About P systems and λ-calculus

Loïc COLSON[1], Nataša JONOSKA[2],
Maurice MARGENSTERN[1], Gheorghe PĂUN[3]

[1] LITA, EA 3097, Université de Metz, UFR MIM,
Île du Saulcy, 57045 METZ, Cédex, FRANCE,
E–mail:{colson,margens}@sciences.univ-metz.fr

[2] Department of Mathematics, University of South-Florida,
4202 E. Fowler Ave., PHY 114,
TAMPA, FL 33620-5700, USA
E–mail: jonoska@math.usf.edu

[3] Institute of Mathematics of the Romanian Academy,
PO Box 1-764, 014700 BUCUREŞTI, ROMANIA,
and
Research Group on Natural Computing,
Department of Computer Science and Artificial Intelligence,
Universitad de Sevilla,
Avda. Reina Mercedes s/n, 41012, SEVILLA, SPAIN
E–mail:gpaun@us.es

Abstract

In this short extended abstract, we remind the implementation of tree operations in P systems and how we used this technique in [5] in order to implement pure λ-calculus. In our conclusion, we indicate the guidelines to implement Gödel's T-system which provides us with a family of P systems each one implementing a family of total recursive functions. The union of the implemented functions coincide with the set of provably total recursive functions in Peano arithmetic.

1 Introduction

P systems are now well known as a distributed parallel computing paradigm. In these systems, multisets of symbol-objects are processed in the compartments defined by a predetermined hierarchy of membranes. These objects evolve by means of rewriting-like rules which are applied in a maximally parallel (all objects which can evolve, do evolve) nondeterministic (the objects to process and the rules to apply are chosen in a random way) manner. For a detailed exposition on P systems we refer the reader to the book [9].

Initially the membrane structure of a P system was fixed such that only the evolution of the objects was considered as essential part of the computation. However,

soon after, it became clear that the active role of the membrane structure carries a very powerful benefit as well as it is naturally occurring. The first observation about the power of the active membranes was introduced in [8] where it was shown that if membrane division and membrane dissolution is allowed in the system, then **NP**-complete problems can be solved in polynomial time. Many authors improved on this result by considering different variants of P systems with active membranes (see for example [11, 13, 7]). However, in each of these cases the evolving membrane structure is used to facilitate the computation and the structure itself is not considered an essential part of the final product. In [3] authors define P systems that generate rectangular grids of membranes (that are not nested) such that the objects inside membranes represent rectangular picture languages. Labels for membranes appear from the initial definitions but most of the time their rôle is simply to differentiate the membranes and to locate them.

In [5], a new type of P systems was introduced where the membrane structure (i.e. its tree structure) is an essential part of the computation and the objects are used as facilitators. This system was called λP in [5]. This is a new type of P systems where the membranes are used in an active way such that they are treated as objects, and the objects inside the membranes are treated as catalysts for the reactions. The P systems with active membranes use the creation of membranes, however λP systems go further: they not only create membranes, but they allow membranes to be inserted into other membranes, possibly in the innermost ones in the tree of the whole membrane structure. This definition brings the ideas from [2, 12] together and brings their concepts to a new level. In [2] authors model mechanosensitive channels of bacterial cells with P systems. Although microbiology and cell biology were the main motivation for P systems, this paper is among the first ones to establish a more closer relationship between these two subjects such that the model fits some of the experimental data available. The P systems in [2] are mainly concerned with modeling of the molecular transfer in mechanosensitive channels as well as the conditions that arise to this transfer. This implies that under different environmental conditions the cell membrane changes its gates between closed, partially opened and fully opened. In this paper, the membranes in λP systems have labels (the labeling is not injective) and according to these labels the membranes act in different ways after subjected to the set of general rules. Hence, the labels can be considered in some sense as "different states" of the membrane channels such that in some cases they can be considered as "open to objects only", in some cases are "open for other membranes" and in some cases are "closed". The rules do not allow relabeling of the membranes, but relabeling is intrinsically included by dissolving and then creating new membranes.

The authors in [12] simulate local electronic computer network by P systems. The network is simulated by the membrane structure and the membrane address is obtained by the presence of certain objects. Hence, the membranes are not labeled, but the structure of the membrane is of essential importance. In [5], the computation in λP systems uses labeled membranes and the final configuration is the result of the computation.

Also in [5], the λP systems are illustrated through a system that simulates β-reductions in pure λ-calculus expressions. As λ-calculus has Turing-complete power

of computation, the authors of [5] obtain that λP systems have the same equivalent computational power. They also show that 3SAT and hence other **NP**-complete problems can be solved in polynomial time by λP systems (though in the process of computation, there is an exponential expansion of the number of membranes involved).

In this paper, we go one step further and one step back. The step further is the simulation of **typed** λ-calculus. This amounts to inforce the rôle of the membrane labelling. Here, labels have a complex structure as they encode **types** and they control the computation in a more strong way than in [5]. The result is the step back: we loose Turing-complete power of computation. But, modelling a special kind of typed λ-calculus, namely Gödel's T system, we get functions whose computation always complete, they are **total** recursive functions, and we get all functions which we would like to keep. In fact, the set of functions computed by Gödel's T system is the set of all recursive functions which are provably total in Peano arithmetic.

This will be obtained by a hierarchy of functions, each family of P systems which we shall consider being attached to a fixed **recursor**, see below for the definition of this notion which informally, is a restricted induction principle.

We have no room in this extended abstract to give details of the implementation in P systems. We shall just give the guidelines of this implementation.

2 λP systems

We refer the reader to [9, 10] for an introduction to P systems and we assume some general knowledge for this computational model. All membranes in a λP system have a label from a predetermined alphabet of labels Λ. The set of objects that can appear in the membranes is denoted with $\mathcal{O}$. The membrane structure is represented as a tree with skin being the root. If a membrane with label α is an inner membrane of β such that there is no inner membrane of β that contains α, then α is an immediate child of β. We write $C(\beta) = \{\alpha_1, \ldots, \alpha_k\}$ to represent that all immediate children of β are $\alpha_1, \ldots, \alpha_k$.

