that is well-founded with respect to $\vdash$. Then $\eta \subseteq \beta$*

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

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

Lemma 2.30 *Every well-founded answer function is a subfunction of some context.*

Proof Let ξ be any well-founded answer function for a state X. Let β be an arbitrary extension of ξ to an answer function whose domain contains all the potential queries for X. Then, since ${\Gamma_\beta}^\infty$ consists of queries, it is included in $\mathrm{Dom}(\beta)$. By Lemma 2.22, there is a context $\alpha \subseteq \beta$. By Lemma 2.29, $\xi \subseteq \alpha$, as desired. □

Corollary 2.31 *The Bounded Exploration Postulate implies that there is a bound, depending only on the algorithm, for the number and length of the queries in any well-founded answer function for any state.*

3 Abstract State Machines — Syntax

3.1 Informal description of ASMs

In this section, we describe the syntax of abstract state machines. This syntax is nearly the same as in [8, Section 6]; the only differences are that we include

- outputs,
- a `let` construction for remembering values, and
- an explicit failure instruction.

There will also be a critical difference in the semantics, namely that interaction with the environment (especially via external functions) can take place within steps, not only between steps as in [8], but we postpone discussion of semantics to Sections 4 and 5.

Except for the explicit failure instruction (on which we shall comment further below), the differences from [8, Section 6] are not new here. Sequential ASMs with outputs were defined in [2]. We slightly extend that definition by allowing several output channels. The `let` construction also occurs briefly in a remark in [6] at the end of Subsection 3.1; a fuller presentation is in [8, Subsection 7.3]. We extend it by allowing several variables to be bound by a single use of `let`, rather than requiring nesting of several `let` rules.

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

As is standard in the recent ASM literature (see for example [7]) and for the most part also in older ASM literature (like [6]), we take the vocabulary Υ and the states of an ASM to be subject to Convention 2.1. Thus, static logic names are available and interpreted correctly in all Υ-structures, in particular in all states.

Our ASMs interact with the environment in two ways, outputs and external functions, both of which involve additional "symbols" (beyond those in the vocabulary Υ). For outputs, an ASM determines a finite set of *output labels*, which we think of as corresponding to different output channels, such as a computer screen or a printer or a message channel to another agent in a distributed computation. The program of the ASM can contain commands

of the form $\texttt{Output}_l(t)$ where l is an output label and t a term; the meaning of this command is to output the value (in the current state) of t on channel l. This extension of basic ASMs to allow output was introduced in [2, Section 2]. The official definition there omitted output labels, in effect allowing only one output channel. This involved no real loss of generality, since the structures used in [2] always included enough set-theoretic background to permit any desired labels to be coded as part of the term t. In the present paper, we do not need so much background for other purposes, so, to avoid an artificial restriction of generality, it is better to allow several output channels.

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

We note that this use of external functions goes beyond that in [6] by allowing nesting of external function symbols. Such nesting is useful in applications, and it also reflects our purpose in this paper, namely to study environmental interactions that occur during a step, rather than between steps, of a computation.

3.2 Syntax of ASMs

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

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

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

Note that external function symbols are allowed alongside the function symbols of the states' vocabulary with no restrictions on nesting. The

Boolean terms are the Boolean variables and the compound terms whose outermost function symbol is relational.

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

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

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

  is a rule, called an *update rule*.

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

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

  is a rule, called an *output rule*.

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

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

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

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

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

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

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

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

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

- `Fail` is a rule.

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

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

Remark 3.4 Other versions of ASM notation, for example in [1] and [6] avoid the need for end-markers like `endif` and `enddo` by using conventions about indentation of lines in programs. We do not impose such conventions here, so some markers are needed to ensure unique parsing. Readers are free to adopt other marking systems; outside this remark, we shall pay no attention to this issue. □

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

`do in parallel` $d :=$ `true`, $d :=$ `false enddo`

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

4 Queries and Templates

4.1 Interaction via external functions

In this section, we prepare for the definition of the semantics of ASMs by discussing the main aspect not already treated in [6, 8], namely the interaction with the environment. Our ASMs interact with the environment by means of external functions and output rules. We treat the case of external functions first and afterward deal with outputs.

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

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

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

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

A second and even more serious problem with the $\langle f, a_1, \ldots, a_n \rangle$ representation of queries is that, under this convention, not all ordinary algorithms

(in the sense defined in Section 2) are equivalent to ASMs. The problem is simply that equivalence of algorithms requires them to issue the same queries under the same circumstances. ASMs, however, would issue only queries of the special form $\langle f, a_1, \ldots, a_n \rangle$ in which a label from Λ occurs in the first position and elements of the state occur in all subsequent positions (and similar queries resulting from output rules). Any algorithm issuing queries of a more general form, say with several components from Λ, would not be equivalent to an ASM.

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

Remark 4.1 The decision in [3] to allow queries of a rather general form, rather than the special form $\langle f, a_1, \ldots, a_n \rangle$, has motivation in actual practice. Queries frequently look like

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

or

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

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

4.2 Templates

As indicated above, in order to simulate all ordinary algorithms, our ASMs must be able to ask queries of the same general form allowed in these algorithms, arbitrary tuples of elements from $X \sqcup \Lambda$; see Definition 2.2. These queries must, for the most part, result from the evaluation of external functions, since the other sort of interaction in our ASMs, via output rules, produces only queries of the trivial sort for which no reply is used by the algorithm. (As discussed in [3, Section 2], such a query is regarded as having an automatic, uninformative "OK" as its reply.)

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

the format of the queries issued by an ASM. Furthermore, by attaching templates to occurrences of external function symbols rather than to the symbols themselves, we can accommodate the possibility that different occurrences of the same f with the same arguments $\mathbf{a}$ produce different queries.

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

An easy answer is that this extra complexity is not needed. Our approach is, as we shall prove in Section 6, adequate to provide ASMs equivalent to all ordinary algorithms.

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

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

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

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

write $t[a_1, \ldots, a_n]$ for the result of replacing each placeholder $\#i$ in t by the corresponding a_i.

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

In the following subsections, we shall discuss various possible conventions governing whether the same query can result from several occurrences of an external function symbol in an ASM program. In this discussion, it will be useful to have a short way to say that two template assignments could, given appropriate values for the arguments, produce the same query. That is the reason for the following terminology.

