# Ordinary Interactive Small-Step Algorithms, I

Andreas Blass[1] Yuri Gurevich[2]

Technical Report
MSR-TR-2004-16

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. Most 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

**Abstract**

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

We indicate how a great many sorts of interaction meet these requirements. We also discuss in detail the structure of queries and replies and the appropriate definition of equivalence of algorithms.

Finally, motivated by our considerations concerning queries, we discuss a generalization of first-order logic in which the arguments of function and relation symbols are not merely tuples of elements but orbits of such tuples under groups of permutations of the argument places.

# Contents



# 1 Introduction

*Small-step algorithms* are characterized by two properties:

- (Step) The computation proceeds in a sequence of discrete steps.
- (Small) The amount of work done by the algorithm in any one step is bounded; the bound depends only on the algorithm, not on the state, nor on the input, nor on the actions of the environment.

Many authors call such algorithms sequential (for example the present authors in [5, 10]), but other authors use "sequential" to mean only the first of these properties (which we call *sequential time*). Because of this ambiguity, we now prefer the term "small-step".

These concepts may be clarified by some non-examples. Parallel algorithms of the sort analyzed in [5] are sequential time algorithms, because the computation proceeds in discrete, global steps, but they are not in general small-step algorithms. When there is no uniform bound on the number of processes operating in parallel, the amount of work done by the algorithm in one step can be unbounded.

Distributed algorithms are, in general, not even sequential time algorithms. Here many agents can proceed asynchronously, each at its own speed, and communications between them may provide the only logical ordering of their activities. Such a computation cannot in general be viewed as a sequence of global steps.

The notion of small-step algorithm is given a precise definition in [10], and the main theorem of [10] is that every such algorithm is equivalent (in a strong sense) to a small-step abstract state machine (ASM). In particular, the small-step ASM step-for-step simulates the given algorithm. The definition in [10] allowed for a limited interaction between the algorithm and its environment. Specifically, a run of the algorithm is a sequence of states in which each (except the last, if there is a last one) is transformed to the next either by the transition function of the algorithm or by an intervention of the environment.

In this paper, we consider small-step algorithms that interact more strongly with their environments. We allow the environment to intervene *during* a step of the algorithm, not just between steps. Indeed, our primary interest here is in interactions during steps. Interactions between steps can be treated as in [10], by allowing a run of the algorithm to contain not only steps in which the algorithm changes the state but also steps where the environment changes the state. Interactions during steps involve more complex issues, which our treatment here is intended to elucidate.

We shall see that there is no loss of generality in taking the interaction to consist of queries from the algorithm and answers from the environment.

It is possible to design algorithms that can complete a step without necessarily having received answers to all the queries issued during the step. It is also possible to design algorithms that make use of the order in which answers are received from the environment. It appears, however, that practical algorithms usually do not behave in such ways. (Real-time algorithms may appear to be a counterexample, but they can be treated by regarding the arrival time of a reply as part of the reply.) Accordingly, we study in this paper those algorithms, the *ordinary* ones, that

- never complete a step until all queries from that step have been answered and
- use no information from the environment beyond the function assigning answers to queries.

Here we count a time-out signal as an answer. A more explicit formulation of the second aspect of ordinariness is that whatever the algorithm does is completely determined by its program, its current state, and the answers the environment has already provided for earlier queries.

The general case of sequential-time algorithms that are not (necessarily) ordinary will be treated in [7].

The present paper begins our study of ordinary, interactive, small-step algorithms. The central contribution is a system of postulates intended to define precisely what these algorithms are. They play the same role as the postulates defining small-step algorithms (previously called sequential) in [10] and those defining wide-step algorithms (previously called parallel) in [5]. We extensively discuss the reasons for our postulates, and we consider some potential variations. We also extensively discuss the concept of equivalence of algorithms, since it involves some subtleties that did not occur in non-interactive situations. Because of the length of these discussions, we postpone to the next paper in this series the connection between algorithms and ASMs. The main result to be proved there is that every algorithm, in the sense defined by the postulates, is equivalent in a strong sense to an appropriate ASM.

Already in the present paper, ASMs play a role, by way of the ASM-based programming and specification language AsmL [1]. Though not limited to small-step algorithms, AsmL provides a prime example of the issues involved in this paper — steps that can include both interaction and parallelism. Indeed, so far as we know, no other imperative language allows both of these simultaneously, with both extensive interaction and parallelism within a single step.

In comparison with the postulates for algorithms in [10], the postulates in the present paper not only allow for interaction with the environment within a step but also make two changes that could — and we believe should — be applied even in the situation of [10]. First, we allow the possibility that an algorithm fails to produce a transition from a certain state (with certain answers from the environment, even if all its queries have been answered).

Second, we take into account trivial updates, those that don't change the state but merely reassert a pre-existing value of some function. These are important in distributed algorithms and are technically useful in the present, small-step situation as well. For details about these changes, see Section 5, and for a general discussion, see [6, Section 7].

**Related work**

Several authors have proposed definitions of what an algorithm is. The one most relevant to our purposes is the definition of sequential algorithms in [10]. Another recent proposal was made by Moschovakis in [13]. Turing's ground-breaking paper [14] also sought to describe algorithms, although the resulting Turing machines no longer look like general algorithms. As far as we know, none of the previous definitions of algorithms dealt with intra-step interaction. For example, [10] dealt only with interactions where the environment acts between steps to modify the state. No environmental interactions were considered in [13]. The generalization of Turing machines to allow oracles amounts to allowing a sort of interaction with an environment, but again not combined with other activity within a single step.

Interaction has also been studied for its own sake, for example in process calculi such as the $\pi$-calculus [12]. But here too, the interaction cannot be described as intra-step; in fact, the interactions are the steps, insofar as there are steps at all. Similarly, the work of Wegner and Goldin (see [15] and the references there), emphasizes interaction but does not involve steps within which repeated interactions can occur.

As indicated above, this paper is to be followed by one establishing that algorithms, in the sense defined here, are behaviorally equivalent to abstract state machines. This idea — the ASM thesis — was first established for small-step algorithms in [10] and subsequently for wide-step algorithms in [5], but in both cases the algorithms under consideration were not interactive.

# 2 Examples and Discussion

We use the word "environment" with a very broad meaning. It covers everything that is relevant to the algorithm's work except for what is internal to the algorithm (and completely specified by the algorithm's program). For example, in a modern computer, a typical algorithm's environment will include

as much of the operating system as might affect the algorithm's operation. In a distributed system, we can consider any one agent as an algorithm and all the other agents as (part of) its environment.

It will be useful to record some typical examples of how an algorithm can interact with its environment, during a step. Ultimately, we shall want a fairly uniform description of these interactions, but first we should look at the diverse possibilities, to see what needs to be uniformized.

**Example 2.1** The algorithm needs a character string to be provided by the user. Prompted by (an implementation of) the algorithm, the user types in a string.

We can pretend that the string has been prepared ahead of time and placed into an input file, before this step of the algorithm, and that the algorithm just reads the string from the file. In this way, we can pretend that the environment — in this case the user — acted between the algorithm's steps. But this is only a pretense; in reality the interaction occurs during the algorithm's step. And the pretense becomes more difficult to maintain if the algorithm could issue any of several prompts, requiring different sorts of responses, or if the algorithm interacts with several users who may respond at different times. □

**Example 2.2** An agent in a distributed computation receives, during one of its steps, a message from another agent. In principle, such a message is similar to the user input in the preceding example, except that no prompt was issued. □

**Example 2.3** During a step, the algorithm prints a string and continues, possibly with additional computation and interaction, to complete the step.

We can pretend that the string is stored somewhere in the algorithm's state and that it is discreetly removed and printed by the environment after the step is complete. That is, we can pretend that the interaction occurred between steps. But, as above, the reality is that the interaction occurs during a step. □

**Example 2.4** An agent in a distributed computation sends a message to another agent. In principle, this is quite analogous to the printing example above. □

**Example 2.5** Non-deterministic choices are, as discussed in [10, Section 9], decisions of the environment. Algorithms are, by nature, deterministic. What is sometimes called a non-deterministic algorithm is really an algorithm that sometimes asks the environment to make a non-deterministic choice for it. That request and the environment's response can happen within a single step of the algorithm. □

**Example 2.6** An algorithm may need to enlarge its state. For example, a graph algorithm may need to add a new vertex to the graph it is working on. Or a Turing machine with a finite tape may have reached the end of its tape and want to attach a new cell in order to continue its computation. We adopt from [8] the convention that a state of an algorithm contains, in addition to its active part, an infinite supply of reserve elements, available to be imported as "new" elements of the active part of the state when needed. This importing, or at least the choosing of the element to be imported, though requested by the algorithm, is actually done by the environment, since it can be non-deterministic (indeed it is in [8]). □

**Example 2.7** Abstract state machines can have so-called external functions [8]. These are similar to any other basic functions in the ASM's state except that their values are not controlled by the algorithm but rather by the environment. It was assumed in [8] that the values of an external function cannot change during a single step of the algorithm, but this assumption was subsequently found to be undesirably restrictive and was therefore removed; in particular it does not apply to AsmL. For example, an important use of external functions is in modeling distributed systems. A value that is written by one agent and read by another is represented as an external function value of the reading agent. Since agents operate asynchronously, the writing agent might work much faster than the reading agent, so, from the latter's point of view, these external function values can change in mid-step. Thus, we consider ASMs for which the values of an external function can change during a step of the algorithm. As a result, if the ASM's program refers several times to the same external function with the same argument tuple within a single step, the values may nevertheless be different. □

With these examples in mind, we now discuss, in general terms, our view of algorithms, environments, and their interaction. These generalities will be replaced, in Section 5, by specific postulates, which will play the central role

in this paper and its sequel. Nevertheless, the preliminary generalities are important for they provide the intuitive basis for the postulates.

What we say here about algorithms, steps, and states is largely a recapitulation of what was already said in [8]. The new material here concerns the interaction with the environment and the two changes mentioned toward the end of the introduction, failures of transitions and trivial updates.

An algorithm is given by a finite text. It provides complete instructions for carrying out a computation.

As already indicated, we are concerned with algorithms that proceed in a discrete sequence of steps (sequential time algorithms). Initially, and again at the end of each step that doesn't fail, the algorithm is in some *state*, which incorporates all the currently available information that will be relevant to the future progress of the algorithm. Note that this meaning of the word "state" is more comprehensive than some other common meanings of the same word. For example, if the algorithm is a Turing machine, then the state in our sense includes not only the state of the finite control but also what is written on the tape and which cell is scanned; that is, it includes what is often called the instantaneous configuration or instantaneous description. Similarly, for a C program, the state in our sense includes not only the values of variables but also the content of the stack and the position of the program counter.

The state changes only at the end of a step. During a step, the algorithm can do scratch work, but unless this work is recorded in the state at the end of the step it will not be available in subsequent steps.

We are concerned with the changes that the algorithm makes to its state in any step. The environment can also intervene, between steps, to change the state, but such changes are outside the scope of our discussion. The role the environment plays in our discussion is to provide information during a step of the algorithm, thereby affecting the state at the end of that step. Thus, we study interactions between the algorithm and its environment taking place entirely within single steps. As mentioned earlier, interactions that take place between the algorithm's steps can be treated as in [8] and [10], so we do not need to reconsider them here. If an interaction appears to span several steps, then it should be regarded as separate interactions within single steps, with appropriate information being recorded in the state so that the algorithm remembers the earlier interaction when the later one occurs. For example, if the algorithm prompts the user for an input during one step and receives the input during a later step, then the prompt should be treated as an outgoing

message, the user's input should be regarded as an incoming message, and the state should, between these two steps, record that a prompt was issued and how the resulting input should be used. In this way, we maintain the principle that the state must contain all the information from the preceding step that is used at any later step.

Everything done by an algorithm during a step is determined, via its program, from the state at the beginning of the step plus information received from the environment during the step.

It is convenient to assume that the interaction between the algorithm and its environment takes the form of *queries* from the algorithm and *replies* from the environment. At first sight, this assumption seems to exclude Example 2.2, where the environment supplies information without being asked, and Examples 2.3 and 2.4, where the algorithm exports information without expecting a reply. These and similar examples can, however, be brought into the query-reply form rather easily as follows.

Consider first a situation where the environment supplies information without being asked for it. Such information cannot have any effect until the algorithm pays attention to it. We regard the algorithm's act of paying attention as an implicit query, and we regard the unsolicited information as a reply to that query. (As indicated above, these considerations are intended for environment actions within a step. If the environment provides information between steps, then it should be viewed as changing the algorithm's state to incorporate this information.)

This view applies in particular if the algorithm is written in the object-oriented style and the incoming information is a method call to one of the objects. The call is regarded as the answer to an implicit query of the form "I'm looking for a method call". It may seem that this lowers our abstraction level, but it doesn't. We don't care in detail how the algorithm inquires about incoming calls. Any implementation of the algorithm will do it in some particular way. But there will always be a query.

Now consider the situation where the algorithm emits information without expecting a reply, for example by sending a message or by printing a string. We can regard such outputs from the algorithm as queries of a degenerate sort, to which the response is a vacuous and automatic "OK". (We could also regard them as queries to which no reply is expected, but that approach would require exceptions in the clause of the definition of "ordinary" algorithms requiring that they complete a step only when all queries have been answered. The "OK" convention means that these queries are automat-

ically answered, so they have no effect on whether a step can be completed. The choice between the two conventions is a matter of convenience.)

With these understandings, we can safely regard every interaction between an algorithm and its environment as being initiated by the algorithm with a query, to which the environment supplies a reply. In accordance with our earlier discussion, the reply will arrive during the same step in which the query was issued. (If it arrived later, we'd have two interactions, one in each of the relevant steps, not a single query and reply. The query would be considered an output, i.e., a query that gets an automatic reply "OK" during the same step. The reply at the later step would be considered an unsolicited input, i.e., a reply to a query (in that same later step) that consists of just paying attention to this input.) It is entirely possible for several such query-reply pairs to occur during a single step, and the later queries may depend on the earlier replies. However, for a small-step algorithm, the number of queries issued in any step must be bounded by a number that depends only on the algorithm, not on the state or on the replies to queries. After all, issuing a query is work, and the total work done during one step is bounded.

A query gets at most one reply. If it appears that several replies are given and are used by the algorithm, then, much as in our discussion of unsolicited input, we regard the algorithm's paying attention to subsequent answers as constituting additional, implicit queries.

It is imaginable that a query gets no reply. For example, the algorithm may request a non-deterministic choice of an element from some set, and the set may be empty. In such a case, the environment may simply fail to respond, though it would be more reasonable to expect a reply in the form of an error message, perhaps of the form "cannot choose from empty set". For another example, the algorithm may prompt the user for some input and the user may fail to respond.

In this paper, we consider only ordinary algorithms, which means in particular that our algorithms never complete a step until all queries issued during that step have been answered. If the environment refuses to answer then the algorithm simply hangs; the step is never finished and there is no next state. A great many practical algorithms are of this sort.

For example, the algorithm may incorporate a "timeout" mechanism whereby, if no answer is received within a certain time, then this fact is itself regarded as a reply, so that the algorithm can proceed (perhaps by retrying the query, perhaps by just making a step with no change to its state, per-

haps by doing something more intelligent, or perhaps by doing something disastrous). Notice that the timeout information, i.e., the information that so and so much time has elapsed since the query, is itself provided by the environment, though probably by a different part of the environment than the part that should be answering the query. This is in keeping with the principle that what the algorithm does during a step depends only on its program, its state, and information supplied by the environment. A clock that keeps time independently of the progress of the algorithm's execution is not part of the program and not part of the state, so it must be part of the environment.

# 3 Motivation

In this section, we address two questions of motivation. First, why study intra-step interaction with the environment? Second, why restrict attention to ordinary algorithms? Readers who are already convinced of the value of studying ordinary intra-step interaction can skip this section without loss of continuity.

## 3.1 Intra-step

It is natural to ask whether intra-step interaction is needed at all. Can't we just consider every interaction with the environment as ending the current step and starting a new one? By subdividing the sequence of steps sufficiently finely, can't we reduce all interaction to the inter-step case?

There are several reasons for not wanting to subdivide steps so finely. First, the subdivision would make some aspects of the situation unpleasantly complicated. For example, the evaluation of a single expression (say as part of an assignment command) could span several steps if the expression involved nested occurrences of a function that is evaluated by calling an external library (or another agent's method). Modeling and reasoning about computations is facilitated by using reasonably large steps rather than subdividing them very finely.

Second, this subdivision would be incompatible with any parallelism within a step. Thus, for example, one could not evaluate unnested occurrences of a library function in parallel. The reason is that any query to the environment by one of the subprocesses would end the current step, and

it is unclear what should then happen to the other parallel subprocesses' computations. Although the restriction (in the present series of papers) to small-step algorithms means that there will not be massive parallelism within a step, a bounded amount of parallelism, such as evaluating several subexpressions in parallel, is permitted and very desirable for modeling and specification (see [1]).

In practice, an algorithm often interacts with its environment in two ways. There is a local environment, often on the same computer or on closely connected machines, and there is a global environment, often out on the internet. Within the local environment, the algorithm has conversations, like method calls and call-backs, that are best viewed as just fulfilling a single request. But while this is going on, the algorithm should maintain a fixed state as far as its long-distance interactions are concerned. It is useful to model this situation by beginning a new step only when the state, as seen by the global environment, changes. Then the conversations with the local environment are mostly intra-step interactions.

This issue is connected with the notion of *transactions* in database theory. The idea here is that many updates are collected and performed all at once. A familiar example is the transfer of money from one bank account to another. The subtraction from the first account and the addition to the second must occur together. If one fails for any reason, then the other should not be executed either. So the state must persist until all the updates are ready, which is likely to be longer than merely from one interaction to the next.

In the example just mentioned, a crucial property of the transaction is that it can be rolled back. If anything goes wrong during the transaction, the state remains as it was initially, not as the already computed updates would have it. Notice, though, that not everything can be rolled back in this sense. The state of the database reverts to the beginning of the step, but any prompts and other messages issued to the user cannot be un-issued. Of course, the user can (and should) be informed that the transaction failed, but nothing can alter the fact that he has seen the previously issued messages. The situation is similar in the computation model we describe in this paper. If a step fails, the state remains unchanged, but any queries already issued cannot be retracted.

**Example 3.1** To give an idea of the aspects of computation modeling that make intra-step interaction desirable we describe a small example reflecting real-world AsmL experience.

Consider the task of painting, i.e., assigning colors to, the parts of some picture. Suppose an algorithm wants to paint two pictures, with these jobs being done in parallel. The actual painting is done by another agent (the operating system) in reponse to a method call from our algorithm. But the picture isn't directly available to our algorithm so it can't send complete instructions for the painting when it issues the method call. Instead, the painting agent, to whom the picture is available, produces callbacks to our algorithm, saying "such and such is in the picture; what color should it be?", and by replying to these, our algorithm gradually specifies how the picture is to be painted. After each reply, the painting agent paints the item in question with the specified color. The entire process of painting a picture should be viewed as a single transaction; if any part of the conversation fails, then all colors should revert to what they were before the conversation began.

When our algorithm issues two parallel calls, to paint two pictures, the resulting conversations should not interrupt each other, for example by ending the step. The best way to model the process is as a single step, with interaction during the step. (The amount of interaction will, of course, be bounded if the algorithm is a small-step one.) □

Another use of intra-step interaction is to model non-determinism. As mentioned in Example 2.5, what is sometimes called a non-deterministic action by an algorithm is really a choice made for the algorithm by the environment. The interaction involved here, namely the algorithm's request for a choice and the environment's reply, is most naturally viewed as occurring within a step of the algorithm. This applies in particular to the creation (or importation from the reserve) of new objects; see Example 2.6.

