Journal of Universal Computer Science, vol. 7, no. 11 (2001), 917-951
submitted: 7/6/01, accepted: 1/10/01, appeared: 28/11/01

Partial Updates: Exploration

Yuri Gurevich
(Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
gurevich@microsoft.com)

Nikolai Tillmann
(Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
t-niktil@microsoft.com)

Abstract: The partial update problem for parallel abstract state machines has manifested itself in the cases of counters, sets and maps. We propose a solution of the problem that lends itself to an efficient implementation and covers the three cases mentioned above. There are other cases of the problem that require a more general framework.

Key Words: Abstract state machine, AsmL, ASM thesis, partial updates, submachines, synchronous parallelism, updates

Categories: F.1.1, F.1.2, F.3.2, D.3.3

1 Introduction

The Abstract State Machine (ASM) thesis states that, for every computer system A, there is an ASM B such that B is behaviorally equivalent to A and in particular step-for-step simulates A [G93, G95]. The thesis inspired numerous applications of ASMs [ASM, Br95, Br99, FSE].

In [G00], the thesis is proved for sequential computer systems. Sequential systems are sequential-time and bounded-parallelism. A system is sequential-time if its runs are finite or infinite sequences of states. A system is bounded-parallelism if there is a fixed bound on the degree of the parallelism of single steps in computations of the system; see details in [G00]. In [BG*], the thesis is proved for computer systems that are unbounded-parallelism but bounded-sequentiality. A system is bounded-sequentiality if there is fixed bound on the lengths of sequences of events that must occur in a particular order during any single step. The abstract state machines of [BG*] are essentially the parallel (but not distributed) ASMs of the Lipari guide [G95]; see details in [BG*].

The problem of partial updates, addressed in this paper, arises in the context of sequential-time computer systems of unbounded parallelism, in other words, synchronous parallel systems. The problem bothered the first author from the time of the Lipari guide [G95]. It manifested itself first in the form of the cumulative updates of a counter. Imagine a synchronous parallel computer system A such that A contains an integer counter c and such that different parts of A increment c during the same step. The value of c at the end of a step of A is the value of c at the beginning of the step plus the sum of all the increment parameters. If you see the individual changes as usual assignments, the semantics is all screwed up: the cumulative effect of increments by 2 and by 3 is a contradiction rather then the expected increment by 5, and the cumulative effect of two increments by 2 is an increment by 2 rather than the expected increment by 4. How should we deal with counters in the ASM model of the computer system A? (One may wonder if there is a real system A of that kind. We have some imperfect examples but one does not need a perfect example to worry about such systems in the context of the ASM thesis.)

The second manifestation of the partial-update problem was related to sets. Imagine that one part of a synchronous parallel computer system treats a set variable s as a unit, subject to usual assignments, while another part, during the same step, treats that same set variable s as an aggregate, subject to insertions and deletions of members. How do we reconcile the two points of view in the ASM model of the computer system?

The third manifestation of the partial-update problem was related to the development of AsmL, a powerful ASM-based specification language [FSE]. In AsmL 2, the current version of AsmL, map variables replace dynamic functions of positive arity. Often you need to change a map variable m only at some particular argument, say at 1. How do you do that in the ASM paradigm, and how do you combine such a partial update with usual updates? So far, the partial-update problem for maps seems similar to that of sets, but it is more complicated. The value $m(1)$ of m at the argument 1 can be again a map. You may want to change $m(1)$ at the argument 2 leaving the rest of m unchanged. And $m(1)(2)$ can be again a map, and so on and so forth.

Partial updates can be eliminated in the traditional ASM setting; see [Section 4]. But the development of AsmL required more practical algorithms for the integration of partial updates. The problem was complicated by the use of submachines in AsmL[1]. In violation of the bounded sequentiality assumption discussed above, the number of the steps of a submachine B of a program A is not bounded a priori. For example B may run to completion (that is until its state stabilizes), whatever number of steps it takes, during one step of A. Furthermore, B may have submachines of its own.

In this paper, we do not eliminate partial updates. We develop a systematic

[1] AsmL is not the first ASM-based language to employ submachines; see [A00, S99]. A theoretical study of submachines was conducted in [BS00].

solution of the partial-update problem that allows the programmer to freely use partial updates to modify counters, sets and maps in the main program and in submachines and in submachines of submachines, and so on, without worrying how submachines will report modifications and how to integrate modifications[2].

The basis of our approach is illustrated best on the example of a counter. How does a counter do in parallel all the increments? Typically it doesn't. Instead, it performs increments sequentially but the result does not depend on the order in which the updates are executed. So the presumed parallel execution is an abstraction.

At this point, we need a couple of mathematical definitions. Consider a collection F of unary operations over a set S. Recall that two operations f and g commute if $f(g(x)) = g(f(x))$ for all x in S. Say that f and g *malcommute* if $f(g(x)) \neq g(f(x))$ for all x in S. Say that F is *apt* if every two operations in F either commute or malcommute.

We view modifications, like counter increments or map alterations, as unary operations. It turns out that in each of the three cases mentioned above — counters, sets and maps — the collection of modifications is apt. In this paper, we restrict attention to *apt cases* where the collection of modifications is apt. Notice that it is easily checkable in apt cases whether two modifications f, g commute; just check $g(f(x)) = f(g(x))$ for any one x.

Now consider an apt case. Call a multiset M of modifications consistent if their composition is order independent. It is easy to check that the composition of the modifications in M is order independent if and only if every two modifications commute. This gives us a simple generic algorithm for checking consistency in apt cases. (In the cases of counters, sets and maps, more efficient algorithms for checking consistency follow from our results in [Sections 10-13]).

Any location ℓ can be modified by the main program as well as by submachines of all levels (submachines of the main program, their submachines, etc.). Consider the computation of any submachine M during a single step of its parent machine. Suppose M makes n steps. Let μ_i be the multiset of modifications of ℓ produced by M or reported by its submachines during the ith step of M. At the end of the ith step, M composes μ_i into one modification f_i. If $i < n$, then M applies f_i to the content of its local version of ℓ to update the current state of M. (Initially, when M is created by its parent, the content of M's version of ℓ is the current content the parent's version of ℓ.) After the nth step, M computes the composition $f_n \circ \cdots \circ f_1$ and reports it to its parent machine. (At this point, M dies, and its version of ℓ dies with it.) The main machine, at every step, composes all modifications produced by it and reported to it into

[2] Our treatment of maps extends that of an earlier version of [GSV01].

one modification which it applies to ℓ to update its own state. If inconsistency is detected in the process then the computation fails. The reporting and integration procedures are described in [Sections 7 and 8].

A disgruntled ASMer may not like that we "pollute" the clean ASM paradigm with partial updates. He/she may be comforted by the fact that partial updates can be eliminated. Besides, the current framework of apt cases is rather restricted. We intend to revisit the partial update problem.

In the rest of the paper, the traditional ASM setting means the setting without partial updates. The setting with partial updates will be called new.

We presume that the reader has some familiarity with abstract state machines (ASMs); the Lipari guide [G95] would do.

2 Preliminaries

2.1 Multisets

Intuitively, a multiset is a set with multiple occurrences of elements. For example, a multiset $\lceil a, b, b, a, b \rceil$ contains a with multiplicity 2 and b with multiplicity 3. The underlying set of a multiset μ will be called the *domain* of μ and denoted $\mathtt{dom}(\mu)$. Thus $\mathtt{dom}(\lceil a, b, b, a, b \rceil) = \{a, b\}$.

Every map m whose range consists of positive integers yields a multiset μ with $\mathtt{dom}(\mu) = \mathtt{dom}(m)$ where the multiplicity of an element a is $m(a)$. Every multiset μ can be obtained this way; the generating map is the *characteristic function* χ_μ of μ. The characteristic function of the empty multiset is the empty map.

Let μ_1 and μ_2 be multisets with domains d_1 and d_2 and characteristic functions χ_1, χ_2 respectively, let $d = d_1 \cup d_2$, and for each $i \in \{1, 2\}$, let f_i be the extension of χ_i to d such that $f_i(a) = 0$ for $a \in d - d_i$. The *sum* $m_1 + m_2$ is the multiset with domain d and characteristic function $f_1(a) + f_2(a)$.

We use ceiling brackets for multisets in the same way as braces are used for sets. By analogy with set comprehension, we use multiset comprehension. Let s be a set or multiset, let p be a unary relation over s, and let f be a function over s. Then

$$\lceil f(x) \mid x \in s \text{ where } p(x) \rceil$$

is a multiset μ with domain $d \rightleftharpoons \{f(x) \mid x \in s \text{ where } p(x)\}$. What is the characteristic function χ_μ of μ? If s is a set then $\chi_\mu(y)$ is the cardinality of the set $\{x \mid x \in s \text{ where } p(x) \wedge f(x) = y\}$. Otherwise

$$\chi_\mu(y) = \sum_{x \in s_y} \chi_s(x) \quad \text{where } s_y = \{x \mid x \in \mathtt{dom}(s) \text{ where } p(x) \wedge f(x) = y\}$$

Until now, we viewed multisets as generalizations of sets. There is another view of the notion of multiset, as an abstraction of the notion of sequence. You abstract from the order of the members of the sequence but not from the number of the occurrences of a member in the sequence. The second view suggests the implementation of multisets as sequences.