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

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

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

4.3 No variation — the Lipari convention

We consider first the convention used in the Lipari guide, [6, Section 3.3.2], where it was formulated as follows: "[T]he oracle should be consistent during the execution of any one step of the program. In an implementation, this may be achieved by not reiterating the same question during a one-step execution. Ask the question once and, if necessary, save the result and reuse it."

In our present context, this means the following. Suppose an ASM program contains several occurrences of terms $f(\mathbf{t})$ that begin with the same

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

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

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

The Lipari convention has, however, a serious disadvantage: In practice, one needs external functions whose value can be different for different occurrences within the same step of a computation. A typical example is the nullary external function symbol `new`, whose intended interpretation is an element chosen from the reserve (to be imported as a new element of the active part of the state; see [6, Section 3.2] for information about reserve and importing, although `new` is not used there). The essential property of `new` is that it produces a different value each time it is evaluated. This directly contradicts the Lipari convention.

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

To model such behavior within the Lipari convention, it is necessary to replace the various occurrences of such a "varying" external function symbol

with different function symbols, perhaps by attaching subscripts. Then one has a separate template for each of the new external function symbols. For the sake of human readability, it may be useful to precede such a modified ASM program with a preamble telling which of the external function symbols in the program correspond to which of the "intended" (unsubscripted) symbols.

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

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

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

4.4 Mandatory variation — the must-vary convention

At the opposite extreme from the Lipari convention is the *must-vary convention*. Under this convention, reflecting current standard programming practice, all occurrences of external function symbols are required to produce distinct queries.

This means that templates must be assigned not to the external function symbols but to their occurrences in the program. Furthermore, the assignment must be such that there is no possibility of issuing the same query from different places in the program. This means that, if two of the assigned templates collide, then either at most one of them is actually used in the execution of the program (for example if the relevant occurrences are on different branches of a conditional rule) or the values of the arguments must be such that the resulting queries differ. In general, this requirement is an undecidable, run-time property of the program, but one can ensure it with fairly simple syntactic requirements. For example, one can require that, if two occurrences have colliding templates and are not in different branches of any conditional rule, then they must occur within the scope of guards that guarantee enough distinctness of the arguments to make the queries distinct. Nevertheless, the requirements on the templates are fairly complicated.

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

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

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

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

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

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

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

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

Unfortunately, `let` does not suffice to handle all patterns of repeated use of the result of a single query. Consider, for example, the following scenario. An algorithm begins by issuing two queries, say q and q'. After receiving any reply r to q, it computes (without waiting for a reply to q') two things that depend on r, say $f_1(r)$ and $f_2(r)$ and issues new queries that depend on these. Similarly, after receiving any reply r' to q', it computes (without waiting for a reply to q) $f_1(r')$ and $f_2(r')$ and issues new queries that depend on these. Finally, if it gets answers to both q and q', then it uses them to compute some $g(r, r')$ and issues a query that depends on this. We can simulate the first part of this, the part that uses only r, with a rule that begins `let` $x = e$ `in` ..., where e is an external function symbol producing the query q. Similarly, the part involving only r' can be handled by `let` $x' = e'$ `in` .... But the computation of $g(r, r')$ would require the program to have $g(x, x')$ in the scope of both occurrences of `let`. That could be achieved only by nesting the two occurrences or by combining them into `let` $x = e, x' = e'$ `in` .... Either way, at least one of the two earlier parts, either the r part or the r' part, will not be executed until after both q and q' are answered. So the causality relation of the ASM is inequivalent to that of the given algorithm and thus the ASM is not behaviorally equivalent to the given algorithm. (See, however, the discussion of "eager let" in Section 7.)

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

4.5 Flexible variation

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

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

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

4.6 Reduction to the Lipari convention

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

4.7 Different external functions

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

On the one hand, we shall define the semantics of ASMs in enough generality to cover even the "strange" template assignments where different external function symbols are assigned colliding templates. In fact, the semantics of such an ASM will be the same as for the ASM obtained by (1) replacing any symbols with colliding templates by a single symbol and (2) permuting the arguments of these symbols to match the permutations of the $\#i$'s involved in the definition of colliding templates.

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

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

4.8 Outputs

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

One difference between the output situation and the external function situation is that we could, if we wanted, insist on the simplest form for the output queries, namely $\langle l, a\rangle$ where a is the value of t. This is a very strong form of the Lipari convention. In the case of external functions, we could not insist on the analogous form $\langle f, \mathbf{a}\rangle$, because we needed ASMs to be able to produce queries of the general form permitted by Definition 2.2. We don't need to match these general queries with output rules, simply because our construction of an ASM equivalent to any given ordinary algorithm will not

use output rules at all.

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

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

5 Abstract State Machines — Semantics

5.1 General features of ASM semantics

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

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

- an ASM program using vocabulary Υ together with some vocabulary E of external function symbols and some set of output labels,
- a template assignment, i.e, a function assigning
  - to each n-ary external function symbol f a template $\hat{f}$ for n-ary functions and
  - to each output label l a template $\hat{l}$ for unary functions,

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

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

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

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

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

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

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

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

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

5.2 Semantics of terms

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

- a causality relation $\vdash^t_X$ between finite answer functions and potential queries,
- for any context α, a value $\mathrm{Val}(t, X, \alpha)$.

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

Isomorphism Postulate for Terms:

- Any isomorphism $i : X \cong Y$ of states preserves causality, i.e., if $\xi \vdash^t_X q$ then $i \circ \xi \circ i^{-1} \vdash^t_Y i(q)$.
- If $i : X \cong Y$ is an isomorphism of states and if α is a context for X, then $\mathrm{Val}(t, X, \alpha) = \mathrm{Val}(t, Y, i \circ \alpha \circ i^{-1})$.

For the Bounded Work Postulate, it means:

Bounded Work Postulate for Terms: There are uniform bounds for the cardinalities of the contexts and the lengths of the queries that occur in them. Furthermore, there is a bounded exploration witness, i.e., a finite set W of terms with the following properties. Assume

- X and X' are states,
- 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).

If $\alpha \vdash_X q$, then also $\alpha \vdash_{X'} q$. If α is a context for X, then $\mathrm{Val}(t, X, \alpha) = \mathrm{Val}(t, X', \alpha)$.

