Membrane computing and combinatory logic

Adrien Ragot

Abstract

In 1998 Gheorghe Păun introduced Membrane computing a model of computation based on local rewriting on trees inspired by the behavior of membranes in biology, the model was shown to be able to simulate Turing machines, and so to be Turing complete. In this note we investigate how membrane computing and the so called P–systems relate to functional programming, for instance to combinatory logic. We exhibit a simple translation of combinators into P–systems with names with low amount of information (essentially polarized binary trees), thus obtaining a new proof of Turing completeness for P–systems. But this translation makes use of "non–elementary" (and so not entirely local) rewriting rules, therefore in the last part of the note we investigate how to simulate non elementary rules using only the elementary ones.

Contents

1 Membrane computing **3**

2 Combinatory Logic **8**

3 *P*–systems and Combinatory Logic **9**

4 Non–elementary rules simulation in reasonable time **11**

Introduction

The modern notion of calculable functions and so of *computing* is generally considered to have been introduced in the 30's, in diverse forms by (to name a few) A. Church, A. Turing and K. Gödel, with concepts such as *Turing machines, λ–calculus* or the *recursive functions*. Since these several *models of computation* where introduced at the same time, a natural question arised: do these models compute the same functions? The answer to this question was positive, meaning that a function is equivalently Turing–computable, λ–computable or recursive. Although, there remained another natural question, does the informal notion of computable function coincide with the formal notions given by these models of computation? This question cannot (mathematically) be answered as it involves an informal concept of computable function, hence a notion that cannot be defined for mathematical use. The Church–Turing thesis states that the answer must be positive, i.e. the informal notion of calculable function is the same as the formal definitions provided by Turing, Church or Gödel. Today, the thesis is widely accepted among computer scientist and has been strengthened with time since each model of computation that has been introduced was never shown to be able to compute more functions than the Turing–computable functions.

It is difficult to precisely identify what was the first notion corresponding to modern *computation*, although we can observe that the study of logic – and of what is called today *proof theory* – is closely related to theoretical computer science and its birth. One famous example is the *Curry–Howard isomorphism* stating that the proofs of intuitionistic logic in natural deduction correspond to a specific class of programs, that is the simply typeable lambda terms. This is not the only occurrence of a link between computing and mathematical logic, for instance one can also mention *Combinatory Logic*, which is also subject to the Curry–Howard isommorphism.

Combinatory logic was introduced by Moses Schönfinkel in 1924, it was designed to clarify the role of quantified variables in first order logic, to do so Schönfinkel eliminated the bounded variables directly from the syntax (i.e. there are no way to bound variables). Meanwhile in the 30's, A. Church invented the lambda calculus, a model of computation rooted in the notions of substitution and bounded variables. From there Haskell Curry who had rediscovered the work of Schönfinkel in 1927 on Combinatory Logic, was able to show that the model of Schönfinkel is equivalent to the lambda calculus of Church (and so that it is a Turing complete model of computation, meaning it can compute any calculable function[1]). Combinatory logic is therefore an extremely simple and complete model of computation: its simplicity directly comes from the fact that bounded variables cannot occur in the calculus but also that we only need a small number of combinators called *basis* (for example the combinators K and S) and their respective reductions to obtain Turing completeness. These first notions of computation are all related to *functional programming* meaning they are models of computation where programs are not running in parallel (although these model could be enriched to consider the execution of programs in a concurrent way).

In 1998 Georges Păun introduced a model of computation called *membrane computing* [Pau00] – that is also widely referred to as *P–systems* – this model of computation is based on local transformation of trees inspired by the behavior of membranes in biology. The model is naturally suited for parallel computation as the transitions or *computations* operate locally on a tree (i.e. the P–system). With respect to Turing Machines, Membrane Computing is a model of computation trading time complexity for space complexity, such complexity results have first been obtained by G. Păun in [Pău01], where he shows that the famously known NP–complete problem SAT can be solved in linear time using membrane computing (with an unlimited number of processors). From there the study of the complexity of P–system has been an active area of research [Por+11],[Lep+21],[GHI21],[Woo+09].

In the present note we give a new proof of Turing completeness for membrane computing by showing it can simulate combinatory logic. In a first time we define P–systems following [Pau06] as labelled trees where computation corresponds to local rewritings. Then we give simple language able to define a class of object with typical behaviors (like moving outside or inside a membrane), but not capturing the whole possible behaviors. We then introduce combinatory logic and give a simple basis $\{\mathsf{K}, \mathsf{B}, \mathsf{M}, \mathsf{T}\}$ with only one combinator of rank 3, as we know this is a necessity, see [Leg88]. The next part then consist in establishing how P–systems can simulate combinatory logic with only the use of polarity and a restricted use of names. The construction we establish makes use of non elementary rules, and so the next section shows how to simulate the non elementary rules using the elementary ones, to our knowledge no such result exists in the literature and it is a key point if we want to simulate combinatory logic, as we may need to erase, move or duplicate entire subtrees and not just leaves or nodes of the P–system. The results are not straightforward to obtain: since P–systems are highly parallel means of computation some conflict can occur between computations. Namely the reduction is highly undeterministic and we need to control the computation in order to be able to *compose* processes. We observe that we can make a choice: either by giving an *explicit strategy* stating which transition should be preferred when multiple transition can be performed or make it so that the labelling of the membranes contains *more information*, such as the size of the P–system the number of direct sub–membranes, etc . . . . It illustrates a trade off between the determinism of the computation and the (space) complexity of the means of computation (in that case the P–system).

[1]That is, assuming Church–Turing thesis.

1 Membrane computing

We present here the notion of P–system due to G. Păun although we differ from the usual definition one can find, and presents them as labelled trees. In the beginning of this section we recall usual notions from graph theory on trees, and give a lemma of factorization of trees as forest with a common root, then we introduce the notion of *tree representation* and show that two trees are isomorphic if and only if they have the same representation. With these tools we define P–systems as labelled trees, and give their respective notions of representation.

Along this section we are given a set $\mathcal{N}$ of elements called *names* or *membrane names*, a set C of elements called *objects*, and a finite set of polarities $\{+, -, 0\}$ respectively called *negative, positive* and *neutral*.

Definition 1.1 (Binary relation, Graph, Tree)**.** Given a set X a *binary relation* R over X is a subset of X^2. Then we define the following:

- Given x and y two points of X, we denote by xRy the fact that the pair (x, y) belongs to R.
- We define the *transitive closure* of R, denoted R^*, the binary relation such that given any two points x, y of X xR^*y if there exist a sequence of points of X $(x_1, ..., x_n)$ such that, $x_1 = x, x_n = y$ and for any $1 \leq i \leq n$ the relation x_iRx_{i+1} holds.
- A relation R is *symmetric* if for any point x and y if xRy holds then so does yRx.
- A relation R is *antisymmetric* if for any two points x and y if xRy holds then yRx cannot hold.

An *oriented graph (without loops)* is pair (V, E) where V is a set of elements called *nodes* or *vertex*, and E is an antisymmetric binary relation over V. A graph is:

- *connected* if given any two vertices x and y we have xR^*y.
- *acyclic* if given any three vertices x, y, z such that xR^*y and xR^*z there cannot exists a vertice w such that yR^*w and zR^*w.
- A *tree* if it is acyclic and connected. The *root* of a tree is a vertex that is connected to each vertex of the tree.
- Given a (disjoint) finite family of trees $\mathcal{T}_1, ..., \mathcal{T}_n$ of respective roots $r_1, ..., r_n$ and a vertex r not belonging to $\bigcup V_i$. we denote by $\bigvee^r_{1\leq i\leq n} \mathcal{T}_i$ the tree (V, E) where:
  - The set of vertices V is defined as $\bigcup_{1\leq i\leq n} V_i$.
  - The set of edges is defined as $E = \bigcup_{1\leq i\leq n} E_i \cup \{(r, r_i)\}$.
- the *size* of a graph is the cardinal $\mid V \mid$ of its set of vertices V.