A membrane with label α is denoted with $[^{\alpha}]$ and the label is denoted with α . The circle around α is used to indicate that α is a label of a membrane visible from outside. In a membrane, the labels of all immediate inner membranes (immediate children) are visible (detectable). All rules can use the objects currently in the membrane and have contextual dependence on the labels of the inner membranes. Three basic types of operations with membranes (tree structure) are performed.

$$(a) \qquad \alpha \to [^{\alpha}]_{in[^{\beta}]}, \qquad (\alpha \to [^{\alpha}]_{out[^{\beta}]})$$

$$(b) \qquad \alpha \to [^{\alpha}]_{in^*[^{\beta}]}, \qquad (\alpha \to [^{\alpha}]_{out^*[^{\beta}]})$$

$$(c) \qquad \alpha \to [^{\beta}[^{\alpha}]]$$

The first operation takes a membrane and all its structure and includes it in a membrane at the same level. This tree operation is performed such that if α and β

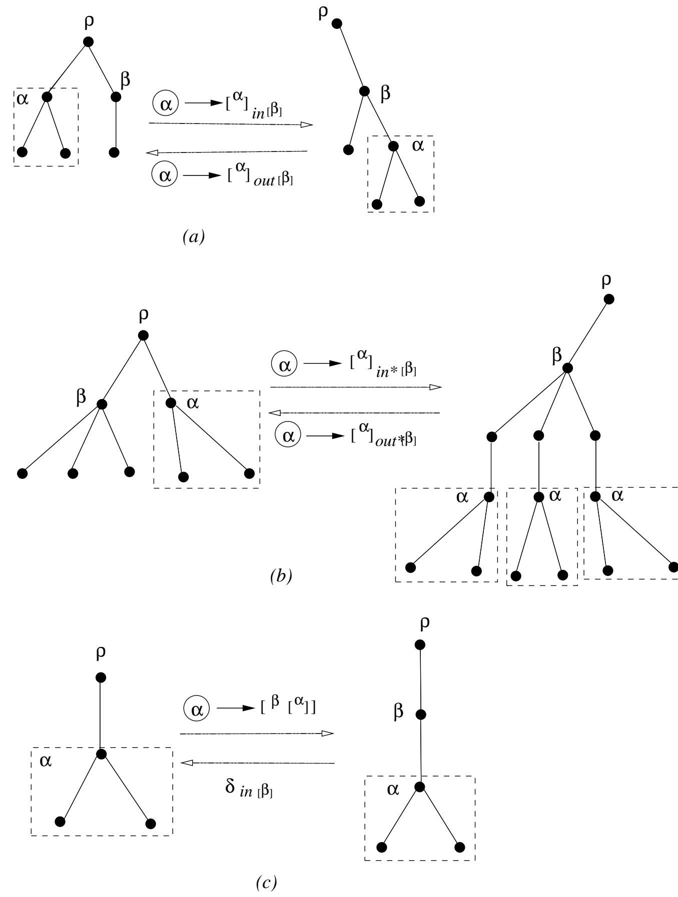

Figure 1: The change of the tree structure by the operations. The subtree that is moved by the operation is included in a dashed box.

are children of ρ, i.e. $\alpha, \beta \in C(\rho)$, then α and all its subtree is removed as a child from ρ and is added to $C(\beta)$ as a child to β (see Figure 1 (a)).

The operation (b) takes the membrane α with all its structure and includes this structure inside the most inner membranes of β (see Figure 1 (b)). As a tree operation, if α and β are children of ρ i.e. $\alpha, \beta \in C(\rho)$, then the operation removes α and its subtree as a child of ρ and adds it to the leaves of β. If the target β is not indicated, then membrane α becomes a subtree of all leaves of its parent ρ. Both of these operations have a variation (in parenthesis) which gives opposite operations to the trees. When the membrane α and its subtree is not removed, it is indicated by the appearance of $[^{\alpha}]$ on the right hand side of the rule.

The third type of rules (c) surrounds the membrane α with a membrane β. The tree structure in this case changes such that if α is a child of ρ, then after the operation of type (c), ρ has a child β which has a child α (see Figure 1 (c)). The reverse of the process (removing β as a child from $C(\rho)$) is obtained by introducing a special symbol δ in membrane β. The presence of this symbol removes the current membrane, but does not change the rest of the structure of that membrane. We

assume that the set of objects (denoted with $\mathcal{O}$) always contains this symbol δ. Another special symbol that removes a membrane together with all its substructure (i.e. removes a whole subtree in the tree structure) is δ^*. We assume that $\delta^* \in \mathcal{O}$.

In addition to the standard targets of objects in P systems $\{in, out, stay\}$ we add two additional targets $\{in^*, out^*\}$. These operations are similar to the ones for the membranes. The target in^* denotes that the object is sent to the innermost membranes of the substructure of the membrane where it currently belongs, and the target out^* indicates that the object is sent to the skin. The standard target $in_{\#}$ where $\#$ is the number of the membrane is substituted with $in_{[^\alpha]}$ where $[^\alpha]$ is a membrane labeled α. There may be several such membranes, and in that case the object is sent to all membranes with the indicated label.

Definition 2.1 *A* **λP rule** *is a rule of the form* $X_1, \ldots, X_s \rightarrow Y^1_{tar_{Z_1}}, \ldots, Y^t_{tar_{Z_t}}$ *where*

- $X_i \in \mathcal{O}$, *or* $X_i = Z$, *for* $Z \in \Lambda$,
- $Y^j \in \mathcal{O}$, *or* $Y^j = [^Z]$, *or* $Y^j = [^Z[^U]]$, *for* $Z, U \in \Lambda$,
- $tar \in \{in, in^*, out, out^*, stay\}$, $Z_i \in \Lambda$.

From now on, in order to simplify the notation, the target *stay* will be omitted.

Definition 2.2 *A* **λP system** *is a construct*

$$\Pi = (\Lambda \cup \mathcal{O}, \mu(V), R, f)$$

where Λ *is an alphabet of the labels of the membranes,* $\mathcal{O}$ *is the alphabet of objects,* μ *is the initial tree of a membrane structure with nodes* V, R *is the set of* λP *rules and* $f : V \rightarrow \Lambda$.

2.0.1 Execution of the rules

The set of rules is fixed for the whole system. The presence of objects and labels of membranes trigger the evolution. All rules that can be applied are applied (note that the rules may depend on the labels of the inner membranes and as such are contextual). Objects in the membranes are acting as catalysts, that is, the presence of an object within a context of a given membrane label, triggers all membranes with that label to take part in the application of the rule. As is standard for P systems, all objects that take part in the operation (i.e. are on the left hand side of the rule) and do not appear on the right hand side of the same rule, are considered to be destroyed in the process. A weak priority is imposed such that all rules that do not depend on the context of membrane labels are applied first.

