

A First Year Report:
Process Calculi and the **Poly★** System

Jan Jakubův

Supervisors: Dr. J. B. Wells, Prof. F. Kamareddine

June 4, 2008

Contents

| | | |
|----------|------------------------------------------------------------------|-----------|
| 1 | Introduction | 2 |
| 1.1 | Report structure overview | 2 |
| 1.2 | Brief history overview | 2 |
| 1.3 | Notations and preliminaries | 3 |
| 2 | Process calculi and their usage | 4 |
| 2.1 | The π -calculus | 4 |
| 2.2 | Mobile Ambients | 7 |
| 2.3 | Variants of the π -calculus and Mobile Ambients | 11 |
| 2.4 | BioAmbients | 14 |
| 3 | Type systems for process calculi and related approaches | 16 |
| 3.1 | Types for Mobile Ambients (TMA) | 17 |
| 3.2 | Flow analysis for BioAmbients (FABA) | 21 |
| 4 | The Poly\star System | 23 |
| 4.1 | A history of Poly \star | 23 |
| 4.2 | Syntax of Meta \star | 24 |
| 4.3 | Free and bound names, a substitution | 25 |
| 4.4 | α -equivalence and structural equivalence | 27 |
| 4.5 | Meta \star reduction rules | 27 |
| 4.5.1 | The π -calculus in Meta \star | 28 |
| 4.5.2 | Mobile Ambients in Meta \star | 29 |
| 4.6 | Shape predicates | 31 |
| 4.7 | Poly \star types | 34 |
| 5 | Work done in the first year | 35 |
| 5.1 | Contributions in the previous sections | 36 |
| 5.2 | Handling of α -equivalence in process calculi | 36 |
| 5.3 | Handling of recursive behavior | 37 |
| 5.4 | Handling of nondeterminism | 40 |
| 5.5 | Implementation issues | 41 |
| 5.6 | Comparison of Poly \star and TMA | 42 |
| 5.7 | BioAmbients in Poly \star and comparison with FABA | 44 |
| 5.8 | Other work on process calculi in Poly \star | 47 |
| 6 | Conclusions and future work | 47 |
| 6.1 | Time plan | 48 |
| A | Analysis: Transcriptional regulation by positive feedback | 49 |
| A.1 | A term to be analyzed | 49 |
| A.2 | A set \mathcal{R} of the FABA analysis | 51 |

1 Introduction

Process calculi are used to model concurrent systems, i.e., systems where several interacting units called processes engage in activity at the same time. The main stress is laid on inter process communication and synchronization. Usage of process calculi is wide spread because their intention is very general. Nowadays they cover various applications in computer science, but the same ideas are also used for modeling of molecular and biological systems or modeling of work flow in business management. The current state of art can suggest to us that more applications outside the computer science can be expected as well for the future importance of the area.

1.1 Report structure overview

The rest of this section provides a brief history overview of formal descriptions of concurrent systems in Section 1.2, and notations and preliminaries used in the rest of the report in Section 1.3. Section 2 provides brief descriptions of the most widespread modern process calculi. It includes the π -calculus, Mobile Ambients, and some of their variants and combinations. Section 3 provides a general introduction to type systems for process calculi, describes one type system for Mobile Ambients in more details, and also presents a description of one related approach. Section 4 provides brief overview of the current state of art of the Poly \star system. This section also contains some new contributions. The new contributions are distinguished from previous results of other authors in Section 5.1. Section 5 describes the work of the author of this report done during the first year of his PhD study at Heriot-Watt University. Conclusions and a future work discussion in Section 6 close the report.

1.2 Brief history overview

The history of formal descriptions of concurrent systems can be traced back to the seventies of the 20th century. Several formalisms intended to capture the concept of a computable function, like Turing Machines and the lambda calculus, were already proposed in the first half of the 20th century. A need for more subtle definition of computation arises with the expansion of Computer Science. A missing aspect here was the notion of interaction especially needed in the situation when the described system can interact with other systems during the computation.