2.2 ASMs

As we said in the introduction, we assume that the reader is familiar with the Lipari Guide [G95]. In particular we assume that the reader knows the notions of vocabulary, state, location, update, update set, and rule. For brevity, elements of an ASM may be called points.

The update set of a rule R at a state X will be denoted $\Delta_X R$, and the value of a term (or expression) t at a state X will be denoted $\nabla_X t$. (This notation is borrowed from an earlier version of [GSV01].) The subscript may be omitted if the state is clear from the context.

Recall that an update set Δ of a state X is *consistent* if it contains at most one update per location. If Δ is consistent then $X + \Delta$ is the state which results from X by executing the updates in Δ.

2.3 The background of a state

In the ASM paradigm, a state is comprehensive. It is not reduced to indicating the current values of the variables. For example, if you deal with sets at all, then the state contains all sets that are or may become relevant later. The concept of state background was introduced in [BG00]; see also [GSV01] and [BG*]. The background of state does not change during state's evolution. It contains `true`, `false`, `undef`. Here we presume that the background includes integers and is closed under finite sets, multisets, sequences and maps. In other words, if x is a finite set of elements of a state X, or a finite multiset of elements of X, or a finite sequence of elements of X, or a finite map from elements of X to elements of X then x is an element of X. In principle, the background may contain some infinite sets, multisets, sequences or maps, but here we are interested only in finite ones. In the rest of the paper, by default, sets, multisets, sequences and maps are finite. (We still speak about the base set of a state even though it may be infinite, but in most cases of possibly infinite sets we try to use the term collection.)

Our map notation follows that of [GSV01]. It is pretty much self-explanatory. For example, the map $\{i_1 \mapsto r_1, i_2 \mapsto r_2\}$ has domain $\{i_1, i_2\}$ and maps i_1 to r_1, and i_2 to r_2. The atomic maps $\{i_1 \mapsto r_1\}$ and $\{i_2 \mapsto r_2\}$ are called *maplets*. An arbitrary map can be thought of as a set of maplets. The empty map is denoted $\{\rightarrow\}$.

Formally, maps are elements of the state, and the mapping information is contained in a static `apply` function: if m is a map and $\texttt{apply}(m, i) = r \neq \texttt{undef}$ then m includes the maplet $\{i \mapsto r\}$. The domain of a map m is the set $\{i \mid \texttt{apply}(m, i) \neq \texttt{undef}\}$ and the range is the set $\{\texttt{apply}(m, i) \mid i \in \texttt{dom}(m)\}$. (Notice that the range never contains `undef`.) We write $x(i)$ instead of $\texttt{apply}(x, i)$.

2.4 Sequential composition and submachines

In ASMs, the default composition of rules is the parallel composition. That is why the rule

```
do in-parallel
   P
   Q
```

is usually abbreviated to

```
P
Q
```

But the sequential composition

```
do sequentially
   P
   Q
```

is used as well though its semantics is more complicated [G00]. It is executed in two successive substeps. The second substep may overwrite some changes made during the first substep. If a clash occurs at either substep, the whole step is aborted.

Börger and Schmid use infix notation P `seq` Q [BS00] for sequential composition. AsmL uses notation

```
step
   P
step
   Q
```

so that the user pays a syntactic price for every substep. The hope is that the user will not use sequential composition without a good reason.

In the rest of this subsection, we recall (though not literally) some definitions from [BS00]. First we define the sequential composition of update sets Δ_1 and Δ_2. The intention is that Δ_1 is executed first, and then Δ_2 is executed in the state resulting from the execution of Δ_1. Recall that an update consists of a location and the new value. If u is an update, let $\mathrm{Loc}(u)$ be the location of u. If Δ is an update set then $\mathrm{Loc}(\Delta) = \{\mathrm{Loc}(u) : u \in \Delta\}$. We don't distinguish between inconsistent update sets.

Definition 2.1 If update sets Δ_1 and Δ_2 are consistent then

$$\Delta_1 \texttt{ seq } \Delta_2 \rightleftharpoons \{u \in \Delta_1 : \mathrm{Loc}(u) \notin \mathrm{Loc}(\Delta_2)\} \cup \Delta_2$$

And if Δ_1 or Δ_2 is inconsistent then the composition is the inconsistent update set. $\square$

Now the semantics of the sequential composition of rules can be defined.

Definition 2.2

$$\Delta_X(P \texttt{ seq } Q) \rightleftharpoons \Delta_X P \texttt{ seq } \Delta_Y Q$$

where $Y = X + \Delta_X(P)$. $\square$

Even though $\Delta_Y Q$ is an update set over Y, it is also an update set over X because X and Y have the same vocabulary and the same elements and therefore the same locations and the same possible updates; see [G00]. Thus the sequential composition of update sets makes sense.

Once the sequential composition is defined, its repeated application leads to the iteration of a rule:

$$R^n \rightleftharpoons \begin{cases} \texttt{skip} & n = 0 \\ R^{n-1} \texttt{ seq } R & n > 0 \end{cases}$$

If some $\Delta_X R^n$ is equal to $\Delta_X R^{n+1}$ then so is any $\Delta_X R^m$ with $m > n$. In this case $\Delta_X R^n$ is denoted $\lim_{k\to\infty} \Delta_X R^k$. If some $\Delta_X R^n$ is inconsistent then $\lim_{k\to\infty} \Delta_X R^k$ is inconsistent. If no $\Delta_X R^n$ is equal to $\Delta_X R^{n+1}$ or inconsistent then $\lim_{k\to\infty} \Delta_X R^k$ is undefined. The `iterate` rule has the following semantics:

$$\Delta_X(\texttt{iterate } R) \rightleftharpoons \lim_{k\to\infty} \Delta_X R^k$$

2.5 AsmL

AsmL (Abstract State Machine Language) is an advanced ASM-based specification language developed at Microsoft Research [FSE]. It extends the language of the parallel ASMs of the Lipari Guide [G95] in a number of directions. Some extensions are briefly described in [GSV01]. For the present paper, the two most relevant extensions are the use of submachines and partial updates. This paper provides the theoretical foundation for the treatment of partial updates in AsmL.

3 Challenges

In the introduction, we mentioned three manifestations of the partial-update problem. Here we expand on that theme.

3.1 Sets

One can think of a set in two different ways.

- A set is one entity. Accordingly it is natural to treat a set variable s as a nullary function and assign new values to it, e.g. `s := {1,2,3}`.
- A set is a composite object. Accordingly it is natural to treat a set variable s as a relation so that we can insert elements into and remove elements from the set, e.g. `s(1) := true`, or `s(2) := false`.

It may be desirable to exploit both points of view in a program so that one can overwrite a whole set as well as insert elements into it and remove elements from it. Furthermore, we should be able to make in parallel consistent changes that use both points of view, like this:

```
do in-parallel
   s := {1, 2, 3}
   s(1) := true
   s(4) := false
```

How to reconcile the two points of view?

3.2 Maps

In the case of maps, we have a similar problem. A map m can be seen as a single entity, or as a set of maplets $\{i \mapsto r\}$. You may want to replace the whole map m. You may want to add, remove or change a single maplet. Actually the map problem is more involved. It involves recursion. The value r of a maplet $\{i \mapsto r\}$ can be a map in its own right. You may want to zoom in and add, remove or change only a maplet of r. Again, you want to make in parallel consistent changes like this:

```
do in-parallel
   m(1) := { 2 ↦ 3 }
   m(2) := { 3 ↦ 4 }
   m(1)(2) := 3
   m(1)(3) := undef
```

Here the first and second clauses partially update m, and the third and fourth clauses partially update the submap $m(1)$ of m. Notice that we cannot see $m(1)$ as a mere abbreviation for $\texttt{apply}(m, 1)$ in these clauses because `apply` is a static function. So what should the clauses mean in the ASM world?

3.3 Counters

In this paper, a counter is integer-valued and the only counter-changing operations are these:

- $\texttt{incr}(k)$ increments the counter value by k.
- $\texttt{overwrite}(n)$ updates the counter value to n.

Again, you want to make consistent changes in parallel, and the question is how to program them properly in the ASM world.

4 Partial Updates in the Traditional Setting

The question arises how to deal with partial updates in the traditional ASM setting which has only total updates. One natural (to ASMers) point of view is that partial updates involve different levels of abstraction. A counter, for example, can be placed outside our system, as a separate vassal agent, with smaller steps. During one step, our program sends instructions to the counter, and the counter performs them in some order. Care should be taken on how our program will read the counter correctly.

Let us rephrase the question: Can partial updates be accommodated (rather than avoided) in the traditional ASM setting? Again, the answer is yes. We mention below two ways to achieve the goal.

Notice that the set challenge is dominated by the map challenge. In fact, sets can be represented by maps whose only possible value is `true`. On the other hand, the map challenge and the counter challenge are quite different. Two identical modifications of a map have the same effect as one of them while two identical increments of a counter accumulate. Changes to a map at two different indices (say at 1 and at 2) affect separate parts of the map. There is nothing like that in the case of a counter.

4.1 Registry Solution

4.1.1 Maps

The following solution is sketched in [GSV01]. Introduce an auxiliary binary dynamic function MR (an allusion to Map Registry). The first argument of MR is intended to represent a map variable; for simplicity we do not distinguish here between the variable and its representative. The second argument of MR is intended to be a sequence. Instead of changing a map variable m directly, register the changes with MR. For example, suppose that currently $m(1)$ is also a map and $m(1)(2) = 7$ and that you want to change $m(1)(2)$ to 11. Use the rule

$$\mathtt{MR}(m, [1, 2]) := 11$$

When all desired changes have been registered with MR, an additional *Appendix* step is made, in which the new values of the map variables are computed from the entries in MR and all locations of MR are reset to `undef`. This step fails if there are conflicting entries in MR.