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

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

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

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

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

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

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

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

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

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

This clearly satisfies the part of the Isomorphism Postulate referring to causality. As for the Bounded Work Postulate, we take as our bounded exploration witness W the union of the witnesses for the t_i augmented by the variant of t itself described in the last part of our normalization of bounded exploration witnesses. (The augmentation is needed only for the sake of the normalization.) It is clear that this choice of W behaves as required with respect to causality. The part of the Bounded Work Postulate concerning

the number and length of queries is also satisfied, as will become clear once we determine, in Lemma 5.6 below, what the contexts are.

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

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

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

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

Lemma 5.6 *Let $\vdash$ be the union of the clean causality relations $\vdash^i$. Then an answer function α is a context for $\vdash$ if and only if it is the union of subfunctions α_i that are contexts for the respective $\vdash^i$'s. Furthermore, these subfunctions are uniquely determined.*

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

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

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

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

Suppose, toward a contradiction, that there are elements $q \in {\Gamma_\alpha}^\infty$ that are not in ${(\Gamma_i)_{\alpha_i}}^\infty$ for any i. Choose such a q that is in ${\Gamma_\alpha}^{n+1}$ for as small an n as possible. (Remember that $\Gamma^0 = \varnothing$ always, so it is correct to write the

exponent as $n+1$.) So $q \in \Gamma_\alpha({\Gamma_\alpha}^n)$, which means that there is $\xi \subseteq \alpha \restriction {\Gamma_\alpha}^n$ such that $\xi \vdash q$. Fix such a ξ and, in view of the definition of $\vdash$, fix i such that $\xi \vdash^i q$. Because $\vdash^i$ is clean, ξ is well-founded with respect to it, and so we have

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

But then

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

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

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

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

According to Lemma 2.22, the subfunction α_i defined by restricting α to ${(\Gamma_i)_\alpha}^\infty$ is a context for $\vdash^i$. Since $\alpha \supseteq \bigcup_i \alpha_i$, it remains only to prove that this inclusion is in fact an equality. But both sides of the inclusion are contexts for $\vdash$ — the left side by assumption and the right side by the part of the lemma already proved. By Lemma 2.22, these two contexts cannot be distinct.

Finally, the uniqueness of the α_i follows immediately from Lemma 2.22. □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and so, by Lemma 2.22, there is a unique subfunction of α that is a context for $\vdash'$, namely $\xi = \alpha \restriction {\Gamma'_\alpha}^\infty$. So $\xi = \bigcup_i \xi_i$ for unique contexts ξ_i for $\vdash^i$. Let $a_i = \mathrm{Val}(t_i, X, \xi_i)$ and let q be the query $\hat{f}[a_1, \ldots, a_n]$. So $\xi \vdash q$ and therefore

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

So α includes $\xi \cup \{(q, r)\}$ for some reply r. We shall show that $\mathrm{Dom}(\xi \cup \{(q, r)\})$ is a pre-fixed point of Γ_α. Then it follows that

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

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

To verify the claimed pre-fixed point, suppose

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

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

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

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

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

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

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

Conversely, every such $\xi \cup \{(q, r)\}$ is a context for $\vdash$. To see this, notice first that, by the argument given above, $\mathrm{Dom}(\xi \cup \{(q, r)\})$ is a pre-fixed point of $\Gamma_{\alpha'}$ for any α' that includes $\xi \cup \{(q, r)\}$; we choose in particular $\alpha' = \xi \cup \{(q, r)\}$. Since ${\Gamma_{\alpha'}}^{\infty}$ is the smallest pre-fixed point of $\Gamma_{\alpha'}$ it is included in $\mathrm{Dom}(\alpha')$. By Lemma 2.22, there is a (unique) $\alpha \subseteq \alpha'$ that is a context for $\vdash$. By what we already proved, this context includes a context for $\vdash'$, which can only be ξ because a single answer function α' cannot include two different contexts for the same causality relation $\vdash'$ (see Lemma 2.22). The rest of our analysis of what a context for $\vdash$ must look like shows that $q \in \mathrm{Dom}(\alpha)$. Thus, $\mathrm{Dom}(\alpha') = \mathrm{Dom}(\xi) \cup \{q\} \subseteq \mathrm{Dom}(\alpha)$. Since $\alpha \subseteq \alpha'$, we conclude that α' is equal to α, which is a context for $\vdash$. This completes the proof that $\xi \cup \{(q, r)\}$ is a context for $\vdash$ and thus completes our description of all contexts for $\vdash$.

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

With the description of contexts available, we are in a position to define $\mathrm{Val}(t, X, \alpha)$ when α is a context for the causality relation $\vdash$ attached to state X and term $t = f(t_1, \ldots, t_n)$. Since α includes a unique context ξ for $\vdash'$, which is in turn a union of unique contexts ξ_i for the $\vdash^i$ (associated to the t_i), there are well-defined elements $a_i = \mathrm{Val}(t_i, X, \xi_i)$. Furthermore, the query $q = \hat{f}[a_1, \ldots, a_n]$ is in $\mathrm{Dom}(\alpha)$. We define $\mathrm{Val}(t, X, \alpha)$ to be $\alpha(q)$.

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

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

5.3 Semantics of rules

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

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

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

In each case of our recursion, we shall verify that the causality relation is clean. Furthermore, we shall establish an explicit description of the contexts, from which the first two items in the Bounded Work Postulate — bounding

the numbers and lengths of queries in any context — will follow. Finally, we shall present, in each case, a bounded exploration witness verifying the remaining parts of the Bounded Work Postulate

Update rules

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

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

It remains to define failures and Δ^+ for $f(t_1 \ldots, t_n) := t_0$, and this is easy. An update rule never fails. Its update set $\Delta^+(X, \alpha)$, for a state X and context α, consists of a single update $\langle f, \langle a_1, \ldots, a_n\rangle, a_0\rangle$, where each a_i is the value $\mathrm{Val}(t_i, X, \alpha_i)$ and the functions α_i are the unique contexts for the respective $\vdash^i$ whose union is α.

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

Output rules

The intuition is that an output rule first issues enough queries to evaluate the term occurring as its argument. Once it has enough information from the environment to evaluate this term, it issues one more query, namely the output itself. Recall from [3, Section 2] that outputs are regarded as queries that receive an automatic and uninformative answer OK. Here is the formal definition.