As the first work that mentions concurrency we can point out Petri nets, for the first time published in the PhD thesis [Pet62] of C. A. Petri in 1962. Petri nets are also used nowadays but they use a different approach to concurrency than the one used in process calculi. As an another important researcher studying the semantics of parallel systems we can mention H. Bekič. Bekič worked for IBM and was well-known for his work on semantics of programming languages in the sixties and seventies. In his paper [Bek84] from 1971 he addresses a parallel execution of processes. He was the first one who used an operator to denote

a parallel composition of processes, in particular to denote what he called a quasi-parallel execution of processes. This parallel composition operator plays a central role in every modern process calculus.

The first process calculi are due to the work of R. Milner and C. A. R. Hoare. The work of Milner between the year 1973 and 1980 culminated in the Calculus of Communicating Systems (CCS) described in his book published in 1980 [Mil80]. CCS already defines operators for sequential, parallel and alternative composition which are milestones of process calculi. In 1978 Hoare published the paper that describes the language Communicating Sequential Processes (CSP) [Hoa78]. CSP provides a way to describe synchronous communication and also has been practically applied in industry to formal verification of the concurrent aspect of several systems. The subsequent development of CSP was influenced by the development of CCS and *vice versa*. Both the theory of CCS and CSP are still the subject of active research. While process calculi like CCS and CSP usually use transition systems to give a semantics to programs, there is also a different approach that uses algebraic equations to describe the behavior of the calculus. These approaches are usually called process algebras. Among them we can mention probably the first one: Algebra of Communicating Processes (ACP) [BK84] of J. Bergstra and J. W. Klop. Furthermore, there exist also algebraic approaches to CCS and CSP. These approaches are used to prove various properties of CCS and CSP and they are elaborated in details. This is one of reasons while CCS and CSP are still used, even if lot of their successors were introduced, which are in some sense either more expressive, simpler or more suitable for different purposes.

From the seventies of the 20th century up to now a large variety of process calculi has been developed. There are several different aspects that they are trying to address. Between these aspects we can mention data treatment, time treatment, probability (a stochastic information treatment), and mobility. Among calculi concerning mobility we can mention probably the most popular modern process calculi: the π -calculus [MPW92, Mil99] of R. Milner, J. Parrow and D. Walker which is the successor of CCS, and Mobile Ambients [CG98] of L. Cardelli and A. D. Gordon. Introduction of the π -calculus and Mobile Ambients has attracted lot of researchers and it has started spreading of the process calculi approach and its applications. Now, for both of them many extensions, variations, and combinations exist. Some of them will be described in Section 2.3. A more detailed history overview can be found in a paper of J. C. M. Baeten [Bae05].

1.3 Notations and preliminaries

All indexes, like i, k, l , are supposed to range over all natural number, i.e., $i \geq 0$. When S_0 and S_1 are sequences then $S_0 \blacktriangle S_1$ denotes their concatenation. $\mathcal{P}_{\text{fin}}(X)$ is the set of all finite subsets of the set X .

A function f is a set of pairs such that $(a, b) \in f$ and $(a, c) \in f$ implies $b = c$. Let $a \mapsto b$ be an alternate notation for pairs used for pairs in functions. Given

the functions f and g and the sets Y and Z we suppose the following definitions:

| | |
|----------------------------------------------------------------------------------------------|-------------------------|
| $\text{Dom } f = \{x \mid (x \mapsto y) \in f\}$ | function domain |
| $Y \xrightarrow{\text{fin}} Z = \{f \in (Y \times Z) \mid f \text{ is finite \& function}\}$ | set of finite functions |
| $S \triangleleft f = \{x \mapsto y \mid (x \mapsto y) \in f \ \& \ x \in S\}$ | domain restriction |
| $f[x \mapsto y] = ((\text{Dom } f \setminus \{x\}) \triangleleft f) \cup \{x \mapsto y\}$ | function extension |

2 Process calculi and their usage

2.1 The π -calculus

In Section 1.2 we have already mentioned that the π -calculus addresses the notion of mobility. In the π -calculus, mobility is abstracted as a change of communication links which connect processes. Abstractions of communication links are called channels. Processes in the π -calculus contain names. These names are used to denote channel names on which communication takes place and also to represent objects of communication themselves. It means that a channel name can be sent by one process to another one that can use this channel name for subsequent communication. This is called *link passing* and it is probably the most important distinction between the π -calculus and its predecessor CCS.