4.1.2 Counters

As in case of maps, we can use an auxiliary registry function, say CR (an allusion to Counter Registry). But this registry function is ternary. The first argument is intended to reflect the variable in question, the second argument is intended to reflect the kind of modification, and the third argument is intended to be a new element (imported from the reserve). There are only two kinds of counter modifications: `incr` and `overwrite`. Instead of incrementing a counter c by n, import a new element r and assign n to $\mathtt{CR}(c, \mathtt{incr}, r)$. The element r is a tag that distinguishes this particular assignment from from any other.

Again, an additional step is required that integrates all modifications of any counter and overwrites all CR-locations to `undef`. It is an error if a nonzero increment and an overwrite are applied to the same counter.

Remark 4.1 One can argue with our policy of disallowing concurrent overwrites and nonzero-increments. There are other possible policies, e.g. add all increments to the overwrite value, or if increments and overwrites both occur then ignore all increments (or ignore all overwrites). Our policy will be justified in [Section 10]. □

4.1.3 The General Picture

The solutions for maps and counter challenges have the following common pattern.

First, register all modifications by means of an auxiliary registry function.

Second, use the registry to check for consistency and compute new values for your variables in a special *Appendix* step.

4.2 A General Result

A more general solution is given in [BG*]. There, different parts of a given computer system can send all kinds of messages to each other. In particular, a message could be a command to increment a counter c by 2 or a command to alter a map m at 1 to 7. Nevertheless, by the main theorem of [BG*], there is a traditional ASM that is behaviorally equivalent to A. Thus partial updates can be eliminated.

The main theorem of [BG*] applies only to bounded-sequentiality systems, with a fixed bound on the lengths of sequences of events that must occur in a particular order during any computation step. The parallel machines of the Lipari guide [G95] are like that. However, the partial-update elimination result is later generalized in [BG*] to the case of computer systems built from bounded-sequentiality systems by means of submachines.

5 Particles

In this and the following three sections, we develop a systematic approach to deal with partial updates directly.

In the traditional ASM setting, location contents are changed by updates and only by updates. That is preserved in the new setting; at the end of each step, an ASM produces a set of updates that are used to change location contents. The new aspect is this. During one step, in addition to updates, an ASM issues modifications, like increment a counter c by 7, or alter a map variable m at index 1 to 7. At the end of every step, all updates and modifications are integrated into updates.

What are these modifications? They can be viewed as unary operations over appropriate domains. The increment-by-7 modification is the operation

$$\texttt{incr}_7(x) \risingdotseq x + 7$$

over integers. The map alteration at index 1 to value 7 is an operation over maps; a map x is transformed into a map $y \risingdotseq \texttt{alter}_{1\mapsto 7}(x)$ that behaves as follows:

$$y(i) = \begin{cases} 7 & \text{if } i = 1 \\ x(i) & \text{otherwise} \end{cases}$$

5.1 Clients Types

ASMs of the Lipari guide are essentially untyped. We say "essentially" rather than "completely" because there is a separate Boolean type there. AsmL, because it is integrated into an industrial environment, is typed. Here we also use a bit of type discipline; this is not necessary but convenient.

We presume that there are types, like counters, sets and maps, which we will call *client types* for brevity. For each client type T, there is an associated type whose elements are modification operations over T. These modifications will be called *particles*, or more exactly T particles. For example, $\mathtt{incr}_7$ is a counter particle, and $\mathtt{alter}_{1\mapsto 7}$ is a map particle.

Remark 5.1 The term particle is admittedly strange. Here are our justifications of it. First, the concept is central to the new setting, and so we wanted it to have a simple name. Second, modifications are combined to produce a single update, and so they are parts, or particles, of the resulting update. Third, we recalled that, in quantum physics, particles can be seen as special functions, and so there is a precedent of calling some functions particles. $\square$

Intuitively, a T particle f is an operation on (the collection of elements of type) T, and so f can be applied to any T element a to produce another T element $f(a)$. But in the ASM paradigm of abstract states, a particle is just an element. To this end, we have a special particle-apply that takes a particle f and an element a and returns the desired $f(a)$.

Remark 5.2 In the ASM paradigm, the collection of states of an ASM is closed under isomorphisms, so that any structure isomorphic to a state is itself a state. Only the isomorphism type of the state matters. Thus we cannot assume that modification operations themselves always belong to the state. We can assume only that they are represented by some elements. For simplicity of exposition, we are going to ignore this pedantic point. $\square$

Further, particles f, g over a given client type T can be composed; $g \circ f$ is a T particle h such that $h(x) = g(f(x))$ for all x of type T. Here $\circ$ is a static binary operation in the state.

Remark 5.3 The application and composition functions may be polymorphic. Alternatively, there may be a separate pair of apply and composition functions for every client type T. We don't care. We will apply particles only to legitimate client elements, and we will compose only particles of the same particle universe. Having separate apply and composition functions for every client type may be useful if one contemplates a generalization where the composition of T particle may differ from the usual function composition. $\square$

5.2 Apt Client Cases

If T is a client type and if f, g are T particles, then a priori we have the following three scenarios:

1. $(g \circ f)(x) = (f \circ g)(x)$ for all x of type T, in other words $g \circ f = f \circ g$, so that f and g commute.
2. $(g \circ f)(x) \neq (f \circ g)(x)$ for all x of type T. In this case, we will say that f and g *malcommute*.
3. $(g \circ f)(x) = (f \circ g)(x)$ for some x of type T, and $(g \circ f)(x) \neq (f \circ g)(x)$ for some x of type T.

We will say that the case of a client type T is *apt* if every two T particles either commute or malcommute.

In this paper, we are primarily interested in apt particle types.

6 Partial Updates

Recall that traditional ASM updates are called *total updates* in this paper. Recall that a total update u is given by a pair (ℓ, v) where ℓ is a location and v is an element (intentionally, the new content of ℓ). Usually u and (ℓ, v) are identified. But here we have a problem. A partial update is also given by a pair (ℓ, f); this time around ℓ is a location of a client type T and f is a T particle. We don't want a partial update given by a pair (ℓ, f) to be confused with a total update given by the pair (ℓ, f). The total update given by a pair (ℓ, v) will be denoted $\mathrm{TU}(\ell, v)$, and the partial update given by a pair (ℓ, f) will be denoted $\mathrm{PU}(\ell, f)$.

Actually we would like to see total updates as special partial updates. To this end, we introduce *overwrite* particles

$$\texttt{overwrite}(y) : x \mapsto y$$

Our intention is to identify $\mathrm{TU}(\ell, v)$ with $\mathrm{PU}(\ell, \texttt{overwrite}(v))$ but we need to be careful. The world of updates is untyped. In a given state, any location ℓ and any point v form an update. The world of partial updates has some type discipline. We don't want to impose any particular type system. So we will just assume the following.

– If ℓ is a location of a client type T and v is a value of type T then $\mathrm{TU}(\ell, v)$ is identified with $\mathrm{PU}(\ell, \texttt{overwrite}(v))$ where the particle $\texttt{overwrite}(v)$ is a T particle.

– If ℓ is any other location and v is a legal content of ℓ according to your type system, then $\mathrm{TU}(\ell, v)$ is $\mathrm{PU}(\ell, \mathtt{overwrite}(v))$ where the type of the particle $\mathtt{overwrite}(v)$ is whatever is appropriate in your type system.

Of course, we will have to ensure that our treatment of overwrite particles is consistent with this identification.

Remark 6.1 One can avoid the second-clause complications by using only client-type overwrite particles. Then total updates of non-client locations do not fit our current partial-update framework. The remainder of this section can be easily adjusted to this. $\square$

For future reference, we note the following.

Lemma 6.2 *Suppose that T is any client type and consider two T particles such that one of them is an overwrite. Then (i) the composition of the two particles, in either order, is an overwrite, and (ii) the two particles either commute or malcommute.*

Proof The first claim is obvious, and the second follows from the first. $\square$

The ostensible meaning of a partial update $\mathrm{PU}(\ell, f)$ is to modify the current content v of ℓ to $f(v)$. But modifications (at least those modifications that are not total updates) do not really change location contents. They are integrated into total updates which can change location contents.

A *partial-update multiset* ψ is a multiset of partial updates. The concept of a partial-update multiset is a generalization of a set of total updates. In the case of partial updates, we really need multisets and not only sets. Think for example about counter increments.

6.1 Partial Update Rules

Traditionally, if you fire a rule R in a state X then the result is a set of updates of X. Now we deal with partial updates. Firing a rule R in a state X results in a multiset of partial updates of X; this multiset will be denoted $\tilde{\Delta}_X R$ (notice the tilde).

Definition 6.3 The *Partial Update Rule* (*Partial Assignment*) R is a rule

$$f(\bar{t}) \leftarrow t_0$$

where $f(\bar{t})$ is a term of some client type T, and t_0 is a term whose type is that of T particles. Let X be a state. Then

$$\tilde{\Delta}_X R \rightleftharpoons \lceil \mathrm{PU}(\ell, g) \rceil$$

where $\ell = (f, \nabla_X \bar{t})$ and $g = \nabla_X t_0$. $\square$

7 Integration

Consider one step of an ASM that transforms a state X to a state Y. At the end of a step, the ASM is supposed to produce an update set, that is a set of total updates. So the multiset ψ of partial updates, generated during the step, should be transformed into an update set. This is done by means of an *integrator* that integrates partial updates separately for each location ℓ.