Finally, we point out a connection with the issue of abstraction levels, as discussed for example in [10]. As in that work, we intend to model algorithms on their natural level of abstraction. There certainly exists a reasonable level of abstraction in which a method call, possible call-backs, and the whole resulting conversation constitute just the evaluation of a single expression. Modeling at that level of abstraction requires intra-step interaction with the environment. To subdivide the steps in order to call this inter-step interaction would be to descend to a lower level of abstraction.

## 3.2 Waiting for replies

When an ordinary algorithm issues a query, it cannot complete its step until it receives a reply. Certainly not all algorithms are like this; why then is it reasonable to restrict attention to ordinary algorithms?

One answer is relevance to practice. Although one can design algorithms that don't wait for replies to all their queries, such situations are uncommon. In particular, programs in AsmL always wait for replies.

Referring back to the discussion, in the preceding subsection, of local and global environments, we recall that intra-step interaction is needed for modeling the algorithm's conversations with its local environment. In those conversations, if the environment failed to provide an answer, the computation normally would hang. An algorithm that sends a message to the global environment may well continue its computation without waiting for an answer, especially since it may expect some delay before the answer arrives, but this is not usually the case for messages to the local environment, such as remote procedure calls.

## 3.3 Ignoring timing

The information from the environment that an ordinary algorithm uses is only the replies to its queries, not, for example, the timing of those replies (or how hard the user bangs the keyboard when typing input). Why is it reasonable to impose this restriction?

Again, there is the justification from practice, including AsmL.

Second, some sorts of additional information could, if one wanted to consider them, be included as part of the replies to a query. For example, nothing prevents replies from including time stamps.

A more technical justification is that it is difficult to simultaneously permit (bounded) parallelism within steps and dependence of the algorithm on the timing of replies. Consider, for example, two parallel subprocesses each of which asks a query and receives a reply from the environment. The relative timing of the two replies, i.e., which came first, is information available to the algorithm but to neither of its two subprocesses. So if this information affects the computation, then it is not clear in what sense the two subprocesses are independent, i.e., in what sense there is genuine parallelism.

For these reasons, we confine attention in the present paper and its sequel to ordinary algorithms. More general algorithms, which can use information

about the timing of replies and which can complete a step even when some queries have not been answered, will be treated in [7].

# 4 Queries and Replies

In the preceding discussion, we have taken for granted that there are clear notions of "query" and "reply". There are, however, some real difficulties with these concepts, and the present section is devoted to discussing these difficulties and resolving them at least for the purposes of the present paper. The central problem is a discrepancy between what looks reasonable when we concentrate on algorithms and what looks reasonable when we concentrate on environments. In this paper, we are interested primarily in analyzing algorithms, so we shall, in the end, work with an algorithm-centered view of queries and replies. But before settling on that view, we must compare it with an environment-centered view, to ensure that what we do here will be applicable in contexts where the environment plays a more central role. An important example of such a context is the study of distributed algorithms, where every agent is part of every other agent's environment.

Let us begin with some basic comments on the queries issued by an algorithm. As the examples above show, there can be many different sorts of queries — prompts, print commands, requests for a non-deterministic choice, paying attention to an input, etc. — so it is reasonable to think of a query as containing a label indicating what type of query it is. Additional labels may also be involved, for example to indicate which part of the algorithm's program issued the query. These labels can be taken from a fixed, finite set that depends only on the algorithm, not on its state.

But a query can also have ingredients that depend on the state. For example, to request the value $f(a)$ of an external function $f$ at an argument $a$, the algorithm should issue a query that contains (at least) the function symbol $f$ (or some other label that specifies $f$) and the argument $a$. The $f$ part is a label as in the previous paragraph; it comes from a finite set fixed by the algorithm. But the $a$ part is different; it comes from the (base set of the) algorithm's state, which is by no means fixed by the algorithm.

The picture emerging from these considerations is that a query can be represented by a finite tuple whose components are either labels or elements of the state. The labels come from a finite set $\Lambda$ determined by the algorithm. Since we treat only small-step algorithms in this paper, there will be a finite

bound, depending only on the algorithm, for the lengths of the tuples that represent queries. The intuition here is that assembling the components of a query is work, so a small-step algorithm can only do a bounded amount of it in any step. For the same reason, the number of queries issued during any step will be bounded.

Two technical points arise here. One is a matter of normalization. We could, for example, insist that, in the tuples representing queries, labels precede state elements. We could also modify our set of labels to include tuples of the original labels and thereby arrange that each query involves only one label. Such normalizations may prove useful in some contexts, but we shall not need them in this paper. We present, in Section 7, a discussion both of such normalization and of what might be called unnormalization, allowing queries of an even more general form than is used here and in the rest of the paper. We shall show that neither normalization nor a moderate degree of unnormalization makes any essential difference.

The second technical point concerns the possibility that the labels given with the algorithm might also be elements of some states. In fact, if we adopt the "abstractness" idea from [10] (also used in [5]), namely that every isomorphic copy of a state is also a state, then such an overlap between states and $\Lambda$ can really occur. This situation could lead to unwanted ambiguity in the queries. The problem can be avoided in several ways. For example, the assumption about isomorphic copies could be restricted to copies that use certain "permissible" entities as elements of the state; members of $\Lambda$ would not be permissible. Alternatively, we could take the components of queries to be elements of the disjoint union $X \sqcup \Lambda$ (where $X$ is the state); "disjoint union" means that the two sets are replaced by disjoint copies. For notational convenience, we shall adopt this second approach, but we shall suppress all mention of the copies. In effect, we write as though $X$ and $\Lambda$ were always disjoint.

How should replies to queries be represented? The simplest answer, and the one that we shall adopt, is that a reply is represented by an element of the algorithm's state. The intuition behind this is that a reply must be something that "makes sense" to the algorithm, and such things ought to be included in the state.

One might argue that, for the sake of symmetry, replies should have the same form as queries, namely tuples. But that situation can easily be accommodated in our picture by (1) adding to the vocabulary names for our labels, so that labels can be represented by state elements, and (2) splitting

any query that asks for a tuple into several queries, each asking for one component of the tuple. Part (2) wouldn't be needed if the states of the algorithm are closed under formation of tuples. Such closure is common for realistic algorithms; after all, programming languages generally provide for tuples. We should also remark that allowing the more general, tuple form of replies would cause no essential change in our work.

In view of these considerations, we adopt the convention that queries for state $X$ are tuples of elements of $X \sqcup \Lambda$ and replies are elements of $X$, where $\Lambda$ is a finite set of labels fixed by the algorithm. This convention describes queries and replies as seen by the algorithm. We must still discuss the environment's point of view, and here an important issue arises, namely abstractness.

Both the theory of ASMs as developed in [8] (and even earlier) and the axiomatic description of algorithms as begun in [10] have among their basic principles the abstractness of states — hence the terminology "abstract state machine." Abstractness means that all the important information about a state must be explicit in its interpretation of function symbols. The specific identity of the elements must never make a difference. In particular, an isomorphic copy of a state $X$ is again a state, and for computational purposes it does not differ from $X$. This principle is reflected in the Abstract State Postulate of [10] and [5].

In our present situation, replacing a state $X$ by another state, say $X'$, with an isomorphism $i : X \cong X'$ would change the queries and the replies. Suppose, for example, that the algorithm requests the value of some external function $f$ at an argument given by some term $t$ (involving no additional external functions, for simplicity). Then $X$ will issue a query that looks like $\langle f, a\rangle$ where $a$ is the value of $t$ in $X$ (and where we've ignored possible additional labels), while $X'$ issues $\langle f, a'\rangle$, where $a' = i(a)$ is the value of $t$ in $X'$. From the environment's point of view, this seems quite unreasonable. The environment would have to look into the states $X$ and $X'$ to find out what those elements $a$ and $a'$ represent, and it would have to formulate its reply as an element of $X$ or of $X'$. Worse yet, the very same tuple could be issued as a query by two isomorphic states but with entirely different meanings. For example, a component of such a tuple could be the element serving as the number 0 in one state and as the number 17 in another. How is the environment to react when presented with such a query? An omniscient environment, knowing what the states are, could handle the problem, but this will never work if, for example, the environment consists simply of the

other agents in a distributed system.

The solution to this difficulty is in two parts — of which the first will be sufficient for our present purpose of setting up suitable postulates to describe ordinary, interactive, small-step algorithms. The second part will, however, be important for making contact with what usually happens in practice.

The first part involves a closer examination of the idea of abstractness that led to the requirement that a state be replaceable by any isomorphic copy. This idea makes good sense when we consider an algorithm in isolation, but, as the preceding discussion indicates, it must be modified in the presence of interaction. The modification, fortunately, is quite simple and natural. What is abstract — what can be replaced by an isomorphic copy — is not the state of the algorithm alone but the entire system, state plus environment. Thus, when we replace a state by an isomorphic copy, we must correspondingly replace the environment in such a way that the entire system is isomorphic to what it was before the replacements. In one of the examples mentioned above, if an element represents 0 in the original state and 17 in the new state, then this element must also undergo the corresponding change in meaning to the environment.

In our axiomatic description of interactive algorithms, the environment will be modeled simply by a function mapping queries to their replies. This will make it easy to describe the change of environment that must accompany an isomorphic change of state so as to produce an isomorphism of the entire system.

The second part of the solution involves looking at what happens in practice, because what we usually regard as the environment of an algorithm — let us call it the proper environment — does not change every time one replaces a state by an isomorphic state. In particular, the environment's view of queries and replies seems quite insensitive to such changes of the state. This discrepancy arises because the proper environment is not really the whole environment. Ordinarily, there is an interface between the algorithm, with its internal representations of queries and answers as described above, and the proper environment with its quite different view of these items. Specifically, the proper environment will have a set $Q$ of queries and a set $R$ of replies. Every state of the algorithm will have an interface consisting of two functions. The first function maps the tuples that (for the algorithm) represent queries to elements in $Q$. The second function maps elements of $R$ to elements of the state, the algorithm's internal representations of answers. For example, we note that functions of this sort form a part of the TCP/IP pro-

tocol, transforming data between a computer's internal representation and a form suitable for internet transmission.

In this picture, an interaction between the algorithm and the proper environment proceeds as follows. The algorithm produces a representation of a query, a tuple of elements from $X \sqcup \Lambda$. The first of the interface functions transforms this representation into a member $q$ of $Q$. The environment recognizes $q$ as a query and responds (if at all) with a reply $r \in R$. The second of the interface functions transforms $r$ into an element of $X$, which the algorithm then uses to continue its step.

Recall, however, that we use the word "environment" to refer to everything that can influence the course of the computation except for the algorithm and the current state. Thus, the interface must be considered a part of the environment. The complete environment contains both the proper environment and the interface. When a state of the algorithm is replaced by an isomorphic copy, the proper environment does not change, but the interface does. Indeed, the interface functions are simply composed with the isomorphism between the two states. In more detail, if $i : X \cong X'$ then the first interface function for $X$, the function $\varphi$ mapping queries for $X$ into $Q$, is replaced with $\varphi \circ i^{-1}$, where we regard $i^{-1}$ as acting on queries by acting on their components in $X'$. Similarly, the interface function $\psi$ mapping $R$ to replies for $X$ is replaced with $i \circ \psi$.

**Remark 4.1** By passing to a lower level of abstraction, one can incorporate the interface functions into the algorithm. That is, they can be programmed. But then, if we replace the state by an isomorphic copy, we usually change the $Q$ and $R$ used by these functions, and so (what remains of) the environment must again adjust to this shift. □

**Remark 4.2** Although it is not immediately relevant to our work in this paper, it seems worth noting that, according to our rather broad view of what constitutes the environment, even the agent executing the algorithm can be regarded as part of the algorithm's environment. Certainly various details about this agent — how fast it works, whether it performs parallel operations by interleaving sequential ones and, if so, how to interleave, etc. — are best viewed as part of the environment. □

# 5 Postulates

In this section, we introduce postulates intended to describe ordinary, interactive, small-step algorithms. Recall that "ordinary" means that all queries issued by the algorithm during a step must be answered before the algorithm can complete its step and that only the answers, not their timing, are relevant. More precisely, the algorithm's actions depend only on its program, its state, and previous answers from the environment. If the environment refuses to respond to a query, then the algorithm hangs, i.e., there is no transition and no next state (not even a repetition of the preceding state).

Our postulates are based on the postulates for (non-interactive) small-step algorithms in [10]. The new material in the present postulates incorporates the view of interaction presented in Section 4. We also allow for the possibility that there is, in certain situations, no transition, either because there are unanswered queries or because of some error conditions. Also, we have modified the postulates to take trivial updates into account. Finally, we have reorganized some of the postulates. For example, the transition function, mentioned in the very first postulate in [10], now occurs later because it involves queries and replies (the transition performed by an algorithm usually depends on the replies it received) and not only states.

We do not repeat here the justifications given in [10] for those parts of the postulates that do not involve the environment, undefined transitions, or trivial updates. When, in our discussions of the following postulates, we say that something is (essentially) taken from [10], this should be interpreted as referring to [10] for further explanation and justification.

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[1] subset $\mathcal{I} \subseteq \mathcal{S}$ of *initial states*,

[1] In [10], $\mathcal{I}$ and $\mathcal{S}$ were not required to be nonempty. This requirement seems, however, to be an essential part of the concept of algorithm. An algorithm without an initial state couldn't be run, so is it really an algorithm? We therefore add "nonempty" to the postulate here.

- a finite vocabulary $\Upsilon$ such that every $X \in \mathcal{S}$ is an $\Upsilon$-structure, and
- a finite set $\Lambda$ of *labels*.

The first two items in this postulate, $\mathcal{S}$ and $\mathcal{I}$, are as in the Sequential Time Postulate of [10]; our third item is part of the Abstract State Postulate of [10]. The fourth item, $\Lambda$, is new and is, as discussed above, used in forming (internal representations of) queries.

**Remark 5.1** We shall require $\mathcal{S}$ to be closed under isomorphisms, so it is really a proper class rather than a set. Except in this remark, we ignore the distinction between sets and classes, as it will be irrelevant to our work. □

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

We adopt the following conventions concerning vocabularies and structures. These come mostly from [8] with some minor modifications to agree with [9] and [10].

**Convention 5.2**
- A vocabulary $\Upsilon$ consists of function symbols with specified arities.
- Some of the symbols in $\Upsilon$ may be marked as *static*, and some may be marked as *relational*. Symbols not marked as static are called *dynamic*.
- Among the symbols in $\Upsilon$ 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 $\Upsilon$-structure, the interpretations of `true`, `false`, and `undef` are distinct.
- In any $\Upsilon$-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 $\{\texttt{true}, \texttt{false}\}$.

□

Since relational symbols are interpreted as taking only the values `true` and `false`, they behave like the predicate symbols commonly used in first-order logic. Partial functions are treated as total functions that take the value `undef` outside the intended domain of definition.

**Definition 5.3** 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 $\Lambda$ 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 $\Lambda$ are always disjoint.

Our definition of potential queries and potential replies incorporates our decision, discussed in Section 4, to adopt the point of view internal to the algorithm. Thus, our queries and replies are what were previously called internal representations.

**Definition 5.4** An *answer function* is a partial map from potential queries to potential replies.

An answer function $\alpha$ 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 shall often need to refer to the restriction of an answer function $\alpha$ to a subset $Z$ of its domain; we use the standard notation $\alpha \restriction Z$ for this restriction. We also write $\beta \subseteq \alpha$ to indicate that $\beta$ is a restriction of $\alpha$; this notation agrees with the ordinary set-theoretic meaning of $\subseteq$ if we regard functions as sets of ordered pairs.

**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.

The intuitive meaning of $\xi \vdash_X q$ is that if, in state $X$, the algorithm has issued the queries in $\mathrm{Dom}(\xi)$ and received the answers given by $\xi$ then it will issue query $q$.

This view of causes implicitly involves our decision to treat only ordinary algorithms. Consider, for example, an algorithm that issues a query $q_1$ and, strictly later but without necessarily waiting for a reply to $q_1$, issues $q_2$. No causality relation exactly models this. Indeed, if we let the empty answer function cause $q_2$, then the fact that $q_2$ is issued strictly later than $q_1$ is lost. If, on the other hand, we take the causes of $q_2$ to be the answer functions that have $q_1$ in their domains, then we lose the fact that $q_2$ could be issued before the reply to $q_1$ is received. Thus, our causality relations cannot capture this sort of timing information and is therefore inadequate for dealing with general algorithms. It is, however, adequate for ordinary algorithms. The situation we have described, where $q_2$ is issued strictly after $q_1$ but without waiting for an answer to $q_1$, though certainly possible in an algorithm, is not possible in an ordinary algorithm, because the algorithm would make use of more than just the program, the state, and answers received from the environment. To see this, consider the situation at two moments, first just before the algorithm issues $q_1$ and second just after $q_1$ is issued (but before it is answered). The program, the state, and the answers received from the environment are exactly the same at both moments. Yet the algorithm can issue $q_2$ at the second but not at the first moment. Thus, such an algorithm is not ordinary. The limitation imposed by ordinariness on the information an algorithm can use is just what is needed to allow our simple representation of causality as a relation between (a state and) answer function and a query. If we were dealing with non-ordinary algorithms or if we were looking into the timing of queries and replies (beyond logical dependences), then we would need a more general notion of causality, where a cause can consist of not only an answer function but also a set of unanswered queries.

**Definition 5.5** 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 $\alpha$ with the following properties:

- For all answer functions $\xi$ 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 $Z = \mathrm{Dom}(\alpha)$.

The intuition behind this definition is that a context describes the complete interaction between the algorithm and its environment during one step. We think of $\mathrm{Dom}(\alpha)$ as the set of queries issued by the algorithm, and $\alpha(q)$ is the environment's answer to query $q$. Intuitively, the first requirement in this definition says that $\mathrm{Dom}(\alpha)$ is closed under causality with respect to $\alpha$. That is, if $\alpha$ includes information $\xi$ that causes the algorithm to issue query $q$, then $q$ has in fact been issued and answered. (The "and answered" clause here is part of what distinguishes our present topic, ordinary algorithms, from general algorithms that don't require answers to all their queries.)

The second part of the definition reflects that $\mathrm{Dom}(\alpha)$ is the smallest set with this closure property. Intuitively, it means that no query is issued without a cause. To connect this intuition with the formulation in the definition, suppose $Z$ were a counterexample to the second part of the definition, and consider some query $q \in \mathrm{Dom}(\alpha) - Z$. If the algorithm had a cause to issue this query, say $\xi \vdash_X q$ and $\xi \subseteq \alpha$, then, by the assumed property of $Z$, there must be a $q' \in \mathrm{Dom}(\xi) - Z$. If the algorithm had a cause to issue $q'$, say $\xi'$, then we get some $q'' \in \mathrm{Dom}(\xi') - Z$. Repeating the argument, we get an infinite regress (possibly a loop) of causes, which means that $q$ wasn't genuinely caused in the first place.

The second requirement in the definition can also be understood as an induction principle. To prove a property for all elements of $\mathrm{Dom}(\alpha)$, it suffices to prove it for an arbitrary $q \in \mathrm{Dom}(\alpha)$ under the assumption that the property holds for all elements in the domain of some $\xi \subseteq \alpha$ that causes $q$.