In the classical π -calculus papers [MPW92, Mil99], the semantics of the language of process terms is given by a *labeled transition system*. The labeled transition system consists of a set of process terms (sometimes called just *states* in general), a set of labels, and a ternary relation that assigns to every state and label a possible next state. We will not present the semantics of the π -calculus in this style but we will follow the *reduction system* style. This approach can be also found in the literature (e.g. [San93, PT97]). Reduction systems (or *rewrite systems*) can be seen as an *unlabeled transition system*. Reduction systems define a binary reduction relation on terms.

The syntax and semantics of the π -calculus is given in Figure 1. Now we describe the syntax. As already mentioned above, *names* serve as abstractions of communicated values as well as identifiers of channels on which communication takes place. Prefixes, denoted α , describe communication actions and are used to prefix processes in order to express a sequential composition, written as $\alpha.P$. An *input* prefix $x(y_1, \dots, y_k)$ expresses an input communication action, i.e., waiting for values for y_1, \dots, y_k on the channel x . Similarly, an *output* prefix $x\langle y_1, \dots, y_k \rangle$ expresses sending of names y_1, \dots, y_k on the channel x . A process $\alpha.P$, where α is an input communication prefix $x(y_1, \dots, y_k)$, waits on the channel x for k -tuple of names and after it receives them, it substitutes received names for names y_1, \dots, y_k in P and continues as this new P . A process $\alpha.Q$, where α is an output prefix, just sends names on specified channel and continues as Q . A process $(P \mid Q)$ is a process that executes processes P and Q in *parallel*. The name *restriction* operator ν restricts the scope of x in $(\nu x)P$ to P , which means that it makes it different from any other name outside P , even one also called x . A *conditioned* process $[x=y]P$ behaves like P if names x and y are

| |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $x, y, z \in \text{Name} ::= a \mid b \mid \dots \mid z \mid \dots$ |
| $\alpha \in \text{Prefix} ::= x(y_1, \dots, y_k) \mid x\langle y_1, \dots, y_k \rangle$ |
| $P, Q, R \in \text{Process} ::= (\nu x)P \mid 0 \mid (P \mid Q) \mid !P \mid \alpha.P \mid [x=y]P$ |
| <i>Structural equivalence:</i> |
| $\frac{P \sim_\alpha Q}{P \equiv Q} \quad \frac{P \equiv Q}{Q \equiv P} \quad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R} \quad \frac{P \equiv Q}{(\nu x)P \equiv (\nu x)Q}$ |
| $\frac{P \equiv Q}{P \mid R \equiv Q \mid R} \quad \frac{P \equiv Q}{!P \equiv !Q} \quad \frac{P \equiv Q}{\alpha.P \equiv \alpha.Q} \quad \frac{P \equiv Q}{[x=y]P \equiv [x=y]P}$ |
| $\frac{}{P \mid Q \equiv Q \mid P} \quad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R} \quad \frac{}{(\nu x)0 \equiv 0}$ |
| $\frac{}{(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P} \quad \frac{x \notin \text{fn}(P)}{P \mid (\nu x)Q \equiv (\nu x)(P \mid Q)}$ |
| $\frac{}{[x=x]P \equiv P} \quad \frac{}{P \mid 0 \equiv P} \quad \frac{}{!P \equiv P \mid !P}$ |
| <i>Reduction relation:</i> |
| $\frac{}{x(y_1, \dots, y_k).P \mid x\langle z_1, \dots, z_k \rangle.Q \rightarrow P\{y_1 \mapsto z_1, \dots, y_k \mapsto z_k\} \mid Q}$ |
| $\frac{P \rightarrow Q}{(\nu x)P \rightarrow (\nu x)Q} \quad \frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R} \quad \frac{R \equiv P \quad P \rightarrow Q \quad Q \equiv R'}{R \rightarrow R'}$ |

Figure 1: Syntax and semantics of the π -calculus

equal, otherwise does nothing. The *null* process ‘0’ does nothing. A *replicated* process ‘!P’ behaves like an arbitrary number of P’s in parallel, that is like ‘P | P | ...’. The intention of the replication operator is to express a repetitive behavior and can be used to encode a recursive behavior.