Proviso 7.1 *All client cases are apt.*

7.1 Integration of Particles

By way of motivation, consider a counter c. Suppose that the current value of c is 0 and let f, g be increment particles with parameters 5 and 7 respectively. We expect that partial updates $\mathrm{PU}(c, f)$ and $\mathrm{PU}(c, g)$ will be integrated into a total update $\mathrm{TU}(c, 12)$. Does it mean that the counter performs the two increments simultaneously? Not necessarily. The counter can perform the two increments in the order they come to it. It is important though that the result does not depend on the order.

Definition 7.2 1. Suppose that T is a client type, and let M be a multiset $\lceil f_1, \ldots, f_n \rceil$ of T particles. M is *consistent* if every composition $f_{i_1} \circ \cdots \circ f_{i_n}$ of its members gives the same particle. Here $(i_1, \ldots, i_n)$ ranges over all permutations of $(1, \ldots, n)$.

2. A multiset of overwrite particles $\texttt{overwrite}(v_1), \ldots, \texttt{overwrite}(v_n)$ is *consistent* if $v_1 = \cdots = v_n$.

$\square$

Concerning the first part, the composition of zero particles is the identity particle. This justifies the obviously desired conclusion that the empty multiset of particles is consistent.

The two parts of the definition overlap but agree on the common multisets.

Lemma 7.3 *Suppose that T is a client type, and let M be a multiset of T particles. The following conditions are equivalent:*

1. *M is consistent.*
2. *Every two members of M commute.*

Proof Clearly *2* implies *1*. It remains to prove that *1* implies *2*. We assume that there are noncommuting particles f, g in M and prove that M is inconsistent. By the Proviso above, the case of T is apt, and so f and g malcommute. Let h be the composition of the other particles of M in any order and let x be any point. Since f and g malcommute, we have

$$(f \circ g \circ h)(x) = (f \circ g)(h(x)) \neq (g \circ f)(h(x)) = (g \circ f \circ h)(x)$$

and so M is inconsistent. $\square$

Notice that $f \circ g \circ h$ and $g \circ f \circ h$ differ at every element of type T.

Remark 7.4 Every particle type is closed under composition and thus forms a semigroup. In any semigroup, the composition of elements $f_1, \ldots, f_n$ is order independent if every two elements f_i, f_j commute. So pairwise commutativity is sufficient for order independence. However it is not necessary. Here is a counterexample. Consider matrices, say 2×2 integer matrices, together with the usual matrix multiplication. Choose some noncommuting matrices f_1, f_2 and let f_3 be the zero matrix. Multiply f_1, f_2, f_3 in any order; the result is always the zero matrix. Instead of matrices we could speak about linear transformations of the appropriate vector space. $\square$

Definition 7.5 If a multiset M of particles is consistent then the *product* (or *parallel composition*) $\Pi(M)$ of M is $f_1 \circ \cdots \circ f_n$; otherwise the product is undefined. $\square$

7.2 Integration of Partial Updates

Let ψ be a partial-update multiset, $\mathrm{Loc}(\psi)$ be the set of locations ℓ such that some partial update of ℓ occurs in ψ. Let ℓ range over $\mathrm{Loc}(\psi)$ and define ψ_ℓ to be the multiset $\lceil f \mid \mathrm{PU}(\ell, f) \in \psi \rceil$.

Definition 7.6 A partial-update multiset ψ is *consistent* if every ψ_ℓ is consistent. $\square$

We assume that ψ is consistent and explain how to integrate ψ. For each location $\ell \in \mathrm{Loc}(\psi)$, the integrator computes the product $\Pi(\psi_\ell)$. The new content of ℓ is $v_\ell \rightleftharpoons (\Pi(\psi_\ell))(\nabla_X \ell)$ and the resulting total update is $\mathrm{TU}(\ell, v_\ell)$.

We use this occasion to define the product $\Pi(\psi)$ of the partial-update multiset ψ:

$$\Pi(\psi) \rightleftharpoons \lceil \mathrm{PU}(\ell, \Pi(\psi_\ell)) \mid \ell \in \mathrm{Loc}(\psi) \rceil$$

8 Reporting

We saw above, in [Subsection 2.4], that the use of submachine requires sequential composition of update sets. In the presence of particles the situation becomes more complicated. We need sequential composition of particles obtained by parallel composition; this is addressed in [Subsection 8.1]. Further, the very notion of submachines needs a generalization; this is done in [Subsection 8.2].

8.1 Sequential Composition

Recall the [Definition 2.1] of the sequential composition of update sets. We are going to define the sequential composition of partial-update multisets ψ and ψ'. The intention is that ψ is executed first and then ψ' is executed in the state resulting from the execution of ψ'.

Definition 8.1 Let ψ and ψ' be two consistent partial-update multisets and let $L = \mathrm{Loc}(\psi) \cup \mathrm{Loc}(\psi')$. For each $\ell \in L$, let $g_\ell = \Pi\lceil f \mid \mathrm{PU}(\ell, f) \in \psi \rceil$ and $g'_\ell = \Pi\lceil f \mid \mathrm{PU}(\ell, f) \in \psi' \rceil$. Then

$$\psi \ \mathtt{seq} \ \psi' \rightleftharpoons \lceil \mathrm{PU}(\ell, g'_\ell \circ g_\ell) \ \mid \ \ell \in L \rceil$$

$\square$

Notice that the resulting partial-update multiset (actually a set) ψ `seq` ψ' is consistent when both ψ and ψ' are consistent. We don't distinguish between different inconsistent partial-update multisets. If ψ or ψ' is inconsistent then ψ `seq` ψ' is the inconsistent partial-update multiset.

Remark 8.2 We have $g'_\ell \circ g_\ell$ rather than $g_\ell \circ g'_\ell$ because of the way our composition works: $(f_2 \circ f_1)(x) = f_2(f_1(x))$. $\square$

Now we are ready to define the semantics of the sequential composition of rules that may involve partial updates.

Definition 8.3

$$\tilde{\Delta}_X(P \texttt{ seq } Q) \rightleftharpoons \tilde{\Delta}_X P \texttt{ seq } \tilde{\Delta}_Y Q$$

where Y is X plus the result of the integration of $\tilde{\Delta}_X P$. □

8.2 Submachines

We write

```
machine
   R
```

to encapsulate a rule R in a submachine. From the point of view of the parent program, a submachine B of a machine A is an oracle for A. During one step of A, B can have a complicated run during which it computes some modifications of the state of A. But B does not change the state of A. Instead, it reports the results to A. In the traditional setting, B reports an update set which is added to the update set of A; recall that we consider only one step of A. What should B report in the new setting? Notice that B may have submachines of its own which may have submachines of their own and so on. It seems reasonable to require that every submachine summarizes partial updates and reports a set of partial updates with at most one partial update per location.

If the partial-update multiset $\tilde{\Delta}_X R$ is consistent then `machine` R produces the set

$$\tilde{\Delta}_X(\texttt{machine } R) \rightleftharpoons \Pi(\tilde{\Delta}_X R)$$

of partial updates.

Remark 8.4 The submachine `machine` R behaves like the sequential composition of R and `skip`:

$$\tilde{\Delta}_X(\texttt{machine } R) = \tilde{\Delta}_X(R \texttt{ seq skip}) = \tilde{\Delta}_X(\texttt{skip seq } R)$$

holds for every rule R and every state X. □

Remark 8.5 In the traditional setting, the encapsulation is useful only if R is a sequential composition or iteration of rules. You can have a submachine

```
machine
   do in-parallel
      x := 3
      y := 5
```

but here the encapsulation does not buy you anything. This changes in the new setting. The following submachine, for example,

```
machine
   do in-parallel
      c  ← incr(1)
      c  ← incr(-1)
```

will report only a zero increment of c. This makes a difference. Suppose that the main program issues an overwrite of c at the same step, and there are no other modifications of c issued at the same step. An overwrite commutes with zero increment but does not commute with any other increment, and so the encapsulation above turns an inconsistent scenario to a consistent one. □

9 Preamble to Examples

In the rest of the paper we apply the theory developed in [Sections 5-8] to counters, sets, maps and one additional client type. In each case, we describe the particles, give a simple commutativity criterion and prove that the case is apt.

Some particles may have individual names. The identity particle

$$\texttt{identity} : x \mapsto x$$

is an example. But typically particles come in families containing many particles. Each particle family is the range of a special static function F of positive arity. For example, the counter incrementing particles form the range of the function `incr` with integer domain. Since the background is closed under tuples, we may assume without loss of generality that every family-generating function F is unary. Thus every particle f of the F family has the form $F(x)$ where F is a particle-family generating function and x is an argument for F. It is presumed that, if f belongs to the family of F, then the equation $f = F(x)$ has a unique solution. This solution will be called the *control* of f in F. It is produced by a special parameter recovery function $\texttt{ctrl}_F(f)$. We will allow ourselves to omit the index when it is obvious from the context.

10 Example: Counters

Think about a counter as a location holding an integer value which can be modified by overwrite and increment particles.