Lemma 1.1 (Uniqueness and existence of the root)**.** *Any tree $\mathcal{T}$ has a unique root.*

Proof. Obtaining uniqueness is straightforward, since if two distinct roots occur in the tree $\mathcal{T}$ we can create a cycle, which leads to a contradiction. For the existence reason by induction: if $\mathcal{T}$ has one node then this only node is its root. If $\mathcal{T}$ has $n + 1$ node let us show it has a root consider a leaf v of $\mathcal{T}$ and consider the induced tree $\mathcal{T}'$ by $V \setminus \{v\}$. Obviously this tree is smaller than $\mathcal{T}$ and so we can apply induction, claiming that r is the root of $\mathcal{T}'$. Now v has necessarily a parent node u belonging to $V \setminus \{v\}$, then since r is the root of $\mathcal{T}'$ we know that r is connected to u and since uEv we can ensure that r is connected to v. This shows that r is the root of $\mathcal{T}$. □

Definition 1.2 (Induced subtree by a vertex)**.** Given a tree $\mathcal{T}$ and a vertex v the *induced subtree* by v is the tree $\mathcal{T}_v = (V_v, E_v)$ where,

- V_v is the set of vertices u such that vE^*u.
- E_v is the set of edges in the intersection $E_v = E \cap V^2$.

Given r the root of $\mathcal{T}$ and $x_1, ..., x_n$ the children of r. A *direct subtree* of $\mathcal{T}$ is any of the subtree induced by one of the vertex x_i.

Lemma 1.2 (Tree Factorization)**.** *Given a tree $\mathcal{T}$ denoting its root r, one of the two cases occur:*

- *$\mathcal{T}$ contains one vertex*
- *Given $\mathcal{T}_1, ..., \mathcal{T}_n$ the direct subtree of $\mathcal{T}$ we have $\mathcal{T} = \bigvee^r_{1 \le i \le n} \mathcal{T}_i$.*

Proof. Assume that $\mathcal{T}$ contains more than one vertex. Let us show that the point two holds, given r the root of $\mathcal{T}$ take each subtree induced by the points x_i directly connected to r. Let us denote $\mathcal{T} = (V, E)$ and $\bigvee^r_{1 \le i \le n} \mathcal{T}_i = (V', E')$ first we show the tree's have the same vertices.

- Given a v a vertex of V either it belongs to a direct subtree and so to V' or it is the root r and so it also must belong to V'.
- On the other a vertex v of V' may be the root r and so belong to V. Otherwise v may be vertex of one the tree $\mathcal{T}_i$ but these trees are subtrees of $\mathcal{T}$ therefore also in that case v belongs to V.

Now let us show the edges are the same:

- Consider an edge (u, v) of E if u is the root then the edge is of the form (r, v) and so v is a children of r i.e. $v = x_i$ for some i. Since x_i is the root of $\mathcal{T}_i$, by definition (r, x_i) belongs to E'. If u is not the root, then u and v belong to the same direct subtree (since otherwise we create a cycle) and so (u, v) belongs to E'.
- Consider an edge (u, v) of E' if that edge belongs to one of the tree $\mathcal{T}_i$ then since $\mathcal{T}_i$ is a subtree of $\mathcal{T}$ it is an edge of E. On the other that edge may be of the the form (r, x_i) where x_i is the root of $\mathcal{T}_i$, but indeed since x_i is a children of r this edge belongs to E

□

Definition 1.3 (Tree representation)**.** Given N a set of nodes. A *tree representation* is a word generated over the alphabet $\{[,]\} \cup N$ by the following grammar:

$$t := []_n \mid [t_1, ..., t_n] \text{ where each } t_i \text{ is itself a tree representation.}$$

Definition 1.4 (Canonical Map)**.** We define the canonical map Φ sending finite trees to their representation as follow.

- If the tree contains only one vertex the mapping is $\Phi((\{v\}, \emptyset)) \mapsto []$
- On the other hand if the tree contains more than one vertex the mapping is defined as $\Phi(\bigvee^r_{1 \le i \le n} \mathcal{T}_i) \mapsto [\Phi(\mathcal{T}_1), ..., \Phi(\mathcal{T}_n)]$

Definition 1.5 (Size of a tree and a tree representation)**.** The size of a tree $\mathcal{T}$, denoted $\mid \mathcal{T} \mid$ is defined as the cardinal of V i.e. the number of nodes of the tree.

The size of a tree representation t is defined inductively as follows:

- $|[]| = 1$
- $|[t_1, ..., t_n]| = \Sigma_{1 \le i \le n} |t_i| + 1$ where each t_i is a tree representation.

Proposition 1.1 (Size–preservation)**.** *The surjective map Φ sending a tree to its representation conserve the size. Meaning given any tree $\mathcal{T}$, it stands that $|\mathcal{T}| = |\Phi\mathcal{T}|$ and vice versa.*

Proof. By doing a straightforward induction. □

Proposition 1.2 (Maps to representations)**.** *Given a set of nodes N. Φ is a surjection from the set of tree (V, E) of which the nodes belong to N to the set of tree representation. And, Φ is such that two trees have the same representation if and only if they are isomorph.*

Proof. First we show the surjective property. We perform induction on the size of the representation. A representation of size 1 can only be $[]$ and indeed is the image of any tree made of only one node, i.e. of the form $(\{v\}, \emptyset)$. Now if we take some representation $[t_1, ..., t_n]$, by induction each t_i has a preimage $\mathcal{T}_i$, then $\bigvee^r_{1 \le i \le n} \mathcal{T}_i$ is a preimage of $[t_1, ..., t_n]$.
To show the isomorphic property, we reason by induction on the number of nodes of a tree $\mathcal{T} = (V, E)$. If V contains only one node, then the set of edges E must be empty since a tree does not contain loops.

Therefore $\mathcal{T}$ corresponds to a tree of the form $(\{v\}, \emptyset)$, to which we associate the representation $[]$. Indeed we note that all trees with one node i.e. of the form $(\{v\}, \emptyset)$, are isomorph. The converse is obviously true.

Now by induction assume V is of size $n + 1$. Denoting r the root of V, since V has more than one node calling the lemma we can ensure that $\mathcal{T} = \bigvee^r_{1 \le i \le n} \mathcal{T}_i$ where for each i the subtree $\mathcal{T}_i$ is generated by r_i where $\{r_1, ..., r_n\}$ are the children of root r. Since each $\mathcal{T}_i$ is a proper subtree of $\mathcal{T}$ their size is smaller than the one of $\mathcal{T}$, so by induction we can associate to each $\mathcal{T}_i$ a representation t_i, then we define the representation of $\mathcal{T}$ to be $[t_1, ..., t_n]$.

Assume we are given two isomorph trees $\mathcal{T}$ and $\mathcal{T}'$ denoting their root r and r' since the trees are isomorphic the set of children of r and of r' must be equipotent i.e. they are of the form $\{r_1, ..., r_n\}$ and $\{r'_1, ..., r'_n\}$. Denote the induced subtree by r_i (resp. r'_i) by $\mathcal{T}_i$ (resp. $\mathcal{T}_i'$). Since $\mathcal{T}$ and $\mathcal{T}'$ are isomorphic for each subtree $\mathcal{T}_i$ there is a subtree $\mathcal{T}_j'$ to which it is isomorphic. To make it easier, assume we rearrange the indexes such that each $\mathcal{T}_i$ is isomorphic to $\mathcal{T}_i'$, since each $\mathcal{T}_i$ is smaller that $\mathcal{T}$ we can apply induction and ensure that $\mathcal{T}_i$ and $\mathcal{T}_i'$ are represented by t_i. Hence since we have $\mathcal{T} = \bigvee^r_{1 \le i \le n} \mathcal{T}_i$ and $\mathcal{T}' = \bigvee^r_{1 \le i \le n} \mathcal{T}_i'$ we can ensure that both trees have the same representation that is $[t_1, ..., t_n]$.

The converse is also shown by induction and is straightforward, we assume two trees have the same representation $[t_1, ..., t_n]$ which allows us to ensure that the induced subtree by the children of the root are isomorph and so that the two trees are isomorph. □

Definition 1.6 (P–system)**.** A *P–system* is a tree $T = (V, E)$, together with a labelling function ℓ associating to any node n a triple $(\iota(n), \mu(n), \rho(n))$ where:

- The image $\mu(n)$ is an element of $\mathcal{N}$ and is called the *name* of the node or membrane.
- the image $\iota(n)$ is a multiset of objects i.e. elements of C, it is called the *content* of the membrane n,
- the image $\rho(n)$ is an element of $\{+, -, 0\}$, it is called the *polarity* of the membrane n.

A triple $(\iota(n), \mu(n), \rho(n)) = (\Gamma, h, \alpha)$ will be denoted $\Gamma \subset h : \alpha$.