Let R be $\mathtt{Output}_l(t)$. The causality relation for R is the union of the causality relation $\vdash'$ of t and the relation $\vdash''$, where $\xi \vdash' q$ means the following. First ξ is required to be a context for $\vdash'$. Let $a = \mathrm{Val}(t, X, \xi)$. Then q is

required to be $\hat{l}[a]$. (Recall that the square-bracket notation here means to substitute a for the unique placeholder #1 in the template $\hat{l}$ that the ASM assigns to the output label l. Recall also that we feel free to omit mention of the state X when it is fixed in a particular discussion.)

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

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

Parallel blocks

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

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

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

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

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

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

Remark 5.7 The parallel block with no components is often denoted by `skip`. Taking $n = 0$ in the previous definition, we find that `skip` has the expected semantics. Its causality relation is empty; its update set in any state (and with the unique context $\varnothing$) is empty; and it doesn't fail.

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

Conditional rules

A conditional rule should first issue whatever queries are needed for the evaluation of its guard. When enough answers have been received for this evaluation, the algorithm should continue by executing the appropriate branch. Here is the formal definition.

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

Lemma 5.8 $\vdash$ *is clean.*

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

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

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

So it remains to prove that $\mathrm{Dom}(\eta) \subseteq {\Gamma_{\xi}}^{\infty}$.

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

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

As in previous cases, we intend to characterize the contexts for $\vdash$. We begin with a consequence of Lemma 2.29, making use of the cleanness of our causality relations.

Corollary 5.9 *Suppose $\vdash$ is a clean causality relation and suppose that a certain answer function α includes both a context β with respect to $\vdash$ and an answer function η such that $\eta \vdash q$ for a certain q. Then $\eta \subseteq \beta$ and $q \in Dom(\beta)$.*

Proof Since $\vdash$ is clean, η is well-founded. So Lemma 2.29 gives us that $\eta \subseteq \beta$. Then from $\eta \vdash q$ we infer that $q \in \Gamma_{\beta}(\mathrm{Dom}(\beta)) = \mathrm{Dom}(\beta)$. □

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

Lemma 5.10 *The contexts for $\vdash$ are the unions $\xi \cup \beta$ where ξ is a context for $\vdash'$ and β is a context for $\vdash^0$ or $\vdash^1$ according to whether $Val(\varphi, X, \xi)$ is* `true` *or* `false`*. Furthermore, for each context α, the corresponding ξ and β are uniquely determined.*

Proof Assume first that α is a context for $\vdash$. We shall produce the ξ and β required by the lemma. As before, we use the notations Γ, Γ', and Γ_i

$(i = 0, 1)$ for the Γ operators associated to the causality relations $\vdash$, $\vdash'$, and $\vdash^i$.

Since $\mathrm{Dom}(\alpha) = {\Gamma_\alpha}^\infty \supseteq {\Gamma'_\alpha}^\infty$, we know by Lemma 2.22 that α includes a unique context ξ for $\vdash'$. We assume that $\mathrm{Val}(\varphi, X, \xi) = \mathtt{true}$; the case of $\mathtt{false}$ is handled in the same way.

We show next that $\mathrm{Dom}(\alpha)$ is pre-fixed by $(\Gamma_0)_\alpha$. Suppose $q \in (\Gamma_0)_\alpha(\mathrm{Dom}(\alpha))$. Thus, $\delta \vdash^0 q$ for some $\delta \subseteq \alpha$. Then $\xi \cup \delta \vdash q$. Since $\xi \cup \delta \subseteq \alpha$, we conclude that $q \in \Gamma_\alpha(\mathrm{Dom}(\alpha)) = \mathrm{Dom}(\alpha)$, where the last equality comes from the assumption that α is a context for $\vdash$.

Since ${(\Gamma_0)_\alpha}^\infty$ is the smallest pre-fixed point of $(\Gamma_0)_\alpha$, we have ${(\Gamma_0)_\alpha}^\infty \subseteq \mathrm{Dom}(\alpha)$. Again invoking Lemma 2.22, we infer that α includes a unique context β for $\vdash^0$. Notice that, at this point, we have established the uniqueness assertion in the lemma.

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

Consider first the case that $\delta \vdash' q$. Apply Corollary 5.9 to the clean causality relation $\vdash'$, the context ξ for $\vdash'$, and the fact that $\delta \vdash' q$. The corollary gives that $q \in \mathrm{Dom}(\xi) \subseteq \mathrm{Dom}(\xi \cup \beta)$ as required.

There remains the case that $\delta \vdash'' q$. This means that, first, δ includes a context for $\vdash'$, which can only be ξ since at most one context for $\vdash'$ can be a subfunction of α. Then, since $\mathrm{Val}(\varphi, X, \xi) = \mathtt{true}$, there must be some η such that $\eta \vdash^0 q$ and $\delta = \xi \cup \eta$. Apply Corollary 5.9 to the clean causality relation $\vdash^0$, the context β for this causality relation, and the fact that $\eta \vdash^0 q$. The corollary gives that $q \in \mathrm{Dom}(\beta) \subseteq \mathrm{Dom}(\xi \cup \beta)$ as required.

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

Suppose, therefore, that $\alpha = \xi \cup \beta$ where ξ is a context for $\vdash'$, where $\mathrm{Val}(\varphi, X, \xi) = \mathtt{true}$, and where β is a context for $\vdash^0$. (The case where $\mathrm{Val}(\varphi, X, \xi) = \mathtt{false}$ and β is a context for $\vdash^1$ is handled analogously.) The argument given in the preceding paragraphs shows that $\mathrm{Dom}(\alpha)$ is pre-fixed by Γ_α. Thus, by Lemma 2.22, α includes a context α' for $\vdash$. That context must, by what we have already proved, have the form $\xi' \cup \beta'$ where ξ' and β' are contexts for $\vdash'$ and the appropriate $\vdash^i$, respectively. Since α can include

at most one context for $\vdash'$, we have $\xi' = \xi$. In particular, the appropriate $\vdash^i$ is $\vdash^0$. Since α can include at most one context for $\vdash^0$, we have $\beta' = \beta$ and therefore $\alpha' = \alpha$. Thus, α is a context for $\vdash$. □

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

We define failures and updates for R in the natural way. Given a state X and a context α, write $\alpha = \xi \cup \beta$ where ξ and β are as in Lemma 5.10. Also, let i be 0 or 1 according to whether $\mathrm{Val}(\varphi, X, \xi)$ is `true` or `false`. So β is a context for the causality relation $\vdash^i$ associated to R_i. Then R fails in X and α if and only if R_i fails in X and β. The update set $\Delta^+(X, \alpha)$ is defined to be the update set of R_i in state X and context β.

