Recursive Calculus with Membranes\footnote{Supported by Spanish Secretaria de Estado de Educacion, Universidades, Investigacion y Desarrollo, project SAB1999-0025}

Adrian ATANASIU
Faculty of Mathematics, Bucharest University
Str. Academiei 14, sector 1
70100 Bucharest, Romania
E-mail: aadrian@pcnet.ro

Carlos MARTÍN-VIDE
Rovira i Virgili University
Pl. Imperial Tarraco 1
43005 Tarragona, Spain
E-mail: cmv@correu.urv.es

Abstract. P systems are computing models, where certain objects can evolve in parallel into a hierarchical membrane structure. Recent results show that this model is a promising framework for solving NP-complete problems in polynomial time. A variant of P systems with active membranes is proposed here. It uses a new operation called “subordination”, based on the process of “endocytosis” of membranes: a membrane can be entirely absorbed by another membrane, preserving its content intact. The generative power of this class of P systems with active membranes covers \textit{RE}. Arithmetical operations defined in [1] can be obtained as particular cases of primitive recursive functions, but with a higher complexity degree.

1 Prerequisites

For the elements of formal languages we shall use definitions and notations from [9]; for the basic notions, notations and results about P systems [4], [6], [7], [8] can be consulted. In this paper we shall use a variant of P systems with active membranes, very close to that defined in [4].

A \textit{P system with active membranes} is a construct $\Pi = (V, T, H, \mu, w_1, \ldots, w_m, R)$ where:

1. $m \geq 1$;

2. $V$ is an alphabet (the total alphabet of the system); its elements are called \textit{objects};

3. $T \subseteq V$ (the terminal alphabet);

4. $H$ is a finite set of labels for membranes;
5. \( \mu \) is a membrane structure, consisting of \( m \) membranes, labeled (not necessarily in a one-to-one manner) with elements of \( H \); there is a (unique) membrane \( s \) called skin; all the other membranes are inside of the skin;

6. \( w_1, w_2, \ldots, w_n \) are strings over \( V \), describing the multisets of objects placed in the \( m \) regions of \( \mu \);

7. \( R \) is a finite set of development rules we define in the next section.

Let \( x = a_1 a_2 \ldots a_k = a_1 \cdot 2^{k-1} + a_2 \cdot 2^{k-2} + \ldots + a_k (a_i \in \{0, 1\}, 1 \leq i \leq k, k \geq 1) \) be a binary integer. An Arithmetical P System (APS for short) is defined in [1] as follows:

\[ \Pi = (V, T, H, \mu, w_1, w_2, \ldots, w_n, R), \]

where the integer \( n_0 \) is a constant fixed by the system (the examples from this paper use the value \( n_0 = k + 1 \)), \( T = \{0, 1\} \), \( V \setminus T = \{f\} \), \( H = \{1, 2, \ldots, n_0\} \), \( \mu = [1 \ldots [a_0]_{n_0} \ldots [a_k]_{n_0}]_2 \), \( w_i = a_{k+1-i} (1 \leq i \leq k) \), \( w_i = f (k + 1 \leq i \leq n_0) \), \( R \) is a set of rules, unspecified in this stage.

The graphic representation of an APS is illustrated in Figure 1.

![Figure 1: The structure of an APS](image)

So, in an APS each membrane contains only one object: a digit (a terminal object) or \( f \) (a special nonterminal object); the digit from the membrane \( i \) is more significant than all digits situated in the membranes \( j \) with \( j < i \) and less significant than all digits situated in the membranes \( j \) with \( j > i \). An \( f \)-membrane is the inermest membrane or it contains only \( f \)-membranes. Every APS contains at least one \( f \)-membrane.

In order to index an APS, we use a nonterminal object \( i \), placed in its membrane 1; it will design a binary integer \( x_i \).

Because an APS will be placed in other P systems, its outermost membrane 1 will not be considered as being the skin.

2 Primary and Secondary Rules

Let \( \Pi \) be a membrane structure defined as above; the set \( R \) of development rules contains rules of the following types (for \( i, j \in H, u, v, w \in V^*, \alpha, \beta, \gamma \in \{0, +, -\} \)):
1. \([u_i]^0 \rightarrow v[i_i]^0\) (filter-out);
2. \(u[i]^0 \rightarrow [v]^0\) (filter-in);
3. \([u_i]^0 \rightarrow v \ (i \neq s)\) (membrane dissolving);
4. \([u_i]^0 \rightarrow [v]^0\ [w]^0 \ (i \neq s)\) (membrane division);
5. \([i_i]^0 [j_j]^0 \rightarrow [i_i]^0 [j_j]^0\ \ (\alpha \neq +, \ i \neq s, \ j \neq s)\) (subordination).