2.0.2 Computation

The λP system stops evolving when none of the rules can be applied. The result of the computation is the final membrane configuration. A formal definition follows. As in the standard definition of a configuration of a P system the membrane tree

structure is indicated with square brackets (see [9, 10]). Let μ be the initial tree representing a membrane structure of a λP system Π. The **initial configuration** of Π is denoted with $\mathcal{C}_0 = [^S \mathcal{O}_S [^{Y_1} \mathcal{O}_{Y_1}] \cdots [^{Y_k} \mathcal{O}_{Y_k}]]$ where $[^S]$ is the skin (i.e. the root of μ) and $[^{Y_i}]$ are membrane structures corresponding to the subtrees of μ with roots in the set of immediate children $C(S)$ of the root S $(i = 1, ..., k)$. The sets $\mathcal{O}_S, \mathcal{O}_{Y_i}$ denote the set of objects in each of the membranes. We write

$$\mathcal{C}_i \to_R \mathcal{C}_{i+1}$$

if by applying all λP rules from R in parallel as described above one changes the configuration $\mathcal{C}_i$ into $\mathcal{C}_{i+1}$. A **computation of the λP system** Π is a sequence: $\mathcal{C}(\Pi) = \mathcal{C}_0, \mathcal{C}_1, \ldots$ such that $\mathcal{C}_i \to_R \mathcal{C}_{i+1}$ for all i. The computation is finite if there is j such that $\mathcal{C}_j = \mathcal{C}_{j+1}$. In this case we say that $\mathcal{C}_j$ is a **final configuration**. We say that the *result* of the λP system is the final configuration. The result for Π is denoted with $\mathcal{F}(\Pi)$.

In the sections that follow we concentrate on specific λP systems which can be used to simulate λ-calculus in parallel and so 3SAT can be solved in linear time with linear input but exponential increase of membranes. The fact that any general λ-calculus expression can be represented and the β-reduction can be simulated shows that λP systems are computationally universal.

3 Simulating pure λ-calculus

Pure λ-calculus was created by Alonso Church in the early thirties of the last century. We refer the reader to [1, 6] for an introduction and for references. However, in order to make the paper self-contained, we give a very short introduction to λ-calculus.

3.1 Pure λ-calculus

This formal system consists in a countable number of **symbols**. Three of them, 'λ', '(' and ')' can be seen as punctuation symbols while the remaining ones are called **variables** which are usually denoted by individual letters with possible indices: a, b, x, y_1, ..., y_k.

λ-terms are defined by induction, as follows:

Definition 3.1 *(i) a variable x is a λ-term being denoted by x; x is* **free** *in this term;*

(ii) if M is a λ-term and if x is a variable, $\lambda x\, M$ is a λ-term, and all occurrences of x are **bounded** *in $\lambda x\, M$; we say that these occurrences of x are* **controlled** *by this λ and that M is the* **scope** *of this λ; $\lambda x\, M$ is called the* **abstraction** *of M with respect to x;*

(iii) if M and N are λ-terms, then $(M)\,N$ is a λ-term; this term is called the **application** *of M to N; the free occurrences of x in M and N are still free in $(M)\,N$.*

This notation of λ-terms is taken from [6]. It has the property that it needs exactly the symbols which we indicated to define the λ-terms. We note that our definition is a variation of the traditional notation $(M\ N)$ for the application that makes use of an additional symbol: the blank. This symbol is used as a separator in the application.

A **sub-term** U of a λ-term M is a λ-term which, as a word, is a factor of M.

The notion of computation for λ-terms is defined through β-reductions. We say that a λ-term of the form $(\lambda\, x\ M)\, N$ is a **redex** and that its **reduction** is the term $M\,[x := N]$ which is obtained from M by replacing all free occurrences of x in M by N. This restriction is due to the following natural requirement: if N contains a variable which is free before the reduction, the variable must remain free after it. We denote the reduction by $(\lambda\, x\ M)\, N \Rightarrow M\,[x := N]$.

When a λ-term contains no redex, we say that it is in **normal form**. The goal of a computation is to transform a λ-term into a λ-term which is in normal form and, by definition, this normal form is called the result of the computation. In order to do so, we apply successively the β-reduction until we arrive to a term without a redex.

For a λ-term M, we denote with $\mathcal{N}(M)$ the normal form obtained by performing β-reductions to M.

Example 3.2 Consider the following λ-term:

$$((\lambda\, x\ \lambda\, y\ (x)\, y)\ a)\ b.$$

This term represents the application of $\lambda\, x\ \lambda\, y\ (x)\, y$ to two arguments: first a, then b. Notice that in pure λ-calculus, $\lambda\, x\ \lambda\, y\ (x)\, y$ is defined as the representation of number 1, see [1, 6]. The reduction goes as follows:

$$((\lambda\, x\ \lambda\, y\ (x)\, y)\ a)\ b \Rightarrow (\lambda\, y\ (a)\, y)\ b \Rightarrow (a)\, b$$

In this computation in two steps, there is no choice in the order of reduction of the redexes. In general, this may not be the case. Assume that a is an abstract term, *e.g.* $\lambda\, zP$. Then, we have

$$((\lambda\, x\ \lambda\, y\ (x)\, y)\ \lambda\, z\ P)\ b \Rightarrow (\lambda\, y\ (\lambda\, z\ P)\, y)\ b$$

and, at this point, there are two redexes at the same stage of the computation (λy-term and λz-subterm) and so there are two possible paths for the continuation of the computation.

Note that a term may have several subterms which are redexes and these redexes, as in the above example, can be nested.

The rules of λ-calculus do not fix the choice and the computation can be executed by any order, even in parallel if it is possible when the redexes are independent. Fortunately, the Church-Rosser theorem (see [1] for instance) indicates that this is not a problem. The theorem says that any path of the computation which leads to a normal term leads to the same normal term. In this case, this normal term is called the **result** of the computation. A path of computations which leads to the result is called a **terminating path**.

Continuing Example 3.2, if we reduce the leftmost redex first, we have:

$$((\lambda x\, \lambda y\, (x)\, y)\, \lambda z\, P)\, b \Rightarrow (\lambda y\, (\lambda z\, P)\, y)\, b \Rightarrow (\lambda z\, P)\, b \Rightarrow P[z := b]$$

Otherwise, by reducing the λz-subterm first, we have:

$$((\lambda x\, \lambda y\, (x)\, y)\, \lambda z\, P)\, b \Rightarrow (\lambda y\, (\lambda z\, P)\, y)\, b \Rightarrow (\lambda y\, P[z := y])\, b$$
$$\Rightarrow P[z := b]$$

as we assumed that y does not appear in P and is free in the λ-term $(\lambda z\, P)\, y$.

What we did on this example can be generalised to a class of λ-terms which have a rather simple structure and which we call **(sub)linear** λ-terms. Their definition is recalled in [5] and, in this paper, advantage is taken of the facility of computation which is attached to this sturcture to prove that $3SAT$ can be simulated in linear time by λP systems.

For more complex terms, it may be a real question whether the computation terminates. The theorem of Church-Rosser says that if there are terminating paths they all lead to the same result. But in general, a terminating path may not exist. If there is a terminating path, by reducing the leftmost redex in a λ-term, the sequence of reductions leads to the normal form (see for example [6] and the theorem of the **leftmost reduction strategy**).

Pure λ-calculus is computationally equivalent to the Turing machines, see for instance [1]. The simulation of a general λ-term is described in the nex section.

4 The simulation of general terms; the system Π''

When the terms contain several occurrences of the same variable, it may happen (and, indeed, it does in many cases), that the reduction leads to contradictions if variables are not renamed in different copies of the same λ-term.