Remark 10.1 Instead of integer counters, one can deal with e.g. real counters. □

The increment particle performs a simple addition:

$$\texttt{incr}(k) : x \mapsto x + k \qquad \text{where } k \text{ is an integer}$$

Notice that in this example $\texttt{incr}(0)$ coincides with the identity particle.

The following theorem implies an algorithm which computes whether two particles commute.

Theorem 10.2 *Two counter particles commute and if and only if one of the following conditions holds.*

- *Both particles are increment particles.*
- *The particles are overwrite particles with the same control.*
- *One particle is an overwrite particle and the other one an increment particle with control zero.*

Proof Obvious. □

Theorem 10.3 *The case of counters is apt. In other words, every two counter particles either commute or malcommute.*

Proof By the previous theorem, one of the noncommuting particle is an overwrite. Use [Lemma 6.2]. □

Remark 10.4 In [Subsection 4.1.2], we mentioned different a-priori-possible policies for handling concurrent overwrites and increments. The particle framework leads us unambiguously to our policy where an overwrite and an increment can be executed in parallel if and only if the increment's control is zero. □

11 Example: Maps

In the case of maps, we have the identity particle and the overwrite and alter particle families. An alter particle changes only parts of the given map, leaving the rest of the map as it is. An overwrite particle, on the other hand, throws away the old value completely and replaces it with a new value.

On the first glance, the matter is simple. Think about a map as a set of maplets $i \mapsto v_i$. A simple alter particle affects only one maplet $i \mapsto v_i$ replacing the old value v_i with a new value v_i'. Accordingly the control should be a pair $[i, v_i']$. A more complicated alter particle affects several maplets replacing old values

v_i with new values v'_i. Accordingly the control should be a set $\{[i, v'_i] : i \in I\}$ and thus a map in its own right. This would work if we were willing to restrict attention to maps say from integers to integers, but we don't. The value v_i may be a map in its own right and you may want just to alter it rather than replace it. Thus the control of f may involve another alter particle g_i such that $v'_i = g_i(v_i)$. The control of g_i may involve yet another alter particle. In that sense the definition of alter particles involves recursion. We give a formal description of such recursive modifications in the next subsection where we abstract from our particles and deal with so-called transformers instead.

11.1 Map Transformers of Finite Rank

This is a mathematical digression whose goal is to identify a family of map-transforming particles called *alterers*.

11.1.1 The Setup

To simplify the exposition, we abstract from some aspects of ASMs so that our framework contains only things needed for this mathematical digression. So we assume the following.

S is a collection whose elements will be called atoms; one of the atoms is called `undef` and none of the atoms is a map. Let $S_0 = S$, and let S_{n+1} be the collection of maps m over $S_0 \cup \cdots \cup S_n$. (According to [Section 2], m is finite, that is domain of m is finite, and its range does not contain `undef`.) Let $S^* = S_0 \cup S_1 \cup S_2 \cup \cdots$. Every map over S^* is an element of S^*. In the rest of this subsection, a map is a map over S^*. Elements of S^* will be called points. Thus a point is either an atom or a map.

A binary operation `apply` on S^* reflects the behavior of maps:

$$\texttt{apply}(x, y) = \begin{cases} x(y) & \text{if } x \text{ is a map and } y \in \texttt{dom}(x) \\ \texttt{undef} & \text{otherwise} \end{cases}$$

Here $x(y)$ means of course the value of the map x at point y. The point of this trivial remark will become apparent in the next paragraph.

X_S is the structure formed by the collection S^*, and the binary operation `apply` and the nullary operations `undef` and `emptymap` (and possibly some additional operations which play no role in this subsection but may be needed in the next subsection). For each point (that is element) x, X_S "knows" whether x is a map, and if yes then what map it is exactly. In that sense, the nature of points is immaterial; instead of X_S, we can work with any isomorphic copy of X_S.

Therefore we cannot exploit the fact that elements of $S^* \setminus S$ are genuine maps, that is finite functions. As far as the structure X_S is concerned, they are just elements. It is the function `apply` that makes them behave as if they are maps. Instead of $\mathtt{apply}(x, y)$, we write $x(y)$, even if x is not a map. This makes every point look like a total function which equals `undef` almost everywhere. As far as `apply` is concerned, every atom looks like the empty map (but only the empty map is the interpretation of the nullary symbol `emptymap` in X_S).

T is the collection of all unary operations over S^*. Elements of T will be called transformers. Even though a transformer f is applied to every point, it is convenient to think about it as primarily a map transformer. In this connection, the argument of the transformer will be often denoted m.

C is the collection of finite functions c from S^* to T. In other words, the domain of any $c \in C$ is a finite subset of S^* and the range is a subset of T. Elements of C will be called controllers. This terminology is justified by the following definition.

11.1.2 Ranked Transformers and Controllers

Definition 11.1 An element $c \in C$ *controls* a transformer f if, for all points m and x,

$$(fm)(x) = \begin{cases} (cx)(mx) & \text{if } x \in \mathtt{dom}(c) \\ mx & \text{otherwise} \end{cases}$$

□

Here $fm = f(m)$, $cx = c(x)$ and $mx = m(x)$. For better readability, we allow ourselves to omit some parentheses when the intended meaning is clear from the context.

Let us try to explain the definition. A transformer f transforms any point m into a map $m' = f(m)$. If m is not a map then $m' = f(\mathtt{emptymap})$. The interesting case is when m is a map. So let m be a map, and let x be any point. The question is what is $m'(x)$. If $x \in \mathtt{dom}(c)$ then $c(x)$ is a transformer; in this case $m'(x)$ is obtained by applying the transformer $c(x)$ to $m(x)$. Otherwise $m'(x) = m(x)$.

Example 11.2 Let $\mathtt{const}_y$ be the constant transformer such that $\mathtt{const}_y(x) = y$. Let c be the controller $\{0 \mapsto \mathtt{const}_1\}$ and let f be the function controlled by c. If m is an atom (say orange; we cannot rule out that some our points are oranges, and oranges are no maps) then $f(m)$ is the map $\{0 \mapsto 1\}$. Indeed $(fm)(0) = (c0)(m0) = 1$ and, if $x \neq 0$ then $(fm)(x) = m(x) = \mathtt{undef}$. □

Definition 11.3 Let c be a controller. For every point x, we define an associated transformer

$$c_x \rightleftharpoons \begin{cases} c(x) & \text{if } x \in \texttt{dom}(c) \\ \texttt{identity} & \text{otherwise} \end{cases}$$

□

Example 11.4 Let c be again $\{0 \mapsto \texttt{const}_1\}$. Then $c_0 = \texttt{const}_1$ and every other $c_x = \texttt{identity}$. □

Corollary 11.5 *A transformer f is controlled by a controller c if and only if $(fm)(x) = c_x(mx)$ for all points m, x.*

Lemma 11.6 *Every transformer has at most one controller.*

Proof By contradiction assume that a transformer f has two distinct controllers c and d. Then there are points x, y such that $c_x(y) \neq d_x(y)$. Now let $m = \{x \mapsto y\}$. We have

$$(fm)(x) = c_x(mx) \neq d_x(mx) = (fm)(x)$$

which is impossible. □

By induction on n, we define controllers and transformers of rank n.

Definition 11.7 – A transformer f is of rank 0 if it is a constant transformer.

- The empty controller is of rank 0. A nonempty controller c is of rank n if its range consists of transformers of rank $\leq n$ and there is at least one transformer of rank n there.
- A transformer f is of rank $n > 0$ if there is a controller of rank $n - 1$ that controls f.

Transformers of positive rank are *alterers*. □

Corollary 11.8 *If c is a controller such that every transformer in the range of c is ranked then c is ranked.*

Proof The empty controller has rank 0. If c is nonempty, then the rank of c is the maximal rank of transformers in the range of c. (We use the fact that the domain of c is finite. It implies that the range of c is finite and so the maximal rank in question exists.) □

Lemma 11.9 *The collection of ranked transformers is closed under composition.*

Proof Let f and g be ranked transformers. By induction on $\text{rank}(f) + \text{rank}(g)$ we prove that the transformer $h \rightleftharpoons g \circ f$ is ranked. If f or g is constant then so is h. So we can assume that f and g have positive ranks. Let c, d be the controllers of f, g respectively. Using [Corollary 11.5], we have

$$(hm)(x) = (g(fm))(x) = d_x((fm)x) = d_x(c_x(mx))$$

We construct a controller e such that $e_x = d_x \circ c_x$ for all points x. (To ease reading, we abbreviate `identity` to *id*.)

- The domain of e is $\texttt{dom}(c) \cup \texttt{dom}(d)$. So if $x \notin \texttt{dom}(e)$ then $e_x = id = id \circ id = d_x \circ c_x$.
- If $x \in \texttt{dom}(c) - \texttt{dom}(d)$ then $e(x) = c(x)$. For such x, $e_x = c_x = id \circ c_x = d_x \circ c_x$. Since c is ranked, $e(x)$ is ranked.
- If $x \in \texttt{dom}(d) - \texttt{dom}(c)$ then $e(x) = d(x)$. For such x, $e_x = d_x = d_x \circ id = d_x \circ c_x$. Since d is ranked, $e(x)$ is ranked.
- If $x \in \texttt{dom}(c) \cap \texttt{dom}(d)$, set $e(x) = d(x) \circ c(x)$. For such x, $e_x = d(x) \circ c(x) = d_x \circ c_x$. Notice that $\text{rank}(cx) \leq \text{rank}(c) < \text{rank}(f)$. Similarly, $\text{rank}(dx) < \text{rank}(g)$. Thus $\text{rank}(cx) + \text{rank}(dx) < \text{rank}(f) + \text{rank}(g)$. By the induction hypothesis, $e(x)$ is ranked.

