Asynchronous Parallel Self-Replication Based on Logic Molecular Model

Katsuhiko Nakamura

School of Science and Engineering, Tokyo Denki University
Saitama-ken,350-0394 Japan.
nakamura@rd.dendai.ac.jp

Abstract

This paper discusses asynchronous parallel universal computation and self-replication based on a computation model, called a logic molecular model, or a parallel production system (PPS). The program in this model consists of extended Horn clause rules, which are used for forward deduction of unit clauses, called molecules, from unit clauses in working memory. All possible deductions in the system are asynchronously executed in parallel. This formalism is also effective in representing a broad class of speed-independent asynchronous computation and systems including parallel parsing and cellular automata. It is shown that for any PPS program P, there is a set of molecules that contains the coded program of P, which replicates itself by asynchronous parallel computation in time proportional to $\log n$, where n is the number of rules in P.

Introduction

The self-replication of complex systems is universal in biology, as cell division and propagation are essential to living organisms. Many biologists believe that the appearance of self-replicating molecules marked the origin of life. Several hypothetical models of the first self-replication have been presented and discussed in evolutionary biology (Dawkins, 2004). In information science, there have been several theoretical models of self-replication intended to clarify the principles and conditions of self-replication (Hutton, 2003; Sipper, 1998). Some of these models can be applied to artificial self-organization in complex systems including amorphous computing (Abelson et al., 2007) and molecular computing.

Von Neumann adopted a cellular automaton (CA) model of self-replication and presented a two-dimensional (2-D) 29-state CA with universal computation power and self-replicating processes in his last note titled "*Theory of Self-Reproducing Automata*" (von Neumann, 1966). A CA is essentially a parallel system used as a model of parallel computation. Transitions in von Neumann's CA, however, are serial and sequential because the universal computation and self-replication are based on a universal Turing machine. After von Neumann, lot of work focused on the parallel computation power of CAs and self-replication on CAs (Sipper, 1998). Nevertheless, there has been little work on parallel universal computation and parallel self-replication, not only using CAs but also with other computation models. Albert and Culik (1987) showed a 1-D CA with parallel universal computation power in the sense that the CA can simulate any 1-D CA in linear time. Nakamura (1997) showed a 1-D CA with parallel self-replication processes and parallel universal computation power in a similar sense. Nehaniv (2002) showed an asynchronous cellular automaton with a self-reproducing pattern known as "Langton's loop."

This paper proposes a parallel computation model, called a *logic molecular model*, or a *parallel production system* (PPS), and shows that this simple formalism is effective in modeling a broad class of asynchronous parallel computations and biological systems. This model is intended to be a simple and general basis not only for parallel universal computation such as universal Turing machines for serial computation but also for the modeling of self-replication in biological systems.

The logic molecular model proceeds as follows.

1. Every global state of the system is represented by a multiset of *molecules*, which are data tokens in working memory from the point of parallel computation.

2. A program in PPS is a set of production rules, or simply rules. The rules specify the interactions of the molecules by forward, data-driven deduction. Deduction by the rules is a kind of hyper-resolution (Robinson, 1992); each rule is described as an extended Horn clause rule and every molecule in the system as a unit clause.

3. All applicable deductions are asynchronously executed in parallel. Therefore, the computation needs to be speed-independent to reach a definite result in spite of the indefinite orders of the transitions of the elements.

Since the pioneering work of von Neumann, CAs have been used for modeling not only biological systems but also other complex systems. However, modeling using CAs has the following limitation.

- In a CA model, the arrangements of the cells and the interconnections among cells are strictly regular and fixed. This restriction prevents us not only from using CAs to model general parallel systems but also from applying CAs to parallel computers.

- Most standard CA models are synchronous systems. Synchronization generally simplifies the construction of deterministic systems. Nevertheless, it is a fundamentally accepted principle that asynchronous systems are generally faster than synchronous systems in large scale parallel systems, because the synchronization period is determined by the maximum delay in the system. As there are no specific synchronous biological systems, asynchronous systems are more appropriate for modeling self-replication.

There has been some researches into extensions of CAs. The Lindenmayer system (or L-system) (Lindenmayer, 1968) is an extended CA where every cell can propagate itself; the L-system is intended to model biological development. Nakamura (1981) showed that any synchronous d-D CA ($d = 1, 2, \cdots$) can be transformed into an asynchronous d-D CA while preserving its parallel computation power.

Recently, there have been several models other than CAs called biologically-motivated systems or natural computing. The chemical abstract machine (CHAM) (Berry and Boudol, 1992) and the GAMMA language (Baâtre and Métayer, 1993) based on *multiset transformation* have some properties similar to our model. In these formalisms as well as in the logic molecular model, every global state of the system is a multiset of data elements. In CHAM, the global state can be considered a solution of molecules that interact with each other. CHAM and GAMMA, in which no data element is deleted from the global states, are intended to provide a simple paradigm of parallel computation. They are not intended to describe speed-independent asynchronous parallel processes as does the logic molecular model.