These rules are called primary rules (operations) and they are applied according to the following principles ([4]):

- All the operations are applied in parallel to all objects to which they can be applied. One object can be used by only one operation, non-deterministically chosen (there is no priority relation among the rules of types (1) – (3)), but any object which can evolve by an operation of any form, should evolve.

- If a membrane is dissolved, then all the objects in its region are left free in the region immediately above it; the operations of a dissolved membrane are no longer available at the next steps. The skin membrane is never dissolved.

- All objects and membranes not specified in an operation and which do not evolve are passed unchanged to the next step.

- At one step, a membrane \(i\) can be the subject of only one operation.

- The operations of types (1) – (3) have a higher priority than division; the subordination has the lowest priority.

Besides these five primary operations, other two secondary operations can be defined:

6. \([u \rightarrow v]^0\);
7. \([i_i]^0 [j_j]^0 \rightarrow [i_i]^0 [j_j]^0\ [i_i]^0 [j_j]^0\].

These two operations are defined in [4] (with minor changes mentioned in [1]).

**Lemma 1** Operations (6) and (7) can be defined in terms of (1) – (5).

**Proof.** Let \(i \underline{\mu} v\) be a new nonterminal object. Rule (6) can be defined by a rule of type (1) and a rule of type (2) as follows:

\([u_i]^0 \rightarrow i \underline{\nu} v[i_i]^0\; \text{ and } \; i \underline{\nu} v[i_i]^0 \rightarrow [v]^0\).

The object \(i \underline{\mu} v\) and the priority of operations of type (1) – (2) assure the consistence of this operation.

For operation (7) the construction is much longer. It should involve:

1. A rule of type (5) which transforms \([i_i]^0 [j_j]^0 \rightarrow [i_i]^0 [j_j]^0\);
2. A new nonterminal object \(z\) and a rule of type (4) which transforms this configuration in \([i \underline{z} i_i]^0 [j_j]^0 \rightarrow [i \underline{z} i_i]^0 [j_j]^0\].
3. A new nonterminal object \( x \) and a rule of type (2) which leads to the configuration 
\[
[x_j^{\gamma}]_j \xrightarrow{\mathcal{R}} [x_j^{\gamma}]_j
\]

4. Using this object \( x \), all the objects and membranes from \([j]_j^{\beta}\) excepting \([j]_j^{\beta}\) are erased. During the last operation, a new nonterminal object \( y \) is generated in the membrane \( j \).

5. With a rule of type (3), \( y \) dissolves the membrane \( j \) polarised \( \gamma \).

\[\square\]

**Remark 1** The set of rules (6), (2), (3), (4), (5) can be considered also primary: rule (1) can be defined in terms of (2), (3), (4) and (6). We shall denote by “I-primary rules” the set of rules (1) – (4), (6), (7) (see [1], [3], [4]) and by “II-primary rules” the set of rules (1) – (5).

**Lemma 2** The subordination rule (5) cannot be obtained in a \( P \) system with active membranes defined by I-primary rules.

**Proof.** Let us associate with a \( P \) system a tree, according to the definition given in [4]. The operations of filtering (1), (2) preserve the tree’s structure unchanged. The dissolving rule erases a vertex together with its edges (thus, the depth of the tree decreases):

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\triangle \\
\end{array}
\]

Operations (4) and (7) add new edges to the tree:

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\end{array}
\]

(4)

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\end{array}
\]

(7)

All these operations preserve the depth of the tree. Thus, any combination of them will have as a result a tree with at most the same depth.

The subordination rule increases the depth of the tree:

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\end{array}
\]

(6)

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\end{array}
\]

(7)

All these operations preserve the depth of the tree. Thus, any combination of them will have as a result a tree with at most the same depth.

The subordination rule increases the depth of the tree:

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\end{array}
\]

(6)

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\end{array}
\]

(7)

All these operations preserve the depth of the tree. Thus, any combination of them will have as a result a tree with at most the same depth.

The subordination rule increases the depth of the tree:

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\end{array}
\]

(6)

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\end{array}
\]

(7)

All these operations preserve the depth of the tree. Thus, any combination of them will have as a result a tree with at most the same depth.