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

Let rules

Consider a let rule R, say

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

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

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

Given an $\Upsilon \cup \dot{\mathbf{v}}$-structure X and given k elements $a_1, \ldots, a_k$ of X, we write $(X \text{ but } \mathbf{x} \mapsto \mathbf{a})$ for the structure with the same base set as X and the same

interpretations of all function symbols except that each $\dot{x}_i$ is interpreted as the corresponding a_i, whether or not $\dot{x}_i$ had a value in X (i.e., whether or not $\dot{x}_i \in \Upsilon \cup \dot{\mathbf{v}}$).

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

The proof that $\vdash_X$ is clean is essentially the same as in the case of conditional rules. Only the following minor differences need to be taken into account. First, in the case of conditional rules, $\vdash'$ was clean by induction hypothesis. In the present situation, the induction hypothesis tells us that each $\vdash^i_X$ is clean, so we must invoke Lemma 5.5 to infer that $\vdash'$ is clean. Second, in the case of conditional rules, the second part of the causality definition used the causality relation $\vdash^0$ or $\vdash^1$ associated, in state X, to R_0 or R_1 according to the truth value $\mathrm{Val}(\varphi, X, \xi)$. In the present situation, the causality relation used in the second part is always obtained from the same rule R_0 but in different states X^* according to the values $\mathrm{Val}(t_1, X, \xi)$. Neither of these differences affects the structure of the proof, so we do not repeat the details.

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

Lemma 5.11 *The contexts for $\vdash_X$ are the unions $\xi \cup \beta$ of a context ξ for $\vdash'$ and a context β for $\vdash^0$, where $\vdash'$ and $\vdash^0$ are as in the definition of $\vdash_X$. Furthermore, for each context α, the associated ξ and β are uniquely determined.*

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

To define failures and updates for the let-rule R in state A and context α, begin by writing $\alpha = \xi \cup \beta$ with ξ and β as in the preceding lemma. By

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

To complete the verification of the postulates, we must produce a bounded exploration witness W. This will be the union $W' \cup W''$ of two parts. The first part, W', is the union of bounded exploration witnesses for the terms t_i. Recall that we saw, when discussing the semantics of terms of the form $f(t_1, \ldots, t_k)$, that if states X and X' agree as to the values of terms from W' when the variables are given values from $\mathrm{Range}(\alpha)$, then the q's such that $\alpha \vdash' q$ will be the same in both states and, if α is a context with respect to $\vdash'$ then the t_i will have the same values a_i in both states. To form W'', start with a bounded exploration witness W_0 for R_0 and, in each of its terms, replace all occurrences of any constant $\dot{x}_i$ with the following variant t_i' of t_i. To get t_i', make the following two substitutions in t_i. Replace all occurrences of variables v from $\mathbf{v}$ with the corresponding constants $\dot{v}$. Replace all subterms that begin with external function symbols by new, distinct variables.

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

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

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

of the postulate in [3, Section 5].) Let $X'^* = (X' \text{ but } \mathbf{x} \mapsto \mathbf{a})$. So we have that $\eta \vdash^0 q$ in X^*, and we must show that the same is true in X'^*. Since W_0 is a bounded exploration witness for R_0, it suffices to show that each term in W_0 has the same values in X^* and X'^* when the variables are given values in Range(η).

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

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

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

We must still verify that our bounded exploration witness behaves properly with respect to failures and updates, but most of the work for this has

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

Fail

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

Remark 5.12 In Definition 5.1 of ordinary ASMs, we required that the set of states be closed under the transition function, which had not yet been defined. The preceding construction of the semantics of ASMs (under the temporary assumption that all structures of the appropriate vocabulary are states) determined the transition functions, via the specification contained in the Update Postulate. Thus, this construction completes Definition 5.1. □

Example 5.13 We give a small but otherwise realistic example of an ordinary interactive algorithm and show how to represent it with an ASM. The algorithm's vocabulary contains nullary dynamic symbols x and y whose values in all states are numbers, which we think of as the coordinates of a point in the plane. The algorithm accepts as input from a user two numbers δx and δy, to be used as increments of x and y. (We write δx rather than the customary Δx to avoid any possible confusion with the notation for update functions.) So at the end of its step, the algorithm will update x and y to $x + \delta x$ and $y + \delta y$. But first, it wants to draw the new point (x, y) on the computer screen, and for this purpose, it must invoke a drawing routine provided by the operating system. So this drawing routine is part of the algorithm's environment. After it is invoked, the drawing routine calls back

to our algorithm, asking for the coordinates of the point to be plotted. When it gets these coordinates, the drawing routine draws the point and confirms to the algorithm that this job has been done. At this point, the algorithm can complete its step.

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

To write this as an ASM, we must decide on a vocabulary of external function symbols and output channels to correspond to the interaction described above. We use δx as a nullary external function symbol, whose associated query (associated by the template assignment) is the query asking for the user's first input. So the number supplied by the user will be the value of the term δx (in agreement with the notation used earlier). Of course, δy is handled analogously. The algorithm's initial call to the drawing routine will be the query assigned to an external function symbol $Draw$, whose value will be the element sent by the drawing routine to confirm that its job is done. Next, we have the two implicit queries by which the algorithm looks for the drawing routine's callbacks. We use external function symbols $ReqX$ (for "request x") and $ReqY$ for these. So the values of these symbols will be whatever signals the drawing routine sends as its callbacks. The algorithm's outputs, in response to the callbacks, are of course the new values of x and y; we use X and Y as names for the output channels on which these numbers are sent.

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

To make the intention behind such equations more evident, we employ the syntactic sugar $t!$ for $t = t$.

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

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

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

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

6 Ordinary Algorithms are equivalent to Abstract State Machines

In this section, we prove the main result of this paper, the ASM thesis for ordinary algorithms.

Theorem 6.1 *Every ordinary, interactive, small-step algorithm is equivalent to an ordinary ASM.*

Recall that ordinary (interactive, small-step) algorithms, ordinary ASM's, and equivalence were defined in Definitions 2.13, 5.1, and 2.18, respectively, and that ordinary ASMs are interpreted as algorithms in accordance with the semantics presented in Section 5.