This problem is usually solved by the introduction of an equivalence relation on λ-terms, called α-conversion [1, 6]. The variables of a λ-term are called **separated** when the occurrences of a given variable are all free or all bounded. The two λ-terms U and V are said α-equivalent if and only if there is a bijection φ from the variables of U onto the variables of V such that replacing each occurrence of a variable x of U in U by $\varphi(x)$ we get V. Performing such a bijection is called **renaming** the variables.

The renaming the variables during the computation is an essential feature of pure λ -calculus, see for example [1, 6]. Now, we address this problem, reproducing the solution given in [5].

In compilers of λ-calculus, several solutions are used in order to perform an automatic renaming of the variables after the execution of a reduction. Almost all these solutions use either names which are *strings* over a finite set of symbols or use an infinite alphabet. In our setting of λP systems the alphabets $\mathcal{O}$ and Λ are finite and the objects are not considered as strings. However, the solution of [5] consists in implementing a 'primitive' renaming by appending numbers, i.e. 'indices' to variables which indicate when the renaming is to be performed. Numbers are not used in the general description of λP systems but objects and the unary representation of numbers as a way to implement indices may be used.

We see that this can be performed by the following mechanism of [5].

When a redex $(\lambda\, x\; M)\, N$ is reduced, we *freeze* the argument N in order to prevent later substitutions of the variables. During the further steps of the computation, the variables of N remain frozen, as long as N does not come to the functional position inside a parenthesis. We solve this by putting a 'freezing membrane' labelled by F around the term to be frozen. Then, in context of F to all variable labeled membranes a 'red light' r is added, which is a new symbol in $\mathcal{O}$. This r may accumlate r's that are already present in the membrane. When this is performed, the freezing membrane F is not needed any more and so, it is dissolved. When N comes in a functional position, then a signal is sent to all variable labeled membranes contained in N which erases exactly one r in these membranes. The number of r's which are present in the variable labeled membranes is reduced by one at each unfreezing step and is treated as an index and a pointer of parentheses.

It is not difficult to check that this mechanism simulates a correct renaming and so, if mixed with the protection mechanism of the λP system of type Π, we obtain an exact simulation of the lefmost reduction strategy of a general λ-term. As this was performed in [5], we can extend λP system in order to obtain a parallel reduction to several terms with several arguments.

4.1 Representing general λ-terms with membranes

4.1.1 Alphabets for the λP system Π''

As before we assume that X_Φ is the set of variables that appear in the λ-term Φ. The set of objects for the λP system Π'' is

$$\mathcal{O} = \{\delta, \delta^*\} \cup \{B, d, d', \eta\nu, \sigma, \eta', \nu', r, g\} \cup \{s_x, c_x \mid x \in X_\Phi\}.$$

The set of membrane labels is the following:

$$\Lambda = \{(), P, F\} \cup \{x, \lambda_x \mid x \in X_\Phi\}.$$

Note that there is only one additional membrane label which is F, that we use to create the membrane which sends the 'red light' r signal.

4.1.2 Initial configurations for Π'

This new setting requires an adjusted definition of the initial membrane representation of a λ-term. In order to keep the same representation pattern of membrane structures corresponding to all λ-terms from initial to the final normal form, we admit both 'red light' r's and/or 'green light' g's in the variable labeled membranes. This comes from the observation that in a λ-term M the appearance of x may be bounded and unbounded. Accordingly, we have the following definition.

<u>**Definition**</u> **4.1** *For each λ-term α we define a membrane structure μ_α inductively,*

(i) For a variable x we have the membrane $[^x s_x]$.

(ii) If M is a λ-term and is represented by a membrane system $M_{[]}$, then the λ-term $\lambda_x M$ is represented by the membrane system $[^{\lambda_x} \bar{M}_{[]}]$ such that $\bar{M}_{[]}$ is

obtained from $M_{[]}$ by adding the object g in all membranes $[^x]$ representing a free appearance of x in M and by adding the object r in all membranes $[^x]$ representing a bounded appearance of x in M.

(iii) If M is represented by a membrane system $M_{[]}$ and N with a membrane system $N_{[]}$, then the λ-term $(M)N$ is represented by the membrane system $[^{()} M_{[]}[^P N_{[]}]]$.

The membrane structure μ_α that corresponds to the λ-term α *with membrane representation $\alpha_{[]}$ is* $\mu_\alpha = \left[\alpha_{[]}\right]$.

Notation. When $\mathcal{A}$ is a set of λ-terms, we write $\mathcal{A}_{[]}$ for the set of membrane representations of $\alpha_{[]}$ for $\alpha \in \mathcal{A}$, and we write $\mu_{\mathcal{A}} = \left[\mathcal{A}_{[]}\right]$.

4.1.3 Rules for the λP system Π''

The rules for this system are essentially the same as those for Π' except that we need to adjust the rules 4 and 5 to reflect the 'freezing' procedure and the use of 'red' and 'green' light. The reduction of one 'red' light when 'unfreezing' appears is simulated with rules 8.1–8.3 which are executed with priority since they have no contextual dependance from membrane labels.

rule 1. $B, () \to B_{\text{in}\,[()]}$
(begin reduction)

rule 2.1. $B, \lambda_x, P \;\to \eta$
(prepare the separation of the arguments)

rule 2.2. $\eta, P \;\to [^{()}\,[^P]], \nu$
(include each argument in an additional membrane $[^{()}])$

rule 2.3. $\nu, \lambda x \;\to [^{\lambda x}]_{\text{in}\,[()]}, \sigma_{\text{in}\,[()]}, \delta, \qquad (x \in X_\Phi)$
(include all functional terms into the membrane with the arguments, destroy the current membrane, send the starting symbol σ to each of the membranes with arguments)

rule 2.4.$\sigma, \lambda x\;,\; P \;\to d, c_{x\text{in}^*_{[\lambda x]}}, [^P]_{\text{in}^*\,[\lambda x]}, \delta_{\text{in}\,[\lambda x]}, \eta'_{\text{in}^*_{[\lambda x]}} \qquad (x \in X_\Phi)$
(for each function in the membrane of a given argument, prepare to destroy the current membrane, search for the current variable into all most-inner membranes, send the argument in the innermost membranes, send a trigger for one r to change in g in each innermost membrane of λx, destroy λ_x)

rule 3. $d,\; P \;\to \delta, \delta^*_{\text{in}\,[P]}$
(destroy the argument, destroy the current membrane $[^{()}])$

rule 4.1 $P\;, g, s_y, c_x \;\to s_y, g, B_{\text{out}^*}, \delta^*_{\text{in}_{[P]}} \qquad (x, y \in X_\Phi, x \neq y)$
(if the right variable is not found, destroy the argument, and send the beginning symbol B to the skin)

rule 4.2 $P\;, r, s_y, c_x \;\to s_y, r, B_{\text{out}^*}, \delta^*_{\text{in}_{[P]}} \qquad (x, y \in X_\Phi, x = y$ or $x \neq y)$

