Generic Parallel Algorithms

Nachum Dershowitz and Evgenia Falkovich

School of Computer Science, Tel Aviv University, Tel Aviv, Israel
nachum.dershowitz@cs.tau.ac.il, jenny.falkovich@gmail.com

Abstract. We develop a nature-inspired generic programming language for parallel algorithms, one that works for all data structures and control structures. Any parallel algorithm satisfying intuitively-appealing postulates can be modeled by a collection of cells, each of which is an abstract state machine, augmented with the ability to spawn new cells. All cells run the same algorithm and communicate via a shared global memory.

1 Introduction

Evolving systems – physical, biological, or computational – are typically viewable on many distinct levels of abstraction. Let us imagine some closed ecosystem as an example. An *ecologist* views species, populations, and their interactions; population growth and shrinkage may be modeled, say, by predator-prey and other resource equations. A *biologist* takes a different viewpoint, based on the individual organisms; she may develop a kinetic model for swarming behavior, for instance. On a lower level still, a *biochemist* sees interacting cell systems; he might use a diffusion-reaction equation to describe the development of the colorings on an animal's coat. The *chemist* looks at reactions on the molecular level; the *physicist* sees atoms and their constituents. The common denominator of all these views is one of a complex of objects that evolve over time and that interact with each other and with their environment according to a set of rules. It is this generic notion of a *system of interacting objects* that we seek to capture.

It has been convincingly argued by Gurevich [15] (presaged by Post [16]) that logical structures are the right way to view evolving algorithmic states, just as they are ideal for capturing the salient features of static entities. The structure stores the values (taken from the structure's domain) of components of the state that are updated during the computation (variables and program counters) as well as the state's functional capabilities (like arithmetic).

That there are multiple levels at which to understand the same overall system necessitates an abstraction mechanism. Atomic physics is of no relevance to the ecologist; the ecologist's view of the system is the same regardless of quantum physics. This means that the behavior of the entities at the ecological level should be modeled independently of the underlying physical model, which translates into the requirement that states qua structures are isomorphism-closed (making them oblivious as to how the domain values they deal with are in fact implemented) and that their evolution respects those isomorphisms. The importance of isomorphism-invariance for purposes of abstraction has been repeatedly emphasized [7,13,15].

A. Beckmann, E. Csuhaj-Varjú, and K. Meer (Eds.): CiE 2014, LNCS 8493, pp. 133–142, 2014.

On each level of the ecological system there are interacting entities: populations interact on the ecology level, organisms on the biological level, cells on the biochemical level, etc. The interacting entities need not all have "algorithmic" behavior. Aspects of the external environment (such as weather conditions) can also be treated as entities with which algorithmic components trade information. Accordingly, we need a model of communication between entities, which we shall refer to as "cells", in addition to a model of their individual evolution. To that end, we can allow the control of one cell to access values in another cell – a shared-memory viewpoint, or request values from another cell – a message-based framework. Similarly, we can allow one cell to set values in another cell or to request those changes from the other cell (depending again on one's viewpoint). Interaction and coöperation have been considered within Gurevich's framework [2,4]. We take the shared memory viewpoint here and assume that cells work in discrete time with a shared clock.

Many systems, be they natural or artificial, create new entities as they evolve in time. We will, therefore, need to model the "birth" of new component cells. But we will not, in this paper, consider changes in channels of communication (the "topology") other than at birth (cf. [12]). Were it not for possible interaction with external agents and for the birth of new components, one might have been tempted to view a software system as one large evolving global "organism", rather than as a conglomerate of many interacting individual cells.

In the next two sections, we characterize parallel algorithms and their cells. Then, in Sect. 4, we give a description of a parallel programming language based on abstract state machines (ASMs) [14]. Section 5 proves that all parallel algorithms, as characterized here, can be programmed with the constructs of the proposed language. We conclude with a brief discussion.

2 System Evolution

Informally, a *parallel algorithm* consists of a (finite or infinite) set of cells, whose individual *states* all evolve according to the same algorithm. The state of each *cell*, at any moment, is a (logical) structure with a tripartite vocabulary $F \uplus F' \uplus G$ consisting of *private (internal)* operations F, *public (global)* G, and *embryonic* F', the latter having the same similarity type as F. (There could be any fixed number of embryonic copies $F', F'', \ldots, F^{(k)}$, but let us leave it simple for now, one child at a time.) The individual cells all run a "classical" (sequential) algorithm in the sense of [15,8].