Like any induction principle, this can be rewritten as a "minimal element" principle by considering the complement of $Z$. That principle reads: If $\varnothing \neq Z \subseteq \mathrm{Dom}(\alpha)$ then there exist $q \in Z$ and $\xi \subseteq \alpha$ such that $\xi \vdash_X q$ and $Z \cap \mathrm{Dom}(\xi) = \varnothing$. This principle can be understood intuitively by observing that in any nonempty subset $Z$ of the set $\mathrm{Dom}(\alpha)$ of issued queries there should be a query $q$ that is issued first (i.e., at least as early as any other member of $Z$). Any cause for this $q$, consisting of strictly earlier queries and their replies, must not involve any queries from $Z$. Since $q$ should be caused by some restriction $\xi$ of $\alpha$, this $\xi$ will be as required in the minimal element principle.

Yet another way to view the definition is that the domain of a context $\alpha$ is generated by the causality relation, in the sense that it is the closure (also called the least fixed point) of a certain monotone operator based on the

causality relation. Specifically, for any answer function $\alpha$, define a monotone operator $\Gamma_{X,\alpha}$, or just $\Gamma_\alpha$ when $X$ is understood, on sets of potential queries by

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

The idea behind this definition is that if a set $Z$ of queries has been issued and if the answers were as given by $\alpha$ (i.e., $\alpha \restriction Z$), then the algorithm will issue the queries in $\Gamma_\alpha(Z)$.

**Definition 5.6** Let $\Gamma$ be a monotone operator on subsets of a set $S$. That is, if $Y \subseteq Z \subseteq S$ then $\Gamma(Y) \subseteq \Gamma(Z)$. Its iteration is defined to be the sequence of sets $\Gamma^n$ given by

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

In this generality, the sequence continues transfinitely (so the $n$ above can be any ordinal number) with

$$\Gamma^\lambda = \bigcup_{\nu<\lambda} \Gamma^\nu$$

for limit ordinals $\lambda$, but we shall never need transfinite iterations in the present paper. The least fixed point, the common value of $\Gamma^\nu$ for all sufficiently large $\nu$, is denoted by $\Gamma^\infty$.

All the operators $\Gamma$ used in this paper will have $\Gamma^\infty = \Gamma^n$ for sufficiently large finite $n$.

Recall from the general theory of fixed points that $\Gamma^\infty$ is the smallest (with respect to $\subseteq$) fixed point and also the smallest pre-fixed point of $\Gamma$. That is, it is the smallest $Z$ satisfying $\Gamma(Z) = Z$ and also the smallest $Z$ satisfying $\Gamma(Z) \subseteq Z$.

**Lemma 5.7** *Let $\alpha$ 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 $\subseteq \alpha$.*

*Proof* Suppose first that ${\Gamma_\alpha}^\infty \subseteq \mathrm{Dom}(\alpha)$ and let $\beta = \alpha \restriction {\Gamma_\alpha}^\infty$. So $\mathrm{Dom}(\beta) = {\Gamma_\alpha}^\infty$. We must show that $\beta$ is a context. The first requirement of the definition of context means that $\Gamma_\alpha({\Gamma_\alpha}^\infty) \subseteq {\Gamma_\alpha}^\infty$, which is true as ${\Gamma_\alpha}^\infty$ is a fixed point of $\Gamma_\alpha$. For the second requirement, suppose $Z$ is as there. To show $Z = {\Gamma_\alpha}^\infty$, it suffices to show that $\Gamma_\alpha(Z) \subseteq Z$, for the least fixed point

${\Gamma_\alpha}^\infty$ is also the least pre-fixed point. But the hypothesis on $Z$ in the second requirement says exactly that $\Gamma_\alpha(Z) \subseteq Z$, so this part of the proof is complete.

Conversely, suppose some subfunction $\beta = \alpha \restriction Q$ of $\alpha$ is a context. We show that its domain $Q$ is ${\Gamma_\alpha}^\infty$ and, in particular, ${\Gamma_\alpha}^\infty \subseteq \mathrm{Dom}(\alpha)$. The first clause in the definition of "context" ensures that $\Gamma_\alpha(Q) \subseteq Q$. As ${\Gamma_\alpha}^\infty$ is the smallest set with this closure property, it follows that ${\Gamma_\alpha}^\infty \subseteq Q$. To establish the reverse inclusion, we apply the second clause of the definition of context with $Z = {\Gamma_\alpha}^\infty$. The hypothesis of that clause is satisfied, since $\Gamma_\alpha({\Gamma_\alpha}^\infty) \subseteq {\Gamma_\alpha}^\infty$. So the second clause gives ${\Gamma_\alpha}^\infty = Q$, as required. □

**Corollary 5.8** *If $\alpha$ and $\beta$ are two distinct contexts for the same state, then there is some $q \in Dom(\alpha) \cap Dom(\beta)$ with $\alpha(q) \neq \beta(q)$.*

*Proof* Otherwise, $\alpha \cup \beta$ would be an answer function that includes two distinct contexts, contrary to the lemma. □

To understand the intuition behind the lemma, think of $\alpha$ as describing the answers that the environment would give if the appropriate queries were issued. That is, if the algorithm were to issue $q \in \mathrm{Dom}(\alpha)$ then the environment would reply with $\alpha(q)$, but if the algorithm were to issue a query not in $\mathrm{Dom}(\alpha)$ then the environment would not reply.

In the "good" case of the lemma, where ${\Gamma_\alpha}^\infty \subseteq \mathrm{Dom}(\alpha)$, the algorithm will issue all the queries in ${\Gamma_\alpha}^\infty$, the environment will reply to all of them, and no additional queries will be issued (because ${\Gamma_\alpha}^\infty$ is closed under causality, i.e., is a pre-fixed point of $\Gamma_\alpha$). The interaction between the algorithm and its environment is complete, and a state transition can occur.

In the "bad" case, where ${\Gamma_\alpha}^\infty \not\subseteq \mathrm{Dom}(\alpha)$, the algorithm will issue at least one query to which the environment fails to reply. In this case, this step of the computation hangs; there is no transition.

The next postulate will describe the transition from one state to the next, in terms of the updates performed by the algorithm during this step. We begin by specifying the format of updates.

**Definition 5.9** A *location* in a state $X$ is a pair $\langle f, \vec{a} \rangle$ where $f$ is a dynamic function symbol from $\Upsilon$ and $\vec{a}$ 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 $f_X(\vec{a})$. 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 $\langle f, \vec{a}, b \rangle$ or $\langle f, a_1, \ldots, a_n, b \rangle$ instead of $(\langle f, \vec{a} \rangle, b)$ or $(\langle f, \langle a_1, \ldots, a_n \rangle \rangle, b)$.

The intended meaning of an update $\langle f, \vec{a}, b \rangle$ 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 $\vec{a}$ is $b$. This intention is formalized in the following postulate.

**Update Postulate:** For any state $X$ and any context $\alpha$ for $X$, the algorithm provides an *update set* $\Delta_A^+(X, \alpha)$. In addition, it either *fails* or produces a *next state* $\tau_A(X, \alpha)$, but not both. If there is a next state $X' = \tau_A(X, \alpha)$, then it

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

In [10], the existence of a transition function was part of the Sequential Time Postulate, and the unchangeability of the base set was part of the Abstract State Postulate. The present Update Postulate incorporates several additional ingredients, which deserve some comments.

First, the input of the transition function is no longer simply a state; it also includes a context. The transition performed by the algorithm at the end of a step can (and usually does) depend not only on the previous state but also on the information received from the environment during that step.

Second, we have incorporated the idea that there is no transition if some queries are issued but remain unanswered. Thus, the domain of the transition function contains only pairs $(X, \alpha)$ where $\alpha$ is a context, i.e., where all the appropriate queries have been issued and answered.

Third, having included in vocabularies the possibility of marking some symbols as static, we make this marking effective by insisting that the interpretations of static symbols cannot change[2] during a state transition. Indeed, since locations begin, by definition, with a dynamic function symbol, only

[2]It was intended in [10] that the logic names be static in this sense, but the formulation there was slightly weaker.

these symbols can be the subject of updates, and so only these can be changed by the algorithm in the transition from one state to the next. If we were also considering changes of state produced by the environment, between the algorithm's steps, then these changes would also be required to leave the base set and the static functions unchanged.

Fourth, our $\Delta^+$ is not quite the same as the $\Delta$ of [10], which is why we added + to the notation. We shall occasionally need to refer to the old $\Delta$, modified to incorporate interaction with the environment, so we define it here and comment on its relation to $\Delta^+$.

**Definition 5.10** If $X' = \tau_A(X, \alpha)$ is defined, then $\Delta_A(X, \alpha)$ is the set of those updates $\langle f, \vec{a}, b\rangle$ such that $f_{X'}(\vec{a}) = b \neq f_X(\vec{a})$.

Thus, $\Delta_A(X, \alpha)$ describes the changes in the dynamic functions between $X$ and $X'$. Inspection of the definition and the Update Postulate yields the following observation.

**Lemma 5.11** *If $\tau_A(X, \alpha)$ is defined, then $\Delta_A(X, \alpha)$ is the set of non-trivial updates in $\Delta^+_A(X, \alpha)$.*

It may seem strange, particularly to a reader acquainted with [10] and [5], to work with $\Delta^+$ rather than $\Delta$. Why should we pay attention to trivial updates that have no effect on the transition computed by the algorithm and therefore have no effect on a run of the algorithm? The answer is that, although trivial updates are irrelevant when the algorithm runs in isolation, they become significant when the algorithm is part of a larger parallel or distributed computation. The reason is that a trivial update can, in such a situation, clash with an update produced by another part of the computation. Since the occurrence of a clash is important, so is the trivial update that caused it. So a major reason for introducing $\Delta^+$ is to prepare the way for future papers in which small-step algorithms occur as agents in larger computational processes. In fact, we shall already need $\Delta^+$ in the sequel to this paper, in order to define parallel composition as an operation on algorithms.

A fifth difference from the set-up in [10] is that we allow explicit failure as an alternative to a state transition, even when $\alpha$ is a context for $X$ so that the algorithm should be ready to complete its step in the situation $(X, \alpha)$. Such failures can arise in several ways. One is that the environment has given

an inappropriate reply, which the algorithm cannot use. For example, if the algorithm asks the environment to (non-deterministically) choose a member from a certain set and the environment's answer is not in that set, then the algorithm may fail to continue its computation.

Another possible reason for failure is that the updates in $\Delta_A^+(X, \alpha)$ contradict each other. We say that $\Delta_A^+(X, \alpha)$ *clashes* if it contains two different updates of the same location, $\langle f, \vec{a}, b\rangle$ and $\langle f, \vec{a}, b'\rangle$ with $b \neq b'$. In this case, the algorithm must fail in the situation $(X, \alpha)$; it cannot have a next state $X'$ because $f_{X'}(\vec{a})$ would be subject to contradictory requirements.

**Remark 5.12** There is an essential difference between these examples of failure, where $\tau_A(X, \alpha)$ is undefined while $\alpha$ is a context, and the situations where $\tau_A(X, \alpha)$ is (necessarily) undefined because $\alpha$ is not a context. In fact, there are three situations to be distinguished .

1. $\alpha$ properly includes a context $\beta$. In this case, there may be a transition to $\tau_A(X, \beta)$, which is what we would ordinarily be interested in. $\alpha$ contains some extra queries (and their answers), which the algorithm would not actually issue. This situation would be of interest only if there were some reason for considering those extra queries, for example if another algorithm were using the same $\alpha$.

2. $\alpha$ includes no context. In this case, there are queries that are caused by subfunctions of $\alpha$ but have not yet been answered — perhaps not even asked. So the algorithm, being an ordinary one, cannot go ahead with a transition. But the situation may change as more answers are received (and possibly more queries issued), leading to enough information for the algorithm to complete its step. Formally, this means that $\alpha$ may be included in some context $\beta$ and $\tau_A(X, \beta)$ may be defined.

3. $\alpha$ is a context but $\tau_A(X, \alpha)$ is nevertheless undefined. In this case, the algorithm's interaction with the environment seems to have gone well, in that all the right queries have been asked and answered. Nevertheless, something has gone wrong. Perhaps the algorithm has produced clashing updates. Perhaps the environment has given an inappropriate answer, which the algorithm cannot use. Perhaps there is some other reason for the algorithm to fail. In any case, there are no prospects for correcting the problem. No further queries are to be asked, so there is nothing the environment can do about the situation, nor does the

algorithm provide a way out. We are dealing not with a delay, as in (2), but with a crash.

It is useful to distinguish case (3) from the other cases, and we have done so by having the algorithm explicitly fail in those situations where $\alpha$ is a context for $X$ but there is no transition. Another way to achieve the same objective would be to introduce a new object, say "Error", as a possible value of $\tau_A$, to require $\tau_A(X, \alpha)$ to be defined whenever $\alpha$ is a context for $X$, and to let $\tau_A(X, \alpha) =$ Error if $\tau_A(X, \alpha)$ would be undefined in our set-up.

This would be useful, for example, if we were dealing with runs in which either the algorithm or the environment can be responsible for a state transition. In such a case, a run that ends with a transition from a certain state $X$ to Error would differ from the same run minus the transition to Error, i.e., ending at $X$. The difference is that the latter run could be extended by an environmental transition to a new state $Y$ (followed perhaps by transitions caused by the algorithm), whereas the former run is definitely ended. Another way to express the difference is that, in the former run the algorithm attempted to progress from state $X$ and ran into an insurmountable problem, whereas in the latter case, the algorithm made no such attempt.

Yet another way to achieve the same goal, instead of introducing "Error", would be to explicitly indicate, in a run, all those places where the algorithm attempted a transition (successfully or perhaps, at the end of a run, unsuccessfully). This approach looks especially reasonable if one wants to indicate, throughout a run, which transitions were caused by the environment and which are the work of the algorithm. □

**Definition 5.13** 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 $\Lambda$ 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 $\xi$ 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 $\alpha$ is a context for $X$, then
  - $i[\Delta^+(X, \alpha)] = \Delta^+(Y, i \circ \alpha \circ i^{-1})$, and
  - the algorithm fails in $(X, \alpha)$ if and only if it fails in $(Y, i \circ \alpha \circ i^{-1})$.

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. It follows further that $i[\Delta(X, \alpha)] = \Delta(Y, i \circ \alpha \circ i^{-1})$.

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

**Convention 5.14** 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. □

The first two parts of the Isomorphism Postulate were in the Abstract State Postulate of [10]. So was the fourth part, formulated in terms of the transition function, without contexts, and without the possibility of explicit failure. So the idea behind this fourth part comes directly from [10]; we have merely adapted it to the present framework. The third part of the postulate is new, but it is the natural thing to require of the causality relations. The whole Isomorphism Postulate can be summarized as "isomorphisms preserve everything".

The last part of the Isomorphism Postulate tacitly uses the fact that $i \circ \alpha \circ i^{-1}$ is a context for $Y$. This fact follows trivially from the earlier parts of the Isomorphism Postulate, which ensure that the isomorphism $i$ preserves everything used in the definition of "context."

**Remark 5.15** In the last two parts of the Isomorphism Postulate, the answer function $\alpha$ is replaced by $i \circ \alpha \circ i^{-1}$, and similarly for $\xi$, when $X$ is replaced by $Y$. This formalizes the fact, discussed in Section 4, that when we replace a state by an isomorphic copy, we must correspondingly adjust the environment, so that the entire system, state plus environment, is isomorphic to what it was before. Since we model environments by answer functions, this adjustment of the environment is formalized as a change of the answer function induced by the isomorphism between the states.

In more detail, suppose that $i : X \cong Y$, that $\alpha$ is an answer function for $X$, that $q \in \mathrm{Dom}(\alpha)$ is a query for $X$, to which $\alpha$ gives the reply $r = \alpha(q)$. Then the corresponding query and reply for $Y$ are $i(q)$ and $i(r)$, respectively. So the adjusted environment (i.e., answer function for $Y$) $\alpha'$ should produce the answer $i(r)$ for the query $i(q)$. That is, $\alpha'(i(q)) = i(r) = i(\alpha(q))$. Since this is to happen for all $q$, we have $\alpha' \circ i = i \circ \alpha$, or equivalently $\alpha' = i \circ \alpha \circ i^{-1}$. This explains the $i \circ \alpha \circ i^{-1}$ in the Isomorphism Postulate.

It may be instructive to analyze the situation in terms of the interface functions described in Section 4. As in that discussion, let $Q$ and $R$ be the sets of queries and replies as seen by the proper environment. So a state $X$ of the algorithm comes with two interface functions. One of these, say $\varphi_X$, maps internal representations of queries (i.e., potential queries in the sense of this section) to members of $Q$. The other, say $\psi_X$, maps members of $R$ to internal representations of replies (i.e., potential replies in the sense of this section). The proper environment's answers are given by a partial function $\theta : Q \to R$. Our answer functions combine $\theta$ with the interfaces to produce a partial function from potential queries to potential replies, namely

$$\alpha = \psi_X \circ \theta \circ \varphi_X.$$

Recall that the interface functions for the isomorphic states $X$ and $Y$ are related via the isomorphism $i$ as

$$\varphi_Y = \varphi_X \circ i^{-1} \qquad \text{and} \qquad \psi_Y = i \circ \psi_X.$$

Now the same function $\theta$ as above produces for the state $Y$ the answer function

$$\begin{aligned} \alpha' &= \psi_Y \circ \theta \circ \varphi_Y \\ &= (i \circ \psi_X) \circ \theta \circ (\varphi_X \circ i^{-1}) \\ &= i \circ \alpha \circ i^{-1}. \end{aligned}$$

This again accounts for the $i \circ \alpha \circ i^{-1}$ in the Isomorphism Postulate. $\square$

We record for future reference an immediate consequence of the Isomorphism Postulate.

**Lemma 5.16** *Suppose $i : X \cong Y$ is an isomorphism of states and $\alpha$ 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 $\Gamma$ 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 $\alpha$ and all states.
- There is a bound, depending only on $A$, for the cardinalities $|\mathrm{Dom}(\alpha)|$ for all contexts $\alpha$ 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,
  - $\alpha$ 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 $\alpha$ is a context for $X$, then $\Delta^+(X, \alpha) = \Delta^+(X', \alpha)$ and, if the algorithm fails for either of $(X, \alpha)$ and $(X', \alpha)$, then it also fails for the other.

The part of this postulate referring to $\Delta^+$ is very similar to the Bounded Exploration Postulate of [10]. It says that the work done by the algorithm in producing updates involves looking only at the values of the finitely many terms in $W$. In [10], these terms were closed terms; here they can contain variables to be replaced by the values given by the environment's replies.

This simply means that the algorithm is permitted to use those replies in computing its updates. We also accommodate the possibility that the algorithm fails to make a transition in one of $(X, \alpha)$ and $(X', \alpha')$; it must then also fail in the other. As in [10], we shall call a set $W$ as in the postulate a *bounded exploration witness* for the algorithm $A$.

Let us amplify a bit the explanation of the role of the variables that can occur in the terms in $W$. As indicated above, these variables serve as place holders for replies to be obtained from the environment. There is, however, no specific connection between the variables and the replies (or the queries leading to the replies). That is, any reply to any query can be used as the value of any variable. The reason for allowing so much freedom in the assignment of values to variables is that the restrictions that one might think of imposing, requiring a particular variable to represent the reply to a particular query, cannot be formulated without knowing what the queries are, and that depends on the state and the replies to previous queries. Indeed, infinitely many queries are possible if we consider all possible states. The bounded exploration witness, on the other hand, must depend only on the algorithm, not on the state or the environment. Thus, the freedom that we allow, in assigning to the variables arbitrary elements of Range$(\alpha)$, is necessary, because we cannot formulate, on the basis of the algorithm alone, reasonable restrictions on that freedom.