Definition 1.7. Inductively we can define the representation of P–systems as the following:

$$P = [\Gamma]^\alpha_h \mid [\Gamma, P_1, ..., P_n]^\alpha_h \quad \text{(with each } P_i \text{ being a representation itself).}$$

Where Γ is a multiset of object, h is a name and α is a polarity.

Definition 1.8 (rewriting rules)**.** In the language of P–systems, the *transitions* or *computations* are defined as rewriting on trees. Each rules operates locally and depends on the objects occuring at the membrane. We define the *rewriting rules* or *transitions* as local rewritings of the following form.

- send in communication $[a]^\alpha_h \to []^\beta_i b$ then we denote the object a by $b \cdot \mathsf{out}(\alpha, h \mapsto \beta, h')$.
- send out communication $a[]^\alpha_h \to [b]^\beta_i$, in that case the object a is denoted $b \cdot \mathsf{in}(\alpha, h \mapsto \beta, h')$
- evolution $[a]^\alpha_h \to [b]^\beta_i$ then a is denote $b \cdot \mathsf{ev}(\alpha, h \mapsto \beta, h')$
- dissolution $[a]^\alpha_h \to b.$ then a is denoted $b \cdot \mathsf{diss}(\alpha, h)$
- division rule $[[]^p ... []^p, []^q ... []^q]^\alpha_h \to [[]^{p'}, ..., []^{p'}]^\beta_i [[]^{q'}, ..., []^{q'}]^\gamma_j$
- dissolve non–elementary membrane $a \cdot \mathcal{T} \to b$ then a is denoted $b \cdot \mathsf{kill}(h, \alpha)$.
- dupplication $[a]^\alpha_h \to [b]^\alpha_h [c]^\alpha_h$. a is then denoted as $[b, c] \cdot \mathsf{duppl}(\alpha, h)$
- endocytosys $[a]^\alpha_h []^\beta_j \to [[b]^\alpha_h]^\beta_j$. In that case we will denote a by $b \cdot \mathsf{endo}(\beta, j)$.
- exocytosys $[[a]^\alpha_h]^\beta_j \to [b]^\alpha_h []^\beta_j$. In that case we will denote a by $b \cdot \mathsf{exo}$

We consider the transitions up to a compatible closure for the tree representation, meaning the transitions are conserved by the following rule

$$\frac{[t] \to [t'] \quad \text{each } t_i \text{ is a P–system}}{[t, t_1, ..., t_n] \to [t', t_1, ..., t_n]} \qquad \frac{t \to t' \quad \text{each } t_i \text{ is a P–system}}{[t, t_1, ..., t_n] \to [t', t_1, ..., t_n]}$$

A membrane rewriting rule always transform an object into another object (eventually the empty object ϵ) and simultaneously moves the object or a subtree from one node to another. Given a rewriting rule the object a is called the *agent* of the rule while the generated objects $b_1, ..., b_n$ are called the *products* of the rule.

- Given an object x the *behavior* of x the set of rules in which x is the agent. The behavior of x is denoted $Beh(x)$.
- A *terminal* rule is a rule with no product, i.e. that generates no object.
- An *atomic* object is an object such that its behavior is made only of terminal rules.
- If the agent and product of a rule are the same then the rule is said to be *structural*.
- A behavior is *cyclic* if it contains rules $R_1, ..., R_n$ such that a is the agent of the transition R_1 and the product of transition R_n.
- A *finite* behavior is a non cyclic behavior made of a finite number of rules.

Definition 1.9 (Atomic object)**.** We introduce a language to describe a wide family of object with finite behavior. We call them *atomic objects*.

$$\mathsf{Atom} = \{\mathsf{out}, \mathsf{in}, \mathsf{endo}, \mathsf{exo}, \mathsf{diss}, \mathsf{dupl}, \mathsf{gen}, \mathsf{name}, \mathsf{test}\}$$

- We define the object out with the following behavior $[\mathsf{out}] \longrightarrow []\epsilon$.
- We define the object $\mathsf{in}(h:\alpha)$ with the following behavior $\mathsf{in}(h:\alpha)[]_h^\alpha \longrightarrow [\epsilon]_h^\alpha$.
- We define exo with the following behavior $[[\mathsf{exo}]_h]_i \longrightarrow []_i[\epsilon]_h$.
- we define $\mathsf{endo}(h:\alpha)$ with the behavior $[\mathsf{endo}(h:\alpha)][]_h^\alpha \longrightarrow [[\epsilon]]_h^\alpha$.
- gen has the behavior $[\mathsf{gen}] \longrightarrow [[\epsilon]]$.
- diss has the behavior $[[\mathsf{diss}]] \longrightarrow [\epsilon]$.
- dupl has the behavior $[\mathsf{dupl}] \longrightarrow [\epsilon][\epsilon]$.
- $\{\}_\alpha^h$ has the behavior $[\{\}_\alpha^h]_\alpha^h \longrightarrow [\epsilon]_\alpha^h$.
- $(h:\alpha)$ has the behavior $[(h:\alpha)] \longrightarrow [\epsilon]_\alpha^h$.

On non–terminating behavior. All the behaviors cannot be captured with the syntax of atomic objects we have introduced. Indeed an object with a cyclic behavior such as $a[] \longrightarrow [a]$ and $[a] \longrightarrow a[]$ cannot be written down as a functional term. This is due to the lack of a fixed point operator in the language we introduced. Although for our work, we will only need functional terms to express combinatory logic, and then later on express non–elementary rules by the means of the elementary one.

Definition 1.10. We define the terms by the following induction

$$t, p \quad = \quad t \text{ is an atomic object} \quad | \quad t + p \quad | \quad t \cdot p$$

The behavior of terms is defined inductively thanks to the following rules:

$$\frac{R \in Beh(t)}{R \in Beh(t+p)} \qquad \frac{R \in Beh(p)}{R \in Beh(t+p)} \qquad \frac{R[t \longrightarrow b] \in Beh(t)}{R[p \cdot t \longrightarrow p \cdot b] \in Beh(p \cdot t)} \qquad \frac{R[t \longrightarrow \epsilon] \in Beh(t)}{R[p \cdot t \longrightarrow p] \in Beh(p \cdot t)}$$

Definition 1.11 (Concatenation and sum)**.** Given two atomic objects a and b where $t \longrightarrow t'[\epsilon]$ is the behavior of a. We denote by $b \cdot a$ (read a then b, or b after a) their *concatenation*, that is the object with the behavior $t \longrightarrow t'[b]$.

A concatenation is a term $a_1 \cdot \cdots \cdot a_n$ where each a_i is an atomic object.

A concatenation $s \cdot \{\}_h^\alpha$ is denoted $\{s\}_h^\alpha$.