Initially, all cells agree on G and their F' are pristine (completely undefined). A single *global step* of the algorithm comprises of the following stages.

1. First, each cell C takes one classical step, producing a set of *updates* U.
2. Cells' private operations F and embryonic operations F' are updated per U.
3. Then the union of all the cell's public updates together are applied to every cell's public G. If there is any disagreement between cells regarding updates to G (the same location getting contradictory new values), the whole system

aborts. (Abortion could be replaced with nondeterministic behavior, should one prefer.)

4. Assuming there are no conflicts, *mitosis* takes place as follows: Each cell C in which the values of the operations F' were modified splits into two, a mother C and daughter C'. The daughter C' inherits G, as updated, from her mother; her F is a copy of her mother's F'. For both mother and daughter, F' is reinitialized to *undefined*.
5. If one wishes, an individual cell can be allowed to *die* and be dropped from the global organism whenever it has no next state, as when it suffers an internal clash.

3 Parallel Algorithms

An algorithm $\mathcal{A}$, in general, is normally viewed as a state-transition system composed of a collection (set, class) $\mathcal{S}$ of states, a (partial) *transition function* $\tau : \mathcal{S} \rightharpoonup \mathcal{S}$ and a (nonempty) subset $\mathcal{S}_0 \subseteq \mathcal{S}$ of *initial states*. We first explain what states of a parallel algorithm look like and then discuss algorithmic transitions.

As explained above, states should be formalized as (first-order) logical structures over some (fixed by the algorithm) vocabulary. On the other hand, we need for systems to comprise multiple local processes, what we called "cells". Each cell has its own unique identity (id), taken from some index set (or class) $\mathcal{I}$. Since we are dealing with parallel algorithms, with both private and shared memory, each cell has a *local* state, which is a structure over a (finite) vocabulary $G \uplus F$, where the (current) values of operations in G are stored (conceptually, at least) in *global* locations, accessible to all cells i, while private data is stored as values of operations in its personal copy of F. The *global* state of the algorithm will be an algebra over the combined (possibly infinite) vocabulary $V = G \cup F^*$, were $F^* = \cup_{i \in \mathcal{I}} F_i$, where each $F_i = \{f_i^1, \ldots, f_i^k\}$ consists of the k local operations of cell i, with f_i^j $(j = 1, \ldots, k)$ of the same arity for all cells. It will be convenient in what follows to denote $F^j = \cup_{i \in \mathcal{I}} \{f_i^j\}$.

With this intuition in mind, we define a global state of *federacy* $\mathcal{I}$, for a given (countable or uncountable) set of *identities* $\mathcal{I}$, to be an algebra X over vocabulary $V = G \cup F^*$. A *global (transition) system* $\mathcal{A}$ *(of federacy* $\mathcal{I}$*)* is composed of a collection $\mathcal{S}$ of global states of federacy $\mathcal{I}$, all over the same vocabulary, a (partial) *transition function* $\tau : \mathcal{S} \rightharpoonup \mathcal{S}$ and a subset $\mathcal{S}_0 \subseteq \mathcal{S}$ of states assumed to be the full collection of *initial states* of $\mathcal{A}$. If τ is undefined for some $X \in \mathcal{S}$ we will say that X is a *terminal state* of $\mathcal{A}$.

Let X be a state of $\mathcal{A}$ with domain $\mathcal{U}$. Let g be some function in V (either in G or in $F^* = F^1 \cup \ldots \cup F^k$) of arity n and $\bar{u} = (u_1, \ldots, u_n)$ be an n-tuple over that domain. If $g(\bar{u}) = w$ in X, we denote this by $g_X(\bar{u}) = w$ or, alternatively, will say that $\langle g, \bar{u}, w \rangle$ is a *location-value* of X. For any ground term t, we write $t_X = w$ to mean that the value of t (as interpreted) in X is w.

We intend that cell i operate over vocabulary $G \cup F_i$ only. So we define the ith *localization* X_i of global state X of federacy $\mathcal{I}$ to be the restriction of X to $G \cup F_i$. The ith cell is expected to manipulate this ith localization only, identifying its

private F with the global F_i. We say that cell (local state) X_i is *empty* if f_i^j is undefined ($\perp$) for all j; on the other hand, we say that X is an *i-cell* if $X = X_i$ and is nonempty. When state X with transition τ is not terminal, we say that $\delta = \langle g, \bar{u}, w\rangle$ is an *update* of X if τ *changes* the value of $g(\bar{u})$ to be w. We define by $\Delta_\tau(X)$ the set of all updates of X.