Notice that this freedom, by allowing very diverse assignments of values to variables, makes the requirements on $X$ and $X'$ in the Bounded Work Postulate, specifically the requirement that elements of $W$ have the same value, rather restrictive. This, in turn, makes the postulate itself rather weak — agreement on causality and updates is demanded only for a restricted collection of pairs $X, X'$. Therefore, when we prove in the sequel to this paper that all algorithms satisfying our postulates are equivalent to ASMs, that theorem will be stronger, i.e., will apply to a broader class of algorithms, than it would without the freedom allowed in assigning values to variables in the Bounded Work Postulate.

The part of the Bounded Work Postulate referring to $\vdash$ says that the computation of queries to issue during a step is subject to the same boundedness requirement as the computation of the transition at the end of the step. These first two items in the postulate say that only a bounded amount of querying occurs in a step, both as regards the number of queries and as regards their complexity, i.e., their length as tuples. As the name suggests, all parts of the Bounded Work Postulate are intended to formalize the essen-

tial property of small-step algorithms that there is a uniform bound on the amount of work, of any sort, that the algorithm can do in one step.

The last sentence of this postulate tacitly uses that $\alpha$ is also a context for $X'$. We now prove that this tacit assumption is correct. It is easy to verify the first clause in the definition of "context." We are given $\xi$ and $q$ such that $\xi \vdash_{X'} q$ and $\xi \subseteq \alpha$, and we must show that $q \in \mathrm{Dom}(\alpha)$. But the preceding part of the Bounded Work Postulate assures us that $\xi \vdash_X q$, so the desired conclusion follows since $\alpha$ is a context for $X$.

The verification of the second clause in the definition of "context" is no harder. We are given a set $Z \subseteq \mathrm{Dom}(\alpha)$ such that

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

and we must show that $Z = \mathrm{Dom}(\alpha)$. Again, the preceding part of the Bounded Work Postulate allows us to replace $\vdash_{X'}$ with $\vdash_X$, and then the desired conclusion follows from the assumption that $\alpha$ is a context for $X$.

The postulates presented above determine the notion of algorithm to be studied in this paper and its sequel.

**Definition 5.17** An *ordinary, interactive, small-step algorithm* is any entity satisfying the States, Interaction, Update, Isomorphism, and Bounded Work Postulates.

We shall often use without explicit mention the following consequence of the Bounded Work Postulate.

**Lemma 5.18** *There is a finite bound, depending only on the algorithm, on the lengths of the iterations ${\Gamma_\alpha}^n$ for all contexts $\alpha$ for all states $X$.*

*Proof* The iteration cannot have more steps than the number of elements in the final set ${\Gamma_\alpha}^\infty$. But for contexts, this final set is $\mathrm{Dom}(\alpha)$, and its cardinality is bounded because of the second part of the Bounded Work Postulate. □

As a consequence of this lemma, we shall never have to deal with transfinite iterations of the operators $\Gamma_\alpha$.

To relate our work here to that in [10] and [5], it is useful to know that the Bounded Work Postulate, formulated here in terms of $\Delta^+$, implies the earlier formulation using $\Delta$. This will not necessarily be true for the same bounded

exploration witness, but it can always be achieved by enlarging the bounded exploration witness. The proof requires knowing that the elements of a state $X$ that occur in updates of $\Delta^+(X, \alpha)$ are among the values of the terms in the bounded exploration witness, and this in turn requires knowing similar information for queries. We therefore begin by establishing this preliminary information, which will also play a role in the sequel paper, in the proof that our algorithms can be simulated by ASMs. The essential idea here is the same as in [10, Lemma 6.2], but the proof is more complicated because we need to take contexts $\alpha$ and their fragments into account.

**Definition 5.19** For any answer function $\alpha$, we write $\alpha_n$ for $\alpha \restriction {\Gamma_\alpha}^n$. (Recall that ${\Gamma_\alpha}^n$ is the $n^{\text{th}}$ stage of the iteration of the operator $\Gamma_\alpha$.)

Thus, in particular, $\alpha_0$ is the empty function. If $\alpha$ is a context and if the iteration of $\Gamma_\alpha$ stabilizes after $n$ steps, then $\alpha_n = \alpha \restriction {\Gamma_\alpha}^\infty = \alpha$.

It will be convenient to normalize our bounded exploration witnesses as follows.

**Convention 5.20** A bounded exploration witness $W$ is always assumed to be closed under subterms and to contain a variable. □

Notice that, if $W$ satisfies the requirements in the postulate for a bounded exploration witness, then so does any finite set of terms that includes $W$. Closing $W$ under subterms and adding a variable certainly preserve finiteness, so our convention entails no loss of generality.

**Definition 5.21** Let $X$ be a state and $\xi$ an answer function for $X$. An element of $X$ is *critical for* $\xi$ if it is the value of some term in $W$ for some assignment of values in Range($\xi$) to its variables.

We note that, since $W$ contains a variable, all elements of Range($\xi$) are critical for $\xi$. Note also that, if $\xi \subseteq \eta$ then anything critical for $\xi$ is also critical for $\eta$.

**Proposition 5.22** *Let $X$ be a state and $\alpha$ a context for $X$. Suppose we have $n$, $\xi$ and $q$ such that $\xi \subseteq \alpha_n$ and $\xi \vdash q$. Then all the components in $X$ of $q$ are critical for $\alpha_n$.*

*Proof* We proceed by induction on $n$. The proof for the base case $n = 0$ is essentially a subproof of the proof of the induction step, so we give only the latter.

So consider $n$, $\xi$, and $q$ as in the proposition, and assume the proposition is true for smaller values of $n$. Let $x \in X$ be a component of $q$. Let $Y$ be the structure obtained from $X$ by replacing the one element $x$ by a new (i.e., not in $X$) element $y$. Since $Y$ is isomorphic to $X$, it is a state.

Could $\xi$ fail to be an answer function for $Y$? One possibility is that one of its reply values is not an element of $Y$. That means that this value is $x$, since all other members of $X$ are in $Y$. But then $x$ is critical, as required.

The other possibility is that some $q' \in \mathrm{Dom}(\xi)$ is not a query for $Y$, i.e., that one of its components in $X$ is not in $Y$ and is therefore $x$. But from $q' \in \mathrm{Dom}(\xi)$ and $\xi \subseteq \alpha_n = \alpha \restriction {\Gamma_\alpha}^n$ and the definition of $\Gamma_\alpha$, we infer that there is $\xi' \subseteq \alpha_{n-1}$ such that $\xi' \vdash q'$. Then, by induction hypothesis, all the components of $q'$, including in particular $x$, are critical for $\alpha_{n-1}$ and therefore also for $\alpha_n$. So we again have the desired conclusion.

Thus, if $\xi$ fails to be an answer function for $Y$, then we have the desired result, namely that $x$ is critical. It remains to consider the case that $\xi$ is an answer function for $Y$.

Could a term in $W$ have different values in $X$ and in $Y$ when its variables are given the same values in $\mathrm{Range}(\xi)$? Certainly not if the term is itself a variable, so suppose the term is $f(t_1, \ldots, t_n)$. We can arrange, by passing to subterms if necessary (since $W$ is closed under subterms), that each $t_i$ has the same value in $Y$ as in $X$. But then, thanks to the definition of $Y$, the only way $f(t_1, \ldots, t_n)$ can have different values in the two states is for its value to be $x$ in $X$ and $y$ in $Y$. Then $x$ is critical, as required.

So it remains only to consider the case that $\xi$ is an answer function for both $X$ and $Y$ and each term in $W$ gets the same values in $X$ and $Y$ when the variables get the same values in $\mathrm{Range}(\xi)$. Then, because $W$ is a bounded exploration witness and because $\xi \vdash_X q$, we have $\xi \vdash_Y q$. In particular, $q$ is a potential query for $Y$. But this is absurd, since one of the components of $q$, namely $x$, is not in $Y \sqcup \Lambda$. □

**Proposition 5.23** *Let $X$ be a state and $\alpha$ a context for $X$. For any update $\langle f, \vec{a}, b\rangle \in \Delta^+(X, \alpha)$, all the components of $\vec{a}$ as well as $b$ are critical for $\alpha$.*

*Proof* Let $x$ be either a component of $\vec{a}$ or $b$. As in the preceding proof, let $Y$ be the state obtained from $X$ by replacing $x$ by a new element $y$.

If $x$ is among the values of $\alpha$ then it is critical for $\alpha$ as required. So we assume from now on that $x \notin \text{Range}(\alpha)$ and thus $\text{Range}(\alpha) \subseteq Y$.

Furthermore, every element $q$ of $\text{Dom}(\alpha)$ is, since $\alpha$ is a context, caused by some $\xi \subseteq \alpha$. Applying Proposition 5.22 with $n$ large enough so that $\alpha_n = \alpha$, we obtain that all of $q$'s components in $X$ are critical for $\alpha$. So if $x$ is such a component then $x$ is critical for $\alpha$, as required. So we assume from now on that $x$ is not a component of any query in $\text{Dom}(\alpha)$. Thus, $\text{Dom}(\alpha)$ consists of potential queries for $Y$. Combining this with the result of the preceding paragraph, we have that $\alpha$ is an answer function for $Y$ as well as for $X$.

As in the proof of Proposition 5.22, we see that, if some term in $W$ has different values in $X$ and $Y$ when its variables are given the same values in $\text{Range}(\alpha)$, then the smallest such term has value $x$ in $X$, and so $x$ is critical for $\alpha$, as required. Therefore, we assume from now on that each term in $W$ has the same values in $X$ and $Y$ when the variables get the same values in $\text{Range}(\xi)$. Since $W$ is a bounded exploration witness and $\alpha$ is a context for $X$, it follows that $\alpha$ is also a context for $Y$ and $\Delta^+(X, \alpha) = \Delta^+(Y, \alpha)$. In particular, $\langle f, \vec{a}, b\rangle \in \Delta^+(Y, \alpha)$, and so $\langle f, \vec{a}, b\rangle$ is an update for $Y$. But this is absurd, as this update involves $x$, which is not in $Y$. □

The next proposition is the same as part of the Bounded Work Postulate except that $\Delta$ has replaced $\Delta^+$.

**Proposition 5.24** *There is a finite set $W$ of terms, depending only on $A$, with the following properties. Assume*

- *$X$ and $X'$ are states,*
- *$\alpha$ 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($\alpha$).*

*If $\alpha$ is a context for $X$, then $\Delta(X, \alpha) = \Delta(X', \alpha)$.*

*Proof* Begin with a bounded exploration witness $W_0$, and obtain $W$ as follows. For each dynamic function symbol $f$, say $n$-ary, and each $n$ terms $t_1, \ldots, t_n \in W_0$, choose some terms $t_i'$ such that (i) each $t_i'$ results from $t_i$ by bijectively renaming variables and (ii) no two of the $t_i'$ have a variable in common. Then form $f(t_1', \ldots, t_n')$ and let $W$ be the union of $W_0$ and the

set of all these terms $f(t'_1, \ldots, t'_n)$ and all the $t'_i$. Clearly, $W$ is finite. We shall show that it is as required in the proposition. Let $X, X', \alpha$ be as in the hypothesis of the proposition.

By Lemma 5.11, the Bounded Work Postulate, and symmetry, it suffices to check that any update in $\Delta^+(X, \alpha) = \Delta^+(X', \alpha)$ that is trivial in $X$ is also trivial in $X'$. Let the update in question be $\langle f, \vec{a}, b\rangle$. So $f_X(\vec{a}) = b$, and we must show that $f_{X'}(\vec{a}) = b$.

By Proposition 5.23, we know that each $a_i$ is the value in $X$ of some term $t_i \in W_0$ for some assignment of values in Range($\alpha$) to the variables. (We may have a different assignment for each $i$.) Let $t'_i$ be as in the definition of $W$ above, for this particular $f$ and $t_1, \ldots, t_n$. Since $t'_i$ is the same as $t_i$ except for the choice of variables, $a_i$ is the value in $X$ of $t'_i$ for some assignment of values in Range($\alpha$) to the variables. Furthermore, since the various $t'_i$ share no variables, we can find a single such assignment that simultaneously gives each $t_i$ the corresponding value $a_i$ in $X$. So that assignment also gives $f(t'_1, \ldots, t'_n)$ the value $f_X(\vec{a}) = b$.

Since all the $t'_i$ and $f(t'_1, \ldots, t'_n)$ are in $W$, they have the same values in $X'$ as in $X$. That is, each $t_i$ denotes $a_i$ and $f(t'_1, \ldots, t'_n)$ denotes $b$ in $X'$. But this means that $f_{X'}(\vec{a}) = b$, as required. □

# 6 Equivalence and Causality

Having defined ordinary, interactive, small-step algorithms by means of our postulates, we devote this section to the question of when two algorithms of this sort should be considered the same. As in [10] and [5], we use a strong form of behavioral equivalence: Two algorithms are the same if they do the same thing in all circumstances. As in [10, 5], this means in particular that equivalent algorithms have the same states, the same initial states, and the same update sets. In addition, since our algorithms, unlike those of [10, 5] issue queries, equivalence of two algorithms should require that they issue the same queries.

We should be more precise about both "the same updates" and "the same queries". In [10, 5], the requirement on updates was that they should be the same for both algorithms, provided, of course, that the states to be updated are the same. For interactive algorithms, the update set depends not only on the state but also on the context, i.e., on the information provided by the environment. So we should require equivalent algorithms to produce the

same updates when they are working in the same state with the same context.

For queries, the requirement is a bit broader. Our algorithms do not issue queries in a context; by the time a context has been obtained, all the queries for that step have been issued and answered. We should require equivalent algorithms to issue the same queries when they are working in the same state and have received the same replies from the environment up to a certain point in the execution of their step. Part of our task in this section will be to give a precise definition of this; another part will be to consider alternative definitions and to show either why they are inappropriate or that they are equivalent to the chosen one.

Notice that this work is logically prior to the "evident" requirement that equivalent algorithms should produce the same updates in any state and context, for the notion of context depends on what queries are issued. In particular, our formulation of the "same queries" requirement must be at least strong enough to ensure that equivalent algorithms have, in each state, the same contexts; only then does the "same updates" requirement make sense.

**Remark 6.1** The definition of equivalence in [10] can be formulated very succinctly as requiring two equivalent algorithms to agree exactly on everything mentioned in the postulates: They have the same states, the same initial states, and the same transition functions. It is tempting to adopt the same strategy here. Require two equivalent algorithms to have the same states, the same initial states, the same causality relation $\vdash_X$ for every state $X$, and the same update set $\Delta^+(X, \alpha)$ for every state $X$ and context $\alpha$. Unfortunately, this definition is inappropriate. Specifically, it is too restrictive, as the following examples show. □

**Example 6.2** Consider two algorithms that are identical except that, in certain states $X$, the causality relation of one algorithm is empty while that of the other consists of a single instance, namely $\{(q, r)\} \vdash_X q'$ for certain queries $q$ and $q'$ and a certain reply $r$. In such a state, the first algorithm obviously issues no queries. The second also issues no queries; although its causality relation is nonempty, the cause $(q, r)$ can never arise and make the algorithm produce $q'$, because $q$ would have to be produced first and there is no way for this to happen. Thus, although the two algorithms have different causality relations, they will, in all possible circumstances, produce the same queries. Since we have assumed that they are identical except for

the causality relations, they always behave the same way and should therefore be considered equivalent. □

It is easy to build more complicated examples based on the same theme, namely that an instance of a causality relation might as well be omitted if the cause involved in it cannot arise. In the example above, the cause $\{(q, r)\}$ cannot arise simply because $q$ has no possible cause. In more complicated examples, $q$ might have a cause, involving queries that have no causes, or perhaps involving queries whose causes involve queries with no causes, etc. The following example is another variation on this theme.

**Example 6.3** Consider again two algorithms that are identical except for the causality relations for certain states. In the one algorithm, this causality relation has only the following instances.

$$\varnothing \vdash q_1, \quad \{(q_1, r_1)\} \vdash q_2.$$

In the other algorithm, there are the same two instances of $\vdash$ plus

$$\{(q_1, r_1'), (q_2, r_2)\} \vdash q_3,$$

where $r_1' \neq r_1$. Here again, the additional instance of causality in the second algorithm can never operate. Indeed, in order for its cause, $\{(q_1, r_1'), (q_2, r_2)\}$ to occur, the query $q_2$ must first be produced, and this requires that the reply to $q_1$ is $r_1$, not $r_1'$. So again, these algorithms should be equivalent despite the difference in their causality relations. □

The next example exhibits a different sort of variation in causality relations; instead of an irrelevant instance of causality, it has an irrelevant ingredient in a cause.

**Example 6.4** Suppose the first algorithm has, in certain states, just the following three instances of causality.

$$\varnothing \vdash q_1, \quad \{(q_1, r_1)\} \vdash q_2, \quad \{(q_2, r_2)\} \vdash q_3.$$

Let the other algorithm have, instead,

$$\varnothing \vdash q_1, \quad \{(q_1, r_1)\} \vdash q_2, \quad \{(q_1, r_1), (q_2, r_2)\} \vdash q_3.$$

So the only difference is that the cause of $q_3$ in the second algorithm includes the reply $r_1$ for $q_1$. But even in the first algorithm, this reply is needed in order for $q_3$ to be produced. Without $(q_1, r_1)$, the query $q_2$ cannot be produced, and so the cause $\{(q_2, r_2)\}$ for $q_3$ cannot arise. Once again, the algorithms behave the same in all circumstances and should be considered equivalent. □

The preceding examples show that our definition of equivalence must impose on the causality relations a more subtle requirement than being identical. The requirement must capture the intuition that the same queries are produced in any circumstances that can actually arise.

**Remark 6.5** We already mentioned that any appropriate definition of equivalence of algorithms must imply that equivalent algorithms have (in any state) the same contexts. One might therefore try using this as the definition, i.e., call two algorithms equivalent if they have the same contexts in all states. Unfortunately, this simple definition is too weak to capture the intuitive notion of equivalence, as the following example shows. □

**Example 6.6** Consider two algorithms that differ only in that, for certain states, one has the causality relation consisting of

$$\varnothing \vdash q \quad \text{and} \quad \varnothing \vdash q'$$

while the other has

$$\varnothing \vdash q \quad \text{and} \quad \{(q, r)\} \vdash q'$$

for all possible replies $r$ to $q$. These algorithms have the same contexts for such a state, because in both cases the contexts must answer both $q$ and $q'$. Yet they should be inequivalent, because if the environment fails to answer $q$, then they behave differently; the first algorithm asks $q'$ and the second does not. □

**Remark 6.7** It would be aesthetically pleasing to be able to define equivalence of algorithms as agreement on everything mentioned in the postulates. As we have seen, this won't work as long as the causality relation is among the things mentioned in the postulates. Why don't we reformulate the postulates using other primitive concepts — ones that are invariant under equivalence

— rather than causality relations? The main reason is that causality is naturally present in algorithms, or at least in the way algorithms are ordinarily described (e.g., programming languages). For example, in the algorithms

```
do in parallel                    issue q
   issue q            and         let r=reply(q) in
   issue q'                          issue q'
```

one clearly sees the two causality structures described in the preceding example. □

## 6.1 Reachability and well-foundedness

In accordance with the preceding discussion, we consider what circumstances can actually arise in the execution of an algorithm. We work, for the time being, with a fixed algorithm, a fixed state, and therefore a fixed causality relation which we denote simply by $\vdash$. The simplest way to describe any circumstances (as seen by the algorithm) is as an answer function $\alpha$ telling what replies the environment has given (so far) to queries. But not every answer function corresponds to possible circumstances. For example, $\alpha$ might contain replies for queries that can never be asked, or for queries that can be asked but not when the environment's answers are as given by $\alpha$. A general answer function is better viewed, intuitively, as indicating the replies that the environment *would give* if the appropriate queries were asked, rather then the replies that the environment *actually gives* to a specific algorithm in some possible circumstances. This subsection is devoted to extracting from a general answer function the part that could actually occur and to characterizing those answer functions that represent possible circumstances.