6.1 Beginning the proof

To prove the theorem, 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.

We begin by making two simplifications, replacing A by equivalent algorithms. First, we normalize the causality relation for every state, as described in [3, Section 6.3]. We recall the definition and essential properties of this construction. The normalization $\vdash'$ of a causality relation $\vdash$ is defined by letting $\alpha \vdash' q$ mean that α is well-founded and q is reachable under α (with respect to $\vdash$). It was shown in [3] that $\vdash'$ 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 $\vdash'_X$, 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 [3, 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 6.2 *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 $\vdash'_X$.*

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' and an answer function α are related as in the Bounded Work Postulate (i.e., all terms from W get the same values in both states when the variables get the same values from $\mathrm{Range}(\alpha)$) and that $\alpha \vdash'_X q$. So, with respect to $\vdash_X$, α is well-founded and q is reachable under α. The latter means (see Definition 2.14) 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 $\vdash'_{X'}$. 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 $\vdash'_{X'}$. Therefore, $\alpha \vdash'_{X'} q$, as required. □

Notice that normalization makes a causality relation clean. Indeed, if $\alpha \vdash' q$ then α is well-founded with respect to $\vdash$ (by definition of the normalization $\vdash'$) and therefore with respect to $\vdash'$ (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 Section 5.2 insofar as that notion applies to rules. (Its last clause is specific to terms and thus irrelevant here.) In addition, we arrange that W contains `true` and `false`.

Proviso 6.3 Summarizing, we **assume from now on, until the end of the proof of Theorem 6.1** 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 Corollary 2.31 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.

Corollary 6.4 *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 corollary admits the following slight improvement, changing $B+1$ to B in the first part of the conclusion.

Corollary 6.5 *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 Corollary 6.4, $q \in {\Gamma_{\xi'}}^B$. 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 [3] about critical elements of a state. First, recall the definition.

Definition 6.6 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 Range(ξ) to its variables.

In particular, since W contains a variable, all elements of Range(ξ) are critical for ξ.

Lemma 6.7 *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 [3, 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 6.8 *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 [3, Proposition 5.24]. □

6.2 Phases

We now 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.

The picture 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 phase n, 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 ${\Gamma_\alpha}^B = {\Gamma_\alpha}^\infty \supseteq \mathrm{Dom}(\alpha)$. 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.

6.3 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.

Recall (Definition 2.11) that X and X' are said to agree over α if they are as in the last part of the Bounded Work Postulate. Thus, 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 Corollary 6.5. 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 6.9 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. $\Box$

Definition 6.10 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; see the discussion preceding the Isomorphism Postulate in Section 2.

Definition 6.11 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 6.9, 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 6.12 *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 2.28. 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 6.13 *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,

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

□

6.4 Uniformity across states and answer functions; similarity

The preceding results show that, to tell what an algorithm will do in a particular state X and (well-founded) answer function α, it suffices to have partial information about (X, α), namely the values of critical terms when their variables get values in $\mathrm{Range}(\alpha)$. Our next goal is to show that, in a sense, far less information suffices. Instead of knowing what the critical values are, we need only to 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 6.7 and 6.8 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 [8, 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 6.12) 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 6.7, 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') ensures 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 its 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.

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 6.14 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 6.15 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 *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 6.16 *For any n, there are only finitely many tags of 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 levels is countably infinite, but we shall never have any real need for tags of 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 level $\leq B$.

Definition 6.17 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 level 0. The following corollary gives the analogous fact for higher phases.

Corollary 6.18 *If a tag has a value at phase n then its level is at most n, and any subtag of depth k has a value at phase $n - k$.*

Proof The first assertion follows from the second because no tag has a value at depth -1. The second assertion is proved by a routine induction on tags. □

Remark 6.19 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. 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 6.20 *If a 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 in Definition 2.6 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 6.21 *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 6.22 *If, in (X, α), a tag has n-value v, then it also has m-value v for all $m \geq n$.*

Lemma 6.23 *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.

Lemma 6.24 *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 6.7, 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 Range(α^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 6.25 *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 6.8, 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. □

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 6.26 (X, α) and (X', α') are *n-similar* if, for each $k \leq n$ and each pair of e-tags t_1, t_2, if t_1 and t_2 have equal k-values in one of (X, α) and (X', α'), then they also have equal k-values in the other.

To avoid possible confusion caused by tags without values, we emphasize that by "have equal values", we mean that they have values and that these values are equal. In particular, taking t_1 and t_2 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 6.27 *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 6.18, the only tags that need to be checked in the definition of n-similarity are those of level $\leq n$; tags of higher levels won't have k-values, for $k \leq n$, in any state and any answer function. Since, by Lemma 6.16, there are only finitely many tags of any single level, there are only finitely many relevant tags and thus only finitely many similarity classes. □

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

Definition 6.28 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 $x, y \in F$, $x \sim y \iff x \sim' y$.

Lemma 6.29 *For any state X, answer function α, and natural number n, the relation "t and t' are e-tags having the same n-value" is a per $\sim_{X,\alpha,n}$ 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 6.22, 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 level $\leq B$, because tags of higher level will not have n-values when $n \leq B$.

Definition 6.30 A per on e-tags of level $\leq B$ is *n-realizable* if it is $\sim_{X,\alpha,n}$, as in the preceding lemma, for some state X and answer function α.

Remark 6.31 In view of Remark 6.19, 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 6.32 *Suppose $\sim_{X,\alpha,n}$ is a restriction of $\sim_{X',\alpha',n}$. Then there is a subfunction 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 6.20, 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 6.20, 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. □

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 6.27, 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 6.33 *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 [8, 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 2.8, from $q \in {\Gamma_\alpha}^{n+1}$ we get $i(q) \in {\Gamma_\beta}^{n+1}$. By Lemma 6.12 we get that ${\Gamma_\beta}^{n+1} = {\Gamma_{\alpha'}}^{n+1}$. By Lemma 6.21 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 6.24, 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 6.34 *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 6.24, q is the $(B-1)$-value of some q-tag p. By Lemma 6.33, 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 6.35 *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 6.33, 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 6.21, each a'_j is the B-value of u_j and b is the B-value of v in (Y, β). By Lemma 6.23, the same holds in (X', α').