To compare different cells we should ignore their individual identities. So we define a *depersonalization* operator $^\sharp$; its application wipes out the id, dropping the id-index from function symbols. Thus, the depersonalized $X_i^\sharp$ is obtained from a cell X_i by replacing its f_i^j symbols by f^j, for all j. So we write $X_i = Y_k$ if $X_i^\sharp = Y_k^\sharp$ for states X and Y and localizations X_i and Y_k. Similarly, we say that transition τ generates the same updates for X_i and Y_k if $\Delta_\tau(X_i)^\sharp = \Delta_\tau(Y_k)^\sharp$. In this case, we will use the notation $\Delta_\tau(X_i) = \Delta_\tau(Y_k)$. We denote by $\Delta_\tau^i(X)$ the set of all updates of locations of $f_i^1, \ldots, f_i^k$ in X. And again, we say that transition τ generates the same $\Delta_\tau^i(X) = \Delta_\tau^l(Y)$ if $\Delta_\tau^i(X)^\sharp = \Delta_\tau^l(Y)^\sharp$.

To capture the uniform behavior of cells, we introduce *templates*, which are terms over an unadorned vocabulary $G \cup \{f^1, \ldots, f^k\}$, where f^i is a symbol of the same arity as the $f_i^j \in F$. For each $i \in \mathcal{I}$, the template t induces a term t_i, obtained by replacing each occurrence of f^j by f_i^j. Given states X and Y from the same transition system and given a template t, we say that $X =_T Y$ if $t_{iX} = t_{iY}$ for any $i \in \mathcal{I}$ (i.e. every term defined by t has the same value in both X and Y). To compare different cells we should again ignore their identities. So let X_i and X_m be distinct localizations of global state X. We say that $X_i =_T X_m$ if $t_{iX} = t_{mX}$ for each $t \in T$. Similarly, we may compare localizations of two distinct global states. Letting X_i be a localization of X and Y_m a localization of Y, we write $X_i =_T Y_m$ if $t_{iX} = t_{mY}$ for each $t \in T$.

Let $\mathcal{A} = (\mathcal{S}, \mathcal{S}_0, \tau)$ be a transition system of federacy $\mathcal{I}$ over vocabulary $V = G \cup F^1 \cup \ldots \cup F^k$. We deem a parallel process $\mathcal{A}$ to be *algorithmic* if it satisfies several postulates, which we now proceed to explicate.

Postulate 1 (Genericity). *The set of states (and also the sets of initial states and of terminal states) is closed under isomorphism (of first-order structures). The set of states is also closed under localizations: if X is a state of $\mathcal{A}$ then X_i is also a state of $\mathcal{A}$, for each $i \in \mathcal{I}$. Transitions preserve the domain (universe) of states, and, furthermore, isomorphic states are either both terminal (have no transition) or else their next states are isomorphic (via the same isomorphism).*

States as structures make it possible to consider any data structure sans encodings. In this sense, algorithms are generic. The structures are "first-order" in syntax, though domains may include sequences, or sets, or other higher-order objects, in which case the state would provide operations for dealing with those objects. (States with infinitary operations, like the supremum of infinitely many objects, are precluded.) Closure under isomorphism ensures that the algorithm can operate on the chosen level of abstraction and that states' internal representation of data is invisible to the algorithm. This means that the behavior of an *algorithm*, in contradistinction with its "implementation" as a program in

some particular programming language, cannot depend on the memory address of some variable.

It must be possible to describe the effect of transitions in terms of the information in the current state.

Postulate 2 (Describability). *There exists a finite set T of* critical *templates such that $\Delta_\tau(X) = \Delta_\tau(Y)$ if $X =_T Y$ for any states X and Y of $\mathcal{A}$.*

The critical templates are those locations in the state named by the algorithm (or program). If every referenced location has the same value in two states, then the behavior of the algorithm must be the same for both those states. This, the essence of what makes a process algorithmic, is a crucial insight of [15].

The updates created by an individual cell may not depend on its id, but only on global and local locations that are available to it. Furthermore, each cell is fully responsible for its dates, and no other cell may change them. Also, all updates of a global state are generated by local cells only.