**Definition 6.8** Let $\alpha$ be an answer function. An $\alpha$-*trace* is a finite sequence $\langle q_1, \ldots, q_n \rangle$ of potential queries such that each $q_i$ is caused by some subfunction of $\alpha \restriction \{q_j : j < i\}$. A query $q$ is *reachable* under $\alpha$ if it occurs in some $\alpha$-trace.

The idea behind this definition is that a trace indicates how a sequence of queries could be asked by our algorithm, using its causality relation $\vdash$, when the environment's answers are in accordance with $\alpha$. Thus, a query

is reachable under $\alpha$ if there is a way for the algorithm to produce this query when the environment's answers are as given by $\alpha$. Although traces are linearly ordered, the next proposition will show that a more general arrangement of queries, for example a partial order or a preorder, will produce the same notion of reachability as long as there is no loop or infinite regress in the causes.

**Remark 6.9** Any initial segment of an $\alpha$-trace is also an $\alpha$-trace. Thus, the definition of reachability would be unchanged if we required the query to occur at the end of an $\alpha$-trace.

It would also be unchanged if we required the trace to contain no repetitions of elements, for deleting the second and later occurrences of any element in an $\alpha$-trace produces another $\alpha$-trace.

If an $\alpha$-trace contains a query not in $\mathrm{Dom}(\alpha)$, then this query cannot contribute to any later queries; removing such a query leaves an $\alpha$-trace. Thus, a query $q$ is reachable under $\alpha$ if it is the last query in some $\alpha$-trace in which all the earlier queries are in the domain of $\alpha$ and there are no repetitions. Of course, the final query $q$ of the trace need not be in $\mathrm{Dom}(\alpha)$. □

The following proposition gives two useful equivalent definitions of reachability. One is similar to that using traces but it does not involve a linear ordering of queries. The other relates the present discussion to the operator $\Gamma_\alpha$ and its least fixed point ${\Gamma_\alpha}^\infty$, as introduced in our discussion of contexts in Section 5.

**Proposition 6.10** *Let $\alpha$ be an answer function and $q$ a query. The following three statements are equivalent.*

1. *$q$ is reachable under $\alpha$.*
2. *There is a set $Z$ of queries, containing $q$, and there is a function assigning to each $q' \in Z$ a function $\xi_{q'} \subseteq \alpha \restriction Z$ such that $\xi_{q'} \vdash q'$ and such that the relation (of $s$ and $q'$) "$s \in Dom(\xi_{q'})$" is well-founded on $Z$.*
3. *$q \in {\Gamma_\alpha}^\infty$.*

*Proof* First, we assume (1) and prove (2). Let $\langle q_1, \ldots, q_n \rangle$ be a trace containing $q$ and without repetitions. Define $Z$ to be $\{q_1, \ldots, q_n\}$, and for

each $q_i \in Z$ let $\xi_{q_i}$ be a cause for $q_i$ that is a subfunction of $\alpha \restriction \{q_j : j < i\}$. Then (2) is satisfied; the relation "$s \in \mathrm{Dom}(\xi_{q'})$" is well-founded because if $q_j \in \mathrm{Dom}(\xi_{q_i})$ then $j < i$. We note for future reference that the $Z$ produced here is finite, so the proposition would continue to hold if we added to (2) the requirement that $Z$ be finite.

Next, we assume (2) and prove (3). Let $q$, $Z$, and the $\xi_{q'}$ be as in (2). We show, by induction with respect to the well-founded relation "$s \in \mathrm{Dom}(\xi_{q'})$", that every element of $Z$ is in ${\Gamma_\alpha}^\infty$. Suppose therefore that $q' \in Z$ and that ${\Gamma_\alpha}^\infty$ contains all elements $s$ of $\mathrm{Dom}(\xi_{q'})$. Then, by definition of $\Gamma_\alpha$ and by the assumption that $\xi_{q'} \vdash q'$, we have that $q' \in \Gamma_\alpha({\Gamma_\alpha}^\infty) = {\Gamma_\alpha}^\infty$.

Finally, we assume (3) and prove (1). We show, by induction on $n$, that every element of ${\Gamma_\alpha}^n$ is reachable under $\alpha$. This is vacuous for $n = 0$ as ${\Gamma_\alpha}^0 = \varnothing$. Assume it holds for $n$ and consider any $q \in {\Gamma_\alpha}^{n+1} = \Gamma_\alpha({\Gamma_\alpha}^n)$. By definition of $\Gamma_\alpha$, we have $\xi \vdash q$ for some $\xi \subseteq \alpha \restriction {\Gamma_\alpha}^n$. Fix such a $\xi$ and, by the induction hypothesis, pick for each $q' \in \mathrm{Dom}(\xi)$ some $\alpha$-trace containing it. Then the concatenation of all these traces, followed by $q$, is easily seen to be an $\alpha$-trace. □

**Remark 6.11** We assumed, in the Interaction Postulate, that causes are finite. That assumption is used in the proof of the implication from (3) to (1) in the preceding proposition, for we need to know that we are concatenating only finitely many traces in order to obtain a trace at the end. If, for the purpose of mathematical generality (not relevance to actual computation), we allowed infinite causes, then in order to maintain Proposition 6.10 we would have to allow traces to be transfinite sequences. Of course, we should then also delete the comment about $Z$ being finite at the end of the proof that (1) implies (2). □

The following easy corollary of Proposition 6.10 is often useful.

**Corollary 6.12** *A query $q$ is reachable under $\alpha$ if and only if it is caused by the restriction of $\alpha$ to some set of reachable (under $\alpha$) queries.*

*Proof* By the proposition, we see that "$q$ is reachable" is equivalent to $q \in {\Gamma_\alpha}^\infty$ and, if we also take into account the definition of $\Gamma_\alpha$, "$q$ is caused by the restriction of $\alpha$ to some set of reachable queries" is equivalent to $q \in \Gamma_\alpha({\Gamma_\alpha}^\infty)$. But ${\Gamma_\alpha}^\infty$ is a fixed point of $\Gamma_\alpha$, so these two conditions are equivalent. □

For the following lemma, recall from Section 5 the notation $\alpha_n$ for $\alpha \restriction {\Gamma_\alpha}^n$. We shall also write $\alpha_\infty$ for $\alpha \restriction {\Gamma_\alpha}^\infty$.

**Lemma 6.13** *If $\alpha$ and $\beta$ 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* The second conclusion follows immediately from the first. We prove the first by induction on $n$. When $n = 0$, the result ${\Gamma_\alpha}^k = {\Gamma_\beta}^k$ is trivial for $k = 0$ as both sides are empty, and the result for $k = 1$ follows exactly as in the general induction step, so we don't treat it separately.

Suppose the lemma is true for $n-1$, and suppose the hypothesis of the lemma is satisfied for $n$. Because ${\Gamma_\alpha}^n$ increases with $n$, the hypothesis of the lemma also holds for $n-1$, and therefore the first conclusion holds for all $k \leq n$. So it remains only to consider the case of $k = n+1$.

Each of the following statements is equivalent to the next, for any $q$.

- $q \in {\Gamma_\beta}^{n+1} = \Gamma_\beta({\Gamma_\beta}^n)$.
- $q \in \Gamma_\beta({\Gamma_\alpha}^n)$.
- $\xi \vdash q$ for some $\xi \subseteq \beta \restriction {\Gamma_\alpha}^n$.
- $\xi \vdash q$ for some $\xi \subseteq \alpha \restriction {\Gamma_\alpha}^n$.
- $q \in \Gamma_\alpha({\Gamma_\alpha}^n) = {\Gamma_\alpha}^{n+1}$.

Indeed, the first two statements are equivalent by the induction hypothesis; the second and third are equivalent by the definition of $\Gamma_\beta$; the third and fourth are equivalent by the hypothesis of the lemma; and the last two are equivalent by the definition of $\Gamma_\alpha$. □

The following definition is intended to capture the notion of an answer function describing the answers actually given by some environment during the execution of a step of an algorithm with causality relation $\vdash$ (in the current state).

**Definition 6.14** An answer function $\alpha$ is *well-founded* if $\mathrm{Dom}(\alpha) \subseteq {\Gamma_\alpha}^\infty$.

In view of Proposition 6.10, well-foundedness means that $\alpha$ does not contain answers to any queries that could never be reached under $\alpha$. Intuitively, $\alpha$ does not contain hypothetical information about how the environment would reply to queries that won't actually occur; it contains only answers to queries that the algorithm would actually ask, given the answers in $\alpha$.

**Proposition 6.15** *For any answer function $\alpha$ and any natural number $n$, $\alpha_n$ is well-founded.*

*Proof* Apply Lemma 6.13 with $\beta = \alpha_n$ to get the equality in the middle of

$$\mathrm{Dom}(\alpha_n) \subseteq {\Gamma_\alpha}^n = {\Gamma_{\alpha_n}}^n \subseteq {\Gamma_{\alpha_n}}^\infty.$$

□

**Corollary 6.16** *$\alpha_\infty$ is well-founded.*

*Proof* $\alpha_\infty$ is $\alpha_n$ for sufficiently large $n$. □

**Proposition 6.17** *$\alpha_\infty$ is the largest well-founded $\beta \subseteq \alpha$.*

*Proof* In view of the preceding corollary, we need only consider an arbitrary well-founded $\beta \subseteq \alpha$ and prove that $\beta \subseteq \alpha_\infty$, which amounts to showing that $\mathrm{Dom}(\beta) \subseteq {\Gamma_\alpha}^\infty$.

For any such $\beta$ we have, from the observation that $\Gamma_\alpha$ is monotone with respect to $\alpha$, that ${\Gamma_\beta}^\infty \subseteq {\Gamma_\alpha}^\infty$. This fact, and the assumption that $\beta$ is well-founded, so $\mathrm{Dom}(\beta) \subseteq {\Gamma_\beta}^\infty$, immediately give what we need. □

In view of this proposition, it is reasonable to call $\alpha_\infty$ the *well-founded part* of $\alpha$.

For well-founded answer functions, the notion of reachability simplifies as follows.

**Proposition 6.18** *If $\alpha$ is well-founded, then a query is reachable under $\alpha$ if and only if it is caused by some subfunction of $\alpha$.*

*Proof* Because $\alpha$ is well-founded, everything in its domain is reachable under it. Thus, the phrase "some subfunction of $\alpha$" in the present proposition is equivalent to the phrase "the restriction of $\alpha$ to some set of reachable queries" in Corollary 6.12. So the present proposition follows immediately from that corollary. □

## 6.2 Equivalence of causality relations

There are two plausible ways to approach the definition of equivalence of causality relations (and thus the definition of equivalence of algorithms) in the light of the examples at the beginning of this section. One way is to consider arbitrary answer functions $\alpha$. Then Examples 6.2, 6.3, and 6.4 show that we must not use the obvious definition of equivalence, namely that every answer function causes the same queries under both algorithms. In each of those examples, there is an answer function that causes different queries under the two algorithms. In Example 6.2, the misbehaving answer function is $\{(q, r)\}$, in Example 6.3 it is $\{(q_1, r'_1), (q_2, r_2)\}$, and in Example 6.4 it is $\{(q_2, r_2)\}$. (In the last of these examples, $\{(q_1, r_1), (q_2, r_2)\}$ also misbehaves, but this misbehavior could be removed by weakening the proposed definition of equivalence to say that, for every $\alpha$, any query caused by a subfunction of $\alpha$ in either algorithm must also be caused by a (possibly different) subfunction of $\alpha$ in the other algorithm.)

In each of the three examples, the indicated misbehavior involves an answer function that cannot actually arise under the algorithm. Instead of considering what queries are caused by subfunctions of $\alpha$, we should consider what queries are caused by subfunctions of $\alpha$ that could actually arise during the execution of the algorithm. According to Corollary 6.12, this amounts to considering what queries are reachable under $\alpha$.

The second way is to confine attention to well-founded answer functions $\alpha$, those that can actually arise. In this approach, it is reasonable to require that the same queries be caused by subfunctions of $\alpha$. The second approach seems more natural, but there is a technical complication, namely, that the notion of well-founded seems to depend on the causality relation. Should we use the answer functions well-founded for the one causality relation, or the other, or both? It turns out that the problem disappears, for we shall show that equivalent causality relations have the same well-founded answer functions. But the second approach would require us to prove this and to give the definition of equivalence simultaneously. To avoid this complication, we adopt the first approach and then prove that the second is equivalent to it.

**Definition 6.19** Two causality relations are *equivalent* if, for every answer function $\alpha$, they make the same queries reachable under $\alpha$.

**Proposition 6.20** *The following statements are equivalent for any pair of causality relations.*

1. *The two causality relations are equivalent.*
2. *They have the same well-founded answer functions $\alpha$, and, for each such $\alpha$, the same queries are caused by subfunctions of $\alpha$.*
3. *They have the same well-founded answer functions $\alpha$, and, for each such $\alpha$, the same queries are reachable under $\alpha$.*
4. *For every answer function that is well-founded for both causality relations, the same queries are caused by subfunctions of $\alpha$.*
5. *For every answer function that is well-founded for both causality relations, the same queries are reachable under $\alpha$.*

To avoid possible confusion, we emphasize that in clauses (2) and (4), "the same queries are caused by subfunctions of $\alpha$" means that any query caused under one of the causality relations by a subfunction of $\alpha$ is also caused under the other causality relation by a possibly different subfunction of $\alpha$.

*Proof of Proposition 6.20* We first observe that Proposition 6.18 gives us the equivalence of (2) and (3) as well as the equivalence of (4) and (5). Also, the implication from (2) to (4) (or from (3) to (5)) is trivial. So we can complete the proof by showing that (1) implies (3) and (4) implies (1).

Assuming (1), to prove (3) it suffices to show that the two causality relations give the same well-founded answer functions; the rest of (3) is included in the definition of equivalence. So suppose, toward a contradiction, that $\alpha$ is an answer function that is well-founded for one of our two causality relations, say ${}^1\vdash$, but not for the other, ${}^2\vdash$. We'll use left superscripts 1 and 2 not only for the two causality relations but also for the associated $\Gamma$ operators. So $\mathrm{Dom}(\alpha) \subseteq {}^1\Gamma_\alpha{}^\infty$ but there are some queries $q$ in $\mathrm{Dom}(\alpha)$ that are not in ${}^2\Gamma_\alpha{}^\infty$. Among such queries, choose one $q$ that is in ${}^1\Gamma_\alpha{}^k$ for the smallest possible $k$. Of course $k = n+1$ for some $n$ (as ${}^1\Gamma_\alpha{}^0 = \varnothing$).

Since $q \in {}^1\Gamma_\alpha{}^{n+1} = {}^1\Gamma_\alpha({}^1\Gamma_\alpha{}^n)$, we have $\xi\, {}^1\vdash q$ for some $\xi \subseteq \alpha \restriction {}^1\Gamma_\alpha{}^n$. Now $\alpha \restriction {}^1\Gamma_\alpha{}^n$ is well-founded with respect to ${}^1\vdash$ by Proposition 6.15. So, by Proposition 6.18, $q$ is reachable under $\alpha \restriction {}^1\Gamma_\alpha{}^n$ with respect to ${}^1\vdash$ and therefore also with respect to ${}^2\vdash$, since (1) is assumed.

By minimality of $k$, we know that the domain of the function $\alpha \restriction {}^1\Gamma_\alpha{}^n$ is included in ${}^2\Gamma_\alpha{}^\infty$, and we have seen that $q$ is reachable, in the sense of ${}^2\vdash$, under this function. By Corollary 6.12, $q$ is caused by some subfunction of $\alpha \restriction {}^2\Gamma_\alpha{}^n$, and this means that

$$q \in {}^2\Gamma_\alpha({}^2\Gamma_\alpha{}^\infty) = {}^2\Gamma_\alpha{}^\infty.$$

This contradicts our choice of $q$, so the proof that (1) implies (3) is complete.

To prove that (4) implies (1), we consider an arbitrary answer function $\alpha$ and show, by induction on $n$, that ${}^1\Gamma_\alpha{}^n = {}^2\Gamma_\alpha{}^n$. This will suffice because, for sufficiently large $n$ we get ${}^1\Gamma_\alpha{}^\infty = {}^1\Gamma_\alpha{}^\infty$, and then Proposition 6.10 tells us that the reachability notions of the two causality relations agree.

For the induction, the base case $n = 0$ is trivial, as both sides of the desired equation are empty. So assume the result for $n$. Thus, $\alpha_n$ is the same with respect to the two causality relations, and it is, by Proposition 6.15, well-founded for both causality relations. By our assumption (4), the same queries are caused by subfunctions of $\alpha$ under the two causality relations. But these queries are exactly the elements of ${}^1\Gamma_\alpha{}^{n+1}$ and ${}^2\Gamma_\alpha{}^{n+1}$. □

**Corollary 6.21** *If two causality relations ${}^1\vdash$ and ${}^2\vdash$ are equivalent, then for all $n$ ${}^1\Gamma_\alpha{}^n = {}^2\Gamma_\alpha{}^n$. In particular, ${}^1\Gamma_\alpha{}^\infty = {}^2\Gamma_\alpha{}^\infty$. Therefore the two causality relations give the same well-founded answer functions and the same contexts.*

*Proof* The first assertion was proved as part of the proof that (4) implies (1) in Proposition 6.20. The second follows by taking $n$ sufficiently large. To deduce the third from the second, note that the definition of well-foundedness and the characterization of contexts in Lemma 5.7 both use the causality relation only via ${\Gamma_\alpha}^\infty$. □

With this corollary, we are in a position to define equivalence of algorithms, because there is no ambiguity in the notion of context.

**Definition 6.22** Two algorithms are *equivalent* if they have

- the same states,
- the same initial states,
- equivalent causality relations in every state, and,

- for every state $X$ and context $\alpha$, the same update set $\Delta^+(X, \alpha)$.

**Remark 6.23** In this definition of equivalence, the parts about the causality relations and the update sets are intended to capture the idea that the two algorithms behave the same way — issue the same queries and perform the same updates — as long as the environment behaves the same way. The notion of "the environment behaves the same way" is represented in the definition by the use of the same answer function $\alpha$ for both algorithms, both in the "update" part of Definition 6.22 and in Definition 6.19. We regard the environment as adequately represented by an answer function because we are dealing only with ordinary algorithms, which cannot be influenced by the environment except through the answer function.

One could raise the following objection to this picture. Although an ordinary algorithm cannot make use of the timing of the replies it receives, there is nothing to prevent the environment from making use of the timing of queries. It might, for example, answer some query $q$ differently depending on whether another query $q'$ had been asked earlier. Or the relative order of the queries might influence whether the environment answers them at all. In such a situation, the environment's replies to the algorithm might be given by different answer functions, depending on the timing of queries, even though it is really the same environment. Intuitively, equivalent algorithms should behave the same even in the presence of such a timing-sensitive environment. Yet our definition of equivalence does not require any agreement when different answer functions are involved.