The subordination rule increases the depth of the tree:

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\end{array}
\]

(6)

\[
\begin{array}{c}
\bullet \quad \bullet \\
i \quad j \\
\end{array} \quad \Rightarrow \quad \begin{array}{c}
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\begin{array}{c}
\bullet \\
i \\
\end{array} \\
\end{array}
\]

(7)

All these operations preserve the depth of the tree. Thus, any combination of them will have as a result a tree with at most the same depth.

The subordination rule increases the depth of the tree:

\[\square\]
Thus, any problem which can be solved using the $I$-primary rules, will be solved also using the set $II$-primary rules.

The importance of the subordination rule (not defined in [4]), consists in the possibility of defining the recursive calculus. It seems that $P$ systems with active membranes defined by $I$-primary rules cannot operate with recursion, although they generate $RE$ too ([4]).

We shall show in the following that $P$ systems with active membranes defined by $II$-primary rules can compute all primitive recursive functions; moreover, the function $\text{min}$ can be implemented in these systems, thus they cover $RE$, like systems with active membranes with $I$-primary set of rules, or $P$ systems on graphs ([5]).

3 Primitive Recursive Functions with Membranes

3.1 Primitive recursive functions

We remember the definition of primitive recursive functions ([2]):

The primitive recursive functions form the smallest class of functions that

(1) includes

(a) the zero function: $z(x) = 0$ for all $x \in \mathbb{N}$ ($\mathbb{N}$ is the set of all non-negative integers);

(b) the successor function: $s(x) = x + 1$ for all $x \in \mathbb{N}$;

(c) the projection functions: $p_i(x_1, \ldots, x_n) = x_i$, $x_i \in \mathbb{N}$, $n \geq 1$, $1 \leq i \leq n$.

(2) is closed under composition: $h(x_1, \ldots, x_n) = v(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$, where $h, g_1, \ldots, g_m : \mathbb{N}^n \rightarrow \mathbb{N}$, $v : \mathbb{N}^m \rightarrow \mathbb{N}$, $x_i \in \mathbb{N}$, $1 \leq i \leq n$.

(3) is closed under primitive recursion: $h(x, 0) = u(x)$, $h(x, y + 1) = g(x, y, h(x, y))$, where $h : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$, $u : \mathbb{N}^n \rightarrow \mathbb{N}$, $g : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$, $x \in \mathbb{N}^n$, $y \in \mathbb{N}$.

All primitive recursive functions are Turing computable, but not all computable functions are primitive recursive (a counterexample is Ackermann’s function).

If

$$(d) \quad \text{min}(u) = \text{min}\{x | u(x) = 0\}$$

is included in (1), then all Turing computable functions are obtained.

3.2 Functions with membranes

In this paper we shall use $P$ systems with active membranes defined with the set of $II$-primary rules. Also, the skin will be ignored in our constructions (its existence goes without noticing).

We know that an integer $a$ is defined as an $APS$ $A$ with $n_0$ membranes and sets of rules associated with each membrane. These rules are of the types (1) – (3) and (6) (by extension) by permitting to some objects to accomplish some special operations. In each membrane $i$ ($1 \leq i \leq n_0$), the objects:

- $y$ changes into 0 the digit situated in the membrane 1 and into $f$ the digits from the other membranes; it is generated in each membrane by the object $z$.

- $t$ filters-out each membrane and dissolves all function-membranes.

35
+ complementates any digit and transforms the object \( f \) in 1.

- \( p \) introduces + in the immediately inner membrane.

- \( x \) filters in each membrane and introduces in the innermost membrane \( n_0 \) the object \( del \).

- \( del \) removes all the objects and dissolves all the membranes of an APS.

In the next section we shall detail these actions.

An APS will be denoted by \( [i A]^i_h \) or by \( [i i A]^i_h \), where \( i = 1, \ldots, n \) is an index for the integer stored in \( A \).

A \( n \)-ary function membrane \( h \) is a P system with \( \mu = [h c_0]^n_h \), having \( 2n + 1 \) nonterminal objects \( c_0, c_1, \ldots, c_{n-1}, \underline{L}, \ldots, \underline{u}, \text{fin} \in V \setminus T \). The whole content of \( V \setminus T \) depends on the function.