Postulate 3 (Locality). *If $X_i =_T Y_j$ for two states X and Y and localizations $i, j \in \mathcal{I}$, then $\Delta_\tau(X_i) = \Delta_\tau(Y_j)$ and $\Delta^i_\tau(X) = \Delta^j_\tau(Y)$.*

Postulate 4 (Globality). *$\Delta_\tau(X) = \cup_{i\in\mathcal{I}}\Delta_\tau(X_i)$ for all states X.*

If some localization of X is empty but is not empty for $\tau(X)$, this indicates that a child has been born.

Postulate 5 (Fertility). *There exists a (input-independent) bound $n \in \mathbb{N}$ such that $\tau(X)$ has at most n non-empty localizations for any local i-cell X, $i \in \mathcal{I}$.*

The idea is that in one step a cell may participate in the creation of only a bounded number of new cells. And each newborn cell has exactly one mother:

Postulate 6 (Motherhood). *For every state X, if a localization X_i is empty, but is non-empty for $\tau(X)$, then there is a $j \in \mathcal{I}$ such that $\Delta^i_\tau(X) \subseteq \Delta_\tau(X_j)$.*

With the above requirements in place, we state what a parallel algorithm is.

Definition 1. *A global system is* algorithmic *if it satisfies Postulates 1–6.*

We say that two parallel algorithmic systems are "congruent" if they are identical, up to permutation of identities $\mathcal{I}$.

Definition 2. *A parallel algorithm* **A** *is a family of all parallel algorithmic systems,* congruent *with some algorithmic system $\mathcal{A}$.*

Proposition 1. *Let $\mathcal{A}$ be an algorithmic system over a* finite *vocabulary. Then $\mathcal{A}$ may be described as an ordinary algorithm.*

Proof. Imagine $\mathcal{A}$ is an algorithmic system over a *finite* vocabulary. Then instead of $V = G \cup F^*$, we may assume that we only have $V = G$ (and we required that G be finite). So for this case, Postulates 3–6 are redundant, and $\mathcal{A}$ is only required to satisfy the **geniricity** and **describability** postulates. Also, our final set of critical templates T is just a finite set of terms over $V = G$. Then $\mathcal{A}$ is a *classical (sequential) algorithm* with *critical* terms T, as defined in [15]. □

4 Parallel Programs

The two basic program statements are assignment and creation. These may be composed in parallel and guarded by conditions.

Assignment. An *atomic assignment* is a rule of the form $h(t^1, \ldots, t^n) := t^0$, where $t^0, \ldots, t^n$ are templates and $h \in G \cup F$.

Let X_i be a localization of X, and suppose that $t^j_{iX} = u^j_i$ for $j = 0, \ldots, n$. If $h \in G$, then application of the assignment on X for i generates a global update $\Delta_a(X_i) = \{\langle h, (u^1_i, \ldots, u^n_i), u^0_i\rangle\}$. If $h \in F$, then the application generates dates $\Delta_a(X_i) = \{\langle h_i, (u^1_i, \ldots, u^n_i), u^0_i\rangle\}$. If any of the t^j is undefined in X_i ($t^j_{X_i} = \perp$) , then $\Delta_a(X_i) = \varnothing$. The application of assignment a to global state X generates the update set $\Delta_a(X) = \cup_{i\in\mathcal{I}}\Delta_a(X_i)$.

Parallel assignment. More generally, a *parallel assignment* rule is a set $\{a_1, a_2, \ldots, a_n\}$ of atomic assignments, written with $\|$ between the atomic a_j.

The update set generated by such parallel assignment a is $\Delta_a(X) = \cup^n_{j=1}\Delta_{a_j}(X)$. If $\Delta_a(X)$ has conflicting updates (different values assigned to the same location), then the rule fails.

Creation. The creation rule $\nu.a$ takes the form **new** a, where a is a parallel assignment, atomic assignments in a are of the form $f^j(t^1, \ldots, t^n) := t^0$, and n is the arity of symbol $f^j \in F$.