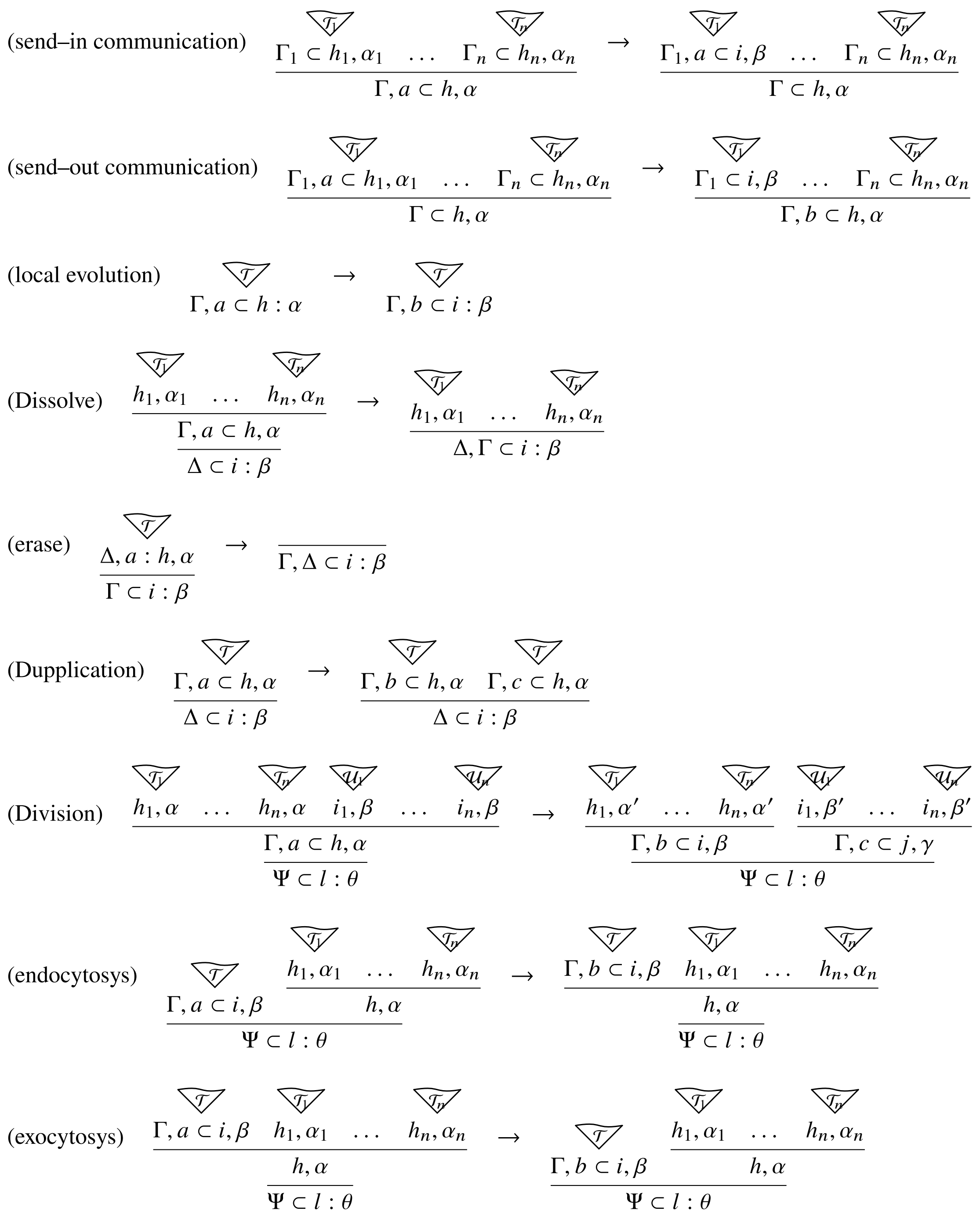


Figure 1: P–systems reductions as local rewriting on trees. The endocytosis, exocytosis duplication and division and erasure are given in their non elementary form. The elementary form is obtained when the subtree $\mathcal{T}$ is of size 1.

2 Combinatory Logic

In this section we exhibit the model of computation due to Schönfinkel of *Combinatory logic*. We give the notion of *basis*, a finite collection of combinators able to express any pure combination of variable (and hence Turing complete). We seek to find a simple base to translate into membrane Computing, while keeping in mind that a basis must contain at least one combinator of rank three (this is result is due to Rémi Legrand [Leg88]). For our purpose, we will consider and define the base $\{\mathsf{B}, \mathsf{M}, \mathsf{T}, \mathsf{K}\}$ Along this section we are given a set denoted $\mathcal{V}$ of elements which are called *variables*, and a set of symbols C called *combinator symbols* or simply *combinators*.

Definition 2.1 (Term, functorial, combinatorial, reductions)**.** A *combinatorial term* (or merely *term*) is any term defined by the following induction:

$$t, u \quad = \quad x \in \mathcal{V} \quad | \quad \mathsf{C} \in C \quad | \quad (t)u.$$

We have the following terminology:

- A *pure combination* is a term in which no combinator occur.
- A *pure combinator* is a term in which no variable occur.
- A *reduction* or *computation* is a rewriting rule of the form $\mathsf{C}x_1, ..., x_n \to F[x_1, ..., x_n]$ which holds for any variable $x_1, ..., x_n$ and where F is a functional term over the variables $\{x_1, ..., x_n\}$. To each combinator we associate a unique reduction.
- A *system of combinator* is a finite set of combinators.
- A system of combinators $\mathcal{S}$ *can express* a combinator $\mathsf{C}x_1, ..., x_n \to E$ if there exists a pure combinator U of $\mathcal{S}$ such that $\mathsf{U}x_1, ..., x_n \to E$.
- A system of combinator $\mathcal{S}$ is a *basis* if for every finite set of variables $\{x_1, \ldots, x_n\}$ and every pure combination E of these variables there exists a pure combinator C of $\mathcal{S}$ such that $Cx_1, \ldots, x_n \to^* E$. Equivalently a basis can express any combinator.

Parenthesis convention in combinatory logic. There is a convention in combinatory logic on how parenthesis are placed in a term, given a finite sequence of variables $(x_i)_{1 \le i \le n}$ the term $x_1 x_2 \ldots x_n$ denotes the term $(\ldots(x_1)x_2)\ldots)x_n)$. In other words we implicitly place the parenthesis on the leftmost subterms of an expression.

Definition 2.2 (Usual combinators)**.** There exists a bunch of usual combinators and their respective reductions, that we present here.

- The identity combinator, $\mathsf{I}x \to x$.
- The erasing combinator, $(\mathsf{K}u)v \to u$.
- The duplication combinators, $\mathsf{W}xy \to xyy$ and $\mathsf{M}x \to xx$.
- The associativity combinator, $\mathsf{B}xyz \to x(yz)$.
- The commutativity combinators, $\mathsf{C}xyz \to xzy$ and $\mathsf{T}xy \to yx$.
- The S combinator, $\mathsf{S}uvw \to (uw)(vw)$.

then the reduction relation – denoted by $\to^*$ – is defined as the compatible closure of these previous relations. If there is no ambiguity we might denote $\to$ for the transitive closure i.e. $\to^*$.

Theorem 2.1 (Canonical Basis for combinatory logic)**.** *The system of combinator* $\{\mathsf{S}, \mathsf{K}\}$ *is a basis.*

Proof. It is a known theorem, its proof can be found in [Ste72]. □

Theorem 2.2 (Turing completeness of combinatory logic)**.** *The system of combinators* $\{\mathsf{S}, \mathsf{K}\}$ *is Turing complete.*

Proof. This is a known proof that is done by showing that combinatory logic is equivalent to lambda calculus (which is known to be turing complete). The proof can be found for example in [Ste72]. □

Basis and Turing Completeness. From this last theorem follows that any basis is Turing complete, since in particular any basis can express S and K. And the previous theorem means it is enough for a system of combinators to be able to express S and K to be a basis, and so putting the two deductions together: to be Turing complete.

Proposition 2.1 (Basis). *The following systems of combinators are basis.*

- $\{\mathsf{C}, \mathsf{W}, \mathsf{B}, \mathsf{K}\}$ *is a basis.*
- $\{\mathsf{B}, \mathsf{M}, \mathsf{T}, \mathsf{K}\}$ *is a basis.*

Proof. To show the first point we merely have to show that the combinator S can be expressed in the system $\{C, W, B, K\}$ to do so we show that $\mathsf{S}_0 = \mathsf{B}(\mathsf{B}(\mathsf{BW})\mathsf{C})(\mathsf{BB})$ has the same normal form than S applied to three variables a, b and c. That means that we have to show that the reduction $\mathsf{S}_0abc \longrightarrow (ac)(bc)$ holds.

$$\begin{aligned}
B(B(BW)C)(BB)abc &\longrightarrow (B(BW)C)[(BB)a]bc \\
&\longrightarrow (BW)(C[(BB)a])bc \\
&\longrightarrow W[(C[(BB)a])b]c \\
&\longrightarrow (C[(BB)a])bcc \\
&\longrightarrow [(BB)a]cbc \\
&\longrightarrow [(BB)a]cbc \\
&\longrightarrow (B)(ac)bc \\
&\longrightarrow (ac)(bc)
\end{aligned}$$

To prove the second point we show that the system of combinators $\{\mathsf{B}, \mathsf{M}, \mathsf{K}, \mathsf{T}\}$ can simulate the system $\{\mathsf{B}, \mathsf{C}, \mathsf{W}, \mathsf{K}\}$. To do so we only have to show that $\{\mathsf{B}, \mathsf{M}, \mathsf{K}, \mathsf{T}\}$ simulates the combinators C and W. We will use multiple times the combinator BBT to obtain the translation, this combinator takes three arguments given in order x, y and z and returns the same sequence in which the first input is now in the last position. More clearly we have the following reduction:

$$\mathsf{BBT}xyz \longrightarrow \mathsf{B}(\mathsf{T}x)yz \longrightarrow (\mathsf{T}x)(yz) \longrightarrow (yz)x \equiv yzx$$

Now we claim that $\mathsf{C} = \mathsf{B}(\mathsf{T}(\mathsf{BBT}))(\mathsf{BBT})$ and $\mathsf{W} = \mathsf{C}(\mathsf{BM}(\mathsf{BBT}))$, for this purpose we will check the computation are the same. This concludes our proof and show that $\{\mathsf{B}, \mathsf{M}, \mathsf{T}, \mathsf{K}\}$ is a basis.