By the Isomorphism Postulate, $\langle f, \mathbf{a}', b' \rangle \in \Delta^+(Y, \beta)$. By Corollary 6.13, $\langle f, \mathbf{a}', b' \rangle \in \Delta^+(X', \alpha')$. □

6.5 Normalizing the algorithm

Proposition 6.33 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 6.35 tells us that failures and updates are determined by the B-similarity class of (X, α) when α is a context. Lemma 6.27 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 only use 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 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 level 0 always have values and tags of higher level cannot have 0-values.

Having computed (the information needed to determine) the 0-similarity class of (X, α), the algorithm knows, via Proposition 6.33, 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 level 0, the algorithm can produce the 0-values of these q-tags p. This is because each p is also at level 0 (because it has a 0-value) and is therefore a tuple of members of Λ and e-tags of 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 6.35, 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 subsection 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 next subsection, we convert this description into an ASM, which will be equivalent to the given algorithm.

6.6 The equivalent ASM

The goal of this subsection is to describe an ASM that is equivalent to a given algorithm A. 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 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.

To describe the program Π of our ASM, it is convenient to introduce syntactic sugar to abbreviate a couple of commonly used constructions. As already mentioned in Remark 5.7, 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`.

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 6.36 We define, for each tag t of 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 level 0, and q-tags consisting entirely of labels from Λ. Note that, for e-tags t of level 0, we have $t^* = t$.

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

Lemma 6.37 *Let X be a state, α an answer function for it, and k a natural number.*

- *For e-tags t of 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 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 level $\leq k-1$ (since t has 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. □

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

Definition 6.38 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 6.39 *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 in that of $\sim_{X,\alpha,B}$. The latter means that t has no B-value. By Lemma 6.37, 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 6.37, 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 6.37, 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 6.37, 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. □

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

To begin, consider the set T_0 e-tags t of 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 6.39, since there is no danger of undefined values for tags of 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 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 6.33, 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 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 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 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 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 6.39 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 6.33, 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 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 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 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. Instead of issuing queries, because of external function symbols in the terms t^* occurring in the guards $\delta(\sim)$, and computing the values of these guards, 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 6.34, 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 6.35 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 6.35 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.

6.7 Proof of equivalence

Let A be an arbitrary algorithm, in the sense of the postulates of Section 2. In the preceding subsection, we have constructed from 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 2.18 of equivalence, we see that the first four 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 6.40 *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 level 0 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, 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 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 level must be $\leq k+1$ (see Corollary 6.18). 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. $\square$

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 2.17, 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 [3, 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 6.41 *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 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 6.37 (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 6.39, $\sim_{k-1}$ 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 6.32 shows that α has a subfunction β such that $\sim_{k-1} = \sim_{X,\beta,k-1}$.

Since the q-tag p is relevant with respect to $\sim_{k-1}$, its $(k-1)$-value q is

in ${\Gamma_\beta}^k \subseteq {\Gamma_\alpha}^k \subseteq {\Gamma_\alpha}^\infty$. This establishes the implication from the first to the second of the allegedly equivalent statements in the lemma.

For the converse, consider any query $q \in {\Gamma_{A,\alpha}}^\infty$; fix some k such that $q \in {\Gamma_{A,\alpha}}^k$. By Lemma 6.24, 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 6.40 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 6.39). 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 6.41 and 2.17, 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 6.39, $\sim$ 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 2.22, 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 6.37, we have that Π produces the same updates as A.

This completes the verification that A and Π are equivalent, so Theorem 6.1 is proved.

Remark 6.42 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 pseudo-code, or something similar), and then it is often possible to convert this written form of the algorithm into an ASM directly, without going through explicit definitions of causality relations, updates, and failures. □

Remark 6.43 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 and Repeated Queries

7.1 Eliminating let

The `let` construct was never used in the ASM program constructed in the proof of Theorem 6.1. 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 [8, 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 [8] 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 2.16 and 2.18 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 [8]. We call this construct *eager let* and use the notation `e-let` for it, because such a rule eagerly proceeds to the body without waiting for all the bindings to be evaluated. If the answer function is insufficient for the evaluation of the bindings, the algorithm will execute as much of the body as it can with the partial information provided. Usually this 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 without the eagerness. (The word "usually" in the preceding sentence cannot be replaced by "always." In

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

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 eager computation could finish the step even without enough answers to evaluate all the t_i.)

7.2 Uneliminability of let

The preceding discussion about eliminating `let` depended crucially on the fact that we adopted the Lipari convention of 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 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 `let` $x = t$ `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 (Subsection 4.5), these many occurrences might produce different queries; under the must-vary convention (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. Eager let 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

`let` $x = p$ `in if` $f(x, x)$ `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 6.1 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 6.1 to the must-vary convention.

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

Example 7.1 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.

```
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 $f(x_1, x_2)$. (We ignore the further problem that, if the replies r_1 and r_2 are equal, then these later queries may be different when they ought to be the same.) But 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
```

doesn't work correctly if the answer function contains a reply to q_2 but not to q_1. 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 eager let. 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 `e-let`.

Actually, this assertion is vague, because eager let 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, eager let would have to be defined as a primitive construct in its own right, not as syntactic sugar. Let us suppose, for the rest of this discussion, that we have a semantics for eager let 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 eager and traditional `let` is that the latter imposes sequentiality, by requiring the binding to be evaluated before work on the body can begin, whereas `e-let` allows them to proceed in parallel. It follows that traditional `let` can be defined in terms of `e-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

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

So it seems reasonable, if one wants to use the must-vary convention, to replace `let` with `e-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 6.1.

Example 7.2 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 ∧ 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 `e-let` as in the previous example leads to

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

This, however, is too eager. It will issue c and d and update F and G without waiting for replies to a and b. Using ordinary `let` or, equivalently as explained above, using conditionals to enforce sequentiality doesn't work. 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. □

Notice that the difficulty here would disappear and the program above would be correct 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 program would be correct, because, despite the eagerness of the let rule, it could not begin to evaluate $c(x)$ until it had a value for x. Benjamin Rossman has suggested a notion of "delayed term" to block eagerness in this manner even when c is nullary and so does not have x as an argument. Specifically, he suggested adding to the ASM syntax terms of the form $t\{s_1, \ldots, s_k\}$, where t and all the s_i are terms. The intended semantics of $t\{s_1, \ldots, s_k\}$ is that

- it cannot be evaluated until all the s_i have been evaluated, and

- when it is evaluated, it gets the same value (and issues the same queries) as t.