Let X_i be a localization, and suppose $t^j_{iX} = u^j_i$ for $j = 0, \ldots, n$ for an atomic assignment. The transition initializes some empty localization X_{k_i} with location-value $\langle f^j_{k_i}, (u^1_i, \ldots, u^n_i), u^0_i\rangle$. So $\Delta_{\nu.a}(X_i) = \{\langle f^j_{k_i}, (u^1_i, \ldots, u^n_i), u^0_i\rangle\}$. If any one of the t^j is undefined in X_i, then $\Delta_{\nu.a}(X_i) = \varnothing$. For each cell i, the transition chooses a unique k_i and the cell's updates are appended to the total set of updates $\Delta_{\nu.a}(X) = \cup_{i\in\mathcal{I}}\Delta_{\nu.a}(X_i)$. If a is a parallel assignment $a_1\|a_2\|\cdots\|a_n$, then application of $\nu.a$ chooses a unique empty X_{k_i} for each X_i in which all arguments t^j are defined, and $\Delta_{\nu.a}(X_i) = \cup^n_{\ell=1}\Delta_{\nu.a_\ell}(X)$. If there is no way to choose k_i for all i so that the rule applies, then it is not applied at all.

Guard. An *atomic guard* is a condition of the form $s = t$ or $s \neq t$. Guard $t = s$ evaluates to T (true) for localization X_i if $t_{iX} = s_{iX}$. Similarly, a guard $t \neq s$ is T if $t_{iX} \neq s_{iX}$. More generally, a *guard* g may be a conjunction of atomic guards $g_1\&g_2\&\cdots\&g_n$, which is T for X_i if each g_j is.

Guarded assignment. This is a rule $g : a$ of form **if** g **then** a, where g is a guard and a is a parallel assignment. Application of $g : a$ to X generates the set of updates $\Delta_{g:a}(X) = \cup\{\Delta_a(X_i) : i \in \mathcal{I} \text{ s.t. } g_{X_i} = \text{T}\}$.

Guarded creation. This is a rule $g : \nu.a$ of form **if** g **then new** a. The rule $\nu.a$ is executed on each X_i for which g evaluates to T.

Definition 3 (Program). *A* (parallel) program *is a finite set P of rules r_i as above, written $r_1\|\cdots\|r_n$. To execute P on state X, all rules are executed in parallel (simultaneously), that is, $\Delta_P(X) = \cup_{r_i\in P}\Delta_{r_i}(X)$. If $\Delta_P(X)$ has conflicting updates, then no updates are applied at all.*

Note that for each application of creation, the program chooses in some fashion new unused indices from $\mathcal{I}$. So for each given initial state, the program may have

multiple runs, depending on the choices made. Each choice is possible and none is preferred. And it does not affect the computation's final result or running time (number of steps). So for each state X, we denote by $P(X)$ any one of the possible (congruent) states obtained by application of P to X.

5 Representation Theorem

A parallel program P is a *characteristic program* of algorithmic system $\mathcal{A}$ if $P(X) = \tau(X)$ for each state X of $\mathcal{A}$. A parallel program P is a *characteristic program* of parallel algorithm $\mathbb{A}$ if it is a characteristic program for each algorithmic system $\mathcal{A}$ in $\mathbb{A}$. We shall presume for simplicity that $\mathcal{A}$ is over a vocabulary $G \cup F^1$ only and denote it by $G \cup F$. We will also assume that in Postulate 5 we have at most one child born per step ($n = 2$). All proofs can be easily extended to the general case.

By **globality**, $\Delta_\tau(X) = \cup_i \Delta_\tau(X_i)$. So we start with i-cells. We first prove that the transitions of any i-cell can be described by a rule composed of assignment and creation rules.

Let X be an i-cell of system $\mathcal{A}$. According to the above simplifying assumption, $\mathcal{A}$ has only one local function. Since X is an i-cell, its non-default locations are over $G \cup \{f_i\}$. Furthermore, any cell may have at most one child in one transition. Hence, non-default locations of $\tau(X)$ are over $G \cup \{f_i, f_j\}$ for some $j \in \mathcal{I}$. So we may consider X and $\tau(X)$ as ordinary states of an ordinary algorithm over finite vocabulary $G \cup \{f_i, f_j\}$ with critical terms $T_i \cup T_j$.