The logic molecular model integrates several paradigms including logic programming, production systems, and functional data-flow programming. Some explanation of the relations between these paradigms is essential. Hyper-resolution (Robinson, 1992) is closely related to unit resolution (Chang, 1970) and, has been studied for bottom-up computation with large data sets including deductive databases. The current work is intended to use deduction in logic programming to represent a broad class of asynchronous parallel computation.

In contrast to logic programming, most production systems, such as OPS-5 (Cooper and Wogrin, 1988), mainly employ forward deduction. Although the purpose and the control mechanisms are essentially different, our computation model has some similarities to production systems: the unit clauses in the global state correspond to data tokens in the working memory, unification to pattern matching and the extended Horn clause rules to production rules for forward deduction.

The control of our computation model is closely related to that used in data-flow programs (Dennis, 1975), as the operations are evoked by data tokens. Our PPS programs are more general and more powerful than the data-flow programs because each rule represents a general pattern of symbolic operations based on unification and unit resolution.

This paper is organized as follows. The next section describes the basic model and its asynchronous transition. The rules and their application to data are defined by using the notions in logic programming. The transitions of the global states are based on asynchronous circuit theory. The third section describes the decomposition of general rules into simpler rules, and extensions of the basic rules so that we can use the models for parallel functional processes. The fourth section shows a PPS that simulates a 1-D bounded synchronous CA. This result is closely related to the synchronous-to-asynchronous transformation of CAs. The fifth section describes a universal computation by the PPS. Based on this universal computation, the sixth section shows several parallel self-replicating molecules and self-replicating programs. The final section gives brief concluding remarks.

The Basic Model and Parallel Derivation

We use basic notions of logic programming such as unification and most general unifier to describe the pattern matching and application of the rules.

Parallel Production Systems

We use the notations and syntax of standard Prolog for variables, terms, lists and operators. A *constant* is either a number or an identifier (an atom in Prolog) that starts with a lower-case character, and a variable starts with an upper-case character and the underscore "_". A *term* is either a constant, a variable, or a complex term of the form $f(t_1, \cdots, t_k)$, where f is an identifier (a function or predicate symbol), and each t_i is a term. An *atom* is a term of the form either $p(t_1, \cdots, t_k)$, or p when $k = 0$, where p is a predicate symbol, and each t_i is a term.

A *substitution* θ is a mapping from a set of variables to a set of terms. For any term t, an *instance* $t\theta$ is a term in which each variable X defined in θ is replaced by its value $\theta(X)$. For any terms s and t, we say that s and t are *variants* of each other, if t is an instance of s and t is an instance of s. A *unifier* for two terms s and t is a substitution θ, such that $s\theta = t\theta$. The unifier θ is the *most general unifier* (mgu), if for every other unifier σ of s and t, $s\sigma$ and $t\sigma$ are instances of $s\theta$ and $t\theta$, respectively.

A *parallel production system* (PPS) is defined by its program and its initial global states. The program is a set of *rules* of the form

$$B_1, \cdots, B_m \rightarrow C_1, \cdots, C_n, \quad m, n \geq 0,\ m + n \geq 1.$$

where each of B_i and C_j is either an atom or a variable. The variable in a rule is instantiated to an atom, when the rule is applied as will be described later. The global state, or the working memory, of the PPS is the multiset of unit clauses (or atoms) called *molecules*. The initial global set generally contains the input information.

A rule $R = (B_1, \cdots, B_m \to C_1, \cdots, C_n)$ is *applicable* to molecules $A_1, \cdots, A_m$ in a global state W, if and only if there is a most general unifier θ such that: $A_i\theta = B_i\theta$ for all $1 \le i \le m$. In this case, we write $W \Rightarrow W'$ for the *result* W' of the application defined by

$$W' = (W - \{B'_1\theta, \cdots, B'_m\theta\}) \cup \{C'_1\theta, \cdots, C'_n\theta\}.$$

The relation $\Rightarrow^*$ denotes the reflective and transitive closure of "$\Rightarrow$". For any initial global state W_0, every global state W with $W_0 \Rightarrow^* W$ is called a *derivable* global state of S.

The application of rule R to molecules $A_1, \cdots, A_m$ is equivalent to the simultaneous hyper-resolution of n Horn clause rules,

$$C_1 \leftarrow B_1, \cdots, B_m, \quad \cdots \quad C_n \leftarrow B_1, \cdots, B_m,$$

and the m unit clauses $A_1, \cdots, A_m$, except that these unit clauses are deleted from the global state. Hence, each resultant unit clause is a logical consequence of the unit clauses in the global state and the Horn clauses.