(whatever the variable is, if there is a 'red light' signal, destroy the argument, and send the beginning symbol B to the skin)

rule 5.1 $P\ s_x, c_x, g \ \rightarrow [^F [^P]], g \qquad (x \in X_\Phi)$

(if there is a 'green light' and if the variable is found, create the membrane inside which further substitutions must be freezed)

rule 5.2 $F\ , g \ \ \rightarrow \delta, r_{\text{in}^*[^F]}, d'_{\text{in}_{[^F]}}$

(when F is present, send inside all variable labeled membranes, i.e. innermost membranes of $[^F]$, an additional 'red light' in order to stop further substitution, destroy the current variable membrane and prepare to destroy the protection of the argument)

rule 5.3 $d',\ P \ \ \rightarrow \delta_{\text{in}[^P]}, \delta$

(destroy the protection and destroy the current membrane $[^F]$)

rule 6. $B, B \ \ \rightarrow B$

rule 7. $\nu, \nu \ \ \rightarrow \nu$

(both rules 6 and 7 are executed with priority)

rule 8.1 $\eta', r \ \ \rightarrow \nu'$

rule 8.2 $s_x, \nu' \ \ \rightarrow g, s_x$

rule 8.3 $r, g \ \ \rightarrow r$

(rules 8.1 and 8.3 are executed with priority; rule 8.1 kills one r; rule 8.2 generates one g, the 'green light'; if there is still at least one r, rule 8.3 kills the g which is produced by rule 8.2; rule 8.3 is used with priority such that if there are no more r's, then the g being produced by rule 8.2 remains, which means that the substitution is allowed).

As before, we denote with $\Pi''(\mathcal{A})$ the λP system of type Π'' that has an initial configuration $\mu_{\mathcal{A}}$ for a set of λ-terms $\mathcal{A}$. We see that the computational power of λP systems of type Π'' is equivalent to the computational power of a universal Turing machine. We have the following.

Theorem 4.2 *Let α be a general λ-term. Then $\mathcal{F}(\Pi''(\alpha)) = \mu_{\mathcal{N}(\alpha)}$, i.e. the λP system $\Pi''(\alpha)$ simulates the reduction steps of the term α.*

Proof. The proof follows from the construction of Π'' and the initial membrane configuration μ_α.

Since λ-calculus is computationally equivalent to any universal Turing machine we have:

Corollary 4.3 *The λP systems can simulate the computation of a universal Turing machine.*

5 λP systems and Gödel's T-system

As indicated in our introduction, P systems may simulate type λ-calculuses. We just indicate here the guidelines of such an implementation. First, we consider

typed λ-calculuses and then Gödel's T-system.

5.1 Typed λ-calculus

A **typed** λ-calculus is a system of computation which, to λ-terms append an additonal information which is called the **type** of the term. Types over λ-terms are defined inductively. We are given a set of types $\mathcal{T}$ which obeys the following rules: σ, τ are in $\mathcal{T}$, $\sigma \rightarrow \tau$ is also in $\mathcal{T}$. Type $\sigma \rightarrow \tau$ has to be interpreted as the type of λ-terms which represents the functions which transforms λ-terms of type σ into λ-terms of type τ. As we shall need the notion a bit later, we define a **subtype** of τ a factor σ of τ considered as a word such that σ is also a type.

By definition, we are given a set of variables for each type in $\mathcal{T}$. If x is a variable and σ a type, we denote by $x : \sigma$ the fact that x has type τ. This relation is also denoted x^σ and, in this latter case, we say that x is decorated by σ.

Next, we inductively define the type of more complex λ-terms by induction on their structure by using the following two rules:

- if $x : \sigma$ and $M : \tau$, then $\lambda x\, M : \sigma \rightarrow \tau$;
- if $U : \sigma \rightarrow \tau$ and $V : \sigma$, then $(U)\, V : \tau$.

We call λ-typed calculus any subset of pure λ-terms which can be typed according to the above rules with types in $\mathcal{T}$. A λ-term M of λ-calculus is called **typable** if there is such a set $\mathcal{T}$ of types that it is possible to type M according to the above rules, starting from the variables of M, appropriately typed with $\mathcal{T}$.

It is well known that it is undecidable to decide whether a given term of the pure λ-calculus is or not typable. However, there are pure λ-terms which are known to be not typable. As an example, $(x)\, x$ cannot be type: the type of x should satisfy the type equation $\sigma = \sigma \rightarrow \sigma$, which is impossible. As a consequence, type λ-calculuses cannot contain fixed-points. This induces a serious restriction on the power of computation of type λ-calculuses. In particular, if we take the simplest typing indicated above, the corresponding typed λ-calculus can only compute polynomial functions, which was first shown by Turing, see [14].

Before turning to the solution which was found out by Gödel in the late fifties of last century, let us notice that we can simulate typed λ-calculus by P systems. It is enough to introduce **typed** labels subjected to observe the type axioms indicated above. The easy details are skipped here. They will be afforded in a forthcoming paper.

5.2 Gödel's T-system

As mentioned above, Gödel provided us in the late fifties of the previous century a typed λ-calculus able to simulate any total recursive functions. see [4].

T-system is typed λ-calculus which is constructed as follows:

The set of types satisfies the axioms of sub-section 5.1, and it contains a type $\mathbf{0}$ which is called type of the **integers**.

Besides variables of all types, as usual, T-system also contains a **constant** term denoted $\mathbf{0}$ of type $\mathbf{0}$, *i.e.* we have $\mathbf{0}$:$\mathbf{0}$. Term $\mathbf{0}$ is called **zero**. T-system also contains

a constant term $\mathbf{S}$ called **successor** which is of type $\mathbf{0} \to \mathbf{0}$. We may interpret $\mathbf{0}$ as integer zero and $\mathbf{S}$ as the function which associates $n+1$ to integer n.

We define **integers** in T-system as the following terms:

- $\mathbf{0}$ is an integer;
- if N is an integer, $(\mathbf{S})\,N$ is also an integer.

And so, the integers in T-sytems are terms $\mathbf{0}$, $(\mathbf{S})\,\mathbf{0}$, $(\mathbf{S})\,(\mathbf{S})\,\mathbf{0}$, and so on.

Next, for each type $s \in \mathcal{T}$, there is a constant term $\mathbf{R}_s$ called **recursor** of type s such that:

- $\mathbf{R}_s$ is of type $\mathbf{0} \to (s \to \mathbf{0} \to s) \to s \to s$;
- $(((\mathbf{R}_s)\,\mathbf{0})\,\mathbf{Step})\,\mathbf{Base} = \mathbf{Base}$;
- $(((\mathbf{R}_s)\,(\mathbf{S})\,n)\,\mathbf{Step})\,\mathbf{Base} = ((\mathbf{Step})\,(((\mathbf{R}_s)\,n)\,\mathbf{Step})\,\mathbf{Base})\,n$.