Let $\delta = \langle h, (u_1, \ldots, u_n), w \rangle$ be an update in $\Delta_\tau(X)$. According to [17, Lemma 5] for each $k = 0, \ldots, n$ there exists $t^k \in T_i \cup T_j$ such that $t^k_X = u_k$. Let X_δ be an ordinary assignment rule $h(t^1, \ldots, t^n) := t^0$. Then $\Delta_{X_\delta}(X) = \delta$. We will call X_δ an *ordinary characteristic assignment* of δ. Denote by $\mathbf{X}$ the ordinary assignment obtained by parallel composition of X_δ for all $\delta \in \Delta_\tau(X)$. Obviously, $\Delta_{\mathbf{X}}(X) = \Delta_\tau(X)$. Take a look at $X_\delta = h(t^1, \ldots, t^n) := t^0$. As we said before, $t^k_X = u_k$ for all $k = 0, \ldots, n$. In particular, t^k is defined over X. Since all non-trivial locations of X are over $G \cup \{f_i\}$, we may conclude that t^k are over T_i for all $k = 0, \ldots, n$. And since all defined locations of $\tau(X)$ are over $G \cup \{f_i, f_j\}$, we may conclude that $h \in G \cup \{f_i, f_j\}$. We partition $\mathbf{X}$ into two parallel assignment rules: a_X for all those rules with $h \in G \cup \{f_i\}$ and n_X for all the rest, rules of the form $f_j(t^1, \ldots, t^n) := t^0$. Obviously, $\mathbf{X} = a_X \| n_X$.

Let $a^\sharp_X$ be obtained from a_X by replacing f_i with f. Then $a^\sharp_X$ is an assignment rule over templates T. From the definition of parallel-assignment application we obtain that $\Delta_{a^\sharp_X}(X) = \Delta_{a_X}(X)$. Let $n^\sharp_X$ be obtained from n_X by replacing f_i and f_j with f. From the definitions of parallel ν-rule application and of comparing updates for different cells, we obtain that $\Delta_{\nu.n^\sharp_X}(X) = \Delta_{n_x}(X)$. Define $\mathbb{X} = a^\sharp_X \parallel \nu.n^\sharp_X$. Then $\Delta_{\mathbb{X}}(X) = \Delta_{a^\sharp_X}(X) \cup \Delta_{\nu.n^\sharp_X}(X) = \Delta_{\mathbf{X}}(X)$.

Proposition 2. *Let X be an i-cell of $\mathcal{A}$. Then $\mathbb{X}(X) = \mathbf{X}(X) = \tau(X)$.*

Proof. That $\mathbb{X}(X)$ is $\tau(X)$ follows from the above. That $\mathbf{X}(X)$ is $\tau(X)$ follows from [17, Lemma 11]. □

Updates of i-cells depend on the values of critical terms only.

Proposition 3. *Let X and Y be i-cells of $\mathcal{A}$, such that $X =_T Y$. Then $\mathbf{Y}(X) = \tau(X)$. And $\mathbb{Y}(X) = \tau(X)$.*

Proof. Since $\mathbb{Y}$ is a rule over T it generates updates based on the values of T only. Hence $\Delta_{\mathbb{Y}}(X) = \Delta_{\mathbb{Y}}(Y)$, since $X =_T Y$. It follows from the earlier discussion that $\mathbb{Y}(Y) = \tau(Y)$. According to **locality**, we have that $\Delta_\tau(Y) = \Delta_\tau(X)$, again since $X =_T Y$. Combining all, we conclude $\mathbb{Y}(X) = \tau(X)$. Hence, we get the following implication for i-cell X and j-cell Y: $X =_T Y \Rightarrow \mathbb{Y}(X) = \tau(X)$. □

Let X be an i-cell. We define an equivalence relation $\sim_X$ on T by $t \sim_X s$ iff $t_X = s_X$. We next show that updates of i-cell X depend on $\sim_X$ only. Let X be an i-cell and Y be a j-cell of $\mathcal{A}$. We write $X \approx_T Y$ if $\sim_X = \sim_Y$.

Proposition 4. *Let X be an i-cell of $\mathcal{A}$ and let Y be a j-cell of $\mathcal{A}$ such that $X \approx_T Y$ and $\mathbf{Y}(X) = \tau(X)$. Then $\mathbb{Y}(X) = \tau(X)$.*

Proof. First assume that $i = j$, i.e. that both X and Y are i-cells. Consider X and Y to be ordinary states over finite vocabulary $G \cup \{f_i\}$ (as we did at the start of this section). By assumption, X and Y each have one child cell in a single transition. Define $\sim^i_X$ on T_i by $t \sim_X s$ iff $t_X = s_X$ for any $t, s \in T_i$. Then $\sim^i_X = \sim^i_Y$. It follows from [17, Lemma 13] that $\mathbf{Y}(X) = \tau(X)$. And by Proposition 2 we conclude that $\mathbb{Y}(X) = \tau(X)$. Recall that we defined $X = Y$ for parallel states X and Y of the same system if they are equal up to a permutation of identities of their cells. The general case follows immediately. □