A function membrane is floating in the skin in a sleeping state, defined by the polarity –. An object \( \text{eval}_h \) situated in the skin makes a copy of the P system which defines that function \( h \):

\[
\text{eval}_h [h]_h^{-} \rightarrow [h]_h^0 [h]_h^{-}.
\]

\( [h]_h \) remains unchanged being preserved for other calls. \( [h]_h^0 \) collects \( n \) APS positively polarised, using a rule of type (5) (in this moment in the skin there is no other rule with a higher priority):

\[
[h]_h^0 [i A]^i_h \rightarrow [h[i A]^i_h]_h^{-}.
\]

In parallel, the object \( c_i \) generates the index \( i \) which will be stored in each APS just entered. All APS have the polarity 0 unless possibly the last one, which will be polarised by +:

\[
[h c_i]_h \rightarrow [c_i]_h^0, \quad [i]_h^0 \rightarrow [i]_h^0 (1 \leq i \leq n - 1),
\]

\[
[h c_n]_h \rightarrow [\text{fin}]_h^0, \quad [u]_h^i \rightarrow [u]_h^i (\text{or } [u]_h^i).
\]

The object \( \text{fin} \) changes the polarity of the function \( h \) (in order to stop other APS to enter) and outputs in the skin the object \( \text{start}_h \) which commands the execution of function \( h \):

\[
[h \text{fin}]_h^0 \rightarrow \text{start}_h [h]_h^+.
\]

After the execution of the function \( h \) is performed, the membrane \( h \) (positively polarised) is dissolved and the APS which contains the result is left free in the outer membrane.

### 4 Implementing Primitive Recursive Functions in P Systems with Active Membranes

In the following we intend to construct function membranes which verify the primitive recursive requirements. We shall not consider here the complexity problem (like in [1]); our goal is just to verify that any primitive recursive function can be obtained using a P system with active membranes (defined by II-\( \text{primary} \) rules).
4.1 The zero function

The zero function acts in the following way (in P systems terms):

\[
\begin{array}{c}
A \\
\text{zero}
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
0
\end{array}
\]

(all APSs will be represented by circles).

The initial membrane structure is \( \mu = [\text{start}_{\text{zero}}[\text{zero}\{1\}A[1]0\}^+]^0 \).

The development rules are:

\[
\begin{align*}
\text{start}_{\text{zero}}[\text{zero}]^+ & \rightarrow [\text{zero}]^+ \\
[y^n]_i^0 & \rightarrow [z^n]_i^0 \\
[y^n]_i^0 & \rightarrow [z^n]_i^0 \\
[y^n]_i^0 & \rightarrow [z^n]_i^0 \\
[y^n]_i^0 & \rightarrow [z^n]_i^0
\end{align*}
\]

The behavior of the objects \( y, z, t \) was presented in Section 3.2.

After the action of this function is over, the function membrane is dissolved.

\textbf{Example 1} The transformations of the function membrane zero when “zero(101)” is computed are the following (in all examples, the skin will be ignored):

\[
\begin{align*}
\text{start}_{\text{zero}}[\text{zero}\{1\}A[1]0\}^+]^0 & \rightarrow [\text{zero}]^+ \\
[\text{zero}]^+ & \rightarrow [\text{zero}]^+ \\
[\text{zero}]^+ & \rightarrow [\text{zero}]^+ \\
[\text{zero}]^+ & \rightarrow [\text{zero}]^+ \\
[\text{zero}]^+ & \rightarrow [\text{zero}]^+ \\
[\text{zero}]^+ & \rightarrow [\text{zero}]^+ \\
[\text{zero}]^+ & \rightarrow [\text{zero}]^+ \\
[\text{zero}]^+ & \rightarrow [\text{zero}]^+ \\
[\text{zero}]^+ & \rightarrow [\text{zero}]^+
\end{align*}
\]

4.2 The successor function

For the successor function \( (\text{succ}(x) = x + 1) \) we have:

\[
\begin{array}{c}
A \\
\text{succ}
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
A + 1
\end{array}
\]

Here, the starting membrane structure is \( \mu = [\text{start}_{\text{succ}}[\text{succ}\{1\}A[1]0\}^+]^0 \).

The development rules for the function membrane successor are:

\[
\begin{align*}
\text{start}_{\text{succ}}[\text{succ}]^+ & \rightarrow [\text{succ}]^+ \\
[p][i]^0 & \rightarrow [i + 1]^0 \\
[p][i]^0 & \rightarrow [i + 1]^0 \\
[t][i]^0 & \rightarrow [t][i]^0
\end{align*}
\]

The behavior of the objects \( t, p, + \) is presented in Section 3.2. The repetition of the object \( \downarrow \) deletes the index of the APS \( A \).
Example 2 Let us compute “suc(101)”. The transformations of the function membrane \( \text{suc} \) are the following:

\[
\text{suc}(1) \rightarrow \text{suc}(1)
\]

\[
\text{suc}(1) \rightarrow \text{suc}(1)
\]

\[
\text{suc}(1) \rightarrow \text{suc}(1)
\]

Thus, suc(101) = 110.