Asynchronous Transition and Speed-Independence

Asynchronous systems generally must be speed-independent to achieve definite computation results in spite of the indefinite order of operations. We represent asynchronous transition in PPSes by applying the terminology of asynchronous circuit theory (Muller and Burtky, 1959) as in defining asynchronous cellular automata (Nakamura, 1981). As several different terms are used for similar notions in term rewriting system (TRS) theories, we have added some comments on these terms in parentheses.

An *allowed sequence* in a PPS is a finite or infinite sequence $W_0, W_1, W_2, \cdots$ of the global states such that $W_i \Rightarrow W_{i+1}$ for $i = 0, 1, 2, \cdots$ and there is no subscript $i_0 \ge 0$ such that a rule is applicable to a subset of molecules in W_i for all $i \ge i_0$. (This notion corresponds to *fair computation* in TRS.) This condition states that all the delays in the application of the rules are arbitrary but finite.

The class G of global states in a PPS is partitioned into subclasses by the equivalence relation $W \Rightarrow^* W'$ and $W' \Rightarrow^* W$ for any $W, W' \in G$. The equivalence class ("strongly connected components" in TRS) is partially ordered by the relation $\Rightarrow^*$. A PPS S is *speed-independent*, if and only if for all allowed sequences $W_0, W_1, W_2, \cdots$ starting with an initial global state W_0, there is an integer j_0 such that all global states $W_j, j \ge j_0$ are in a common equivalence class. In a speed-independent system, if there is a finite allowed sequence $W_0, \cdots, W_t$, then all the allowed sequences starting with W_0 terminate with W_t, which we call the *terminal state*.

A PPS S is *race-free*, if and only if for any derivable global states W and W' such that a rule R is applicable to some molecules in W and $W \Rightarrow W'$, either W' has the result of the application of R, or R is still applicable to the same molecules in W'.

A PPS S has the *Church-Rosser* (*diamond*) property, if and only if for any derivable global states W, X and Y with $W \Rightarrow X$ and $W \Rightarrow Y$, there is a global state Z such that:

$$\begin{tikzcd} W \arrow[r, Rightarrow] \arrow[d, Rightarrow] & X \arrow[d, Rightarrow] \\ Y \arrow[r, Rightarrow] & Z. \end{tikzcd}$$

Proposition 1 *Any race-free PPS is Church-Rosser, and any Church-Rosser PPS is speed-independent. The converses of these relations do not hold.*

Proof It is obvious from the definitions that any race-free PPS is Church-Rosser. We omit the proof that any Church-Rosser PPS is speed-independent because it is similar to the corresponding propositions in asynchronous circuit theory (Muller and Burtky, 1959) and in the theory of TRS.

To prove that the converse does not hold, consider the PPS with program, $p, q \to s$; $s, r \to u$; $q, r \to t$; $p, t \to u$, and initial global state $\{p, q, r\}$. This system is Church-Rosser but not race-free. Consider another PPS with program, $p, q \to s$; $s, r \to u$; $p, q, r \to u$, and initial global state $\{p, q, r\}$. This system is speed-independent, but not Church-Rosser. □

Synchronous Transition

A *synchronous transition sequence* of a PPS is a subsequence $W_0, W_1, W_2, \cdots$ of an allowed sequence such that all applicable rules in W_i, and no other rule, have applied in W_{i+1} for $i = 0, 1, 2, \cdots$. In any race-free PPS, there is a unique synchronous transition sequence for any initial global state. The length of the synchronous transition sequence represents the number of steps, or time, of asynchronous computation where all applications of the rules require a constant time.

Example: Parallel Parsing of a CFL

The first example is parallel bottom-up parsing of the parenthesis languages, i.e, the set of strings with the same number of a's and b's such that no prefix contains more b's than a's. Each rule in the following program represents a production rule for a context free grammar, as in definite clause grammars (DCGs) (Imada and Nakamura, 2010).

[Parsing parenthesis language]

```
a(I,J), b(J,K)→ s(I,K,s(a,b)).
s(I,J,P),s(J,K,Q)→ s(I,K,s(P,Q)).
a(I,J),s(J,K,P),b(K,L)→ s(I,L,s(a,P,b)).
```

Suppose that the initial global state contains the following molecules representing the string *aababb*.

```
a(0,1).a(1,2).b(2,3).a(3,4).b(4,5).b(5,6).
```

The computation proceeds as follows and terminates with a molecule that has a term representing the derivation tree.

```
  a(0,1),a(1,2),b(2,3),a(3,4),b(4,5),b(5,6)
⇒ a(0,1),s(1,3,s(a,b)),a(3,4),b(4,5),b(5,6)
⇒ a(0,1),s(1,3,s(a,b)),s(3,5,s(a,b)),b(5,6)
⇒ a(0,1),s(1,5,s(s(a,b),s(a,b))),b(5,6)
⇒ s(0,6,s(a,s(1,5,s(s(a,b),s(a,b))),b))
```

This computation is speed-independent and terminates with a single molecule having the definite derivation tree for any initial state representing a string in the language. As the grammar is ambiguous, the computation with other initial global states, for example, those for parsing a string *ababab*, cannot be speed-independent. Nevertheless, parsing terminates with a final molecule containing one of the possible derivation trees.