The operator ‘ $(\nu x)P$ ’ binds the name x in P . The prefix ‘ $x(y_1, \dots, y_k).P$ ’ binds names y_1, \dots, y_k in P . With respect to these two kind of binders the definition of *free names* of a term, written as $\text{fn}(P)$, is given in the standard way. This definition gives rise in standard way to the definition of the α -equivalence relation on terms that we denote like ‘ $P \sim_\alpha Q$ ’. A substitution in the π -calculus is a finite mapping from names to names. Based on these, the application of a substitution to a term is defined as usual, renaming names according to the substitution and providing α -renaming of bound names when necessary to avoid a wrong name capture. In order to be able to use the application of a substitution as a function, the α -renaming of bound variables has to be done in a consistent way. It means that, the application has to return the syntactically same result whenever a substitution is applied to the syntactically same term. This can be achieved, for example, by a choice function that consistently picks a new name for a bound name in a particular context (i.e., a context given by a term and by a substitution). The application of a substitution we write in

postfix, like $P\{x \mapsto y\}$, and we assume that the application binds more tightly than other operators. For example, $\langle P \mid Q\{x \mapsto y\} \rangle$ means $\langle P \mid (Q\{x \mapsto y\}) \rangle$.

In the middle of Figure 1 the binary relation \equiv on processes is defined. Its most important intention is to express that \mid should be handled as an associative and commutative operator as 0 as its unit. This relation is an equivalence relation and is often referred as the *structural congruence relation*. It handles commutativity and associativity of parallel composition and determines that the null process acts as a unit w.r.t. parallel composition. Other rules handle behavior of the ν operator and condition statements.

In the third part of Figure 1 the reduction relation is defined. The most interesting is the first rule that handles communication. It says that communication takes place only when there are two processes running in parallel where the first one is waiting on the channel x to receive a k -tuple of names and the second one is sending a k -tuple on the same channel x .

There are many versions and variations of the π -calculus. The one that we have presented here can be called choice-free polyadic synchronous π -calculus with replication. Choice-free means that we did not present the alternative choice operator that allows non-deterministic choice from one of two possibilities. This will be discussed later. Polyadic means that it allows communication on k -tuples for arbitrary k , in contrast to the monadic π -calculus where only a single name can be sent. Synchronous means that an arbitrary continuation of the output action is allowed, in contrast to the asynchronous π -calculus where the output action can continue only as a null process, i.e., $\langle x \langle y \rangle . 0 \rangle$. And finally ‘with replication’ means that the replication operator is used to express repetitive behavior. Other alternatives to the replication operator will be discussed later.

Example 2.1 Imagine that we want to model a system in which there is a printer that can print documents, a server that provides access to the printer, and a user that wants to print some document. The system without the user is modeled by two processes running in parallel. The process P (printer) is a replicated process and it is waiting for a document on a channel \mathfrak{p} (like printer port). Because in the π -calculus only communication is to be modeled, then the printing itself will be modeled by the null process. The server process S is modeled by a replicated process that works as follows. It waits on a channel \mathfrak{c} (like channel) for a pair of names that consists of an authorization key and a channel name on which a response is to be send. After the authorization key is verified (this action is again modeled by the null process), the server sends the printer port name using the received channel. In order to disallow for an unverified processes to communicate directly with the printer on the channel \mathfrak{p} , we can restrict the scope of the name \mathfrak{p} only to processes P and S . Thus the system without the user is modeled by a process $(\nu \mathfrak{p})(S \mid P)$ where definitions for P and S are as follows:

$$\begin{aligned} P &= !\mathfrak{p}(x).0 \\ S &= !\mathfrak{c}(u, v).v\langle \mathfrak{p} \rangle . 0 \end{aligned}$$

Now, the user process U is modeled by a process that sends the authorization key and a channel name r (like the response channel) to the server, waits for a printer port name, and after receiving it, sends a document (modeled by a name d) to the printer using the received channel. That is:

$$U = c\langle a, r \rangle . r(y) . y\langle d \rangle . 0$$

Now we will describe behavior of this model briefly. At first, the user U enters the scope of the name p and the server S replicates one copy of itself by the structural congruence rules:

$$\begin{aligned} U \mid (\nu p)(S \mid P) & \equiv \\ (\nu p)(U \mid !c(u, v) . v\langle p \rangle . 0 \mid P) & \equiv \\ (\nu p)(U \mid c(u, v) . v\langle p \rangle . 0 \mid !c(u, v) . v\langle p \rangle . 0 \mid P) & \end{aligned}$$