By [Corollary 11.5], e controls h. By the construction of e, the range of e consists of ranked transformers. By [Corollary 11.8], e is ranked. Thus h is ranked. □

11.2 Maps and Map Particles

Now we are in a position to define the relevant maps and the relevant particles properly. Let X be a state and S any collection of non-map elements of X that contains `undef`. For example, S could be the collection of integers or binary strings extended with `undef`. Our X is like X_S except that it contains more elements and its vocabulary is richer. In the rest of this section, we deal only with maps that belong to S^*. Strictly speaking, we should speak about S-maps. In order to simplify the exposition, we just speak about maps. In any case, we have a well defined notion of alterers.

Remark 11.10 Let us clarify the sentence about X being like X_S. Since X is closed under maps (see [Subsection 2.3]), X includes the whole collection S^*

constructed in the previous section. As a result, the structure X_S of the previous section is a substructure but not of X itself but rather of the structure obtained from X by removing all function names except for `apply`, `undef` and `emptymap` from the vocabulary of X. (In logic terms, X_S is a substructure of a reduct of X). $\square$

The map particles of a state X consist of the identity particle, (map-) overwrite and alter particles of X. The alter family of particles consists of alterers, that is of positive-rank transformers. The controller of an alterer f is the control of the particle f. Notice that the control itself is a finite map from elements of X to elements of X. The particle $f = \texttt{alter}(\{\rightarrow\})$ with the empty control is similar to but does not coincide with the identity particle over maps: If x is an atom (that is an element of A), then $f(x) = \{\rightarrow\} \neq x$. Thus we do need a separate identity particle.

The overwrite particles are exactly transformers of rank 0. Such transformers do not have controllers in the sense of the previous subsection but they do have controls: the control of $\texttt{overwrite}(y)$ is the point y. (It is because of overwrite particles that we have to distinguish between controls and controllers.)

Lemma 11.11 *Let f and g be alter particles and let $c = \texttt{ctrl}\, f$ and $d = \texttt{ctrl}\, g$. Then f and g commute if and only if c_x and d_x commute for all $x \in \texttt{dom}(c) \cap \texttt{dom}(d)$.*

Proof If $x \notin \texttt{dom}(c)$ then $c_x = \texttt{identity}$ and therefore c_x and d_x commute. Similarly, c_x and d_x commute if $x \notin \texttt{dom}(d)$. Thus it suffices to prove that f and g commute if and only if, for every point x, c_x and d_x commute. Let $h = g \circ f$, $h' = f \circ g$, $e_x = d_x \circ c_x$ and $e'_x = c_x \circ d_x$. We show that h and h' differ at some point if and only if there exist points x, y such that $e_x(y) \neq e'_x(y)$.

First suppose that h and h' differ at some m which means that the maps hm and $h'm$ differ at some point x. Using [Corollary 11.5], we show that $e_x(mx) \neq e'_x(mx)$.

$$\begin{aligned} e_x(mx) = d_x(c_x(mx)) = d_x((fm)(x)) = (g(fm))(x) = (hm)(x) \\ \neq (h'm)(x) = (f(gm))(x) = c_x((gm(x)) = c_x(d_x(mx)) = e'_x(mx) \end{aligned}$$

Second suppose that, for some points x, y, $e_x(y) \neq e'_x(y)$. We show that h and h' differ at $m \rightleftharpoons \{x \mapsto y\}$.

$$\begin{aligned} (hm)(x) = (g(fm))(x) = d_x((fm)(x)) = d_x(c_x(mx)) = e_x(mx) = e_x(y) \\ \neq e'_x(y) = e'_x(mx) = c_x(d_x(mx)) = c_x((gm)(x)) = (f(gm))(x) = (h'm)(x) \end{aligned}$$

$\square$

The following Theorem implies an algorithm which computes whether two particles commute.

Theorem 11.12 *Two map particles f and g commute if and only if one of the following conditions holds.*

1. *f or g is the identity particle.*
2. *Both f and g are overwrite particles and $\mathtt{ctrl}\, f = \mathtt{ctrl}\, g$.*
3. *f is an alter particle, g is an overwrite particle and $f(\mathtt{ctrl}\, g) = \mathtt{ctrl}\, g$.*
4. *f is an overwrite particle, g is an alter particle and $g(\mathtt{ctrl}\, f) = \mathtt{ctrl}\, f$.*
5. *Both f and g are alter particles with $c = \mathtt{ctrl}\, f$, $d = \mathtt{ctrl}\, g$ and c_x and d_x commute for all $x \in \mathtt{dom}(c) \cap \mathtt{dom}(d)$.*

Proof First, assume that f and g commute. If f or g is the identity particle, condition *1* holds. Otherwise, if f or g is an overwrite particle, it is obvious that condition *2*, *3* or *4* holds. For the remaining case that both f and g are alter particles it follows from [Lemma 11.11] that condition *5* holds.

Second, each of the conditions *1-4* obviously implies commutativity. Condition *5* implies commutativity by [Lemma 11.11]. □

Theorem 11.13 *The case of maps is apt. In other words, every two map particles either commute or malcommute.*

Proof Let f and g be map particles which do not commute. We will show that they malcommute.

Since the identity particle commutes with every other particle, the claim is trivial in the case when f or g is the identity particle. We assume that neither of them is the identity particle and prove the claim by induction on $\text{minrank}(f, g) \rightleftharpoons \min\{\text{rank} f, \text{rank} g\}$.

The case $\text{minrank}(f, g) = 0$. At least one of f, g is an overwrite. Use [Lemma 6.2].

The case $\text{minrank}(f, g) > 0$. Both f and g are alter particles. Let $c = \mathtt{ctrl}\, f$ and $d = \mathtt{ctrl}\, g$ be the controllers. Assume that f, g do not commute. By [Lemma 11.11], there is a point $x \in \mathtt{dom}(c) \cap \mathtt{dom}(d)$ such that c_x and d_x do not commute. Clearly $\text{minrank}(c_x, d_x) < \text{minrank}(f, g)$. By the induction hypothesis c_x and d_x differ at every point. Let m be any point. We show that $h(m) \neq h'(m)$. It suffices to show that $(hm)(x) \neq (h'm)(x)$. We have

$$\begin{aligned} (hm)(x) &= g(f(m))(x) = d_x((fm)x) = d_x(c_x(mx)) \\ &\neq c_x(d_x(mx)) = c_x((gm)x) = f(g(m))(x) = (h'm)(x) \end{aligned}$$

□

12 Example: Counter Maps

Counter maps are a marriage of maps and counters. In addition to overwrite and alter particles, counter maps admit increment particles. You can see counter maps as a generalization of multisets in two directions: negative multiplicities and nested map-like structure.

First we extend the background structure to accommodate counter maps. The rest of this section is similar to the previous section; we explain the necessary changes.

12.1 Counter Map Background

Recall that `undef` does not belong to the range of a map. The meaning of $m(x) =$ `undef` is that m is not defined at x. In this section we consider a similar collection of finite functions except that the role of `undef` is played by number zero.

Let us make this more precise. In our abstract states, a map is any element x such that either x is the interpretation of the name `emptymap` or else the set $\{y \mid \mathtt{apply}(x, y) \neq \mathtt{undef}\}$, the domain of x, is finite and nonempty. Counter maps are defined in exactly the same way except that (i) another apply function, $\mathtt{apply}'$, is used and (ii) the set $\{y \mid \mathtt{apply}(x, y) \neq \mathtt{undef}\}$ is replaced with the set $\{y \mid \mathtt{apply}'(x, y) \neq 0\}$, the *domain* of counter map x. One can say that $\mathtt{apply}'$ defaults to 0 while `apply` defaults to `undef`. In this section, $\mathtt{apply}'(x, y)$ is abbreviated to $x(y)$.

In [Section 2] we assumed that the state background is closed under maps. Here we assume that, in addition, the state background is closed under counter maps. In other words, if m is a counter map from elements of the state to elements of the state then m itself is an element of the state.

12.2 The Setup

As in the setup of the previous section, S is a collection whose elements will be called atoms and which contains `undef`; this time we assume that it includes the integers as well. No atom is a counter map.

We build collections $S_0, S_1, \ldots$ and the collection $S^* = S_0 \cup S_1 \cup S_2 \cup \cdots$ as in the previous section except that this time we use counter maps rather than usual maps. As before, T is the collection of all unary functions over S^*, called transformers, and C is the collection of finite functions from S^* to T, called controllers.