Indeed, $\mathbf{R}_s$ defines a recursion. It has three arguments: the first one is an integer and the second is a function of two arguments and the third is a term of type s. The first argument is called the **recursion parameter**. The second argument is called the **step function** and the third argument is called the **base case**. When the recursion parameter is $\mathbf{0}$, we get the base case. When the recursion parameter is positive, it is then of the form $(\mathbf{S})\,n$ for some integer n, then we apply the step function to the previous value of the recursor and to n which is the previous value of the recursion parameter.

It is well known that $\mathbf{R_0}$ defines exactly primitive recursive functions. Next, $\mathbf{R_{0 \to 0}}$ contains Ackermann function which is known to be not primitive recursive. It is also known that the hierarchy of functions defined by the set of functions which are defined by all $\mathbf{R}_s$ with s being a type of a finite subset of $\mathcal{T}$ defines an infinite hierarchy. It is also known that the union of the members of the hierarchy is the set of recursive functions which are provably total in Peano arithmetic.

5.3 Implementing T-system in P systems

The basic idea consists in using the frame of [5] to implement P systems by using a translation of T-system terms into pure λ-calculus.

5.3.1 Pure λ-calculus translation of P system

Consider the developement of a term $(((\mathbf{R}_s)\,(\mathbf{S})\,(\mathbf{S})\,(\mathbf{S})\,n)\,\mathbf{Step})\,\mathbf{Base}$. We get:
$(((\mathbf{R}_s)\,(\mathbf{S})\,(\mathbf{S})\,(\mathbf{S})\,n)\,\mathbf{Step})\,\mathbf{Base}$
$= ((\mathbf{Step})\,(((\mathbf{R}_s)\,(\mathbf{S})\,(\mathbf{S})\,n)\,\mathbf{Step})\,\mathbf{Base})\,(\mathbf{S})\,(\mathbf{S})\,n$
$= ((\mathbf{Step})\,((\mathbf{Step})\,(((\mathbf{R}_s)\,(\mathbf{S})\,n)\,\mathbf{Step})\,\mathbf{Base})\,(\mathbf{S})\,n)\,(\mathbf{S})\,(\mathbf{S})\,n$
$= ((\mathbf{Step})\,((\mathbf{Step})\,((\mathbf{Step})\,(((\mathbf{R}_s)\,n)\,\mathbf{Step})\,\mathbf{Base})\,n)\,(\mathbf{S})\,n)\,(\mathbf{S})\,(\mathbf{S})\,n$

Consider now the formation of couples in pure λ-calculus. We define $<U, V>$ to be the term $\lambda x\,((x)\,U)\,V$. It is not difficult to see that we can extract U and V from $<U, V>$ using the following terms which are the first and second projection: $\pi_1 == \lambda x\,\lambda y\,x$ and $\pi_2 == \lambda x\,\lambda y\,y$. Indeed, $(<U, V>)\,\pi_1 \Rightarrow U$ and $(<U, V>)\,\pi_2 \Rightarrow V$.

Consider the term $c_n = <(((\mathbf{R}_s)\,n)\,\mathbf{Step})\,\mathbf{Base},\ n>$. Then:
$c_{n+1} = <(((\mathbf{R}_s)\,(\mathbf{S})\,n)\,\mathbf{Step})\,\mathbf{Base},\ (\mathbf{S})\,n)>$
$\quad = <((\mathbf{Step})\,(((\mathbf{R}_s)\,n)\,\mathbf{Step})\,\mathbf{Base},\ n)\,n,\ (\mathbf{S})\,n)>$

$= <((\mathbf{Step})\,(c_n)\,\pi_1)\,(c_n)\,\pi_2,\ (\mathbf{S})\,(c_n)\,\pi_2>.$

And so,

$c_{n+1} = <((\mathbf{Step})\,(c_n)\,\pi_1)\,(c_n)\,\pi_2,\ (\mathbf{S})\,(c_n)\,\pi_2> =\ (\mathbf{L})\,c_n.$

From this, using classical arguments, we infer that

$c_n =\ ((\overline{n})\,\mathbf{L})\ <\mathbf{Base},\ \mathbf{0}>,$

where $\overline{n}$ is the integer of Church corresponding to integer n. Church integers are a way to represent non-negative integers in pure λ-calculus. They are defined by the following induction axioms:

- $\lambda x\, \lambda y\, y$ is an integer of Church, namely the representation of 0;
- if $\overline{n}$ represents n, then $n{+}1$ is represented by $\lambda x\, \lambda y\, (x)\, ((\overline{n})\, x)\, y$.

And so $\overline{1} = \lambda x\, \lambda y\, (x)\, y$, $\overline{2} = \lambda x\, \lambda y\, (x)\, (x)\, y$, $\overline{3} = \lambda x\, \lambda y\, (x)\, (x)\, (x)\, y$, and so forth. Notice that $((\overline{1})\, x)\, y = (x)\, y$, $((\overline{2})\, x)\, y = (x)\, (x)\, y$, $((\overline{3})\, x)\, y = (x)\, (x)\, (x)\, y$ and so on.

As a consequence, a representation of $(((\mathbf{R}_s)\, n)\, \mathbf{Step})\, \mathbf{Base}$ in pure λ-calculus is given by: $(((\mathbf{R}_s)\, n)\, \mathbf{Step})\, \mathbf{Base} = (\pi_1)\, ((\overline{n})\, \mathbf{L})\ <\mathbf{Base},\ \mathbf{0}>$.

At this point, there is a problem: how to transform integers in T-system into the corresponding integers of Church? We have to find the simplest type for Church integers which are typable: it is not difficult to see that $\overline{0} : \tau \to (\sigma \to \sigma)$ and next: $\overline{1} : (\sigma \to \tau) \to (\sigma \to \tau)$, $\overline{2} : (\sigma \to \sigma) \to (\sigma \to \sigma)$. The simplest solution is $\sigma = \mathbf{0}$ and so, by induction on n, we can see that we obtain that $\overline{n} : (\mathbf{0} \to \mathbf{0}) \to (\mathbf{0} \to \mathbf{0})$. Define $\mathbf{N} == (\mathbf{0} \to \mathbf{0}) \to (\mathbf{0} \to \mathbf{0})$. The translation requires that we have $\mathbf{R_N}$ at our disposal. Then the translation can be performed by: $\overline{n} = (((\mathbf{R_N})\, n)\, \mathbf{Step})\, \mathbf{0}$ where $\mathbf{Step} == \lambda x\, \lambda a\, \lambda b\, (a)\, ((x)\, a)\, b$.