Lemma 1. *For each parallel algorithmic transition system there exists a characteristic parallel program.*

Proof. Let $\sim$ be some binary relation on T. Then for any pair of distinct term-templates $s, t \in T$ we have that either $s \sim t$ or $s \not\sim t$. For each $s, t \in T$ we define $\beta_\sim(s, t)$ to be an atomic guard $s = t$ if $s \sim t$ and $s \neq t$ otherwise. Define a guard $\beta_\sim$ to be a conjunction of all atomic guards $\beta_\sim(s, t)$ for all $s, t \in T$. Choose an i-cell X of $\mathcal{A}$ for some $i \in \mathcal{I}$ such that $\sim_X$ is $\sim$. Denote it by $X_\sim$ Define a rule $R_\sim =$ **if** $\beta_\sim$ **then** $\mathbb{X}_\sim$. Obviously $\beta_\sim$ evaluates to T on X. and hence $R_\sim(X_\sim) = \mathbb{X}_\sim$. According to Proposition 4, we have that $\mathbb{X}_\sim = \tau(X_\sim)$ and so $R_\sim(X_\sim) = \tau(X_\sim)$.

Define P to be a parallel program consisting of rules $R_\sim$ for all binary relations $\sim$ of T. Note that since T is finite, it has only finitely many distinct binary relations and so program P is finite. We claim that P is a characteristic program of $\mathcal{A}$, that is, $P(X) = \tau(X)$ for any state X of $\mathcal{A}$. Assume first that $X = X_\sim$ for some binary relation $\sim$ on T. Let $\sim'$ be another binary relation on T, distinct from $\sim$. Then for some $s, t \in T$ we have that $\beta_\sim(s, t) \neq \beta_{\sim'}(s, t)$. So $\beta_{\sim'}(s, t)$ is false for X and so is $\beta_{\sim'}$. Then $\Delta_{R_{\sim'}}(X) = \varnothing$ and that is for any binary relation on T other than $\sim$. Hence $P(X) = R_\sim(X)$ and according to the previous discussion $R_\sim(X) = \tau(X)$, as desired. Assume next that X is an i-cell for some i. Denote $\sim_X$ by $\sim$. As in the previous item, $\Delta_{R_{\sim'}}(X) = \varnothing$ for any binary

relation $\sim'$ on T, other then $\sim$. And so $P(X) = R_\sim(X)$. Let $X_\sim$ be as above. Then $R_\sim(X) = \mathbb{X}_\sim(X)$ (by the definition of $R_\sim$). According to Proposition 4 we have that $\mathbb{X}_\sim(X) = \tau(X)$. Combining everything together we conclude that in this case again we have that $P(X) = \tau(X)$. Assume finally that X is a general state of $\mathcal{A}$. According to **globality**, the update of X is a union of updates of all its localizations X_i, i.e. $\Delta_\tau(X) = \cup_{i\in\mathcal{I}}\Delta_\tau(X_i)$. By the **genericity** axiom, X_i is a state in $\mathcal{A}$. According to **locality**, updates for X_i do not depend on whether X_i is considered as a standalone state or a localization of a general state. So it is enough to show that $\Delta_P(X_i) = \Delta_\tau(X_i)$ for all $i \in \mathcal{I}$. And that follows from the previous paragraph. □

Theorem 1 (Main). *For each parallel algorithm, there exists a characteristic parallel program.*

Proof. Let $\mathcal{A}$ be an algorithmic system in $\mathbb{A}$. By Lemma 1, there exists a characteristic parallel program $P_\mathcal{A}$ of $\mathcal{A}$. If $\mathcal{B}$ is another algorithmic system in $\mathbb{A}$, then B is identical to $\mathbb{A}$, up to permutation of indices in $\mathcal{I}$. Then, obviously, $P_\mathcal{A}$ is a characteristic program of B as well. □

6 Discussion

The starting point for this research was the desire to characterize parallel computation in as generic a form as possible, with an eye especially towards the effective special case. Blass and Gurevich [1,3] successfully characterized parallel algorithms within the abstract-state-machine framework, but their approach is not easily restricted to the effective case. In their setup, an unbounded number of children may be created by a single cell in a single step.