Extensions of the Basic Model

This section describes transformations and extensions of the rules in the basic model. Transformed PPSes simulate the original PPSes in the following sense. A PPS S' *simulates* a PPS S, if and only if there is a computable function $c : U' \rightarrow U$, where each of U and U' is the class of global states of S and S', respectively, such that if for any allowed sequence $W_0, W_1, W_2, \cdots$ in S', the sequence $c(W_0), c(W_1), c(W_2), \cdots$ is an allowed sequence in S, provided that we ignore any repetitions.

Decomposition of Rules

Any rule $B_1, \cdots, B_m$ -> $C_1, \cdots, C_n$ with $m > 2$ and/or $n > 2$ can be decomposed into simpler rules with at most two atoms on each of the left and right hand sides. First we recursively transform a rule $B_1, \cdots, B_m \rightarrow C_1, \cdots, C_n$ with $m > 2$ into the following three rules.

$$B_1, \cdots, B_{m/2} \rightarrow r_1(X_1, \cdots, X_k),$$
$$B_{m/2+1}, \cdots, B_m \rightarrow r_2(X_1, \cdots, X_k),$$
$$r_1(X_1, \cdots, X_k), r_2(X_1, \cdots, X_k) \rightarrow C_1, \cdots, C_n$$

where r_1 and r_2 are unique predicate names and $X_1, \cdots, X_k$ is a list of all the variables in the rule. Secondly, we recursively decompose the rule of the form $B_1, B_2 \rightarrow C_1, \cdots, C_n$ with $n > 2$ into the three rules with unique predicate names q_1 and q_2:

$$B_1, B_2 \rightarrow q_1(X_1, \cdots, X_k), q_2(X_1, \cdots, X_k),$$
$$q_1(X_1, \cdots, X_k) \rightarrow C_1, \cdots, C_{n/2},$$
$$q_2(X_1, \cdots, X_k) \rightarrow C_{n/2+1}, \cdots, C_n.$$

Non-Deleting Molecules

We can extend the basic rule so that any molecule matching with an atom on the left side of the rule remains undeleted from a global state. Any molecule unifying the atom with the prefix operator `*`, as in

$$B_1, \cdots, *B_i, \cdots, B_m \rightarrow C_1, \cdots, C_n,$$

is the *non-deleting molecule*, which is not deleted from the global states when this rule is applied. The asterisk can be prefixed in any atoms on the left-hand side. This rule can be replaced by the rule

$$B_1, \cdots, B_i, \cdots, B_m \rightarrow B_i, C_1, \cdots, C_n.$$

We can apply this transformation to any number of non-deleting molecules in the program. Note that any PPS in which all atoms are non-deleting is race-free.

Evaluable Predicates and Terms

We can extend the use of programs in PPS from pure logical deduction to functional computation by adding some functions to test conditions and evaluate the arithmetic expressions. For the first extension, atoms on the left side can be terms with "external predicates" to test conditions and converting data term. In this paper, we represent these atoms by deterministic Prolog goals with the prefix operator `#`, e.g., the term `#(X > Y+1)`, where operator $>$ is the external predicate. These terms are evaluated after all necessary variables in the condition have been instantiated. We consider the term to be a non-deleting atom and the system to have an implicit model of the external predicate, a possibly infinite set of ground unit clauses.

For the second extension, we allow the rules to contain evaluable terms of arithmetic expressions with the prefix `$` in the atoms, e.g., `$(2.0*X+1.0)`. The arithmetic expression can be placed on both the left and right hand sides of a rule, and it is evaluated and replaced by its value when the rule is applied.

Simulating 1-D Cellular Automata

This section shows a PPS that simulates a 1-dimensional synchronous cellular automaton (1-D CA) and has an identical computational result. We suppose that the CA is bounded in the sense that the leftmost and rightmost cells are fixed and have the special boundary state $\natural$. The CA with three neighbors is defined by a set Q of cell states including $\natural$ and a local function $f : Q^3 \rightarrow Q$. We represent n cells by the numbers $1, 2, \cdots, n$ and each configuration at time i by $\natural q_1^i\ q_2^i \cdots q_n^i \natural$.

We construct a PPS P_Z simulating a 1-D CA Z as follows.