And so, to translate the recursors we need to translate the integers, which also requires a recursor.

5.3.2 Implementation of T-system by P systems

In order to break the above circle, we shall perform the translation of Gödel integers directly into Church integers by the P system under consideration. At this point, notice that the translation of types requires special symbols and so, as an explicit P system has only finitely many symbols, a P system can handle only a certain type of recursors, namely the recursors $\mathbf{R}_\sigma$ for all σ which are subtypes of a fixed type τ.

This shows that the implementation we consider will define a family of P systems which are indexed by the types of recursors and a family is attached to a type σ: it will simulate the functions which are definable by this recursor and by all the other recursors which can be defined by the subtypes of σ.

Now, as we cannot here give the implementation details due to the lack of room, we indicate the main guidelines of it.

We use the same frame as we did for λ-calculus, appending types to the labels of membranes as we already mentioned. The input of a P system Π of family σ where σ is a possible type for a recursor in T-system is a term V of T-system defined by a recursor of type σ. Term V may be considered of the form $(\ldots((((\mathbf{R}_\sigma)\, n)\, \mathbf{Step})\, Base)\, m_1)\ldots)\, m_k$ where n, m_1, ..., m_k are Gödel integers. In the new frame, we need only to introduce a new type of membranes labelled by both $\mathbf{R}$ and the indication of the type of the recursor which this membrane implements.

A molecule traverse the representation of term V in Π and proceeds to the translation of each term, according to its label. Integers are identified by their label **0** and recursors by the label **R** which is introduced for this purpose. The type part of the label of **R** indicates the type of the recursor which this label represents. The order of the terms in a recursor facilitates the translation as far as the recursion parameter is the active membrane inside the membrane which represents the application where the recursor is involved. The particle carries the set B of possible types for resursors of the family in order to compare it with the label of each recursor it will meet. If the label matches with a type contained in B, the translation is performed. If not, the particule stops the computation. Now, in our frame, in order to simplify things, we may assume that V is not only a correct term of T-system but it does not contain a recursor of type higher than type σ which delimits the considered family of P systems.

When the molecule completed the traversal, it sends back another molecule in order to start the computation. This sending back mechanism was already implemented in [5] and so, this part of the implementation raises no difficulty.

Next, when the pure λ-calculus computation stops, we start another translation mechanism which converts Church integers into their representation in Gödel's T-system. As we know that any term in T-system represents a **total** recursive function, we know that the computation of the translation of V in pure λ-calculus will eventually stop and the leftmost reduction strategy which is implemented in λP systems also garantees the termination of the computation. This backward translation raises no difficulty: it is the same as the direct translation. It is simply processed by a different molecule.

6 Conclusion

With this paper we improved the new model of P systems which was introduced in [5] by enlarging the structure in two ways.

First, we introduced more complex labels for membranes, introducing the types of the membranes. Second, we introduce a new kind of membrane, the recursor kind, and a translation mechanism which transforms the representation of a term in T-system into the representation in λP systems of its translation in λ-calculus. When the computation completes, we have another translation mechanism to get the result in the same format as the input.

With this mechanism, we have for the first type an infinite hierarchy of infinite families of P systems which represents terms whose computation always complete. We have a **secure** computation within P systems and as we know that the recursive functions provably total in Peano arithmetics contain all the functions we need for practical purposes, we have a secure and powerful enough P system. We have now a frame ready for real applications.

The path to conrete applications is probably a bit long. The first step on this path is a control mechanism which filters the input of our P system in order to garantee that the input is actually a term of T-system. This part of the path is already known by implementors of functional languages where some of them have a

structure much richer than T-system.

At last, we have given a solution for secure and powerful P systems. Probably it is not the single one. There will be others and, among them, the one which will end on a concrete application.

Acknowledgement

This work has partially been suported by grants EIA-0086015 and EIA-0074808 from the National Science Foundation, USA. It was also partially suported by NATO grant PST.CLG.976912 for the second and third authors and by European project *MolCoNet* IST-2001-32008 for the second and fourth author.

References

[1] Barendregt, H. P. The Lambda Calculus: Its Syntax and Semantics. *North Holland*, Amsterdam, (1984).

[2] D. Besozzi, I.I. Ardelean, G. Mauri, *The Potential of P Systems* Preproceedings of the Fourth Workshop on Membrane Computing (A. Alhazov, C. Martín-Vide, Gh. Paun *Editors*), Report GRLMC 28/03, Universitat Rovira i Virgili, Tarragona, Spain, (2003), 84-102.

[3] R. Ceterchi, R. Gramatovici, N. Jonoska, K.G. Subramanian, Tissue-like P Systems for Picture Generation, *Fundamenta Informaticae*, **56** (2003), 311–328.

[4] K. Gödel, *Über eine bischer noch nicht benützte Erweiterung des finiten Standpunktes*, Dialectica, **12**, (1958), 280-287, (English translation in *On a hitherto unexploited extension of the finitary standpoint*, Journal of Philosophical Logic, **9**, (1980)).

[5] Nataša Jonoska, Maurice Margenstern, *Tree operations in P systems and λ-calculus*, Fundamenta Informaticae, **59**, N°1, (2004), 67-90.

[6] J.L. Krivine, Lambda-calculus, types and models, *Ellis Horwood*, (1993).

[7] A. Păun, *On P Systems with Membrane Division*, in vol. Unconventional Models of Computation (I. Antoniou, C.S. Calude, M.J. Dinneen, eds.), Springer-Verlag, London, (2000), 187-201.

[8] Gh. Păun, *P Systems with Active Membranes: Attacking NP-Complete Problem s*, J. Automata, Languages and Combinatorics , vol. **6**, 1 (2001), 75-90.

[9] Gh. Păun, *Membrane Computing: An Introduction*, Springer, Berlin, Heidelberg, 2002.

[10] Gh. Păun, G. Rozenberg, *A guide to Membrane Computing*, Theoretical Computer Science, **287**, (2002), 73-100.

[11] M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, Solving VALIDITY Problem by Active Membranes with Input, *Brainstorming Week on Membrane Computing*, Tarragona, February 5-11, 2003, in Report GRMLC 26/03, Universitat Rovira i Virgili, Tarragona, Spain, 279-290.

[12] V. Rogozhin, E. Boian, *Simulation of mobile ambients by P systems* Preproceedings of the Fourth Workshop on Membrane Computing (A. Alhazov, C. Martn-Vide, Gh. Paun *Editors*), Report GRMLC 28/03, Universitat Rovira i Virgili, Tarragona, Spain, (2003), 404-427.

[13] P. Sosík, Solving a PSPACE-Complete Problem by P Systems with Active Membranes, *Brainstorming Week on Membrane Computing*, Tarragona, February 5-11, 2003, in Report GRMLC 26/03, Universitat Rovira i Virgili, Tarragona, Spain, 305-312.