1. $\mathsf{B}(\mathsf{T}(\mathsf{BBT}))(\mathsf{BBT})xyz \to (\mathsf{T}(\mathsf{BBT}))((\mathsf{BBT})x)yz \to (\mathsf{BBT}x)(\mathsf{BBT})yz \to (\mathsf{BBT}y)xz \to xzy$
2. $\mathsf{C}(\mathsf{BM}(\mathsf{BBT}))xy \to (\mathsf{BM}(\mathsf{BBT}))yx \to \mathsf{M}((\mathsf{BBT})y)x \to ((\mathsf{BBT})y)((\mathsf{BBT})y)x \to ((\mathsf{BBT})y)xy \to xyy$

$\square$

Combinators as rewriting on binary trees. Combinators can be seen as rewriting on binary trees of which leaves are labelled by combinators or variables. For instance, the identity combinator (W) and its reduction $\mathsf{W}x \longrightarrow xx$ can be written as in the figure 2.

3 *P*–systems and Combinatory Logic

In this section we show how the computation of the combinators from combinatory logic can be simulated in membrane computing i.e. by the means of *P–systems*. We first define how to translate terms from combinatory logic – or equivalently polarized binary tree – into P–systems: to do so we use the polarity internal to the language of P–systems as presented in [Pau06]. Secondly we construct objects that correspond to each combinator of the base $\{\mathsf{K}, \mathsf{B}, \mathsf{M}, \mathsf{T}\}$ thus obtaining an alternative proof of the Turing

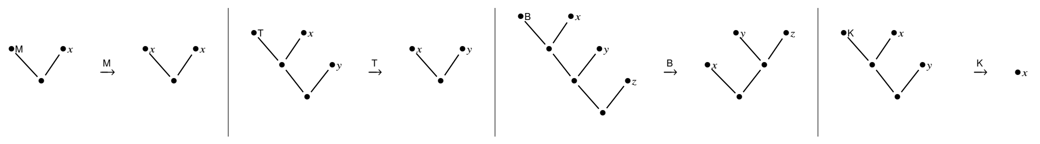


Figure 2: Combinators from the basis {B, M, T, K} as rewriting on labelled binary trees. From left to write *duplication, commutation, associative* and *erase* operators.

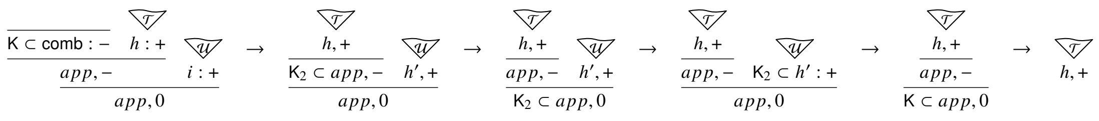
completeness of the P–systems of G. Păun. We consider P systems with no names, and the usual polarity +, − and 0. Then the elementary objects are rule($\alpha \mapsto \beta$). The polarity of a P–system $\mathcal{T}$ is the polarity of its root node. We denote by $\mathcal{T} : \alpha$ the fact that $\mathcal{T}$ is of polarity α.

Definition 3.1 (Representation of combinators as P–systems)**.** We define the representation of terms from combinatory logic as P–systems inductively:

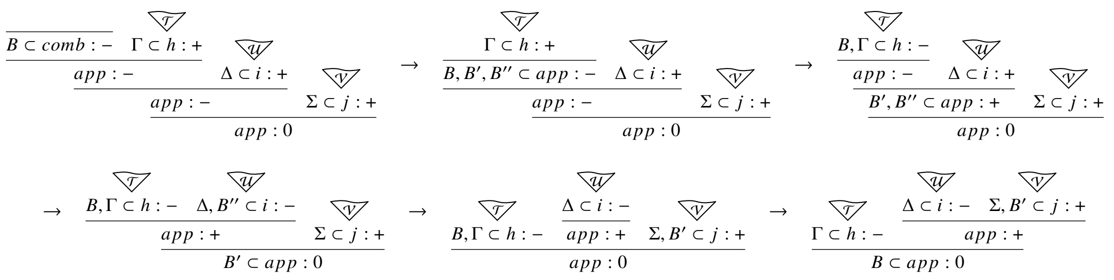
- x corresponds to $[x]^0$.
- S and K corresponds respectively to the elementary membranes $[\mathsf{S}]^0$ and $[\mathsf{K}]^0$.
- $(t)u$ corresponds to $[\mathcal{T}_t : -, \mathcal{T}_u : +]^0$

Theorem 3.1 (Simulation)**.** *Combinatory Logic can be simulated by membrane computing although the reduction are not one–to–one. More precisely this means that given two terms t and u form Combinatory logic such that $t \longrightarrow u$ we have $[\![t]\!] \longrightarrow^* [\![u]\!]$ for their representations as P–system.*

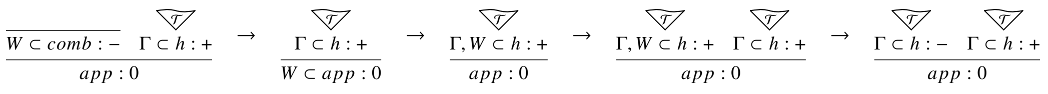
Proof. We present the reduction using tree representation. Here is the first reduction where the erasing combinater K is translated as the object diss · out · in$_+$ · kill · in · diss · diss.



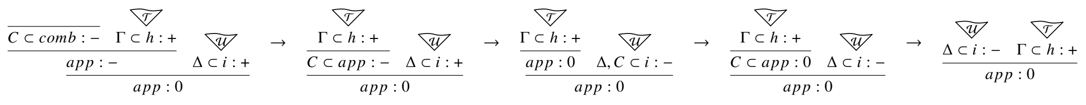
Reduction of the associativity combinator B. We translate the combinator B as $[exo \cdot diss \cdot in, in_(+ \mapsto -) \cdot out, endo_+ \cdot in_+ \cdot out \cdot out] \cdot diss$



In the case of the duplication combinator M we define the corresponding object as the concatenation pol(−) · dupl · in · diss.



Simulation of the commutation combinator T in this case we define the corresponding object with the concatenation diss · in(0) · out · pol(−) · in(+) · out · pol(0) · diss.



□

Non elementary rules. Note that this translation does not use elementary rules only, indeed in some cases to simulate a combinator we may need to *erase or duplicate* an entire subtree, this type of rule are indeed not purely local hence they are usually not considered in regular P–systems. The object of the next section is therefore to show that this type of non–elementary rule can be simulated in polynomial time using the elementary rule.

4 Non–elementary rules simulation in reasonable time

The goal of this section is to show that the non–elementary dissolution and duplication can be simulated by their elementary counterparts. First, we exhibit an object able to erase a subtree of size n in a polynomial number of transition $P(n)$. To do so we must use the notion of names on membranes, we consider P–systems with membranes labelled by the size of the upper–tree they naturally induce (as in the definition 1.2).

Definition 4.1 (Size–labelled P–system). A *size–labelled* P–systems is a P–system constructed by the following induction.

$$t \quad = \quad []_1^\alpha \quad | \quad [t_{1,k_1}, ..., t_{n,k_n}]^\alpha_{\sum k_i + 1}.$$

The set of trees constructed by this induction is denoted by $\mathsf{P_c}$.

Definition 4.2 (Marker membrane). Given a set $\mathcal{M}$ of elements called *marks* the names of membranes may be of the form h where h is an integer or of the form (h, i) where i is a mark belonging to $\mathcal{M}$. A membrane with a name of the form (h, i) is said to be *marked*, and *unmarked* otherwise. If we use only one mark we denote the name of marked membranes by $(h, \bullet)$.

We now introduce and define useful terms for the next results.