Now, the user sends the request to the server on the channel c and the server will respond on the channel r by sending the printer port to the user:

$$\begin{aligned} (\nu p)(c\langle a, r \rangle . r(y) . y\langle d \rangle . 0 \mid c(u, v) . v\langle p \rangle . 0 \mid S \mid P) & \rightarrow \\ (\nu p)(r(y) . y\langle d \rangle . 0 \mid r\langle p \rangle . 0 \mid S \mid P) & \rightarrow \\ (\nu p)(p\langle d \rangle . 0 \mid 0 \mid S \mid P) & \end{aligned}$$

Now, the inactive remainder of server's copy is removed by the structural congruence and the user is now ready to send the document d on the channel p to the printer.

$$\begin{aligned} (\nu p)(p\langle d \rangle . 0 \mid S \mid p(x) . 0 \mid P) & \rightarrow \\ (\nu p)(0 \mid S \mid 0 \mid P) & \equiv \\ (\nu p)(S \mid P) & \end{aligned}$$

Now the system continues from its initial state.

2.2 Mobile Ambients

In this section we will describe another influential process calculus, Mobile Ambients. Mobile Ambients [CG98] was presented by L. Cardelli and A. D. Gordon in 1998. The main aim is again to capture the notion of mobility. In comparison with the π -calculus, Mobile Ambients add a hierarchy of *locations* in which processes are placed. The most important notion here is the notion of an *ambient*. An ambient is a bounded place where other processes are running. An ambient can also contain other ambients and thus create a hierarchy tree. In addition to this, a process running inside an ambient can instruct the ambient that surrounds it to move in the hierarchy or to dissolve a boundary of a nested ambient. Thus the hierarchy tree can change dynamically. So, in comparison with the π -calculus, in Mobile Ambients whole processes can move. In addition to the aspects above, processes are also allowed to perform communication in the style of the π -calculus. There are two main differences. The first is that channels do not have names (but many ambient variants allow channel names). The second is that not only single names can be sent in communication but also

capabilities that allow the instructions for the change of the hierarchy tree to be sent. Capabilities can also be composed to form a more complex capabilities by putting two capabilities in sequence.

| |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $x, y \in \text{Name} \quad ::= \mathbf{a} \mid \mathbf{b} \mid \dots \mid \mathbf{z} \mid \dots$ |
| $M \in \text{Capability} \quad ::= x \mid \text{in } M \mid \text{out } M \mid \text{open } M \mid \varepsilon \mid M.M'$ |
| $P, Q, R \in \text{Process} \quad ::= (\nu x)P \mid 0 \mid (P \mid Q) \mid !P \mid M.P \mid M[P] \mid (x).P \mid \langle M \rangle$ |
| <i>Structural equivalence:</i> |
| $\frac{P \sim_\alpha Q}{P \equiv Q} \quad \frac{P \equiv Q}{Q \equiv P} \quad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R} \quad \frac{P \equiv Q}{(\nu x)P \equiv (\nu x)Q}$ |
| $\frac{P \equiv Q}{P \mid R \equiv Q \mid R} \quad \frac{P \equiv Q}{!P \equiv !Q} \quad \frac{P \equiv Q}{M.P \equiv M.Q} \quad \frac{P \equiv Q}{M[P] \equiv M[Q]}$ |
| $\frac{P \equiv Q}{(x).P \equiv (x).Q} \quad \frac{}{\varepsilon.P \equiv P} \quad \frac{}{(M.M').P \equiv M.M'.P}$ |
| $\frac{}{P \mid Q \equiv Q \mid P} \quad \frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R} \quad \frac{}{P \mid 0 \equiv P}$ |
| $\frac{}{(\nu x)0 \equiv 0} \quad \frac{}{(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P} \quad \frac{x \notin \text{fn}(P)}{P \mid (\nu x)Q \equiv (\nu x)(P \mid Q)}$ |
| $\frac{x \neq y}{(\nu x)(y[P]) \equiv y[(\nu x)P]} \quad \frac{}{!P \equiv P \mid !P} \quad \frac{}{!0 \equiv 0}$ |
| <i>Reduction relation:</i> |
| $\frac{}{x[\text{in } y.P \mid Q] \mid y[R] \rightarrow y[x[P \mid Q] \mid R]}$ |
| $\frac{}{y[x[\text{out } y.P \mid Q] \mid R] \rightarrow x[P \mid Q] \mid y[R]} \quad \frac{}{\text{open } x.P \mid x[Q] \rightarrow P \mid Q}$ |
| $\frac{}{(x).P \mid \langle M \rangle \rightarrow P\{x \mapsto M\}} \quad \frac{P \rightarrow Q}{(\nu x)P \rightarrow (\nu x)Q}$ |
| $\frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R} \quad \frac{P \rightarrow Q}{x[P] \rightarrow x[Q]} \quad \frac{R \equiv P \quad P \rightarrow Q \quad Q \equiv R'}{R \rightarrow R'}$ |