4.3 The projection function

The projection function \( \text{pr}_i(x_1, x_2, \ldots, x_n) = x_i \) (1 ≤ i ≤ n) can be represented by:

![Diagram of A1, ..., An]

We start with a membrane structure \( s = [s_{\text{pr}_i,n}, \text{start}_{\text{pr}_i,n}, [1\ldots 1A]^0, [1\ldots \ldots nA]^0] \)

The developing rules are:

\[
\text{start}_{\text{pr}_i,n} \rightarrow [\text{pr}_i]^{[1\ldots 1\ldots 1]} (1 \leq i \leq n),
\]

\[
i[i]^0 \rightarrow [i]^0 (1 \leq i \leq n),
\]

\[
x_k^0 \rightarrow x^0_k (1 \leq k \leq n),
\]

\[
[\text{del}]^n_0 \rightarrow [\text{del}]^n_0 (1 \leq k \leq n),
\]

The objects del, x, t have the behavior described in 3.2. The integer i placed in the function membrane selects (via the object \( i \)) the APS which finally remains. All the other APS are deleted. The membrane function \( \text{pr}_i,n \) is dissolved after all integers \( x_j \) (j ≠ i) (codified in APS \([1\ldots 1A]^0\)) are deleted.

Example 3 Let us compute “pro2(11,10,0)” (remember, all integers are in basis 2). The transformations of the membrane function \( \text{pro}_2,3 \) are the following:

\[
\text{start}_{\text{pro}_2,3} \equiv \text{pro}_2,3(1) \rightarrow \text{pro}_2,3(1)
\]

\[
\text{start}_{\text{pro}_2,3} \equiv \text{pro}_2,3(1) \rightarrow \text{pro}_2,3(1)
\]

\[
\text{start}_{\text{pro}_2,3} \equiv \text{pro}_2,3(1) \rightarrow \text{pro}_2,3(1)
\]

The objects del, x, t have the behavior described in 3.2. The integer i placed in the function membrane selects (via the object \( i \)) the APS which finally remains. All the other APS are deleted. The membrane function \( \text{pro}_2,3 \) is dissolved after all integers \( x_j \) (j ≠ i) (codified in APS \([1\ldots 1A]^0\)) are deleted.
4.4 The composition

The closure under the composition is represented by the diagram:

\[
\begin{array}{c}
A_1 \cdots A_n \\
h
\end{array} \quad \longrightarrow \quad \begin{array}{c}
A_1 \cdots A_n \\
g_1 \cdots g_m
\end{array}
\]

The developing rules are:

1. \( \text{start}_h \left[ h \right]_h^+ \longrightarrow [h \alpha_1]_h^+, \quad [h \alpha_i]_h^+ \longrightarrow [h w_i]_h^+ [h \alpha_{i+1}]_h^+ \) (1 \( \leq i < m \)),
   \( [h \alpha_m] \longrightarrow [\beta_m w_m]_h^+, \quad [h \beta_m]_h^+ \longrightarrow \text{eval} [w_m]_h^+ \).

The evaluation of the function \( h \) begins. The first step consists in generating \( m \) copies of \( h \), each of them having a specific object \( w_i \). The last object \( w_m \) is generated together with \( \beta_m \) which will produce (via \( \text{eval} \)) a functional copy of the function \( v \); now, a membrane \([v]_v^0 \) is available.

2. \( [h w_i]_h^+ \longrightarrow \text{eval} g_i [h]_h^+ \).

After this step, \( m \) function membranes \([g_1]_v^0, \ldots, [g_m]_v^0 \) will be generated. These rules produce also an “enzyme” which deletes the object \( \alpha_i \) from each \( g \) (this action can be also formalised, but will complicate the writing rules). Thus, the APS \( A_1, \ldots, A_n \) which are already indexed in all new function membranes \( g_i \), will not be indexed again.

3. \( [g_i]_v^0 [h]_h^+ \longrightarrow [g_i[h]_h^+]_v^+, \quad [v]_v^0 [g_i]_v^+ \longrightarrow [v[g_i]_v^0]_v^0, \) (1 \( \leq i \leq m \)).

After this step, the function membrane \( v \) becomes

\( [v[g_1[h]_h^+]_v \cdots [g_m[h]_h^+]_v]. \)

After the deletion of the membrane \( h \), those \( n \) APS from \( h \) will be left free in all membranes \( g_i \).