- $K_0 = \mathsf{n}(1) \cdot \mathsf{diss}_\emptyset \cdot \mathsf{n}(\varepsilon : 0)$.
- The following transition is valid $\{K_0\}_1 []_h \xrightarrow{in} [\{K_0\}_1, \overline{\mathsf{m}} \cdot \mathsf{out}]_{hF\bullet}$, meaning that $\{K_0\}_1$ can only enter in unmarked membranes and by doing so marks the membrane in which it enters, and unmark the membrane from which it got out. This is to avoid two objects $\{K_0\}_1$ to visit the same membrane
- A membrane with the name ε has the following behavior $[x]_\varepsilon \rightarrow [\epsilon]_\varepsilon$ when x is not $(1) \cdot \mathsf{diss}_\emptyset$.
- The object $\overline{\mathsf{m}}$ unmarks membranes i.e. we have the transition $[\overline{\mathsf{m}}]_{h\bullet} \rightarrow []_h$ and $[\overline{\mathsf{m}}]_h \rightarrow []_h$.

Definition 4.3 (Multiplicative form). Given any object a we define its *multiplicative form* $?a$ with the following behavior $[?a]_h \rightarrow [a^h]_h$

Lemma 4.1 (Cardinal decrementation). *Given some P–system $\mathcal{T}$ and $\mathcal{T}'$ one of its subsystem. If $\{K_0\}_1$ occurs at the root of the subtree $\mathcal{T}'$, then it reduces to itself with a destructed leaf by consuming the object $\{K_0\}_1$ (and so the cardinal of $\mathcal{T}'$ decreases).*

Proof. We do so by induction on the size of $\mathcal{T}'$. Let's first assume that $card(\mathcal{T}') = 1$, in that case indeed the the subtree corresponds to a leaf, therefore K_0 erases it, therefore the cardinal of that subtree is now 0, since now that subtree corresponds to $(\emptyset, \emptyset)$. Assume now that the subtree is of cardinal $n + 1$, and let us show its cardinal decreases:

$$\dfrac{\dfrac{\overset{\mathcal{T}_1}{\Sigma_1 \subset h_1 : \alpha_1} \quad \dots \quad \overset{\mathcal{T}_n}{\Sigma_n \subset h_n : \alpha_n}}{\Gamma, \{K_0\}_0 \subset h : \alpha}}{\Delta \subset h+1 : \beta} \xrightarrow{in} \dfrac{\dfrac{\overset{\mathcal{T}_1}{\Sigma_1, \{K_0\}_0 \subset h_1 : \alpha_1} \quad \dots \quad \overset{\mathcal{T}_n}{\Sigma_n \subset h_n : \alpha_n}}{\Gamma \subset h : \alpha}}{\Delta \subset h+1 : \beta} \xrightarrow{(1)} \dfrac{\dfrac{\overset{\mathcal{T}_1'}{\Sigma_1 \subset h_1 : \alpha_1} \quad \dots \quad \overset{\mathcal{T}_n}{\Sigma_n \subset h_n : \alpha_n}}{\Gamma \subset h : \alpha}}{\Delta \subset h+1 : \beta}$$

The reduction goes as shown above. The first step moves the object K_0 in an inner membrane, say $\mathcal{T}_1$ [2], after this transition K_0 occurs at the root of $\mathcal{T}_1$, and since $\mathcal{T}_1$ is a proper subtree of $\mathcal{T}$ we can ensure that its cardinal is smaller than the one of $\mathcal{T}$ i.e. $n+1$. Hence, we can call the induction hypothesis and claim that $\mathcal{T}_1$ reduces to a tree $\mathcal{T}_1'$ of cardinal $\mid \mathcal{T}_1 \mid -1$ (this is the reduction denoted (1)). Now note that since $\mid \mathcal{T} \mid = \sum \mid \mathcal{T}_i \mid +1 = \mid \mathcal{T}_1 \mid + \sum_{2 \leq i} \mid \mathcal{T}_i \mid$ then $\mid \mathcal{T}' \mid = \mid \mathcal{T} \mid -1$ and so the cardinality of the tree has decreased. □

NB. The last lemma ensures that the cardinality of the subsystem has decreased, and so it comes handy for performing induction.

Proposition 4.1 (Polynomial time simulation of the erase transition using elementary dissolution)**.** *Given P_1 and P_2 two size labelled P–systems from $\mathsf{P_c}$ If $P_1 \longrightarrow P_2$ from a non elementary erase–transition erasing a subtree $\mathcal{T}$ then, if $?\{K_0\}_1$ occurs at the root of $\mathcal{T}$ and $\mathcal{T}$ is correctly size–labelled, it follows that $P_1 \longrightarrow^* P_2$ using a number of dissolutions polynomial in the size of the erased sub–P–system.*

Proof. We assume the erase transition occurs at the root level, this is enough to conclude the general case since the transition in P–systems are compatible with the construction (i.e. the induction steps).

Therefore we assume we want to simulate the following situation, where h and i are integers that correspond to the complexity of the tree. Note that in that case h and i are depend on one another, more precisely $i = h + 1$.

$$\dfrac{\overset{\mathcal{T}}{\Delta, a \subset h : \alpha}}{\Gamma \subset h+1 : \beta} \longrightarrow \dfrac{}{\Gamma \subset 0 : \beta}$$

We show the simulation is possible by induction, assuming that h copies of $\{K_0\}_1$ occur at a tree of size h we want to erase. If $h = 1$ the reduction is the following.

$$\dfrac{\Delta, \{K_0\}_1 \subset 1 : \alpha}{\Gamma \subset 2 : \beta} \xrightarrow{name \cdot test} \dfrac{\Delta, (1) \cdot \mathsf{diss}_{|\emptyset} \subset \varepsilon : \alpha}{\Gamma \subset 2 : \beta} \xrightarrow{\varepsilon} \dfrac{(1) \cdot \mathsf{diss}_{|\emptyset} \subset \varepsilon : \alpha}{\Gamma \subset 2 : \beta} \xrightarrow{\mathsf{diss}} \dfrac{}{\Gamma, (1) \subset 2 : \beta} \xrightarrow{\mathsf{name}} \dfrac{}{\Gamma \subset 1 : \beta}$$

To complete and illustrate our induction we treat the case where $h = 2$ i.e. when the erased tree (or P–system) contains one sub–membrane.

$$\dfrac{\dfrac{\Sigma \subset 1 : \gamma}{\Gamma, \{K_0\}_1, \{K_0\}_1 \subset 2 : \alpha}}{\Delta \subset 3 : \beta} \xrightarrow{in} \dfrac{\dfrac{\Sigma, \{K_0\}_1 \subset 1 : \gamma}{\Gamma, \{K_0\}_1 \subset 2 : \alpha}}{\Delta \subset 3 : \beta} \xrightarrow{\{K_0\}_1} \dfrac{\Gamma, \{K_0\}_1 \subset 1 : \alpha}{\Delta \subset 3 : \beta} \longrightarrow \dfrac{}{\Delta, (1) \subset 3 : \beta} \xrightarrow{\mathsf{name}} \dfrac{}{\Delta \subset 1 : \beta}$$

Now we assume the result to be true for P–system of complexity h and show it still holds for P–systems of complexity strictly lower than h. Note that in this case $h = \sum h_i$.

$$\dfrac{\dfrac{\overset{\mathcal{T}_1}{\Sigma_1 \subset h_1 : \alpha_1} \quad \dots \quad \overset{\mathcal{T}_n}{\Sigma_n \subset h_n : \alpha_n}}{\Gamma, \{K_0\}_0^{h+1} \subset h+1 : \alpha}}{\Delta \subset h+2 : \beta} \xrightarrow{diss} \dfrac{\dfrac{\overset{\mathcal{T}_1}{\Sigma_1, \{K_0\}_0 \subset h_1 : \alpha_1} \quad \dots \quad \overset{\mathcal{T}_n}{\Sigma_n \subset h_n : \alpha_n}}{\Gamma, \{K_0\}_0^{h} \subset h+1 : \alpha}}{\Delta \subset h+2 : \beta} \xrightarrow{(1)} \dfrac{\dfrac{\overset{\mathcal{T}_1'}{\Sigma_1 \subset h_1 : \alpha_1} \quad \dots \quad \overset{\mathcal{T}_n}{\Sigma_n \subset h_n : \alpha_n}}{\Gamma, \{K_0\}_0^{h} \subset h+1 : \alpha}}{\Delta \subset h+2 : \beta}$$

[2] Note that it could be any direct subtree $\mathcal{T}_i$ and the proofs would remain the same.