Our model is simpler than Blass and Gurevich for the cases we consider. As we do not have message passing, algorithms need not deal with process ids. Though we bound the number of new cells created by a cell in a step, an infinite number of initial cells for a non-effective parallel algorithm poses no problem. For example, one can imagine a cell for each of uncountably many points on a line segment in 3D space and an algorithm that applies, in parallel, an affine transformation to the coordinates of each point, resulting in a translated segment.

We've considered discrete-time systems, where all cells progress in lockstep with each other, as in [1,3]. We plan to expand this work in several directions:

- Characterize what makes a parallel algorithm effective. Analogous to prior work on classical effectiveness [11,5], we need to demand that the initial global state be finitely describable. This decomposes into two main requirements: (i) each cell itself be an effective classical algorithm; (ii) there be only finitely many cells initially, though their number may depend on the input.
- Prove the extended Church-Turing thesis for parallel algorithms: all effective parallel models of computation can be polynomially simulated by a standard model (like PRAM), as has been done for classical algorithms [10].

– Distributed systems, where cells each progress at their own rate, require separate treatment. This will require a sense of identity for cells and a means of communication between them. Cf. [4].
– Systems that evolve in continuous time are a subject of ongoing research [6,9].

References

1. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Trans. on Computation Logic 4, 578–651 (2003)
2. Blass, A., Gurevich, Y.: Ordinary interactive small-step algorithms (Parts I–III). ACM Trans. on Computational Logic 7, 363–419; 8: art. 15–16 (2006-2007)
3. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms: Correction and extension. ACM Trans. on Computation Logic 9, Art. 19 (2008)
4. Blass, A., Gurevich, Y., Rosenzweig, D., Rossman, B.: Interactive small-step algorithms (Parts I–II). LMCS 3: ppr. 3; 4: ppr. 43 (2007)
5. Boker, U., Dershowitz, N.: Three paths to effectiveness. In: Blass, A., Dershowitz, N., Reisig, W. (eds.) Fields of Logic and Computation. LNCS, vol. 6300, pp. 135–146. Springer, Heidelberg (2010)
6. Bournez, O., Dershowitz, N., Falkovich, E.: Towards an axiomatization of simple analog algorithms. In: Agrawal, M., Cooper, S.B., Li, A. (eds.) TAMC 2012. LNCS, vol. 7287, pp. 525–536. Springer, Heidelberg (2012)
7. Chandra, A.K., Harel, D.: Computable queries for relational data bases. Journal of Computer and System Sciences 21, 156–178 (1980)
8. Dershowitz, N.: The generic model of computation. In: Proc. DCM, pp. 59–71 (2012)
9. Dershowitz, N.: Res Publica: The universal model of computation. In: Computer Science Logic 2013, Turin, Italy. Leibniz International Proceedings in Informatics, vol. 23, pp. 5–10 (2013)
10. Dershowitz, N., Falkovich, E.: A formalization and proof of the Extended Church-Turing Thesis. In: Proc. 7th International Workshop on Developments in Computational Models. EPTCS, vol. 88, pp. 72–78 (2011)
11. Dershowitz, N., Gurevich, Y.: A natural axiomatization of computability and proof of Church's Thesis. Bulletin of Symbolic Logic 14, 299–350 (2008)
12. Dowek, G.: Around the physical Church-Turing thesis: Cellular automata, formal languages, and the principles of quantum theory. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 21–37. Springer, Heidelberg (2012)
13. Gandy, R.: Church's thesis and principles for mechanisms. In: The Kleene Symposium, vol. 101, pp. 123–148. North-Holland (1980)
14. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Börger, E. (ed.) Specification and Validation Methods, pp. 9–36. Oxford University Press, Oxford (1995)
15. Gurevich, Y.: Sequential abstract state machines capture sequential algorithms. ACM Trans. on Computational Logic 1, 77–111 (2000)
16. Post, E.L.: Absolutely unsolvable problems and relatively undecidable propositions. In: Davis, M. (ed.) Solvability, Provability, Definability: The Collected Works of Emil L. Post, pp. 375–441. Birkhaüser, Boston (1994)
17. Reisig, W.: On Gurevich's theorem on sequential algorithms. Acta Informatica 39, 273–305 (2003)