4. \( [g_i]_v^0 \longrightarrow [g_i \text{fin} g_i \text{del} h]_v^+ \), \( [g_i \text{fin} g_i]_v^+ \longrightarrow v \text{start}_h g_i [g_i]_v^+ \) (1 \( \leq i \leq m \)),
   \( \text{del} h [h]_h^0 \longrightarrow [\lambda]_h^+, \quad [\lambda]_h^+ \longrightarrow \lambda, \quad [g_i]_v^0 \longrightarrow \lambda. \)

The objects \( i \) are generated in \( v \) in order to index their own APS. Because here we have membrane functions, not APS, these objects will penetrate membranes \( g_i \) and begin – in the same time – the dissolution of the membrane \( h \) and the start of the function \( g_i \).

5. \( v \) is an object (“enzyme”) which temporarily stops the function evaluation \( v \). If \( \text{start}_v[v]_v^+ \longrightarrow [v]_v^+ \), then the blocking rule will be
   \( [v v]_v^+ \longrightarrow v v]_v^+. \)

6. \( [g_i t]_v^0 \longrightarrow \text{not} v \text{del} t]_v^0, \quad [t]_v^0 \longrightarrow [t]_v^0, \quad [t]_v^0 \longrightarrow [t]_v^0, \) (1 \( \leq i \leq m, \) 1 \( \leq j \leq n \)).

The object \( t \) shows that the function evaluation \( g_i \) is over; the membrane \( g_i \) is dissolved but – at the same time – the index of the new obtained APS becomes \( i \) instead of \( j \).
7. \([\text{sum}_\lambda \cdot v \rightarrow \lambda] v\).

The object \(\text{not}_\lambda w\) neutralises \(w\) and the evaluation of the function \(v\) may begin.

**Example 4** A very known property of the addition is the commutativity. This can be represented by

\[
\text{sum}(x, y) = \text{add}(\text{pro}_2(x, y), \text{pro}_1(x, y)) = \text{add}(y, x).
\]

In our arithmetic with membranes, this computing will be developed as follows:

\[
\begin{align*}
\text{start} & \rightarrow \text{sum} \rightarrow \text{add} \\
\text{eval} & \rightarrow \text{pro}_2 \rightarrow \text{add} \\
\end{align*}
\]

4.5 The primitive recursion

The closure under the primitive recursion has two stages, represented by

\[
\begin{array}{c}
A \quad 0 \\
\hline
h \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
A \\
\hline
u \\
\end{array}
\]

and

\[
\begin{array}{c}
A \quad B + 1 \\
\hline
h \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
A \quad B \quad A \quad B \\
\hline \quad g \quad h \\
\end{array}
\]

We shall consider here only one APS \(A\), but the construction is true for \(n\) APS \(A_1, \ldots, A_n\) (if \(h\) is a \((n+1)\)-ary primitive recursive function and \(g\) is a \((n+2)\)-ary primitive recursive function). The differences between a general \(n\) and the case \(n = 1\) are minimal.

The developing rules are:
1. \[ \text{start}_h[n]_h^0 \rightarrow [n \text{check} \text{ check}^+]_h, \]
\[ \text{check}[i]_h^0 \rightarrow [i a]_h^0, \quad \begin{bmatrix} a_1 & \rightarrow \frac{1}{2} \text{nil}^0 \\ a_2 & \rightarrow \frac{1}{2} \text{nil}^0 \end{bmatrix}, \quad [\text{nil}]_h^0 \rightarrow [1]^+. \]

The two APS situated into the function membrane \( h \) are identified. The first membrane is polarised +. The object \( h \) will initiate in the second APS a checking procedure which verifies if its value is 0 or not.

2. \[ \begin{bmatrix} 1_h & \rightarrow \text{dec} \\ 0_h & \rightarrow \text{nil} \end{bmatrix}_1^0, \quad [z]_2^0 \rightarrow [z]_2^0, \quad \begin{bmatrix} f_c & \rightarrow \text{init} \\ n_c & \rightarrow \text{nil} \end{bmatrix}_2^0 (n = 0, 1), \]
\[ [\text{null}]^0 \rightarrow \text{null}, \quad [\text{null}]_2^0 \rightarrow \text{null}. \]

If the integer contained by the second APS is nonzero, then its decrementation begins (for details concerning the decrementation rules, see [1]). If the integer value is 0, the goal is to activate the function membrane \( u \), via the object \( \text{null} \).