1. For all even time points $t \geq 0$, the state q_j^t of each cell $j, 2 \leq j \leq n-1$ is represented by three molecules $c(j, q_j^t), l(j, q_j^t), r(j, q_j^t)$, and the state of the leftmost

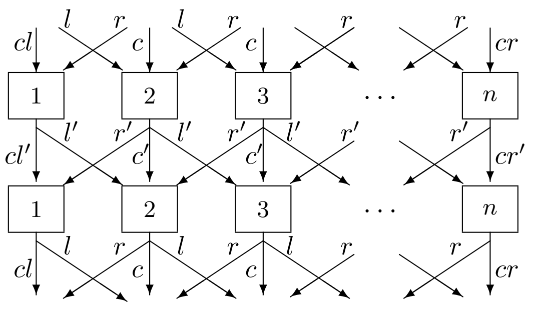

Figure 1: A hypothetical data flow diagram of PPS P_Z for simulating a 1-D CA.

and rightmost cells 1 and n by $cl(1, q_1^t), l(1, q_1^t)$, and $cr(n, q_n^t), r(n, q_n^t)$, respectively. For all odd time points t, the state q_j^t of each cell is represented similarly, except that the predicate symbols c, l and r are replaced by c', l' and r', respectively.

2. The initial global state is the set of the following molecules, which represents the initial configuration $\natural q_1^0\ q_2^0 \cdots q_n^0 \natural$.
   (a) $cl(1, q_1^0),\ l(1, q_1^0)$.
   (b) $l(j, q_j^0),\ c(j, q_j^0),\ r(j, q_j^0),\ 2 \le j \le n-1$.
   (c) $r(n, q_n^0),\ cr(n, q_n^0)$.
3. The program of P_Z is the set of the following rules, where (V `is` $f(L, C, R)$) is a Prolog expression that unifies V with the value of the local function f.

[Program of P_Z for simulating 1-D CA]

```
cl(1,C),r(2,R),#(V is f(♮,C,R))→
                        cl'(1,V),l'(1,V).
c(J,C),#(2 ≤ J ≤ n−1),l($(J−1),L),
    r($(J+1),R),#(V is f(L,C,R))→
                        r'(J,V),c'(J,V),l'(J,V).
cr(n,C),l(n−1,R),#(V is f(L,C,♮))→
                        cl'(n, V),r'(n,V)).
cl'(1,C),r'(2,R),#(V is f(♮,C,R))→
                        cl(1, V),l(1,V).
c'(J, C),#(2 ≤ J ≤ n−1),l'($(J−1), L),
    r'($(J+1),R),#(V is f(L,C,R) →
                        r(J,V),c(J,V),l(J,V).
cr'(n,C),l'(n−1,R),#(U is f(L,C,♮))→
                        cl(n,V),r(n,V).
```

Fig. 1 shows a hypothetical data flow diagram for transitions in P_Z.

Proposition 2 *If the synchronous transition in the 1-D CA Z terminates at time t with the configuration $\natural q_1^t\ q_2^t \cdots q_n^t \natural = \natural q_1^{t+1}\ q_2^{t+1} \cdots\ q_n^{t+1} \natural$, then all the allowed sequences in the PPS P_Z fall into the final equivalence class, in which every global state represents this configuration.*

Proof (*Outline*) We can prove the following two lemmas by mathematical induction on the number of applications of the rules.

1. The proposition holds, if we restrict the allowed sequences to one that includes the synchronous transition sequence.
2. The PPS P_Z is race-free, i.e., all the applications of rules to two molecules and three molecules are not affected by the other operations.

These lemmas imply that the proposition is true for all allowed sequences by Proposition 1. □

We restrict the 1-D CA to the bounded CA in order to simplify the construction of P_Z. It is not difficult to extend the CA model to a more general one such that the boundaries expand with time.

Parallel Universal Computation

A universal program U for PPS is an interpreter such that for any program P, U inputs molecules for a coded program of P and (coded) data molecules D and outputs the molecules that are equivalent to the result of the computation of P for D. The universal program not only describes how the programs are computed, but also makes it possible to easily extend the language. Furthermore, using the universal program, PPS programs can generate programs to be executed later. In particular, the universal program for PPS provides an environment with fixed interaction rules, in which the codes of rules are active molecules that interact with the data molecules.

In this section, we show a universal program for race-free PPSes. We represent the internal code of a rule $B_1, \cdots, B_m \rightarrow C_1, \cdots, C_n$ without evaluable predicates by the molecule,

$$\mathtt{rbc}([B_1, \cdots, B_m], [C_1, \cdots, C_n]),$$

where the list can be an empty list $[\,]$ when $m = 0$ or $n = 0$ and $[B_1]$ or $[C_1]$ are also written B_1 and C_1. For example, $\star\mathtt{rbc}(B, C)$ and $\star\mathtt{rbc}(B, [\,])$ are codes for $B \rightarrow C$ and $B \rightarrow$, respectively. We represent a rule having an atom $\#P$ with an evaluable predicate by

$$\mathtt{rbpc}([B_1, \cdots, B_m], P, [C_1, \cdots, C_n]),$$

The following universal program uses list operations in Prolog to process sequences of atoms.