The reduction (1) is possible using the lemma of cardinal decrementation, and the same lemma ensure that the cardinal has decreased by 1, namely that $| \mathcal{T}_1' |=| \mathcal{T}_1 | -1$. Since $| \mathcal{T}_1' |=| \mathcal{T}_1 | -1$ the size of of the subtree $\mathcal{T}$ has decreased by one, i.e. it size is now of h. This enables us to call the induction hypothesis on the obtained tree after the reduction (1) thus we can ensure that we have the following reduction.

$$\frac{\dfrac{\overset{\overset{\mathcal{T}_1'}{\bigtriangledown}}{\Sigma_1 \subset h_1 : \alpha_1} \quad \dots \quad \overset{\overset{\mathcal{T}_n'}{\bigtriangledown}}{\Sigma_n \subset h_n : \alpha_n}}{\Gamma, \{K_0\}_0^h \subset h+1 : \alpha}}{\Delta \subset h+2 : \beta} \quad \xrightarrow{(IH)} \quad \frac{}{\Delta \subset 1 : \beta}$$

This shows the original subtree of size $h+1$ has been erased, an so completes our inductive proof showing that $\{K_0\}_0$ erases the upper tree at which it occurs.

From this induction we can conclude that if the erased sub–tree $\mathcal{T}$ is correctly size labelled and $?\{K_0\}_1$ occurs at its root then $\mathcal{T}$ gets erased, it follows since then the $?\{K_0\}_1$ generates h copies of $\{K_0\}_1$ and $\mathcal{T}$ being correctly labelled the label of its root i.e. h correspond to its size.

□

Upper bound. We have shown that the erasing object $?\{K_0\}_1$ can erase any subtree when it occurs at it root, but moreover we have a precise upper bound on how many steps are required for that operation to be computed, this computation takes a polynomial time at worst namely $n(n-1)$ where n is the size of the tree. We can ensure this upper bound since an object $?\{K_0\}_1$ fulfills its computation in at most the size of the tree (in the worst case where the tree is a branch).

How to mark the direct sub–membranes. In some cases we might want to mark all the direct sub membrane of a given membrane, to do so we can give a simple object with the following behavior $S[\,]_h \xrightarrow{r_1} [S \cdot \mathsf{out}]_{h^\bullet}$ if h is not $\bullet$–marked. This object visits all the direct sub–membrane and marks them – indeed, by induction we can ensure that such an object terminates its computation. But there is a problem of *compositionality*, namely if we were to mark the direct submembrane and then only perform some other computation given by an other object, say T, we cannot say when S has ended its computation – and so we cannot compose S and T, since we have no way to know when S terminates we cannot know when to call T. Sure we know that S will end its computation but we do not receive the information when S does so, we simply know the computation terminates, nothing more. To remedy this problem we indicate when S starts its computation by introducing another object R such that $[R]_h \longrightarrow [S]_{h^\bullet}$ marking the membrane where we want to compute, and we add an *terminating rule* as $[S]_{h^\bullet} \xrightarrow{r_2} [\epsilon]_h$ to the behavior of S consuming S and unmarking the membrane. Our intention is that the presence of the mark means exactly that the computation is going on, and so that its absence means the termination of the computation. But, this rule cannot be left uncontrolled or S could call the rule r_2 before terminating its goal, i.e. without marking all the direct sub–membrane. Therefore, we need a *preference ordering* on the rules of S indicating which rules S performs first when it could perform several rules. In our case $r_1 < r_2$ meaning that S will first perform r_1 in the case where both rule can be performed, indeed with this constrains we retrieve the information of when S terminates its computation and so *compositionality*.

This is similar to giving a strategy of reduction, if one thinks of lambda calculus a strategy is an arbitrary choice, usually one desires that no preference occurs on reductions *a priori*. In the case of membrane computing we see with that simple example that implementing this preferences is essential if we want to have the compositionality of the processes. Also, we cannot just reject compositionality from a model of computation as the composition of programs is fundamental: for instance, to compute a simple operation like $x = 3 \quad \mathsf{in} \quad 4 \times x + 2$ we compose the multiplication with the addition.

Although using the preferences is not the only way to deal with this problem. A solution consists in marking each membrane with the number of direct sub–membrane it has: in that way we can know when S has terminated. More precisely we would need to parametered objects $(S(k))_{k\in\mathbb{N}}$ and $(T(k))_{k\in\mathbb{N}}$ with the following behavior $S(i)[]_h \to [T(i+1)\cdot\mathsf{out}]_h\bullet$, $[T(i)]_{(h,m)} \to [S(i)]_{(h,m)}$ and $[T(m)]_{(h,m)} \to [\epsilon]_{(h,m)}$. Then the computation would start by placing $S(0)$ in the membrane to which we want to mark its direct sub–membrane, when S has marked m membranes and so $S(0)$ has become $T(m)$ the computation ends and we know that m membranes have been marked since we know that the number of direct sub–membrane is exactly m we know the computation has been terminated.

This example illustrates an important idea, to retrieve the information of termination we can make a choice: either introducing *preferences* that we might want to reject because it introduce strategy or determinism in the computations, or adding more *information* to the membrane names making them more complex. This illustrates a relation between reduction strategies and complexity of the input and objects in membrane computing: we can trade one for the other depending on the context and the goal we have in mind. We can see that a similar phenomenon happens in automata theory when a non deterministic automata is translated into a deterministic automata via the Rabin–Scott powerset construction: the number of states can increase from n up to 2^n. This suggest a link between space complexity of the means of computation, and the amount of determinism of the computation.

For the next simulation we introduce some new objects first $\overline{\mathsf{exo}}$ and $\overline{\mathsf{endo}}$ that perform exocytosis and endocytosis while preserving the size labelling, namely:

$$[[\overline{\mathsf{exo}}]_1]_h \to []_{h-1}[]_i \quad \textit{and} \quad [\overline{\mathsf{endo}}]_1[]_h \to [[\overline{\mathsf{exo}}]_1]_{h+1}$$

We also enrich the name of the membrane so that they can be of the form $h@s$ where s is a sequence of marks.

Proposition 4.2 (Simulation of exocytosisn endocytosis and duplication)**.** *Given $\mathcal{T}$ a subtree of some tree $\mathcal{T}_0$ we can construct an object E that performs exocytosis (resp. endocytosis) when occuring at the root of $\mathcal{T}$.*

Proof. Once again we will reason by induction on the size of the tree on which occurs exocytosis. If the tree $\mathcal{T}$ is of size one an elementary exocytosis suffice.

Assume the tree to be of size $n+1$, consider the object $\mathcal{N}$ which marks each node with the a name of the previous ones. and such that once it reach a leaf the object performs k exocytosis (k is the number of transition to reach the leaf). Once the subtree has been deconstructed we start a second phase called reconstruction which essentially performs all the transitions $[]_{i1}[]^{(i_1,\dots,i_n)} \to [[]^{(i_2,\dots,i_n)}]_{i1}$.
For the deconstruction phase we use a (collection of) objects X_s where s is a sequence of names, with the following behavior

- $X_s(k)[]_h \to [X_{s\cdot h}(k+1)]_{\overline{h}@s}$ and $X_s(k)[]_{h@s} \to [X_{s\cdot h}(k+1)]_{\overline{h}@s}$.
- $[X_s(k)]_{1@s'} \xrightarrow{e1} [\overline{\mathsf{exo}}^k]_{1@s'}$ and $[X_s(k)]_1 \xrightarrow{e2} [\overline{\mathsf{exo}}^k]_1$.