3. \[ [i \text{ call } u]_1^0 \rightarrow \text{call } u, \quad [i \text{ init}]_1^0 \rightarrow \text{eval } u, \quad [i \text{ del}]_1^0 \rightarrow \lambda, \quad [i \text{ nil}]_1^0 \rightarrow \lambda. \]

The nonterminal object \( x \) starts the dissolution of the second APS (see the details given for the function \( \text{proc } x \)); it will be finished using a new rule, which dissolves membrane 1.

The object \( \text{eval } u \) produces a copy of the function membrane \( u \), which will subordinate (rule of type (5)) the first APS (positive polarised). The object \( \bot \) generated by \( c_0 \) (see 4.2) is absorbed by the last rule we have defined here.

4. \[ [\text{call } g]_h^+ \rightarrow \text{eval } g_{[h]}^+, \quad \text{eval } g_{[h]}^+ \rightarrow [g]_g^+ [g \alpha x]_g^0, \]
\[ [s]_g^0 [h]_h^0 \rightarrow [s [h]_h^+]_g^0, \quad \begin{bmatrix} \alpha 1 & \rightarrow \alpha \\ \alpha 2 & \rightarrow \alpha \\ \alpha 3 & \rightarrow \lambda \end{bmatrix}_g^0, \]
\[ \text{call } g \rightarrow \text{start } h. \]

When the decrementation of the second APS is finished, it will generate a new object \( \text{call } g \) which initiates the wake-up procedure of the function membrane \( g \). The object \( \alpha \) absorbs all indexes generated by \( c_i \) in this function membrane (see 4.2); \( \alpha \) duplicates the function membrane \( h \). The dissolution of the membrane \( h \) positively polarised leaves free two APS in \( g \) and – at the same time – starts the evaluation of its inner function membrane \( h \).

5 The Minimality

The primitive recursive functions are not enough to cover all \( RE \) functions. We need to implement also the function \( \text{min}(u) \) defined as follows, for a recursive function \( u : \mathbb{N} \rightarrow \mathbb{N} \):

\[ x \leftarrow 0; \]
\[ \text{while } u(x) \neq 0 \text{ do succ}(x); \]
\[ \text{output}(x); \]

In terms of function membranes, this can represented as follows:
The initial membrane structure has the form \( \mu = [\mu_{\text{min}}[u[0][1]]]_{\text{min}} \); the developing rules are the following:

1. \( \text{start}_{\text{min}}[\mu_{\text{min}}]^{+} \rightarrow [\mu_{\text{min}} \pi \sigma]^{+} \),
   \( \pi[i]^{0} \rightarrow [i \pi]^{0}, \quad [i \pi]^{0} \rightarrow [i \pi]^{0} [1])^{+} \),
   \( [u[0][1]]^{0} \rightarrow [u[0][1]]^{0}, \quad \sigma[u] \rightarrow [\sigma[u]]^{0}, \quad [i \tau]^{0} \rightarrow \text{start}_{\text{u}}[i]^{0}, \quad [u \tau w] \rightarrow \text{check}_{\text{u}}^{+} \).

   The \( \text{APS} \ A \) is duplicated; one copy – indexed by \( 1 \) – is introduced in \( u \). After the evaluation of the function membrane \( u \), the object \( t \) (see 4.2) which usually dissolves the function membranes, is neutralised by \( w \).

2. \( \text{check}[i]^{0} \rightarrow [i]^{0}, \quad \left[ \begin{array}{c}
   0 \rightarrow x; \quad n \rightarrow 0 \\
   0 \rightarrow n; \quad f \rightarrow x
   \end{array} \right]^{0} \), \( n = 0, 1, \),
   \( \text{maybe}[2] \rightarrow [2]^{0}, \quad [u \text{del}] \rightarrow [t_{u}]^{0}, \quad [u \text{del}] \rightarrow t, \quad [\text{min}]^{0} \rightarrow \lambda \).

   \text{check} \ verifies \ if \ the \ \text{APS} \ situates \ in \ the \ function \ membrane \ \mu \ - \ obtained \ as \ the \ result \ of \ the \ evaluation \ of \ this \ function \ - \ has \ the \ value \ 0. \ If \ the \ answer \ is \ affirmative, \ then \ the \ membranes \ \mu \ and \ \text{min} \ are \ dissolved \ and \ the \ second \ copy \ of \ \text{APS} \ \ A \ \ (neutral \ polarised) \ is \ left \ free \ in \ the \ skin.