The standard semantics of terms imposes some sequentiality, since a term cannot be evaluated until its subterms have been evaluated. Delayed terms introduce additional sequentiality by requiring the evaluation of t to wait until after the evaluations of the s_i, even though the s_i are not subterms of t.

In more general terms, delayed terms weaken the coupling between the temporal order in which calculations are done and the logical order of interdependencies between the calculation. Obviously, if one calculation depends on others, then the former must wait until the latter are done. But with delayed terms the converse no longer holds.

Delayed terms allow us to express the algorithm in the preceding example as

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

In more complicated situations, one would need more complicated delays than just a list of terms $s_1, \ldots, s_k$ as above. For example, a causality relation might require that the evaluation of a term t (which issues some query) be delayed until at least one (not necessarily both) of s_1 and s_2 have values. More generally, one can envision arbitrary disjunctions of conjunctions of delays.

It may be that, with this generalized form of delayed terms and with eager let, the must-vary convention can support a theorem like Theorem 6.1. We have not checked this in detail. For the present purpose, namely capturing the notion of ordinary, interactive, small-step algorithms in an ASM framework, Theorem 6.1, proved under the Lipari convention, is sufficient.

We close this section 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

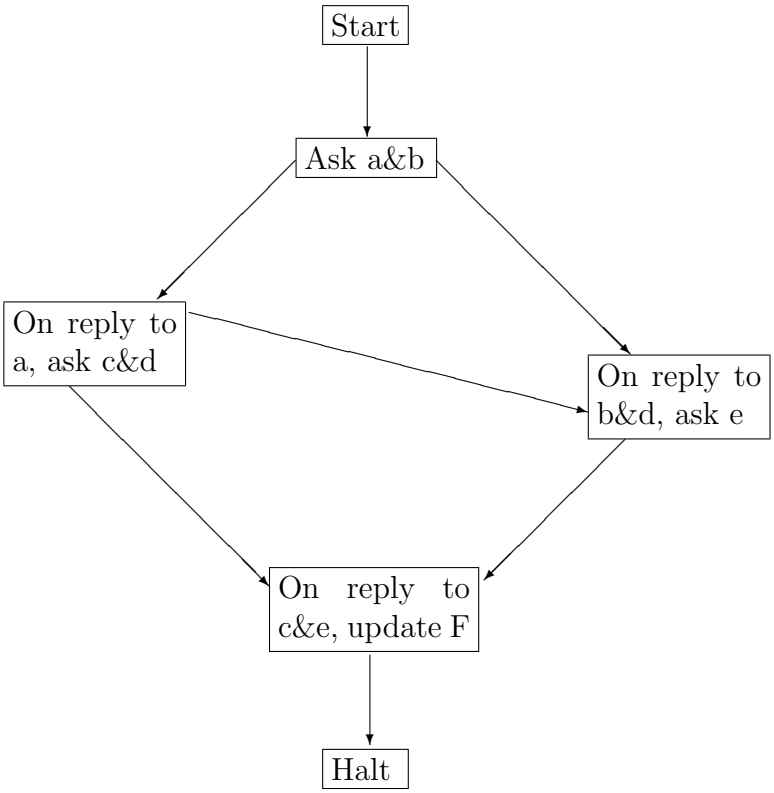

Figure 1: Control flow of the algorithm of Example 7.3

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 eager let, under the must-vary convention.

Example 7.3 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 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. Finally, we use the abbreviation $t!$ for $t = t$, as in Example 5.13.

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

This ASM was obtained by going through the construction of Π in the proof of Theorem 6.1 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 eager let 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.)

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

□

8 Additional ASM Constructs

In this section, we present two constructs, which were omitted from the definition of ASMs in Sections 3 and 5 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 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 6.1, 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 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 $\vdash'$ 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 Γ 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 2.22 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 2.22 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 Lemma 5.10, 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 Lemma 5.10, 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 2.22 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 Lemma 5.10. □

As in 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 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 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 W'.

Remark 8.2 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 might not be the case that $f_X(a_1, \ldots, a_n)$ is the value of $f(\bar{t}_1, \ldots, \bar{t}_n)$ under some 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 8.3 According to Remark 2.10, we can arrange that no variable is repeated in any term from W_1 (or even in the whole set W_1). 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). Then 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 8.4 In our definition of W as $W_1 \cup W_2 \cup W'$, 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. □

To see why $W = W_1 \cup W_2 \cup W'$ 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 8.5 *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 8.5, 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 2.22) and is also a context with respect to $\vdash_{X'}$ (by Part 2 of Lemma 8.5). 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 8.5 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 8.6 *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 8.5), 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 8.6 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 8.7 *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$(t, X^*, \sigma) =$ 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 $\bar{\sigma}$ gives 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 $\bar{t}_i \in W' \subseteq W$ 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 8.5. 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 8.5, we also have $\langle f, \mathbf{a}, b \rangle \in \Delta^+_{R_1}(X', \xi)$. By Lemma 6.8, 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 Range(ξ) 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 8.6 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 8.7, 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 $\bar{t} \in W' \subseteq W$ 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 8.8 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 Lemma 5.10, 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 `true` or `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

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

[2] Andreas Blass and Yuri Gurevich, "Abstract state machines capture parallel algorithms," *A. C. M. Trans. Computational Logic* 4 (2003) 578–651.

[3] Andreas Blass and Yuri Gurevich, "Ordinary Interactive Small-Step Algorithms, I," *A. C. M. Trans. Computational Logic*, to appear.

[4] Andreas Blass, Yuri Gurevich, and Benjamin Rossman, "General interactive small-step algorithms," (in preparation).

[5] Egon Börger and Robert Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag (2003).

[6] Yuri Gurevich, "Evolving algebra 1993: Lipari guide," in *Specification and Validation Methods*, E. Börger, ed., Oxford Univ. Press (1995) 9–36.

[7] Yuri Gurevich, "ASM guide," Univ. of Michigan Technical Report CSE-TR-336-97, at [9].

[8] Yuri Gurevich, "Sequential abstract state machines capture sequential algorithms," *A. C. M. Trans. Computational Logic* 1 (2000) 77–111.

[9] James K. Huggins, ASM Michigan web page, `http://www.eecs.umich.edu/gasm`