The answer to this objection is in two parts, according to the source of the difference in timing of queries. Let us consider first the case where the algorithms specify different timings. As a typical example, suppose algorithm $A$ begins by issuing query $q$ and, after receiving any reply, issues $q'$, while algorithm $B$ begins by issuing only $q'$, postponing $q$ until after it has received an answer to $q'$. (Recall here, from the discussion following the Interaction Postulate in Section 5, that an ordinary algorithm cannot enforce a particular order between two queries except by making one dependent on receipt of an answer to the other.) The objection involves an environment whose answers to $q$ and $q'$ depend on the order in which the queries are received, and which can therefore produce two different answer functions, say $\alpha$ and $\beta$, for these two algorithms. Nothing in the definition of equivalence requires the two algorithms to behave the same in the presence of these two different answer functions.

Yet the definition of equivalence shows, in accordance with intuition, that $A$ and $B$ are inequivalent. Though it doesn't require equivalent algorithms to behave the same when they get different answer functions $\alpha$ and $\beta$, it does require them to behave the same when both get the answer function $\varnothing$. And here $A$ and $B$ behave differently; $A$ issues $q$ and $B$ issues $q'$.

More complicated examples of this sort can be handled similarly. If two algorithms are intuitively different because they issue their queries in a different order, then our definition of equivalence will detect this, not by looking at the different answer functions that may result from the different orderings, but from the single answer function that preceded the difference — an answer function from which the two algorithms produce different queries.

The other part of the answer to the objection concerns the situation where the difference in timing is not specified by the algorithm but is accidental. For example, if an algorithm begins its step by issuing both $q$ and $q'$ (caused by $\varnothing$), then, in the actual execution of the algorithm, these two queries might reach the environment in either order or simultaneously, depending on details of the operating system under which the algorithm runs, the communication channels through which it sends the queries, etc. And again these accidents might influence the answers provided by the environment, so it would seem that such an environment is not adequately modeled by a single answer function.

We must remember, though, that the details that determine the order in which the environment receives such a pair of queries are themselves a part of the environment, albeit usually a different part from the answerer of queries. Indeed, the properties of the operating system, the communication channels, etc., are not part of the algorithm; they are something external that can influence the execution of the algorithm, and this is precisely what "environment" means.

Thus, if the environment produces different answers because it receives two such queries in different orders, then it is not really the same environment. The query-answering part of the environment may be the same, but the timing-deciding part of the environment is different. Therefore, it is appropriate not to require equivalent algorithms to behave the same when the timing of their queries is different for environmental reasons (as opposed to reasons in the algorithms' own causality relations). Indeed, requiring the same behavior in such circumstances could make an algorithm inequivalent to itself. □

## 6.3 Normalized causality relations

In this subsection, we develop a normal form for causality relations. We show how to transform each causality relation $\vdash$ into an equivalent one in normal form, $\vdash'$, and we show that two causality relations are equivalent if and only if their normal forms are the same.

**Definition 6.24** Given a causality relation $\vdash$ define its *normal form* $\vdash'$ by letting $\alpha \vdash' q$ if and only if $\alpha$ is well-founded (with respect to $\vdash$) and $q$ is reachable under $\alpha$ (also with respect to $\vdash$).

The intuition behind this definition is that a cause $\alpha$ for $q$ in the sense of $\vdash'$ contains complete information about how $q$ comes about under $\alpha$. That is, $\alpha$ must contain not only a cause for $q$ (in the sense of $\vdash$) but also causes for the queries involved in those causes, causes for the queries involved in those causes, etc., all in a well-founded arrangement. This is essentially what part (2) of Proposition 6.10 tells us.

Recall also that, by Proposition 6.18, the last part of the definition of $\alpha \vdash' q$, namely that $q$ is reachable under $\alpha$, can be equivalently stated as: there is some $\xi \subseteq \alpha$ such that $\xi \vdash q$.

**Proposition 6.25** *Any causality relation $\vdash$ is equivalent to its normal form.*

*Proof* We verify equivalence in the form (4) of Proposition 6.20. Fix an $\alpha$ that is well-founded with respect to both $\vdash$ and $\vdash'$. If some subfunction of $\alpha$ causes $q$ in the sense of $\vdash$, then $\alpha$ itself causes $q$ in the sense of $\vdash'$. For the converse implication, suppose $\xi \subseteq \alpha$ and $\xi \vdash' q$. Then, by the definition of $\vdash'$, as reformulated just before this proposition, there is $\eta \subseteq \xi$ such that $\eta \vdash q$. Then, as $\eta \subseteq \alpha$, the proof is complete. □

**Proposition 6.26** *If ${}^1\vdash$ and ${}^2\vdash$ are equivalent, then ${}^1\vdash' = {}^2\vdash'$.*

*Proof* This is immediate by inspection of the definition of normalization and the implication from (1) to (3) in Proposition 6.20. □

Combining the two preceding propositions, we find that normalization is an idempotent operation on causality relations.

**Corollary 6.27** $\vdash''=\vdash'$.

*Proof* Apply Proposition 6.26 to the equivalence given by Proposition 6.25. □

We close this section with an easy result that suggests our motivation for introducing the normalization $\vdash'$. The idea is that, once we have a context $\alpha$, enlarging it cannot lead to new queries. This will be important in the sequel to this paper, when we consider the operation of sequential composition of algorithms. In the execution of a step of such a composition, say $A$ followed by $B$, first the environment must provide a context for $A$, so that $A$ can complete its step, and then $B$ continues by asking more queries and getting replies. Things can get a bit confusing if those replies cause new queries in $A$, which is supposed to have already finished its step. The following corollary shows that, if we use a normalized causality relation $\vdash'$, then such confusion cannot arise.

**Proposition 6.28** *Let $\vdash$ be a causality relation and $\vdash'$ its normalization. Suppose that $\alpha$ is a context, that $\alpha \subseteq \beta$, and that $\xi \vdash' q$ for some $\xi \subseteq \beta$. Then $\xi \subseteq \alpha$.*

*Proof* Observe first that the word "context" in the hypothesis is unambiguous, since $\vdash$ and $\vdash'$ have the same contexts, by Proposition 6.25 and Corollary 6.21. Note also that "well-founded" is unambiguous for the same reason.

Since $\alpha$ is a context, its domain includes ${\Gamma_\alpha}^n$ for all $n$. Since also $\alpha \subseteq \beta$, the hypothesis of Lemma 6.13 is satisfied, and therefore so is the conclusion of that lemma, for all $n$. In particular, $\beta_\infty = \alpha_\infty = \alpha$. Now since $\xi \vdash' q$, we know by definition of $\vdash'$ that $\xi$ is well-founded. But $\beta_\infty$ is the largest well-founded subfunction of $\beta$, by Proposition 6.17. So $\xi \subseteq \beta_\infty = \alpha$. □

# 7 Abstract Queries and Queries in Normal Form

In this section, we address two questions that could arise from our definition of potential queries as tuples of state elements and labels.

The first question is whether so much generality is needed. Couldn't we use, for example, tuples where the first component is a label and the other components are elements of the state?

The second question goes in the opposite direction. Couldn't we have queries of an even more general sort? One can imagine queries whose structure is not a simple tuple but something more complicated. In fact, Benjamin Rossman has suggested that the general form of a query could be an arbitrary first-order structure.

We shall show that the answer to both questions is essentially that it doesn't matter. The definition we adopted, the restriction advocated in the first question, and some of the generalizations advocated in the second are all equivalent. Here equivalence means intuitively that queries in one of these senses can be represented by queries in another sense without any loss of information. Of course the notion of algorithm would have to be defined, in each case, to use the corresponding notion of query in the postulates.

## 7.1 Query systems

To establish the equivalence of various notions of query, we begin by defining a general notion of rigid query system, intended to be broad enough to encompass most of the possibilities envisioned by the second question. Then we shall show how to replace the queries of any such system by ones of the restricted form suggested in the first question. Since our original definition of potential queries lies between the two modifications, the equivalence we establish between the latter also encompasses the former. We shall also discuss what would be needed to accommodate non-rigid query systems.

The central idea of a query system is that there should be, for each state $X$, a set $\mathcal{Q}(X)$ of potential queries in $X$. An obvious requirement for such an assignment $\mathcal{Q}$ is that it should respect isomorphisms. That is, if $i : X \cong Y$ is an isomorphism between two states, then there should be an associated bijection from $\mathcal{Q}(X)$ to $\mathcal{Q}(Y)$. Furthermore, these bijections should be coherent in the sense that if we also have an isomorphism $j : Y \cong Z$, then transporting queries from $X$ to $Z$ via $Y$ should be the same as transporting them directly by means of the composite isomorphism $j \circ i$. All of this, plus the trivial requirement that the bijection associated to the identity isomorphism $X \cong X$ should be the identity bijection of $\mathcal{Q}(X)$, can be expressed succinctly by saying that we have a functor $\mathcal{Q}$ from the category of states and isomorphisms to the category of sets.

Another aspect of query systems is that each potential query $q$ for $X$ should directly involve only finitely many elements of $X$; we call the set of these elements the query's support and denote it by $\mathrm{supp}(q)$. The crucial

property of supports, apart from being preserved, like everything else, by isomorphisms, is that the result of applying an isomorphism $i : X \cong Y$ to some $q \in \mathcal{Q}(X)$ should not depend on all of $i$ but only on its restriction to $\mathrm{supp}(q)$.

Up to this point, our description of query systems has imposed no constraints at all on the queries associated to non-isomorphic states. The potential queries could look entirely different for each isomorphism class of states. This situation is unsatisfactory both intuitively and technically. At the intuitive level, we expect that a query system will associate queries to states in a uniform (or at least somewhat uniform) manner, not totally differently for each isomorphism class. Technically, to formulate and use the Bounded Work Postulate, we need to be able to compare queries for non-isomorphic states, for example in saying that the same $\alpha$ is an answer function for two such states. Notice that the queries introduced in Section 5 were very uniform, in that whether a tuple is a query for $X$ depends only on whether its non-label components are elements of $X$; the rest of the structure of $X$ played no role here. In general, the rest of the structure might play a role, but a limited one. For example, we might want tuples of a certain form to be queries just in case a certain component belongs to a certain subuniverse of the state. Thus, the same tuple may serve as a query in one state and not in another, even if the relevant elements are present in both states but don't belong to the same subuniverse in both states. Our definition of query systems will allow for this sort of dependence on the state, but only a finite amount of information about the state should be relevant — not the whole isomorphism type of the state.

To describe the finite amount of information that is relevant, we use a finite set of terms, rather similarly to the use of bounded exploration witnesses in the Bounded Work Postulate. We begin by defining the necessary concepts relating to such a set of terms and the associated partial agreement between states; afterward, we give the formal definition of a query system and of rigidity.

**Convention 7.1** Fix, until further notice, a vocabulary $\Upsilon$ and a finite set $U$ of $\Upsilon$-terms. The notions of partial isomorphisms, query-systems, etc., that we define below should all be regarded as relative to $\Upsilon$ and $U$. $\square$

Although we are ultimately interested only in the potential queries associated to states, our definition of query systems will require that they associate

sets of potential queries to all $\Upsilon$-structures. The reason for this is to reduce the dependence of the notion of query system on any particular algorithm. With our definitions, the only information we need about an algorithm, in order to begin talking about its potential queries, is its vocabulary $\Upsilon$.

**Definition 7.2** A *partial isomorphism* from one $\Upsilon$-structure $X$ to another $X'$, written $i : X \rightharpoonup X'$, is a bijection $i$ between some $A \subseteq X$ and some $A' \subseteq X'$ with the following property. For any two terms $t_1, t_2 \in U$ and any assignment $v$ of values in $A$ to their variables, $t_1$ and $t_2$ have the same value in $X$ using $v$ if and only if they have the same value in $X'$ using $i \circ v$.

**Remark 7.3** It appears reasonable to strengthen the condition for a partial isomorphism by requiring the property in the last sentence also when different assignments, say $v_1$ and $v_2$ are used for the two terms $t_1$ and $t_2$. This strengthening can, however, be subsumed by our formulation if we just enlarge $U$ so that, whenever it contains $t_1$ and $t_2$, it also contains copies of them that use disjoint sets of variables. In fact, we don't even need to enlarge $U$; just modify all the terms in it so that no two of them have a common variable.

Another equivalent reformulation of the definition is to start with a finite set $U'$ of Boolean terms and to require that, for each $t \in U'$ and each assignment $v$ of values in $A$ to its variables, $t$ has value `true` in $X$ using $v$ if and only if it has value `true` in $X'$ using $i \circ v$. To go from our definition of this new one, take $U'$ to consist of all equations between terms from $U$. To go from the new definition to the original one, take $U$ to consist of $U'$ plus `true`.

A rough way to express our condition is that, if you try to extend $i$ from $A$ to more elements of $X$ by the rule $i(f_X(\vec{a})) = f_{X'}(i(\vec{a}))$, then the resulting extension is, at least up to and including terms from $U$, well-defined and one-to-one. □

We remark that the empty function is a partial isomorphism from $X$ to $X'$ if and only if the closed terms in $U$ satisfy the same equations in both structures. Notice also that any restriction of a partial isomorphism to a subset of its domain is again a partial isomorphism of the same structures.

The following definition formalizes our earlier, informal discussion. For brevity, we write simply "query" rather than "potential query".

**Definition 7.4** A *query system* $\mathcal{Q}$ (with respect to $\Upsilon$ and $U$) consists of

- for each $\Upsilon$-structure $X$, a set $\mathcal{Q}(X)$ of *queries* for $X$,
- for each structure $X$ and each $q \in \mathcal{Q}(X)$, a finite subset $\mathrm{supp}_X(q)$ of $X$ (written just $\mathrm{supp}(q)$ when $X$ is understood), called the *support* of $q$,
- for each partial isomorphism $i : X \rightharpoonup X'$ whose domain includes $\mathrm{supp}_X(q)$, a query $i_*(q) \in \mathcal{Q}(X')$

subject to the following requirements.

- If a partial isomorphism $i : X \rightharpoonup X$ is the identity map of its domain, then $i_*$ is the identity map of its domain also.
- If $i : X \rightharpoonup X'$ and $j : X' \rightharpoonup X''$, then $(j \circ i)_* = j_* \circ i_*$.
- If $i$ is a partial isomorphism whose domain includes the support of the query $q$, then $\mathrm{supp}(i_*(q)) = i[\mathrm{supp}(q)]$.
- If $i$ and $j$ are two partial isomorphisms $X \rightharpoonup X'$, if $q$ is a query for $X$ with support included in $\mathrm{Dom}(i) \cap \mathrm{Dom}(j)$, and if $i \restriction \mathrm{supp}(q) = j \restriction \mathrm{supp}(q)$, then $i_*(q) = j_*(q)$.

**Remark 7.5** This definition admits an elegant reformulation in terms of the category $\mathbb{S}$ of sets and a category $\mathbb{P}$ whose objects are pairs $(X, A)$ where $X$ is an $\Upsilon$-structure and $A$ is a subset of (the base set of) $X$. A morphism $(X, A) \to (X', A')$ in $\mathbb{P}$ is a partial isomorphism $i : X \rightharpoonup X'$ with $\mathrm{Dom}(i) = A$ and $\mathrm{Range}(i) \subseteq A'$ (so $i$ is a function $A \to A'$).

A query system gives rise to a functor $\mathcal{Q} : \mathbb{P} \to \mathbb{S}$ given on objects by

$$\mathcal{Q}(X, A) = \{q \in \mathcal{Q} : \mathrm{supp}(q) \subseteq A\}$$

and on morphisms by $\mathcal{Q}(i) = i_*$. It also gives rise to a natural transformation $\mathrm{supp} : \mathcal{Q} \to \mathcal{P}_{\mathrm{fin}} \circ \mathcal{D}$, where $\mathcal{P}_{\mathrm{fin}} : \mathbb{S} \to \mathbb{S}$ is the covariant finite power set functor and $\mathcal{D} : \mathbb{P} \to \mathbb{S}$ is the functor sending $(X, A) \mapsto A$ and $i \mapsto i$.

The functor $\mathcal{Q}$ and the natural transformation supp satisfy the following two requirements. First, if $i : (X, A) \to (X, A')$ is an inclusion (note that $X$ is the same and $A \subseteq A'$) then $\mathcal{Q}(i)$ is also an inclusion. Second, if $q \in \mathcal{Q}(X, A)$ then $q$ is also in $\mathcal{Q}(X, \mathrm{supp}(q))$ (which is a subset of $\mathcal{Q}(X, A)$ by the first requirement).

One can verify that any functor $\mathcal{Q}$ and natural transformation supp satisfying these requirements arise from a query system. So our definition of query system could be replaced with this functorial one. $\square$

**Remark 7.6** If $i : X \rightharpoonup X'$ is a partial isomorphism, then consider the structure $Y$ obtained as follows. For each element $a \in X$ that is the value in $X$ of a term $t \in U$ for some assignment $v$ of values in $\mathrm{Dom}(i)$ to its variables, replace $a$ by the value $a'$ in $X'$ of the same term $t$ with the assignment $i \circ v$ of values to its variables. This replacement is well-defined and one-to-one, by the definition of partial isomorphism. The structure $Y$ obtained in this way is isomorphic to $X$, via the map $i'$ that, in the notation above, sends each $a$ to the corresponding $a'$ and is the identity map on the rest of $X$. Furthermore, the identity function on $\mathrm{Range}(i)$ is easily seen to be a partial isomorphism $\mathrm{id} : Y \rightharpoonup X'$ such that $\mathrm{id} \circ i' = i$. Thus, every partial isomorphism is obtainable by composing a (total) isomorphism and a partial isomorphism that is an identity map.

One could take advantage of this fact to avoid talking about general partial isomorphisms, using instead only the two special cases of isomorphisms and partial isomorphisms that are identity maps. The definition of query systems could be formulated just in terms of these two special cases. □

## 7.2 Rigid query systems are simple

Not all query systems can be reduced to the standard form introduced in Section 5. To see where the difficulty occurs, consider a system where a query for $X$ is simply an unordered set of two elements from $X$. The support of such a pair is the pair itself, and the action of partial isomorphisms is the obvious one, applying the isomorphism to both elements of the pair. To represent these queries by tuples, we would have to choose an ordering for the two elements of each pair, and in general there is no isomorphism-invariant way to do this.

There are at least three ways around the problem. One is to assume that our structures include a background, in the sense of [3], adequate to form arbitrary subsets of a structure. Then a query, in the tuple sense of Section 5, could have a two-element set as its only non-label component, rather than having the two elements of the pair as two components.

A second approach, to which we devote this subsection, is to impose an additional requirement on query systems. This requirement, which we call rigidity, will exclude the troublesome examples.

Finally, a third approach, about which we shall comment in the next subsection, generalizes the tuple approach from Section 5 to allow queries structured in a somewhat more general way than just tuples. In order for this

generalization to cohere with the ASM model, external functions must also allow arguments structured in a more general way than mere tuples. Such a generalization of the usual argument structures was introduced, though for predicates rather than functions, in [4, Section 3], under the name of thesauri.

We begin by defining rigidity. It is the converse of the last requirement in the definition of query systems.

**Definition 7.7** A query system is *rigid* if, whenever $i$ and $j$ are partial isomorphisms $X \rightharpoonup X'$, $q$ is a query for $X$, $\mathrm{supp}(q) \subseteq \mathrm{Dom}(i) \cap \mathrm{Dom}(j)$, and $i_*(q) = j_*(q)$, then $i \restriction \mathrm{supp}(q) = j \restriction \mathrm{supp}(q)$.