[14] Alan M. Turing, *Computability and lambda-definability*, J. Symb. Log. **2**, (1937), 153-163.

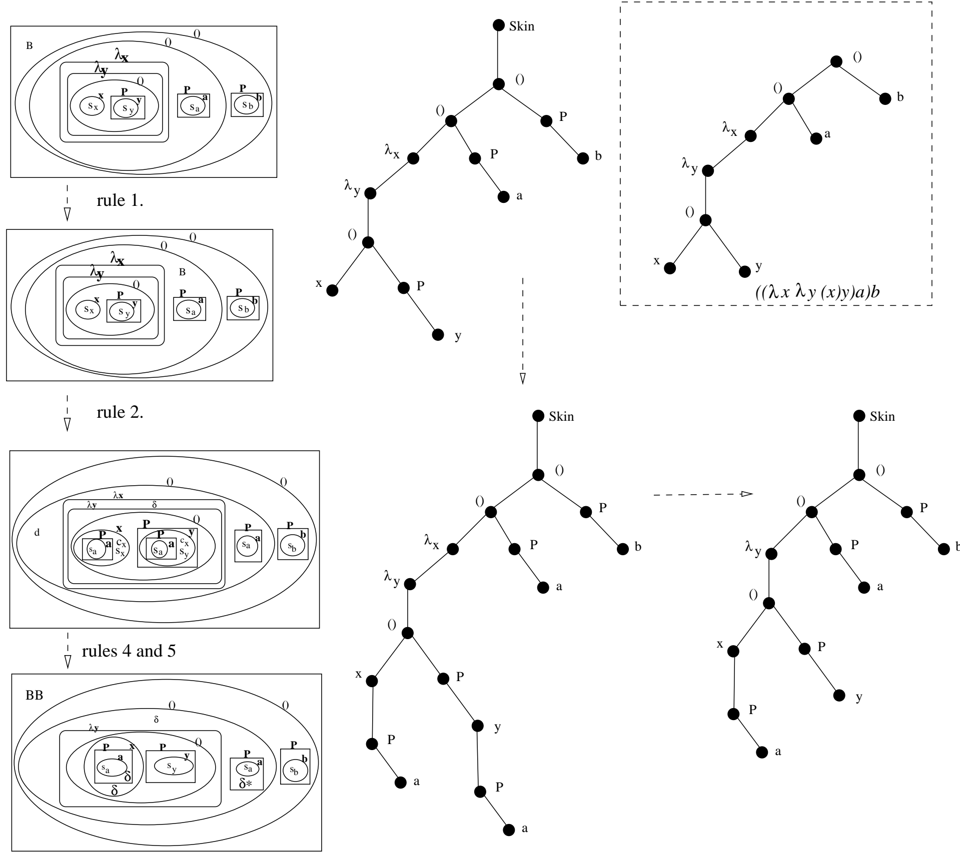

Figure 2: Computation steps corresponding to the β-reduction of the λ-term from Example 3.1 are represented in two ways: to the left with the membrane structure and to the right with the tree structure. The λ-tree corresponding to this term is boxed by a dashed line. The first two membrane structures are the same except the beginning symbol B has moved inside the membranes $[^{()}]$. Note that the only difference between the λ-tree and the corresponding tree of membrane structures is in the skin (root) and in the membrane (node) P which is protecting the variables to the right of the left most λ. The membrane structure goes through several intermediate steps between two representations of λ-terms.

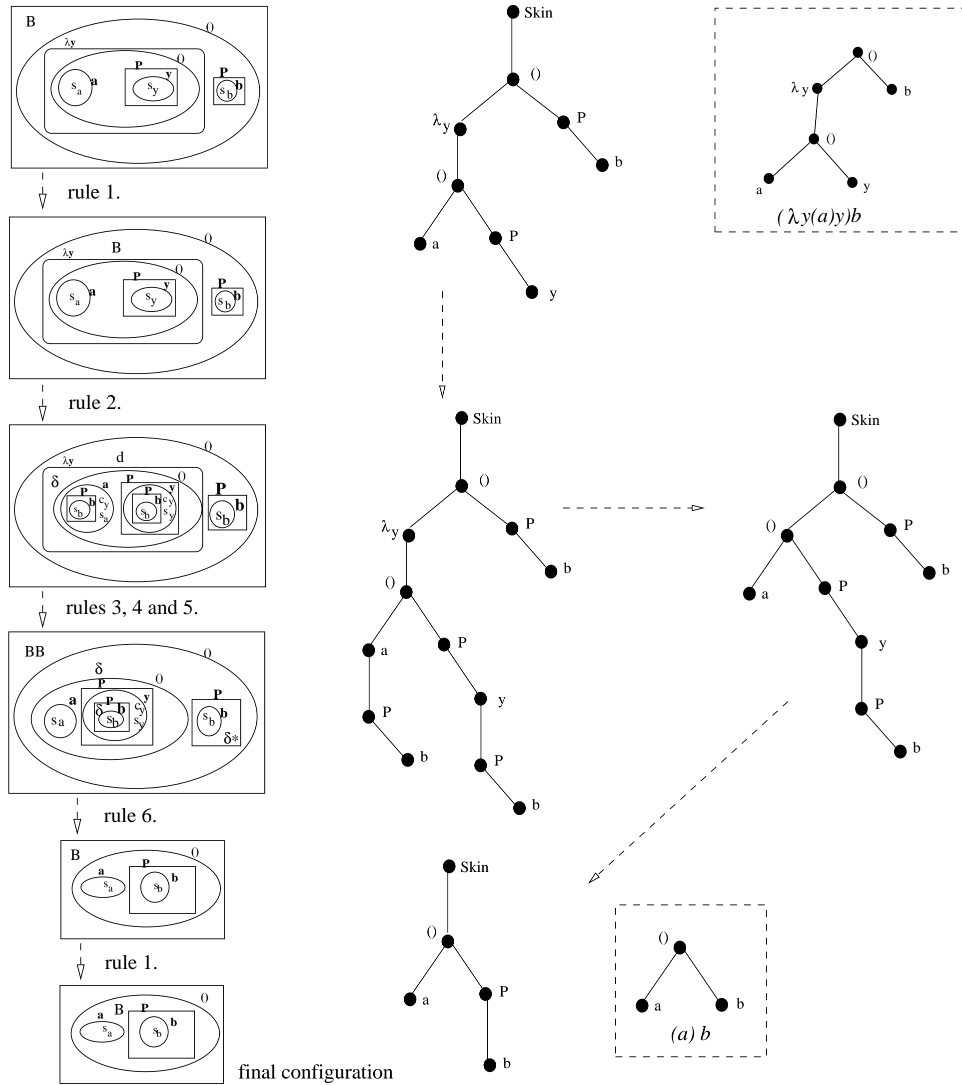

Figure 3: Continuation of the computation from Figure 2.