Figure 2: Syntax and semantics of Mobile Ambients

The syntax of Mobile Ambients is presented in the top part of Figure 2. Again, there are many variations of Mobile Ambients in the literature. In the terminology of Section 2.1, the one presented here can be called choice-free monadic asynchronous Mobile Ambients with replication. We will describe the syntax concentrating on the changes between Mobile Ambients and the π -calculus. The set of *names* is the same. *Capabilities* can be seen as instructions to change the hierarchy tree. Capabilities can be executed by processes. A

single name as a prefix can be viewed as a capability that is to be instantiated by some non-single name capability during a computation by communication. We say that two ambients are *sibling*, when they are contained as children in the same ambient. The capability ‘in y ’ instructs a surrounding ambient x (i.e., the ambient that contains the process that executes the capability) to move inside a sibling ambient named y . The capability ‘out y ’ instructs a surrounding ambient x that is contained inside an ambient y to move out of y and continue as a sibling of y . The capability ‘open x ’ instructs the surrounding ambient to dissolve a boundary of its child ambient x . The empty capability ε does nothing and capabilities can be sequenced using a dot.

Now we will describe the syntax of processes. Name restriction, the null process, parallel composition and replication share the same syntax as well as the same meaning with the π -calculus. In addition, capabilities can prefix processes. The syntax that expresses that a process P runs inside an ambient x is ‘ $x[P]$ ’. The syntax category ‘ $M[P]$ ’, that also allows terms like ‘(in x)[P]’, is given only to make the definition of a substitution application more uniform and terms like ‘(in x)[P]’ do not have any meaning and are usually excluded by type systems. The two remaining categories serve as communication primitives, similarly to those prefixes in the π -calculus with differences mentioned above: communication takes place only on an anonymous channel and the output is synchronous, i.e., does not have any continuation.

Again, there are two kinds of binders: the restriction operator ν and the input communication prefix (x). These binders bind names like in the π -calculus. This gives rise to the definition of free names of a term in the standard way. A substitution in Mobile Ambients maps names to capabilities. An application of a substitution to process terms is defined as expected, substituting capabilities for variables and renaming bound names when necessary. The result of an application of a substitution to a term is always syntactically well-defined term. However, in situations when substituting ‘in x ’ for y in ‘ $y[0]$ ’ or in ‘out y ’, a meaningless terms like in these cases ‘(in x)[0]’ or ‘out (in y)’ can be computed. Such a meaningless result is syntactically allowed to be formed in order to have application of substitution to process terms always defined. Such *anomalies* can be excluded by a type system. We will discuss type systems later.

In the original paper [CG98] the following approach is taken. There is stated that processes are identified up to the renaming of bound names:

$$\begin{aligned} (\nu x)P &= (\nu y)P\{x \mapsto y\} && \text{if } y \notin \text{fn}(P) \\ (x).P &= (y).P\{x \mapsto y\} && \text{if } y \notin \text{fn}(P) \end{aligned}$$

By this is meant that these processes are identical. Then, one can think about processes like about classes of α -equivalence. Note that this is a different approach than the one applied in the π -calculus from the previous section, where the α -equivalence relation is defined but α -equivalent processes are not supposed to be identical. We won’t follow this approach because it causes problems when comparing the system with systems that does not identify terms up to α -equivalence. Here, we present Mobile Ambients in the style of the previous section. Again, the α -equivalence relation is denoted \sim_α . We just mention

the approach from the original paper in order to show different techniques of handling of α -conversion. These will be discussed later.