The concept of rigidity may be clarified by a non-example arising from the discussion above. If queries are unordered pairs of elements of the state, with each pair serving as its own support, then a permutation that interchanges the two elements of a pair is a counterexample to rigidity. It maps the pair to itself, but its restriction to the pair is not the identity. In general, the idea behind rigidity is to prohibit such non-trivial automorphisms of the structure involved in a query.

Our next goal is to show that every rigid query system is equivalent to one where queries have the very restricted form suggested in the first question at the beginning of this section. We first define these restricted systems and the (natural) notion of equivalence.

**Definition 7.8** A query system (for $\Upsilon$ and $U$) is *simple* if there is a set $\Lambda$ of *labels* such that:

- Each query for a structure $X$ is a tuple whose first component is an element of $\Lambda$ and whose other components are elements of $X$.
- The support of a query is the set of all its components but the first.
- Partial isomorphisms act on queries by acting on all components but the first.

**Remark 7.9** The definition does not require that every tuple consisting of a label followed by elements of $X$ be a query for $X$. There might be some restrictions, such as that certain components come from certain subuniverses of $X$ or that certain components be distinct. Simple query systems of this generality are needed in order to match arbitrary rigid query systems.

However, any simple query system can be enlarged to one in which all tuples of the indicated form are queries. Any algorithm based on the given simple query system can equally well be based on the enlarged one. The extra queries added by the enlargement would never be caused by any answer function and thus would never be involved in a context and would never be relevant to a computation. □

Observe that all simple query systems are rigid.

**Definition 7.10** Two query systems $\mathcal{Q}$ and $\mathcal{Q}'$ are *equivalent* if there is, for each structure $X$, a bijection $e_X : \mathcal{Q}(X) \to \mathcal{Q}'(X)$ such that, for all $q \in \mathcal{Q}(X)$,

- $\mathrm{supp}(e_X(q)) = \mathrm{supp}(q)$ and
- $i_{*\mathcal{Q}'}(e_X(q)) = e_{X'}(i_{*\mathcal{Q}}(q))$ whenever $i : X \rightharpoonup X'$ is a partial isomorphism whose domain includes $\mathrm{supp}(q)$.

(In the second of these requirements, the extra subscript $\mathcal{Q}$ or $\mathcal{Q}'$ on $i_*$ indicates which query system's $i_*$ is intended.) When no confusion can result, we may write simply $e$ instead of $e_X$.

**Remark 7.11** From the category theoretic point of view outlined in Remark 7.5, the definition of equivalence of query systems just says that $e$ is a natural isomorphism between the functors $\mathcal{Q}$ and $\mathcal{Q}'$ and that $e$ commutes with supp. □

It follows immediately from the definition that equivalence of query systems is an equivalence relation.

**Proposition 7.12** *Every rigid query system is equivalent to a simple one.*

*Proof* Let $\mathcal{Q}$ be an arbitrary query system. To construct an equivalent simple query system, our first task is to define an appropriate set $\Lambda$ of labels. We first form the collection $\mathcal{T}$ of all triples $\langle X, q, \vec{a}\rangle$ where $X$ is a structure, $q \in \mathcal{Q}(X)$, and $\vec{a}$ is a list, without repetitions, of all the elements of $\mathrm{supp}(q)$.

**Definition 7.13** For $\langle X, q, \vec{a}\rangle$ and $\langle X', q', \vec{a}'\rangle$ in $\mathcal{T}$, we define $\langle X, q, \vec{a}\rangle \sim \langle X', q', \vec{a}'\rangle$ to mean that there is a partial isomorphism $i : X \rightharpoonup X'$ such that $\mathrm{supp}(q) \subseteq \mathrm{Dom}(i)$, $i_*(q) = q'$, and $\vec{a}'$ is obtained by applying $i$ componentwise to $\vec{a}$.

Obviously $\langle X, q, \vec{a}\rangle \sim \langle X', q', \vec{a}'\rangle$ implies that $\vec{a}$ and $\vec{a}'$ have the same length.

**Lemma 7.14** *This $\sim$ is an equivalence relation on $\mathcal{T}$.*

*Proof* Reflexivity, symmetry, and transitivity follow from the facts that identity maps, inverses of partial isomorphisms, and composites of partial isomorphisms are again partial isomorphisms. For symmetry, notice also that the equation $(i^{-1})_* = (i_*)^{-1}$ follows from the first two requirements in the definition of query system (functoriality). □

**Lemma 7.15** *If $\langle X, q, \vec{a}\rangle \sim \langle X, q, \vec{a}'\rangle$ (with the same $X$ and $q$) then $\langle X, q, \vec{a}\rangle = \langle X, q, \vec{a}'\rangle$.*

*Proof* By definition of $\sim$, there must be a partial isomorphism $i : X \rightharpoonup X$ that sends $\vec{a}$ to $\vec{a}'$ componentwise and satisfies $i_*(q) = q$. Rigidity of $\mathcal{Q}$ then requires that $i$ fixes $\mathrm{supp}(q)$ componentwise. But, by definition of $\mathcal{T}$, all components of $\vec{a}$ are in $\mathrm{supp}(q)$. So $\vec{a} = \vec{a}'$. □

The *permutations* of an element $\langle X, q, \vec{a}\rangle$ of $\mathcal{T}$ are the elements obtainable from it by permuting the components of $\vec{a}$ while leaving $X$ and $q$ unchanged. It is almost true that the permutations of $\langle X, q, \vec{a}\rangle$ constitute its orbit under the action of the symmetric group on the $\vec{a}$ part; the only thing wrong is that the meaning of "the symmetric group" varies with $\langle X, q, \vec{a}\rangle$. Different $\langle X, q, \vec{a}\rangle$'s have $\vec{a}$ of different lengths, so different symmetric groups have to act. It is trivial to check that, if $\langle X, q, \vec{a}\rangle \sim \langle X', q', \vec{a}'\rangle$, then each permutation of $\langle X, q, \vec{a}\rangle$ is $\sim$ to the corresponding permutation of $\langle X', q', \vec{a}'\rangle$. Thus, it makes sense to speak of one equivalence class in $\mathcal{T}/\!\sim$ being a permutation of another.

**Definition 7.16** The set $\Lambda$ of *labels* is defined as a subset of $\mathcal{T}/\!\sim$ consisting of one representative from each class of members that are permutations of each other. Equivalently, it is a subset of $\mathcal{T}/\!\sim$ that is maximal with respect to containing no two elements that are permutations of each other.

Readers appalled by the possible size of this $\Lambda$ are asked to wait until after the proof of the proposition is finished; Remark 7.18 will address this size issue.

Using $\Lambda$, we are in a position to define the simple query system $\mathcal{Q}'$ that will be equivalent to $\mathcal{Q}$. In fact, the definition of simple query systems leaves us little choice about the definition. We can and will choose which tuples of the form $\langle \lambda, \vec{a} \rangle$ with $\lambda \in \Lambda$ and all components of $\vec{a}$ in $X$ should be queries for $X$. The other ingredients of the query system, i.e., the support function and the bijections induced by partial isomorphisms, are then determined by the definition of simple.

**Definition 7.17** $\mathcal{Q}'$ is the simple query system with label set $\Lambda$ defined by putting $\langle \lambda, \vec{a} \rangle$ into $\mathcal{Q}'(X)$, for a given $X$, if and only if $\lambda$ has an element of the form $\langle X, q, \vec{b} \rangle$ where $X$ is the given structure, $\vec{b}$ is a permutation of $\vec{a}$, and the function $i$ sending the components of $\vec{b}$ to those of $\vec{a}$, in order, is a partial isomorphism $X \rightharpoonup X$.

Recall that elements of $\Lambda \subseteq \mathcal{T}/\sim$ are equivalence classes of elements of $\mathcal{T}$, so to say, as we did in this definition, that $\lambda$ has an element of the form $\langle X, q, \vec{b} \rangle$ makes good sense. Recall also that, according to the definition of $\mathcal{T}$, we know that $\vec{b}$ is a listing without repetitions of the members of $\mathrm{supp}(q)$; therefore so is $\vec{a}$ since it is just a permutation of $\vec{b}$.

For the sake of brevity, we shall write $i(\vec{b}) = \vec{a}$ or $i : \vec{b} \mapsto \vec{a}$ to mean that $a$ is the result of applying $i$ componentwise to $\vec{b}$. So the last part of the definition of $\mathcal{Q}'(X)$ says that the function defined by $i : \vec{b} \mapsto \vec{a}$ is a partial isomorphism.

We repeat that, as required by the definition of simple, the query system $\mathcal{Q}'$ gives each of its queries $\langle \lambda, \vec{a} \rangle$ the support consisting of the elements of $\vec{a}$, and that if $i$ is a partial isomorphism then $i_*$ acts on queries by applying $i$ to all components except the initial $\lambda$. It is easy to check that $\mathcal{Q}'$ is a query system and therefore a simple one.

We next define the functions $e_X$ that will witness that $\mathcal{Q}'$ is equivalent to $\mathcal{Q}$. Let $\langle \lambda, \vec{a} \rangle \in \mathcal{Q}'(X)$, so, according to the definition of $\mathcal{Q}'$, we have some query $q \in \mathcal{Q}(X)$ and some listing $\vec{b}$ of $\mathrm{supp}(q)$ such that $\lambda$ is the $\sim$-class of $\langle X, q, \vec{b} \rangle$. Furthermore, $\vec{a}$ is a permutation of $\vec{b}$ and thus also lists $\mathrm{supp}(q)$ without repetitions. And finally, we have a partial isomorphism $i : X \rightharpoonup X$ that sends $\vec{b}$ to $\vec{a}$.

We define $e_X(\langle \lambda, \vec{a} \rangle)$ to be $i_*(q)$. Since the domain of $i$ consists of all the components of $\vec{b}$ and these are all the members of $\mathrm{supp}(q)$, $i_*(q)$ is well-defined. But to know that $e_X(\langle \lambda, \vec{a} \rangle)$ is well-defined, we must show that it is independent of the choices made in its definition.

Suppose, therefore, that for the same $X$ and $\langle \lambda, \vec{a} \rangle$, we had another $\langle X, q', \vec{b'} \rangle \in \lambda$, where $\vec{b'}$ is also a permutation of $\vec{a}$, and where $i' : \vec{b'} \mapsto \vec{a}$ is a partial isomorphism. We must show that $i_*(q) = i'_*(q')$, i.e., that we get the same value for $e_X(\langle \lambda, \vec{a} \rangle)$ as before.

Since both $\langle X, q, \vec{b} \rangle$ and $\langle X, q', \vec{b'} \rangle$ belong to the $\sim$-class $\lambda$, the definition of $\sim$ gives us a partial isomorphism $j : X \rightharpoonup X$ such that $j : \vec{b'} \mapsto \vec{b}$ and $j_*(q') = q$. Then $i \circ j = i'$, because $i \circ j$ sends $\vec{b'}$ first to $\vec{b}$ and then to $\vec{a}$, so it agrees with $i'$. Therefore, $i'_*(q') = i_*(j_*(q')) = i_*(q)$, as required. This completes the proof that $e_X$ is well-defined.

Next, we show that it is surjective. Consider any $q \in \mathcal{Q}(X)$, and list its support, without repetitions, as $\vec{c}$. Thus, $\langle X, q, \vec{c} \rangle \in \mathcal{T}$. Its equivalence class in $\mathcal{T}/\sim$ might not be in $\Lambda$, but it can be transformed to an element $\lambda \in \Lambda$ by suitably permuting the components of $\vec{c}$. Let $\vec{a}$ be this permutation of $\vec{c}$. Then $\langle \lambda, \vec{a} \rangle \in \mathcal{Q}'(X)$, and the definition of $e$ immediately gives (with an identity map as $i$) that $e_X(\langle \lambda, \vec{a} \rangle) = q$. Since $q$ was arbitrary, this shows that $e_X$ is surjective.

Before showing that $e_X$ is one-to-one, we observe that it respects supports, as required in the definition of equivalence. Indeed, using the notation from the definition of $e_X$ we have that the support of $e_X(\langle \lambda, \vec{a} \rangle) = i_*(q)$ is $i[\mathrm{supp}(q)]$ because $\mathcal{Q}$ is a query system. This support is listed by $\vec{a}$, because $\vec{b}$ lists $\mathrm{supp}(q)$ and $i : \vec{b} \mapsto \vec{a}$. But the set listed by $\vec{a}$ is, by definition, the support (in the sense of $\mathcal{Q}'$) of $\langle \lambda, \vec{a} \rangle$. Thus, $e_X$ respects supports.

Now we are ready to prove that $e_X$ is one-to-one. Suppose we have two elements $\langle \lambda, \vec{a} \rangle$ and $\langle \lambda', \vec{a'} \rangle$ in $\mathcal{Q}'(X)$, both mapped by $e_X$ to the same query $q_0 \in \mathcal{Q}(X)$. By definition of $\mathcal{Q}'$ and $e_X$, we have queries $q, q' \in \mathcal{Q}(X)$ and listings of their respective supports $\vec{b}$ and $\vec{b'}$ such that $\langle X, q, \vec{b} \rangle \in \lambda$, $\langle X, q', \vec{b'} \rangle \in \lambda'$, and we have partial isomorphisms $i, i' : X \rightharpoonup X$ such that $i : \vec{b} \mapsto \vec{a}$, $i' : \vec{b'} \mapsto \vec{a'}$, and $i_*(q) = i'_*(q') = q_0$. Because $e_X$ respects supports, we know that both $\vec{a}$ and $\vec{a'}$ list the support of $q_0$ without repetitions. So they differ by a permutation of the coordinates; let $n$ be the common length of $\vec{a}$, $\vec{a'}$, $\vec{b}$ and $\vec{b'}$, and let $\pi$ be the permutation of $\{1, 2, \ldots n\}$ that sends $\vec{a}$ to $\vec{a'}$. It is convenient to use the notation $\vec{a'} = \vec{a} \circ \pi$, which is strictly correct if we view a list of length $n$ as a function on $\{1, 2, \ldots n\}$. The partial isomorphism $i^{-1} \circ i'$ sends $\vec{b'}$ first via $i'$ to $\vec{a'} = \vec{a} \circ \pi$ and then via $i^{-1}$ to $\vec{b} \circ \pi$. The induced bijection on $\mathcal{Q}(X)$ sends $q'$ first to $q_0$ and then to $q$. So $i^{-1} \circ i'$ witnesses that $\langle X, q', \vec{b'} \rangle \sim \langle X, q, \vec{b} \circ \pi \rangle$. Thus, the $\sim$-class $\lambda'$ of $\langle X, q', \vec{b'} \rangle$ differs from the $\sim$-class $\lambda$ of $\langle X, q, \vec{b} \rangle$ by a permutation. But both are in $\Lambda$, which contains

only one representative from each family of permutation-related $\sim$-classes. Therefore $\lambda = \lambda'$ and $\langle X, q, \vec{b} \circ \pi \rangle \sim \langle X, q, \vec{b} \rangle$. By Lemma 7.15, the latter implies $\langle X, q, \vec{b} \circ \pi \rangle = \langle X, q, \vec{b} \rangle$, i.e, $\pi$ is the identity permutation. In view of the definition of $\pi$, this means $\vec{a} = \vec{a}'$. This completes the proof that $\langle \lambda, \vec{a} \rangle = \langle \lambda', \vec{a}' \rangle$ and so $e_X$ is one-to-one.

Finally, we must show that $e$ commutes with partial isomorphisms. So suppose $\langle \lambda, \vec{a} \rangle \in \mathcal{Q}'(X)$ and $j : X \rightharpoonup Y$ is a partial isomorphism whose domain includes the support (in the sense of $\mathcal{Q}'$) of $\langle \lambda, \vec{a} \rangle$, namely the set listed by $\vec{a}$. Using again the notation of the definition of $e_X$, we have that $\lambda$ contains $\langle X, q, \vec{b} \rangle$ with $\vec{b}$ a permutation of $\vec{a}$, and we have a partial isomorphism $i : X \rightharpoonup X$ with $i : \vec{b} \mapsto \vec{a}$ and $i_*(q) = e_X(\langle \lambda, \vec{a} \rangle)$. Notice that, since $\vec{b}$ is a permutation of $\vec{a}$, $j$ is defined on it, and so $j$ witnesses that $\langle X, q, \vec{b} \rangle \sim \langle Y, j_*(q), j(\vec{b}) \rangle$. So $\langle Y, j_*(q), j(\vec{b}) \rangle \in \lambda$, $j(\vec{a})$ is a permutation of $j(\vec{b})$, and $j \circ i \circ j^{-1}$ is a partial isomorphism $j(\vec{b}) \mapsto j(\vec{a})$. Therefore

$$e_X(j_*(\langle \lambda, \vec{a} \rangle)) = e_X(\langle \lambda, j(\vec{a}) \rangle) = (j \circ i \circ j^{-1})_*(j_*(q)) = j_*(i_*(q)) = j_*(e_X(\langle \lambda, \vec{a} \rangle)).$$

This completes the proof that $e$ gives an equivalence between the query systems $\mathcal{Q}'$ and $\mathcal{Q}$. $\square$

**Remark 7.18** The simple query system $\mathcal{Q}'$ constructed in the preceding proof seems to use a huge number of labels, since arbitrary $\Upsilon$-structures can occur as the first components of members of $\mathcal{T}$. But the situation is not so bad as it seems; although $\mathcal{T}$ can be huge, passing to $\sim$-classes reduces it considerably. The reason is that, instead of considering arbitrary states $X$ and lists of elements $\vec{a}$ from $X$, it suffices to choose just enough such pairs so that every pair is related to a chosen one by a partial isomorphism. Then every $\langle X, q, \vec{a} \rangle$ will be $\sim$ to one with $(X, \vec{a})$ among the chosen pairs, because the partial isomorphisms can be used to obtain the appropriate $q$'s. Furthermore, getting enough pairs $(X, \vec{a})$ in this sense just means getting enough to realize every possible pattern of equations and inequations between terms from $U$ with values from $\vec{a}$ for the variables. As $U$ is finite, we need only finitely many pairs $(X, \vec{a})$ for any fixed length of $\vec{a}$. So the only ways to get infinitely many labels in our $\Lambda$ are to have infinitely many lengths of $\vec{a}$'s or to have infinitely many $q$'s for a single $(X, \vec{a})$. For this to happen, $\mathcal{Q}$ would have to have, in some state, either infinitely many queries with the same support or queries with arbitrarily large supports. In such a case, of course, an infinite $\Lambda$ is unavoidable in any equivalent simple query system.

The query systems used in Section 5 have queries with arbitrarily large supports, but, by the Bounded Work Postulate, any particular algorithm will use queries with lengths only up to a certain bounded size. Thus, for any particular algorithm, we could truncate the query system, deleting all queries whose lengths are too large to be relevant for this algorithm. Then the supports of queries are of bounded size, and for each support there are only a finite number of queries. The new query system, still adequate for the given algorithm, is equivalent, by the argument above, to a simple query system with only finitely many labels. More generally, the intuition underlying the Bounded Work Postulate shows that, even with more general rigid query systems than those in Section 5, any particular algorithm should have a bound for the sizes of supports of queries, since building queries with large supports takes large amounts of work. Furthermore, any particular algorithm should use only finitely many queries with a given support in a specific state, for these queries would have to be distinguished in the text of the algorithm. The argument above shows that, for such an algorithm, a simple query system with finitely many labels is adequate. □

**Remark 7.19** The intuition behind the definition of the labels in the proof of Proposition 7.12 is that, if $q \in \mathcal{Q}(X)$, then the label occurring in the corresponding query in $\mathcal{Q}'(X)$ should contain all information about $q$ except for the particular elements in its support. The label should completely describe $q$ up to partial isomorphisms of the support. Borrowing terminology used in [2], we can say that the label is intended to capture the form of $q$, while the listing of the support captures its matter. □

## 7.3 Avoiding rigidity

The definition of "query system" is intended to summarize what one would expect to be true for any reasonable notion of query. Rigidity, on the other hand, seems rather an ad hoc assumption, needed to make the proof of Proposition 7.12 work. It is not clear why one should not permit, say, an unordered pair of elements of $X$ to serve as a query in $X$. So it is worthwhile to analyze query systems that are not necessarily rigid. They cannot be equivalent to simple query systems, since the latter are rigid. But are they equivalent to some variant of simple query systems?