3. \( [2 \text{no}]^{0} \rightarrow [x]^{0}, \quad [1 \text{no}]^{0} \rightarrow \text{urm \ er}, \quad [u \text{er \ del}] \rightarrow \lambda^{0} \). \n
   \text{no} \ shows \ that \ the \ value \ obtained \ by \ the \ evaluation \ of \ function \ \mu \ is \ nonzero. \ Thus, \ the \ \text{APS} \ situates \ in \ the \ function \ membrane \ \mu \ is \ deleted. \ The \ object \ \chi \ assures \ the \ 

   dissolving \ (see 4.3) \ of \ \text{APS}, \ while \ \text{er} \ saves \ the \ function \ membrane \ \mu.

4. \( [\text{urm}]^{0} \rightarrow [p']^{0}, \quad [p'][1]^{0} \rightarrow [1]^{0}, \quad [1]^{0} \rightarrow \pi \sigma[1]^{0}. \)

   The object \( p' \) introduces the second \( \text{APS} \ A \) the nonterminal object ‘+’ which starts the incrementation (4.2). The object \( q \) generates the objects \( \pi \) and \( \sigma \) in the function membrane \( \text{min} \) and restarts this function with the new value calculated in \( A \).

\textbf{Example 5} \textit{Let us consider the function \( u(x) = 1 - x \). Here } u(0) = 1, \ u(1) = 0. \ \text{A } P \text{ system which computes “} \text{min}(u) \text{”} \textit{ goes through the following sequential steps:}

\begin{align*}
\text{start}_{\text{min}}[\mu_{\text{min}}[\mu[0][2]^{0}]_{\text{min}}]^{+} & \rightarrow [\mu_{\text{min}} \pi \sigma][u[0][2]^{0}]^{+} \\
[\mu_{\text{min}}][u][w][1][0][2]^{0}^{+} & \rightarrow [\mu_{\text{min}}][u][w][1][0][2]^{0}^{+} \\
[\mu_{\text{min}}][w][1][0][2]^{0}^{+} & \rightarrow [\mu_{\text{min}}][w][1][0][2]^{0}^{+} \\
[\mu_{\text{min}}][w][1][0][2]^{0}^{+} & \rightarrow [\mu_{\text{min}}][w][1][0][2]^{0}^{+} \\
[\mu_{\text{min}}][w][1][0][2]^{0}^{+} & \rightarrow [\mu_{\text{min}}][w][1][0][2]^{0}^{+} \\
[\mu_{\text{min}}][w][1][0][2]^{0}^{+} & \rightarrow [\mu_{\text{min}}][w][1][0][2]^{0}^{+}
\end{align*}

\( (u(0)) \) is computed; an \( \text{APS} \) containing the value 1 and an object \( t \) results in the function membrane \( u \).}
\[
\min_u \left[ \sum_{t} f(u) \right]_{u}^{0,0,0} \quad \Rightarrow \quad \min_u \left[ \sum_{t} f(u) \right]_{u}^{0,0,0} = \left( u(1) \right) \quad \text{is computed; an APS containing the value 0 and an object t results in the function membrane u)\n\]
\[
\min_u \left[ \sum_{t} f(u) \right]_{u}^{0,0,0} \quad \Rightarrow \quad \min_u \left[ \sum_{t} f(u) \right]_{u}^{0,0,0} = \left( u(1) \right) \quad \text{is computed; an APS containing the value 0 and an object t results in the function membrane u)\n\]

Thus, \( \min(u) = 1 \).

6 Final Remarks

A variant of \( P \) systems with active membranes was proposed here. Its power covers \( RE \).

Arithmetical operations defined in [1] can be obtained as particular cases of primitive recursive functions.

There are some differences between this construction and that from [1]. Here the \( II \)-\emph{primary} set of rules is used, which offers the possibility to define the composition of functions and the recursivity, which is a mathematically powerful tool.

We have not intended here to discuss the complexity of computations. The order of complexity in the primitive recursive arithmetic is greater than in the arithmetic used by computer chips. Because the rules in \( I \)-\emph{primary set} does not involve the recursivity, the complexity of computing will be lower in that case.

It seems that the power of parallel computing helps the iterative computing to solve also NP-complete problems ([3], [4]). We do not know whether \( P \) systems with active membranes defined here are more powerful than the systems defined in [4]; both of them cover \( RE \). It remains to prove whether or not special functions exist which cannot be computed in \( P \) systems with active membranes defined by one of the primary rules set, but they can be computed using the other set of primary rules.

References