We show the deconstruction phase by induction. To do so we show that the presence of an object $X_0(1)$ at the root of a subtree leads to a decrementation of the size of three.

$$\cfrac{\Gamma_2, X_0(1) \subset 1 : \alpha_2}{\cfrac{\Gamma_1 \subset 2 : \alpha_1}{\Gamma_0 \subset 3 : \alpha_0}} \xrightarrow{e2} \cfrac{\Gamma_2, \overline{\mathsf{exo}} \subset 1 : \alpha_2}{\cfrac{\Gamma_1 \subset 2 : \alpha_1}{\Gamma_0 \subset 3 : \alpha_0}} \xrightarrow{exo} \cfrac{\Gamma_1 \subset 1 : \alpha_1 \quad \Gamma_2 \subset 1 : \alpha_2}{\Gamma_0 \subset 2 : \alpha_0}$$

For the induction step. We show that $X_0(1)$ decrements the size of a tree of size $k+1$ assuming it decreases the size of the trees with a size $l < k+1$.

$$\dfrac{\dfrac{\dfrac{\overset{\mathcal{T}_1}{\Delta_1 \subset (h_1, i_1) : \beta_1} \quad \dots \quad \overset{\mathcal{T}_n}{\Delta_n \subset (h_n, i_n) : \beta_n}}{\Gamma_2, X_0(1) \subset 1 : \alpha_2}}{\Gamma_1 \subset 2 : \alpha_1}}{\Gamma_0 \subset 3 : \alpha_0} \quad \xrightarrow{in} \quad \dfrac{\dfrac{\dfrac{\overset{\mathcal{T}_1}{\Delta_1, X_{i_0}(2) \subset (h_1, i_1)@i_0 : \beta_1} \quad \dots \quad \overset{\mathcal{T}_n}{\Delta_n \subset (h_n, i_n) : \beta_n}}{\Gamma_2, \subset 1 : \alpha_2}}{\Gamma_1 \subset 2 : \alpha_1}}{\Gamma_0 \subset 3 : \alpha_0}$$

$$\xrightarrow{(IH)} \quad \dfrac{\dfrac{\dfrac{\overset{\mathcal{T}_1'}{\Delta_1 \subset (h_1, i_1)@i_0 : \beta_1} \quad \dots \quad \overset{\mathcal{T}_n}{\Delta_n \subset (h_n, i_n) : \beta_n}}{\Gamma_2, \subset 1 : \alpha_2}}{\Gamma_1 \subset 2 : \alpha_1} \qquad \Sigma \subset (1, j) : \gamma}{\Gamma_0 \subset 3 : \alpha_0}$$

With the decrementation result it follows by induction aswell that the presence of h copies of object $X_0(1)$ leads to the deconstruction of the tree.
The reconstruction is done with a collection of objects Y with the following behavior

- $[Y]_{h@\epsilon} \to []_h$
- $[Y]_{h@s \cdot i}[]_i \to [[Y]_{h@s}]_i$

To mark a tree we use the collections $(M_i)_{1 \le i \le n}$, $(T_i)_{1 \le i \le n}$ and $(I_i)_{1 \le i \le n}$.

- $[M_i^k]_h \to [T_{i+1}^0 \cdot out^k]_{h:j}$.
- $I_i^k[]_h \xrightarrow{R_1} [T_i^{k+1}]_h$ and $I_i^k[]_{h:j} \xrightarrow{R_2} [T_i^{k+1}]_{h:j}$. with the ordering $R_1 < R_2$
- $[T_i^k]_h \to [M_i^k]_h$ and $[T_i^k]_{h:j} \to [I_i^k]_{h:j}$ and $[T_{h+1}^0]_h \xrightarrow{em} []_h$.

From these families of object and the associated transformation we can obtain exocytosis and endocytosis to do so we modify some of the transition rules. After marking a tree we must start the process of deconstruction to do so we modify the transition em into $[T_{h+1}^0]_h \xrightarrow{em} [X_0(1)^h]_h$. The rules e_1 and e_2 for X become $[X_s(k)]_{1@s'} \xrightarrow{e1} [Y \cdot F \cdot \overline{\mathsf{exo}}^k]_{1@s'}$ and $[X_s(k)]_1 \xrightarrow{e2} [Y \cdot F \cdot \overline{\mathsf{exo}}^k]_1$ where F is either $\overline{\mathsf{exo}}$ or $\overline{\mathsf{endo}}\bullet$ or a duplicator δ, respectively for exocytosis endocytosis or duplication. Where δ has the following behavior $[Y \cdot \delta]_h \to [Y]_1[Y]_h\circ$ where $h = 1$ or $h = 1@s$, then Y makes the membrane marked by $\circ$ interact only between themselves and the same for the unmarked membrane.

□

Bibliography

[Ste72] Sören Stenlund. “Combinators, Lambda-Terms and Proof Theory”. In: 1972.

[Leg88] Remi Legrand. “A basis result in combinatory logic”. In: *Journal of Symbolic Logic* 53.4 (1988), pp. 1224–1226. DOI: 10.1017/S0022481200028048.

[Pau00] Gheorghe Paun. “Computing with Membranes”. In: *J. Comput. Syst. Sci.* 61.1 (Aug. 2000), pp. 108–143. ISSN: 0022-0000. DOI: 10.1006/jcss.1999.1693. URL: https://doi.org/10.1006/jcss.1999.1693.

[Pău01] Gheorghe Păun. “P Systems with Active Membranes: Attacking NP-Complete Problems”. In: *J. Autom. Lang. Comb.* 6.1 (Jan. 2001), pp. 75–90. ISSN: 1430-189X.

[Pau06] Gheorghe Paun. “Introduction to Membrane Computing”. In: *Applications of Membrane Computing*. 2006.

[Woo+09] Damien Woods et al. “Membrane Dissolution and Division in P”. In: *Unconventional Computation*. Ed. by Cristian S. Calude et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 262–276. ISBN: 978-3-642-03745-0.

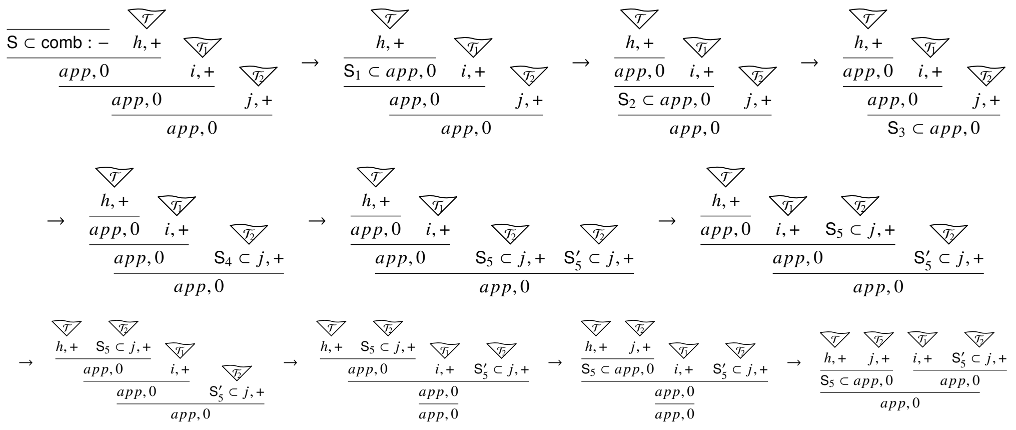
[Por+11] Antonio E. Porreca et al. “P Systems Simulating Oracle Computations”. In: *Int. Conf. on Membrane Computing*. 2011.

[GHI21] Zsolt Gazdag, Károly Hajagos, and Szabolcs Iván. “On the power of P systems with active membranes using weak non-elementary membrane division”. In: *Journal of Membrane Computing* 3.4 (Dec. 2021), pp. 258–269. ISSN: 2523-8914. DOI: `10.1007/s41965-021-00082-2`. URL: `https://doi.org/10.1007/s41965-021-00082-2`.

[Lep+21] Alberto Leporati et al. “Depth-two P systems can simulate Turing machines with NP oracles”. In: *Theoretical Computer Science* (2021). ISSN: 0304-3975. DOI: `https://doi.org/10.1016/j.tcs.2021.11.010`. URL: `https://www.sciencedirect.com/science/article/pii/S030439752100671X`.

Simulating the S–combinator with membrane computing

Reduction for the S combinator interpreted as the catalyst $\mathsf{diss} \cdot \mathsf{out} \cdot \mathsf{out} \cdot \mathsf{in}^{+} \cdot \mathsf{duppl}(\mathsf{C}, \mathsf{D})$. where $\mathsf{C} = \mathsf{endo}_0 \cdot \mathsf{endo}_0 \cdot \mathsf{out} \cdot \mathsf{exo}$ and $\mathsf{D} = \mathsf{endo}_0$



S ⊂ comb : −
h, +
i, +
j, +
app, 0
S₁ ⊂ app, 0
S₂ ⊂ app, 0
S₃ ⊂ app, 0
S₄ ⊂ j, +
S₅ ⊂ j, +
S′₅ ⊂ j, +
S₅ ⊂ app, 0