12.2.1 Ranked Transformers and Controllers

We reuse the previous-section's definitions of controlled transformers and constant transformers (except that now we are talking about counter maps rather than usual maps). In addition, we use an additional category of transformers:

Definition 12.1 A transformer f is an *increment*, if there exists an integer number k such that $f(x) = x \oplus k$ for every point x where

$$x \oplus k \rightleftharpoons \begin{cases} x + k & \text{if } x \text{ is an integer} \\ \texttt{undef} & \text{otherwise} \end{cases}$$

□

The operator $\oplus$ is a modified addition operator to be applied not only to integers but also to other elements of S^*. Notice that, contrary to $\texttt{apply}'$, $\oplus$ defaults to `undef`. We make the following observations on the composition of increment and controlled transformers:

Lemma 12.2 *Let f be a controlled transformer and g an increment transformer. Then $g \circ f$ and $f \circ g$ are constant transformers, and f and g malcommute.*

Proof First, consider $g \circ f$. By the definition of controlled transformers, $f(x)$ is always a counter map. Hence, by definition of $\oplus$, $g(f(x)) = \texttt{undef}$ because $f(x)$ is not an integer.

Second, consider $f \circ g$ and let x be any element of S^*. Then $g(x)$ is an integer, if x is an integer, or `undef` otherwise. In any case, $g(x)$ is an atom; let's call it a. Since a is an atom, $a(y) = 0$ for all points y. Further, let c be the controller of f. By [Definition 11.1] (adjusted to counter maps), $f(a)$ is a counter map and

$$(fa)(y) = \begin{cases} (cy)(0) & \text{if } y \in \texttt{dom}(c) \\ 0 & \text{otherwise} \end{cases}$$

and so $f(g(x)) = f(a)$ does not depend on x and differs from $g(f(x)) = \texttt{undef}$. □

Modify the previous-section's notion of ranked transformers by including the increment transformers into the category of transformers of rank 0.

Lemma 12.3 *The collection of (revised) ranked transformers is closed under composition.*

Proof The proof of [Lemma 11.9] remains valid except that the base of induction needs two additional clauses.

1. If both f and g are increment transformers then $g \circ f$ is an increment transformer.
2. If one of the transformers is an increment and the other is an alterer, then, by [Lemma 12.2], their composition, in either order, is a constant transformer.

□

12.3 Counter Maps and Counter-Map Particles

The revised definitions and lemmas allow us to define the relevant counter maps and counter-map particles properly. Let X be a state and S any collection of non-counter-map elements of X that contains `undef` and includes integers. Our X is like the previous-section's X_S (adjusted to counter maps) except that it contains more elements and its vocabulary is richer. Thus we have again a well defined notion of alterers; this time we also have the notion of increment.

As in the previous section, alterers give rise to the family of alter particles. The family of increment particles consists of increment transformers. The control of an increment particle $f : x \mapsto x \oplus k$ is k. Given an integer k, we write $\texttt{incr}(k)$ for the increment particle with control k.

Notice that, in contrast to the counter case, $\texttt{incr}(0)$ is not the identity particle, as $\texttt{incr}(0)(x) = \texttt{undef} \neq x$ for every non-integer point x.

The counter-map particles of a state X consist of the identity particle and (counter-map) overwrite, increment and alter particles. Notice that the overwrite particle as well as the increment particle is a transformer of rank 0. We explore the relation of increment and overwrite particles in the next definition and lemma.

Definition 12.4 Particles $\texttt{incr}(k)$ and $\texttt{overwrite}(y)$ are *compatible* if

1. $y = \texttt{undef}$, or
2. y is an integer and $k = 0$.

□

Lemma 12.5 *An increment and an overwrite particle are compatible if and only if they commute.*

Proof Let $f = \texttt{incr}(k)$ and $g = \texttt{overwrite}(y)$.

First, let f and g commute. We show that one of the conditions of [Definition 12.4] holds. By contradiction assume that y is neither `undef` nor an integer;

then so is $g(f(x)) = y$. The other composition $f(g(x)) = y \oplus k$ is either `undef` or an integer by definition of $\oplus$. Thus $g(f(x)) \neq f(g(x))$ which cannot be as f and g commute. Therefore our assumption was wrong and y is either `undef` or an integer. If y is `undef`, condition *1* holds. So we may assume that y is an integer. By contradiction assume that $k \neq 0$. Then $f(g(x)) = y \oplus k = y + k \neq y = g(f(x))$. This cannot be as f and g commute. Therefore our assumption was wrong and $k = 0$, so that condition *2* holds.

Second, suppose that condition *1* or *2* of [Definition 12.4] holds. We show that f and g commute.

- If $y = \texttt{undef}$, then for every x holds

$$f(g(x)) = f(\texttt{undef}) = \texttt{undef} \oplus k = \texttt{undef} = y = g(f(x))$$

- If y is an integer number and $k = 0$, then for every x holds

$$f(g(x)) = f(y) = y \oplus 0 = y + 0 = y = g(f(x))$$

□

The following theorem implies an algorithm which computes whether two particles commute.

Theorem 12.6 *Two counter-map particles commute if and only if one of the following conditions holds.*

1. *Any case mentioned in [Theorem 11.12].*
2. *Both are increment particles.*
3. *They are compatible increment and overwrite particles.*

Proof Let f and g be counter-map particles.

First, assume that f and g commute. If neither of them is an increment particle, then the first part of the proof of [Theorem 11.12] remains valid unchanged and implies that condition *1* must hold, so we may assume without loss of generality that one of them, say f, is an increment particle. If the other, say g, is also an increment particle, condition *2* holds obviously. If g is an overwrite particle it follows from [Lemma 12.5] that condition *3* holds. Finally it follows from [Lemma 12.2] that g is not an alter particle.

Second, assume that one of the conditions holds. If condition *1* holds, the second part of the proof of [Theorem 11.12] remains valid unchanged. Condition *2* obviously implies commutativity. If condition *3* holds then it follows from [Corollary 12.5] that f and g commute. □

Theorem 12.7 *The case of counter maps is apt. In other words, every two map particles either commute or malcommute.*

Proof Let f and g be counter-map particles. The proof of [Theorem 11.13] is fine except that the base of induction changes as follows:

The case $\text{minrank}(f, g) = 0$. At least one of f, g is either an overwrite or an increment particle.

If one is an overwrite particle, use [Lemma 6.2]. So we can assume that one is an increment particle, say f, and g is not an overwrite particle. Then the claim is trivial if g is the identity or an increment particle. If g is an alter particle, use [Lemma 12.2]. □

13 Example: Sets

Let X be a state. In this section, we operate on a nonempty and possibly infinite collection S of sets (that is finite sets) in the state X. Each of these sets consists of elements of X and is an element of X. We presume that the collection S is closed under union and set difference (and thus under intersection).

We consider the insertion of one or more elements and the removal of one of more elements as special operations on sets in this section. Insertions and removals may occur in parallel. They are consistent as long as no element is inserted and removed at the same time. We call the joint insert-and-remove operation *insrem*.

Example 13.1 If S contains all finite subsets of integers then we have, in particular, the following insrem operations: $f(x) \rightleftharpoons x \cup \{1, 7, 13\}$, $g(x) \rightleftharpoons x \setminus \{42\}$ and $(g \circ f)(x) = (x \cup \{1, 7, 13\}) \setminus \{42\}$. □

Remark 13.2 In applications, we have encountered only insertions, removals, the combinations of the two, and overwrites. One may want to consider various additional operations and first of all intersection. We notice only that the integration of the intersection operation into the particle framework of this section is straightforward. Instead of *insrem*, there would be the combined operation of insertion, removal and intersection, say *insremint*. □

The set particles consist of overwrite particles and insrem particles. The control of an insrem particle consists of a pair $[y_+, y_-]$ of disjoint sets where y_+ contains the elements which are to be inserted and y_- those which are to be removed.

$$\texttt{insrem}([y_+, y_-]) : x \mapsto x \cup y_+ \setminus y_- \qquad \text{where } y_+ \cap y_- = \emptyset$$

Remark 13.3 When we use the set operations union and difference together without parentheses, we assume left associativity. So $x \cup y_+ \setminus y_-$ means $(x \cup y_+) \setminus y_-$. $\square$

Notice that in this example $\texttt{insrem}([\emptyset, \emptyset])$ coincides with the identity over sets, so `identity` is one of set particles.

Lemma 13.4 *The collection of set particles is closed under composition.*

Proof Let f and g be set particles. If f or g is an overwrite particle, the composition $g \circ f$ also is an overwrite particle, so we may assume that both f and g are insrem particles. Let $f = \texttt{insrem}([x_+, x_-])$ and $g = \texttt{insrem}([y_+, y_-])$. Define

$$h = \texttt{insrem}([x_+ \setminus y_- \cup y_+,\ x_- \setminus y_+ \cup y_-])$$

We first show that h is a legal insrem particle, that is that the part to be inserted and the part to be removed are disjoint. Second, we will show that $h = g \circ f$.

1. By contradiction assume that some $\alpha \in (x_+ \setminus y_- \cup y_+) \cap (x_- \setminus y_+ \cup y_-)$. Then $\alpha \in (x_+ \setminus y_-)$ or $\alpha \in y_+$.
   (a) If $\alpha \in x_+ \setminus y_-$, then $\alpha \in x_+$. Hence $\alpha \notin x_-$ and therefore $\alpha \notin x_- \setminus y_+$. As α belongs to the union $x_- \setminus y_+ \cup y_-$ but not to the first summand, it must belong to the second summand: $\alpha \in y_-$. But then $\alpha \notin x_+ \setminus y_-$ which contradicts the assumption of this case.
   (b) If $\alpha \in y_+$, then $\alpha \notin y_-$ by definition of insrem. Also, $\alpha \notin x_- \setminus y_+$. Therefore $\alpha \notin x_- \setminus y_+ \cup y_-$ which contradicts the choice of α.
2. For every set z the following holds:

$$\begin{aligned} g(f(z)) &= z \cup x_+ \setminus x_- \cup y_+ \setminus y_- \\ &\overset{(1)}{=} z \cup (x_+ \setminus y_-) \setminus x_- \cup y_+ \setminus y_- \\ &\overset{(2)}{=} z \cup (x_+ \setminus y_-) \cup y_+ \setminus (x_- \setminus y_+) \setminus y_- \\ &= z \cup (x_+ \setminus y_- \cup y_+) \setminus (x_- \setminus y_+ \cup y_-) \\ &= h(z) \end{aligned}$$

   where (1) holds because y_- is removed from the set in the end of the expressions anyway, and (2) because $z \cup y_+ \setminus (x_- \setminus y_+) = z \setminus x_- \cup y_+$ for every set z.