Let us begin by looking at how rigidity was used in the proof of Proposition 7.12. It was used only in the proof of Lemma 7.15, which in turn was

used in the proof that $e_X$ is one-to-one. In the absence of rigidity, the proof of Lemma 7.15 breaks down; we can no longer infer that $\vec{a} = \vec{a}'$. But we can still infer that $\vec{a}$ and $\vec{a}'$ are permutations of each other. Indeed, they are, by definition of $\mathcal{T}$, one-to-one listings of $\mathrm{supp}(q)$. And the permutation by which they differ might not be entirely arbitrary, because $i : \vec{a} \mapsto \vec{a}'$ has to be a partial isomorphism and $i_*$ has to fix $q$. This suggests that we consider the *symmetry group* or *stabilizer* of $\langle X, q, \vec{a}\rangle$, namely the set of permutations $\pi$ of $\{1, 2, \ldots, n\}$ where $n$ is the length of $\vec{a}$ and where $i : \vec{a} \mapsto \vec{a} \circ \pi$ is a partial isomorphism and satisfies $i_*(q) = q$. An equivalent way to express this condition on $\pi$ is that $\langle X, q, \vec{a}\rangle \sim \langle X, q, \vec{a} \circ \pi\rangle$. The name "symmetry group" is justified, because it is easy to check that this is a group of permutations.

In the proof of Proposition 7.12, the argument showing that $e_X$ is one-to-one breaks down, in the absence of rigidity, at the place where Lemma 7.15 was invoked. As a result, if $e_X(\langle \lambda, \vec{a}\rangle) = e_X(\langle \lambda', \vec{a}'\rangle)$, we can still conclude $\lambda = \lambda'$, but instead of $\vec{a} = \vec{a}'$, we obtain only that $\vec{a}'$ is obtainable from $\vec{a}$ by applying a permutation in the stabilizer of $\langle X, q, \vec{b}\rangle$.

This suggests that we could repair the difficulty caused by the lack of rigidity by taking the queries of $\mathcal{Q}'$ to be pairs where the first component is a label (as before) but the rest is not merely a list $\vec{a}$ of elements but rather an orbit of such lists under the appropriate permutation group. In other words, in the situation described in the preceding paragraph, we would have identified $\langle \lambda, \vec{a}\rangle$ with $\langle \lambda, \vec{a}'\rangle$, and so $e_X$ would once again be one-to-one.

This idea works, but a few technicalities need to be checked; these are the next two lemmas.

**Lemma 7.20** *If two elements of $\mathcal{T}$ are related by $\sim$ then they have the same stabilizer.*

*Proof* Suppose $\langle X, q, \vec{b}\rangle \sim \langle X', q', \vec{b}'\rangle$. Then the same partial isomorphism that witnesses this instance of $\sim$ also witnesses $\langle X, q, \vec{b} \circ \pi\rangle \sim \langle X', q', \vec{b}' \circ \pi\rangle$ for any permutation $\pi$ of the appropriate $\{1, 2, \ldots, n\}$. If $\pi$ is in the stabilizer of $\langle X, q, \vec{b}\rangle$, so that $\langle X, q, \vec{b}\rangle \sim \langle X, q, \vec{b} \circ \pi\rangle$, then this instance of $\sim$ and the two in the preceding sentence give, by symmetry and transitivity of $\sim$, that $\langle X', q', \vec{b}'\rangle \sim \langle X', q', \vec{b}' \circ \pi\rangle$. Thus $\pi$ is also in the stabilizer of $\langle X', q', \vec{b}'\rangle$. $\square$

In view of this lemma, it makes sense to speak of the stabilizer of an equivalence class in $\mathcal{T}/\!\sim$, meaning the stabilizer of any of its members. In particular, we can speak of the stabilizer of a label $\lambda \in \Lambda$; it will be convenient to introduce the following notation for it.

**Definition 7.21** For any label $\lambda \in \Lambda$, say the $\sim$-class of $\langle X, q, \vec{b}\rangle$ where $\vec{b}$ has length $n$, the *stabilizer* of $\lambda$ is the group $S_\lambda$ consisting of those permutations $\pi$ of $\{1, 2, \ldots, n\}$ such that $\langle X, q, \vec{b}\rangle \sim \langle X, q, \vec{b} \circ \pi\rangle$.

The preceding lemma shows that this definition does not depend on the choice of the representative $\langle X, q, \vec{b}\rangle$ from $\lambda$.

As indicated earlier, we want to modify the definition of $\mathcal{Q}'$ in the proof of Proposition 7.12 by identifying $\langle \lambda, \vec{a}\rangle$ with $\langle \lambda, \vec{a} \circ \pi\rangle$ where they differ only by a permutation $\pi \in S_\lambda$ of their non-$\lambda$ components. To avoid confusion, it will be useful to know that anything identified with a query in this way is itself a query in the sense of (the original) $\mathcal{Q}'$.

**Lemma 7.22** *If $\langle \lambda, \vec{a}\rangle \in \mathcal{Q}'(X)$ and $\pi \in S_\lambda$, then $\langle \lambda, \vec{a} \circ \pi\rangle \in \mathcal{Q}'(X)$ .*

*Proof* The hypothesis that $\langle \lambda, \vec{a}\rangle \in \mathcal{Q}'(X)$ means that $\lambda$ contains some $\langle X, q, \vec{b}\rangle$ where $\vec{b}$ is a permutation of $\vec{a}$ and $i : \vec{b} \mapsto \vec{a}$ is a partial isomorphism. Clearly, then $\vec{b}$ is also a permutation of $\vec{a} \circ \pi$, so all that remains to be checked is that $\vec{b} \mapsto \vec{a} \circ \pi$ is a partial isomorphism. But we know, since $\pi \in S_\lambda$, that there is a partial isomorphism $j : \vec{b} \mapsto \vec{b} \circ \pi$. Since $i : \vec{b} \mapsto \vec{a}$, we also have $i : \vec{b} \circ \pi \mapsto \vec{a} \circ \pi$. Therefore $i \circ j$ is the desired partial isomorphism $\vec{b} \mapsto \vec{a} \circ \pi$. □

**Definition 7.23** Define $\mathcal{Q}''(X)$ to be the quotient of $\mathcal{Q}'(X)$ obtained by identifying $\langle \lambda, \vec{a}\rangle$ with $\langle \lambda, \vec{a} \circ \pi\rangle$ whenever $\pi \in S_\lambda$. We write $[\lambda, \vec{a}]$ for the equivalence class of $\langle \lambda, \vec{a}\rangle$. All elements of $[\lambda, \vec{a}]$ have the same support in the sense of $\mathcal{Q}'$, namely the set listed by $\vec{a}$; we define this set to be the support of $[\lambda, \vec{a}]$ in the sense of $\mathcal{Q}''$. If $i : X \rightharpoonup Y$ is a partial isomorphism whose domain includes this support, then all elements of $[\lambda, \vec{a}]$ are mapped by $i_{*\mathcal{Q}'}$ into the same equivalence class $[\lambda, i(\vec{a})] \in \mathcal{Q}''(Y)$; we define this class to be $i_{*\mathcal{Q}''}([\lambda, \vec{a}])$.

It is easy to check that this defines a query system $\mathcal{Q}''$.

**Proposition 7.24** *For any query system $\mathcal{Q}$, let $\mathcal{Q}'$ be defined as in the proof of Proposition 7.12 and let $\mathcal{Q}''$ be defined as above. Then $\mathcal{Q}''$ is equivalent to $\mathcal{Q}$.*

*Proof* The proof is the same as the proof of Proposition 7.12 except for two points. The first is that the functions $e_X$ defining the equivalence need to be defined on $\mathcal{Q}''(X)$, where previously they were defined on $\mathcal{Q}'(X)$. We define $e_X([\lambda, \vec{a}])$ to be what was previously called $e_X(\langle \lambda, \vec{a} \rangle)$.

**Lemma 7.25** *This new $e_X$ is well-defined*

*Proof* We must check that, if $\pi \in S_\lambda$, then the old $e_X$ sends $\langle \lambda, \vec{a} \rangle$ and $\langle \lambda, \vec{a} \circ \pi \rangle$ to the same query in $\mathcal{Q}(X)$. Using the notation from the proof of the preceding lemma, we have, by the definition of the old $e_X$,

$$e_X(\langle \lambda, \vec{a} \circ \pi \rangle) = (i \circ j)_*(q) = i_*(j_*(q)) = i_*(q) = e_X(\langle \lambda, \vec{a} \rangle),$$

where the equation $j_*(q) = q$, used to obtain the third equality, comes from the definition of $S_\lambda$. □

The only other change from the proof of Proposition 7.12 is that, in proving that $e_X$ is one-to-one, instead of invoking Lemma 7.15 to infer $\vec{a} = \vec{a}'$, we instead use, as explained above, the fact that $\vec{a}$ and $\vec{a}'$ differ by a permutation from $S_\lambda$. Thus, although we cannot obtain $\langle \lambda, \vec{a} \rangle = \langle \lambda, \vec{a}' \rangle$ in $\mathcal{Q}'(X)$, we do obtain $[\lambda, \vec{a}] = [\lambda, \vec{a}']$ in $\mathcal{Q}''(X)$. □

Unlike the $\mathcal{Q}'$ of Proposition 7.12, the $\mathcal{Q}''$ of Proposition 7.24 is not a simple query system, because the queries are not tuples of a label and elements of the state but rather equivalence classes of such tuples under certain permutations of the state elements. The price we pay for starting with a non-rigid $\mathcal{Q}$ is the need for equivalence classes in a tuple-based equivalent system. By the way, since equivalent elements of $\mathcal{Q}'(X)$ have the same label as their first component, we could have defined $\mathcal{Q}''(X)$ by leaving the labels alone and taking equivalence classes only of the lists of elements of $X$. That is, we could write $\langle \lambda, [\vec{a}] \rangle$ instead of $[\lambda, \vec{a}]$, where of course $[\vec{a}]$ represents the orbit of $\vec{a}$ under the permutation group $S_\lambda$.

This generalized notion of queries easily covers the example mentioned earlier, where a query is given by an unordered pair of elements of the state. Such a pair amounts to an equivalence class of ordered pairs under the group consisting of both permutations of the two positions. More generally, an equivalence class of $n$-tuples under the full symmetric group of permutations of $\{1, 2, \ldots, n\}$ amounts to an $n$-element multiset; by starting with $n$-tuples of distinct elements we get $n$-element sets. Other choices of subgroups of the symmetric group give other sorts of structures, for example cyclically (rather than linearly) ordered tuples, or ordered tuples of unordered sets, or unordered sets of ordered tuples.

## 7.4 Thesauri

If we were to generalize Section 5 by allowing queries that involve permutation orbits of tuples of state elements, then in order to simulate such algorithms with ASMs, as we intend to do in a sequel to this paper, we would have to make a similar generalization for the external functions. That is, an $n$-ary external function could take as its argument not an $n$-tuple of elements but rather an orbit of such $n$-tuples with respect to a certain group of permutations of the positions in the $n$-tuple. Equivalently, we could regard the function as still being defined on $n$-tuples but required to be invariant under the given group of permutations of the argument places. A familiar example of such a function is ordinary addition of numbers which, by commutativity, can be viewed as acting not on ordered pairs of numbers but on orbits of such pairs under the symmetric group of two permutations. That is, addition can be viewed as an operation on two-element multisets.

If external functions are generalized in this way, then it would seem natural to allow the same generalization for the state's own functions, the interpretations of the symbols from $\Upsilon$. This would mean, for example, that, if a dynamic function is regarded as acting on tuples in an invariant manner, then when it is updated at one tuple of arguments, it is automatically updated at all the equivalent tuples with respect to the given permutation group.

A rather similar situation arose in [4], though in connection with relations rather than functions. We introduced there a notion of thesaurus, which was essentially a relational vocabulary equipped with a specification of the desired invariance group for each relational symbol. Actually, for the purposes of [4], we also generalized from "relational", i.e., two-valued, to many values, with the number of values being specified for each relation symbol; we had the permutation group act also on the set of "truth" values; and we included a probability distribution on these truth values in order to define random structures.

Our present situation involves functional rather than relational vocabularies, and it needs none of these extra items. So we present here a simpler (than in [4, Section 3]) notion of thesaurus adapted to our present needs. To avoid confusion with the earlier notion, we say "functional thesaurus"; perhaps what was called simply a "thesaurus" in [4] should be called a "relational thesaurus," or even a "many-valued relational thesaurus with probabilities."

**Definition 7.26** A *functional signum* is a triple $\langle f, n, G\rangle$ where

- $f$ is an arbitrary symbol,
- $n$ is a natural number, called the *arity* of the signum, and
- $G$ is a group of permutations of $\{1, 2, \ldots, n\}$.

A *functional thesaurus* is a finite set of signa with distinct first components.

In this paper, we'll often omit the adjective "functional" since these are the only signa and thesauri that we will consider. The first component $f$ in a signum is analogous to the function symbols in traditional vocabularies. In fact, we sometimes write $f$ when we really mean the whole signum. The rest of the signum, $\langle n, G\rangle$, is called the *type* of the signum; it is analogous to the arity in traditional vocabularies.

We next define the intended semantics of signa and thesauri.

**Definition 7.27** An *operation* of type $\langle n, G\rangle$ on a set $X$ is a function $F : X^n \to X$ such that, for each $(x_1, \ldots, x_n) \in X^n$ and each $\pi \in G$,

$$F(x_1, \ldots, x_n) = F(x_{\pi(1)}, \ldots, x_{\pi(n)}).$$

If $\Upsilon$ is a functional thesaurus, then an $\Upsilon$-structure $X$ consists of a nonempty base set $|X|$ and, for each $\langle f, n, G\rangle \in \Upsilon$, an operation $f_X$ of type $\langle n, G\rangle$ on $|X|$.

We have chosen to regard operations as being defined on $n$-tuples and being invariant under the permutation group $G$, but we could also view them as being defined on the $G$-orbits of $n$-tuples. Semantically, the two approaches are equivalent. Syntactically, the approach we have adopted seems a bit simpler, since we can continue to use the traditional notation $f(t_1, \ldots, t_n)$ for terms, subject to the convention that such a term is to be identified[3] with the other terms that arise by permuting the arguments according to $G$. The permutation invariance requirement in the definition of operations of type $\langle n, G\rangle$ is exactly what is needed to ensure that, even with this identification, terms have well-defined values in suitable situations.

With the generalization from vocabularies of function symbols to thesauri of signa, our work in the preceding sections extends straightforwardly to

[3]To regard two different strings of symbols as being syntactically the same may seem strange, but other authors have done this also. For example, many people identify syntactically any two expressions that differ only by renaming bound variables.

the new context. In particular, we would define potential queries to be of the form $\langle \lambda, [\vec{a}] \rangle$ considered above in $\mathcal{Q}''$, with the understanding that each label $\lambda$ has associated with it a particular permutation group to be used in producing the equivalence classes $[\vec{a}]$ that go with $\lambda$ in queries. As we showed in Proposition 7.24, once this generalized notion of query is in place, we can handle any notion of query that forms an abstract query system.

In a sequel to this paper, we shall show that every algorithm, as defined here, is equivalent to an ASM. That work also extends to generalized query systems, provided we allow the ASMs to have external function signa from a thesaurus (rather than external function symbols from a vocabulary). In this situation, it seems natural to also replace the (non-external) function symbols of an ASM's state with signa, so that thesauri would replace vocabularies throughout the ASM theory.

**Remark 7.28** We used, in Section 6, a very strong notion of equivalence of algorithms, implying in particular that each state must have the same contexts for both algorithms. As a result, replacing one query system by an equivalent one in the obvious way does not produce an equivalent algorithm. But there is a natural, weaker notion of equivalence that still conforms to the intuitive idea that the algorithms do the same thing. This notion of equivalence would permit replacing a query system by an equivalent one. It would also permit renaming the function symbols (or the signa) of $\Upsilon$, without of course changing the arities (or types). □

**Remark 7.29** Although it is not needed in this paper, it should be pointed out that, given the generalization from traditional function symbols to signa, there is a further generalization that looks natural. For functions $f : X^n \to X$, as used in traditional structures, there is an asymmetry in that the inputs can be tuples of arbitrary length while the outputs are single elements of $X$. One could allow more general functions, $X^n \to X^m$, but there would be little point in doing so because such a function is equivalent to $m$ functions $X^n \to X$. In the context of signa, however, the inputs are (from one point of view) not simply tuples but orbits of these under some group of permutations of the positions in the tuples. One could symmetrize the situation by also allowing orbits (for a different group in general) as outputs. Such an orbit-to-orbit operation is not simply equivalent to several orbit-to-element operations. So in this context, the symmetrization seems like a natural extension of the framework. We do not pursue this extension here, partly because, as

mentioned above, we do not need it, and partly because it complicates such syntactic matters as substituting terms for variables in other terms. □

# References


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

[2] Andreas Blass, Yuri Gurevich, and Saharon Shelah, "Choiceless polynomial time," *Ann. Pure Appl. Logic* 100 (1999) 141–187.

[3] Andreas Blass and Yuri Gurevich, "Background, reserve, and Gandy machines," in *Computer Science Logic (14th International Workshop, CSL 2000, Annual Conference of the EACSL, Fischbachau, Germany, August, 2000, Proceedings)* P. Clote and H. Schwichtenberg, eds., Springer-Verlag, Lecture Notes in Computer Science 1862 (2000) 1–17.

[4] Andreas Blass and Yuri Gurevich, "Strong extension axioms and Shelah's zero-one law for choiceless polynomial time," J. Symbolic Logic 68 (2003) 65–131.

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

[6] Andreas Blass and Yuri Gurevich, "Algorithms: A quest for absolute definitions," *Bull. Europ. Assoc. Theoret. Comp. Sci.* 81 (2003) 195–225.

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

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

[9] Yuri Gurevich, "ASM guide," Univ. of Michigan Technical Report CSE-TR-336-97, in [11].

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

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

[12] Robin Milner, *Communicating and Mobile Systems: the $\pi$-Calculus*, Cambridge Univ. Press (1999).

[13] Yiannis N. Moschovakis, "What is an algorithm?" in *Mathematics Unlimited*, B. Engquist and W. Schmid, eds., Springer-Verlag (2001) 919–936.

[14] Alan M. Turing, "On computable numbers, with an application to the Entscheidungsproblem", *Proceedings of London Mathematical Society*, series 2, vol. 42 (1936–1937), 230–265; correction, *ibid.*, vol. 43, 544–546. Available online at http://www.abelard.org/turpap2/tp2-ie.asp.

[15] Peter Wegner and Dina Goldin, "Interaction as a framework for modeling," in *Conceptual Modeling*, P. Chen et al., eds., Springer-Verlag, Lecture Notes in Computer Science 1565 (1999), 243–257.