[Parallel universal program]

```
⋆rbc([B|L],CL),B  → rbc(L,CL).
rbc([],[]) →.
rbc([],[C|L]) → C, rbc([],L).
rbc([B|L],CL),B  → rbc(L,CL).

⋆rbpc([B|L],P,CL),B  → rbpc(L,P,CL).
rbpc([],P,[C|L]), #P → C, rbc([],L).
rbpc([B|L],P,CL),B  → rbpc(L,P,CL).
```

Consider that the global state contains a coded rule rbc($[B_1, \cdots, B_m]$, $[C_1, \cdots, C_n]$) and molecules $B'_1, \cdots, B'_m$. If B_i unifies with B'_i for each i by an mgu θ_i, the universal program generates molecules $(C_1, \cdots, C_n)\theta_1 \cdots \theta_m$. This process proceeds correctly in race-free computation of a PPS.

Self-Replication

In this section, we show not only small simple self-replicating sets of molecules, but also self-replication of coded programs composed of a number of *labelled* molecules, such that each replicated molecule is a variant of the original molecule except that the label is different from that of the original. By labeling groups of molecules, the global state can have two or more groups of equivalent coded programs working independently. We add a common label to either the first element of coded rules or the first argument of molecules in a group.

A set S of molecules is *self-replicating*, if and only if there are a simple "start command" molecule p and a set S' of molecules such that:

1. $S \cup \{p\} \Rightarrow^* S \cup S'$, and $S \cup S'$ is a terminal state; and
2. each member in S' is either a variant, or a variant with different label, of a member of S and vice versa, and hence, S' is also a self-replicating set.

In this section, we represent the coded rules using the more readable form ($[B_1, \cdots, B_m]$, #P $\rightarrow$ $[C_1, \cdots, C_n]$) in stead of the form rbpc($[B_1, \cdots, B_m]$, #P, $[C_1, \cdots, C_n]$).

Simple Self-Replicating Molecules

One common method of self-replicating programs is based on the doubling of a part within a program.

[Self-replication by doubling a term]

```
rep → p((p(R)→[(rep → p(R)),R])).
p(R)→ (rep → p(R)), R.
```

When the molecule rep is given, the first rule generates the molecule p((p(R)→[(rep→p(R)),R])). From this molecule, the second rule generates a pair of molecules, which is a variant of the coded program.

Self-Replicating Molecules with Labels

We can transform the simple self-replicating program above to a self-replicating set of molecules identified by a unique label.

Because of the restriction known as *single assignment rule* in logic programming, it is not straightforward to change part of a term without reconstructing the term. To assign the labels in the molecules to different labels, we use *mutable terms*, which are proposed to realize global variables in Prolog (Nakamura, 2009). We consider the mutable term as a variable with assignable values[1]. We represent mutable terms with a value v by $mt($v$), and suppose that its value can be changed to v' by evaluating the term alter($mt($v$), v').

The following rules constitute self-replicating molecules with label l.

[Self-replication with labels by doubling a term]

```
rep($mt(l),L1) → p($mt(l),L1,
   (p($mt(l),L1,R), #alter($mt(l),L2) →
   [(rep($mt(l),L2)→ p($mt(l),L2,R)),R])).

p($mt(l),L1,R),#alter($mt(l),L1)→
   (rep($mt(l),L2)→ p($mt(l),L2,R)),R.
```

For the starting molecule rep($mt(l),m), this program generates two molecules that are equivalent to the original program except that the mutable term $mt(l) is changed to $mt(m). We can repeat this self-replication process by giving the starting command rep($mt(m),n).

Self-Replication by Copying Molecules

Another common method for self-replicating programs is copying such that each part of the program alternately copies the other parts or a program code exists with the capability to inspect and copy itself (Laing, 1976; Ray, 1992; Hutton, 2003).

In the following self-replicating program, two coded rules copy each other.

[Self-replication by copying]

```
rep,*([rpl|B]→D)→([rpl|B]→D),rpl
rpl,*([rep|B]→D)→([rep|B]→D).
```

Note that the term ([rpl|B]→ D) on the left side of the first rule unifies with the second rule. For the starting command rep, the first rule generates a replicated coded rule of the second rule and the molecule rpl, which starts the second rule. The second rule generates a copy of the first rule.

There is also another type of self-replicating molecules that use copying.