$\square$

Lemma 13.5 *Let $f = \mathit{insrem}([x_+, x_-])$ and $g = \mathit{insrem}([y_+, y_-])$. Then f and g malcommute if $(x_+ \cup y_+) \cap (x_- \cup y_-) \neq \emptyset$.*

Proof Let $\alpha \in (x_+ \cup y_+) \cap (x_- \cup y_-)$. Without loss of generality, $\alpha \in x_+$. It follows that $\alpha \notin x_-$. Hence $\alpha \in y_-$. For every set z', $\alpha \in z' \cup x_+ \setminus x_-$ and $\alpha \notin z' \setminus y_-$. This implies the following where the expressions in parentheses play the role of z'.

$$\alpha \in (z \cup y_+ \setminus y_-) \cup x_+ \setminus x_- = f(g(z))$$
$$\alpha \notin (z \cup x_+ \setminus x_- \cup y_+) \setminus y_- = g(f(z))$$

Therefore $f(g(z)) \neq g(f(z))$ for every set z. $\square$

Lemma 13.6 *Let $f = \mathit{insrem}([x_+, x_-])$ and $g = \mathit{insrem}([y_+, y_-])$. Then f and g commute if and only if $(x_+ \cup y_+) \cap (x_- \cup y_-) = \emptyset$.*

Proof First, let f and g commute. By contradiction assume $(x_+ \cup y_+) \cap (x_- \cup y_-) \neq \emptyset$. Then, by [Lemma 13.5], f and g malcommute. That is a contradiction and so the set is empty.

Second, let $(x_+ \cup y_+) \cap (x_- \cup y_-) = \emptyset$. If follows that x_- and y_+ are disjoint, and so are x_+ and y_-. Thus

$$z \setminus x_- \cup y_+ = z \cup y_+ \setminus x_- \tag{1}$$
$$z \cup x_+ \setminus y_- = z \setminus y_- \cup x_+ \tag{2}$$

for every set z. Therefore the following holds for every set z:

$$\begin{aligned} g(f(z)) &= z \cup x_+ \setminus x_- \cup y_+ \setminus y_- \\ &\overset{(1)}{=} z \cup x_+ \cup y_+ \setminus x_- \setminus y_- \\ &= z \cup (x_+ \cup y_+) \setminus (x_- \cup y_-) \\ &= z \cup (y_+ \cup x_+) \setminus (y_- \cup x_-) \\ &= z \cup y_+ \cup x_+ \setminus y_- \setminus x_- \\ &\overset{(2)}{=} z \cup y_+ \setminus y_- \cup x_+ \setminus x_- \\ &= f(g(z)) \end{aligned}$$

$\square$

The following theorem implies an algorithm which computes whether two particles commute.

Theorem 13.7 *Two set particles f and g commute if and only if one of the following conditions holds.*

1. *Both are overwrite particles with the same control.*
2. *f is an insrem particle, $g = \texttt{overwrite}(y)$ and $f(y) = y$.*
3. *$f = \texttt{overwrite}(y)$, g is an insrem particle and $g(y) = y$.*
4. *Both are insrem particles with $f = \texttt{insrem}([x_+, x_-])$ and $g = \texttt{insrem}([y_+, y_-])$ such that $(x_+ \cup y_+) \cap (x_- \cup y_-) = \emptyset$.*

Proof First, assume that f and g commute. If f or g is an overwrite particle it is obvious that one of conditions *1* to *3* must hold. If both f and g are insrem particles, [Lemma 13.6] implies condition *4*.

Second, assume that one of the conditions holds. Each of the conditions *1* to *3* obviously implies commutativity. If condition *4* holds, then the commutativity follows from [Lemma 13.6]. $\square$

Theorem 13.8 *The case of sets is apt. In other words, every two set particles either commute or malcommute.*

Proof Let f and g be set particles. If one of them is an overwrite particle, use [Lemma 6.2]. So we may assume that f and g are insrem particles. Let $[x_+, x_-] = \texttt{ctrl}\, f$ and $[y_+, y_-] = \texttt{ctrl}\, g$. Assume that f and g do not commute. Then, by [Lemma 13.6], the set $(x_+ \cup y_+) \cap (x_- \cup y_-) \neq \emptyset$. It follows from [Lemma 13.5] that f and g malcommute. $\square$

Remark 13.9 There exists a mapping of sets to maps with range $\{\texttt{true}\}$ which turns set particles into appropriate map particles. We give an informal description of this mapping.

A set x becomes the map $\hat{x} = \{\alpha \mapsto \texttt{true} \mid \alpha \in x\}$. (Clearly, the set can be reconstructed from the map.)

A set-overwrite particle $\texttt{overwrite}(x)$ corresponds to the map-overwrite particle $\texttt{overwrite}(\hat{x})$. An insrem particle $\texttt{insrem}([x_+, x_-])$ corresponds to the map particle

$$\begin{aligned}\texttt{alter}(\quad &\{\alpha \mapsto \texttt{overwrite}(\texttt{true}) \mid \alpha \in x_+\}\\ &\cup \{\alpha \mapsto \texttt{overwrite}(\texttt{undef}) \mid \alpha \in x_-\})\end{aligned}$$

$\square$

Acknowledgment

Discussions with all our colleagues in the Microsoft Research group on Foundations of Software Engineering and especially with Andreas Blass were very helpful. Margus Veanes worked on map modifications early on. Last-moment remarks of the editor, Egon Börger, contributed to clarity.

References

[A00] Matthias Anlauff, "XASM — an extensible component-based abstract state machine language", in *Abstract State Machines: Theory and Applications*, Y. Gurevich et al., editors, Springer-Verlag, Lecture Notes in Computer Science 1912 (2000), 69–90.

[ASM] ASM Michigan web page, `http://www.eecs.umich.edu/gasm`, maintained by Jim Huggins.

[BG00] Andreas Blass and Yuri Gurevich, "Background, reserve, and Gandy machines," in *Computer Science Logic*, P. Clote and H. Schwichtenberg, editors., Springer-Verlag, Lecture Notes in Computer Science 1862 (2000), 1–17.

[BG*] Andreas Blass and Yuri Gurevich, "Abstract state machines capture parallel algorithms," to appear. In the meantime, the paper can be found at `http://research.microsoft.com/~ gurevich`.

[BGS99] Andreas Blass, Yuri Gurevich, and Saharon Shelah, "Choiceless polynomial time," *Annals of Pure and Applied Logic* 100 (1999), 141–187.

[Br95] Egon Börger, "Why Use Evolving Algebras for Hardware and Software Engineering?", in *Theory and Practice of Informatics*, M. Bartosek, J. Staudek, J. Wiedermann, editors., Springer Verlag, Lecture Notes in Computer Science 1012, 1995, 236–271.

[Br99] Egon Börger, "High level system design and analysis using Abstract State Machines" in *Current Trends in Applied Formal Methods*, D. Hutter, W. Stephan, P. Traverso, M. Ullman, editors., Springer Verlag, Lecture Notes in Computer Science 1641, 1999, 1–43.

[BS00] Egon Börger and Joachim Schmid, "Composition and submachine concepts for sequential ASMs," in *Computer Science Logic*, P. Clote and H. Schwichtenberg, editors., Springer-Verlag, Lecture Notes in Computer Science 1862 (2000), 41–60.

[Bx97] Don Box, "Essential COM", Addison Wesley Longman, 1997.

[FSE] Foundations of Software Engineering, Microsoft Research, web page, `http://research.microsoft.com/fse/`.

[G93] Yuri Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", Bull. European Assoc. for Theoretical Computer Science, no. 43, Feb. 1991, 264–284. [A slightly revised version appeared in *Current Trends in Theoretical Computer Science*, G. Rozenberg and A. Salomaa, editors., World Scientific, 1993, 266–292.]

[G95] Yuri Gurevich, "Evolving algebra 1993: Lipari guide," in *Specification and Validation Methods*, E. Börger, ed., Oxford Univ. Press (1995) 9–36.

[G97] Yuri Gurevich, "May 1997 Draft of the ASM guide," Univ. of Michigan Tech Report CSE-TR-336-97, found at [ASM].

[G00] Yuri Gurevich, "Sequential abstract state machines capture sequential algorithms," *ACM. Trans. Computational Logic* 1 (2000), 77–111.

[GSV01] Yuri Gurevich, Wolfram Schulte, and Margus Veanes, "Toward industrial strength abstract state machines," Tech. Report MSR-TR-2001-98, Microsoft Research.

[S99] Joachim Schmid, "Executing ASM specifications with AsmGofer", `http://www.tydo.de/AsmGofer, 1999`.