[Parallel self-replication by copying]

```
rep → rpa,rpb.
rpa,*([rpb|B]→D)→([rpb|B]→D).

rpb,*([rpa|B]→D),*([rep|B1]→D1) →
    ([rpa|B]→D),([rep|B1]→D1).
```

For the starting command rep, the first rule generates two molecules rpa and rpb, which start the second and third rules, respectively. The second and third rules generate copies of the third and second rules. As this process can run in parallel, the second and third molecules can be used

[1]The mutable terms can be realized by using lists terminated by variables so that the last element E_k of the list $[E_1, \cdots, E_k|X]$ represents the value. This method is simple but not efficient.

```
%%%%%%%%%%%%%%  Rule 1  %%%%%%%%%%%%%%%%%%%
 rep_j($mt(l),L)→
      rpa_j($mt(l),L),rpb_j($mt(l),L).

%%%%%%%%%%%%%%  Rule 2  %%%%%%%%%%%%%%%%%%%
rpa_j($mt(l),L),
*([rpb_j($mt(l),L)|B],#P → D),
*([rule_{2j-1},$mt(l)|B1]→D1),
*([rule_{2j},$mt(l)|B2]→D2),
#alter($mt(l),L)
→
([rpb_j($mt(l),L1)|B],#alter($mt(l),L1)→D),
([rule_{2j-1},$mt(l)|B1]→D1),
([rule_{2j},$mt(l)|B2]→D2),
rep_{2j}($mt(l),L), rep_{2j+1}($mt(l),L).

%%%%%%%%%%%%%%  Rule 3  %%%%%%%%%%%%%%%%%%%
rpb_j($mt(l),L),
*([rep_j($mt(l),L1)|B] →D),
*([rpa_j($mt(l),L1)|B1],#P →D1),
#alter($mt(l),L)
→
*([rep_j($mt(l),L1)|B],#alter($mt(l),L1)→D),
([rpa_j($mt(l),L1)|B1],#alter($mt(l),L1))→D1).
```

Figure 2: Three rules in module M_j for self-replication of the coded program.

simultaneously as rules and objects of the operation. Therefore, the second and third rules should be non-deleting to keep this PPS race-free.

Self-Replication of Coded Programs

Based on the self-replication by copying shown in the last subsection, we can transform a labelled PPS program to a self-replicating set of molecules.

Let P be any program of N rules. We suppose that each j-th rule in P is unified with the term `([rule`$_j$`,$mt(l)|B]→D)` with the initial label `l`. The transformed program is the union of $M_1, M_2, \cdots, M_{N/2}$ and P, where M_j is a module of the three rules in Fig. 3 for $1 \leq j \leq N/2$, and the second rule contains:

1. the term `([rule`$_{2j}$`,$mt(l)|B2]→D2)` in both sides of the rule, if and only if $2j \leq N$; and
2. the terms in the right hand side `rep`$_{2j}$`($mt(l),L)` and `rep`$_{2j+1}$`($mt(l),L)`, if and only if $j \leq N/2$.

For the starting command `rep`j`($mt(l),m)`, each rule in M_j works as follows:

1. The first rule generates molecules `rpa`$_j$`($mt(l),m)` and `rpb`$_j$`($mt(l),m)`;
2. The second rule replicates the $(2j-1)$-th rule and the $2j$-th rule of P, if $2j \leq N$, and generates the molecules `rep`$_{2j}$`($mt(l),m)` and `rep`$_{2j+1}$`($mt(l),m)`, if $j \leq N/2$; and
3. The third rule replicates the first and second rules.

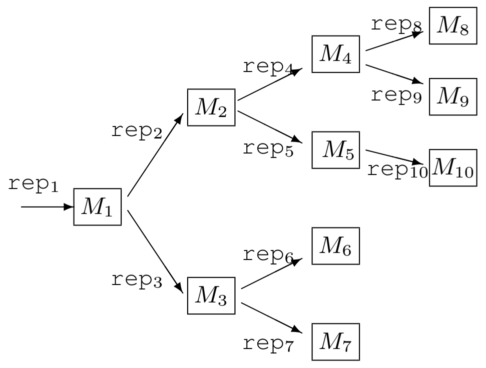

Figure 3: A data flow diagram for the self-replication of a coded program with 20 rules. Each of 10 modules replicates two program rules and three rules of the module itself.

Fig.3 illustrates the data flow in the self-replication of program T for the case $N = 20$. Each modules M_j, with $1 \leq j \leq 4$ generates two molecules `rep`$_{2j}$`($mt(l),m)` and `rep`$_{2j+1}$`($mt(l),m)`, while M_5 generates only `rep`$_{10}$`($mt(l),m)`. Every module replicates two rules in P and three rules of the module.

The following proposition summarizes the discussion in this section.

Proposition 3 *For any PPS program P, we can construct a set T of labelled molecules such that*

1. $|T| \leq 2.5 \cdot |P|$.
2. *T contains a coded program equivalent to P.*
3. *T replicates itself in time $O(\log |P|)$ by race-free computation of T and the start command molecule: it generates all the modules each of which is a variant of the corresponding element in T with a different label .*

We can reduce the factor of 2.5 for the size $|T|$ to less than 2 by changing the module to copy three or more rules.

Concluding Remarks

In this paper, we discussed asynchronous parallel universal computation and self-replication based on a computation model, called the logic molecular model, or the parallel production system. The model is based on the parallel application of production rules, which is forward deduction based on extended Horn clauses.

We showed that for any PPS program, there is a set of molecules that contains the coded program, which replicates itself by asynchronous parallel computation in time proportional to $\log n$, where n is the number of rules in the program. This type of self-replication is important for a theoretical model of biological systems, in which the most processes including self-replication seem asynchronous and parallel.

The essential features of the molecular model are summarized as follows.

- The PPS is a simple model for parallel functional computation as well as for parallel logical deduction.
- The programs in the PPS are compact, as the rules represent patterns of deductions and do not specify the order of the deductions. The PPS is effective for specifying several parallel computations, including parallel parsing, simulating 1-D CA and universal computation.
- As a universal Turing machine and universal programs for sequential computation, the parallel universal program suggests the generality and the computation power of the parallel computation model. By the universal program, the molecules are not only data tokens but also coded rules that can generate other molecules of coded rules. The coded rules are similar to enzymes in biological systems because these molecules control the interactions of other molecules.

We tested several PPS programs including parallel sorting using bitonic sort in addition to the example programs in this paper by using a serial interpreter of PPS in Prolog.

An interesting question regarding self-replication is the cost required to transform a coded program into a self-reproducing set of molecules. The transformed self-replicating coded program in the previous section requires extra $1.5N$ rules for a program with N rules. Reducing the number of rules in parallel self-reproducing programs is a topic for future work to address. Other future problems include:

- implementation of PPS in a concurrent environment;
- machine learning of self-replicating PPSes by extending methods of learning definite clause grammars (DCGs) (Imada and Nakamura, 2010); and
- application of this paper's approaches to amorphous computing (Abelson et al., 2007), to DNA and molecular computing and to chemical kinematics.

Acknowledgement

The author would like to thank Keita Imada for his help in preparing the manuscript. This work is partially supported by KAKENHI 21500148 and the Research Institute for Technology of Tokyo Denki University, Q09J-06.

References

Abelson, J., Beal, J., and Sussman, G. (2007). Amorphous computing. Computer Science and Artificial Intelligence Laboratory Technical report, MIT-CSAIL-TR-2007-030.

Albert, J. and Culik, K. (1987). A simple universal cellular automaton and its one-way and totalistic version. *Complex Systems*, 1:1–16.

Baâtre, J.-P. and Métayer, D. (1993). Programming by multiset transformation. *Comm. ACM*, 36:98–111.

Berry, G. and Boudol, G. (1992). The chemical abstract machine. *Theoretical Computer Science*, 96:217–248.

Chang, C. (1970). The unit proof and the input proof in theorem proving. *Jour. of ACM*, 17:689–707.

Cooper, T. and Wogrin, N. (1988). *Rule-Based Programming with OPS-5*. Morgan Kaufmann.

Dawkins, R. (2004). *The Ancestor's Tale: A Pilgrimage to the Dawn of Life*. Weidenfeld & Nicolson, London.

Dennis, J. (1975). First version of data flow procedure language. MIT/ LCS/ TM-61, MIT.

Hutton, T. (2003). Evolvable self-replicating molecules in an artificial chemistry. *Artificial Life*, 8:341–356.

Imada, K. and Nakamura, K. (2010). Search for semi-minimal rule sets in incremental learning of contextfree and definite clause grammars. *IEICE Trans.*, E93-D:1197–1204.

Laing, R. (1976). Automaton inspection. *Journal of Computer and System Sciences*, 13:172–183.

Lindenmayer, A. (1968). A mathematical model for cellular interaction in development i, filament with one-side inputs. *Jour. of Theoretical Biology*, 18:280–289.

Muller, D. and Burtky, S. (1959). A theory of asynchronous circuits. In *Proc. of International Symposium on the Theory of Switching 29*, pages 204–243. Annuals of the Computation Laboratory of Harvard University.

Nakamura, K. (1981). Synchronous to asynchronous transformation of polyautomata. *Jour. of Computer and System Sciences*, 23:22–37.

Nakamura, K. (1997). Parallel universal computation and self-reproduction in cellular spaces. *IEICE Trans.*, E80-D:547–552.

Nakamura, K. (2009). Proposal for global variable in prolog. ISO/IEC Draft PDTR 13211-X:2010 http://www.sju.edu/ jhodgson/wg17/Drafts/pdtr10.pdf.

Nehaniv, C. L. (2002). Self-reproduction in asynchronous cellular automata. *Proc. of NASA/DOD Conference on Evolvable Hardware (EH02)*, pages 201–209.

Ray, T. S. (1992). Evolution, ecology and optimization of digital organisms. Technicall Report, Santa Fe Institute 92-08-042.

Robinson, J. (1992). Logic and logic programming. *Comm. ACM*, 35:40–65.

Sipper, M. (1998). Fifty years of research on self-replication: An overview. *Artificial Life*, 4:237–257.

von Neumann, J. (1966). *Theory of Self-Reproducing Automata, (Edited and completed by A. W. Burks)*. University of Illinois Press, Urbana and London.