In the middle part of Figure 2 the structural congruence relation on processes is defined. Concentrate on differences between the corresponding definition from the π -calculus. In addition to those rules that traverse a term structure, there are rules that handle the empty capability and decomposition of composed capabilities. A new rule is also the one that allows the restriction operator to skip an ambient boundary and the rules which make ‘0’ and ‘!0’ structurally equivalent.

In the bottom part of Figure 2, the reduction relation on processes is defined. The first three axioms handle in/out/open capabilities while the fourth one handles communication. Of the other rules, there is only one that is not contained in the π -calculus, and this allows reduction to be performed inside an ambient.

Example 2.2 Let us model the same system as in Example 2.1 now in terms of Mobile Ambients. The printer is modeled by a process P running inside an ambient p . The process P is ready to open an ambient named r (again like response) a to read the document that is inside. The server is modeled by a process S running inside an ambient s . The process S waits for an ambient a (authorization ambient), opens it and then sends a sequence of capabilities whose execution will move the surrounding ambient to the printer. The system without the user is modeled by a process ‘ $(\nu p)(s[S] \mid p[P])$ ’ where S and P are as follows:

$$\begin{aligned} S &= !\text{open } a.\langle \text{out } s.\text{in } p \rangle \\ P &= !\text{open } r.(x).0 \end{aligned}$$

The user itself is modeled by an ambient a instructed to enter the server, obtain capabilities guiding to the printer, execute them inside the response ambient r and after entering the printer send the document modeled by the name d . That is like follows:

$$U = a[\text{in } s.(y).r[y.\langle d \rangle]]$$

Now, lets track reductions of the modeled system. At first, the user enters the scope of p by structural congruent rule, server replicates its body S , and the user ambient a enters the server:

$$\begin{aligned} U \mid (\nu p)(s[S] \mid p[P]) & \equiv \\ (\nu p)(a[\text{in } s.(y).r[y.\langle d \rangle]] \mid s[\text{open } a.\langle \text{out } s.\text{in } p \rangle \mid S] \mid p[P]) & \rightarrow \\ (\nu p)(s[a[(y).r[y.\langle d \rangle]] \mid \text{open } a.\langle \text{out } s.\text{in } p \rangle \mid S] \mid p[P]) & \end{aligned}$$

Now, the server is ready to open the ambient a and to send the sequenced capability. The remainder of user’s process is ready to receive it:

$$\begin{aligned} (\nu p)(s[a[(y).r[y.\langle d \rangle]] \mid \text{open } a.\langle \text{out } s.\text{in } p \rangle \mid S] \mid p[P]) & \rightarrow \\ (\nu p)(s[(y).r[y.\langle d \rangle] \mid \langle \text{out } s.\text{in } p \rangle \mid S] \mid p[P]) & \rightarrow \\ (\nu p)(s[r[(\text{out } s.\text{in } p).\langle d \rangle] \mid S] \mid p[P]) & \end{aligned}$$

Now, the sequenced capability is desequenced by the structural congruence in order to allow other reductions to be applied. After that, the response ambient r is instructed to leave the server and to enter the printer:

$$\begin{aligned} (\nu p)(s[r[\text{out } s.\text{in } p.<d>] \mid S] \mid p[P]) &\rightarrow \\ (\nu p)(r[\text{in } p.<d>] \mid s[S] \mid p[P]) &\rightarrow \\ (\nu p)(s[S] \mid p[r[<d>] \mid P]) & \end{aligned}$$

Now, after the process in the printer is replicated, it's ready to open the response ambient which is then ready to send the document. The printer receives the document and finishes.

$$\begin{aligned} (\nu p)(s[S] \mid p[r[<d>] \mid \text{open } r.(x).0 \mid P]) &\rightarrow \\ (\nu p)(s[S] \mid p[<d> \mid (x).0 \mid P]) &\rightarrow \\ (\nu p)(s[S] \mid p[0 \mid P]) & \end{aligned}$$

Now, right after the structural congruence is used to removed the inactive process, the system continues from its initial state:

$$(\nu p)(s[S] \mid p[P])$$

2.3 Variants of the π -calculus and Mobile Ambients

There are many variants of both the π -calculus as well as of Mobile Ambients. In this section we will discuss some of them.

One of the variants of the π -calculus is Higher-Order π -calculus ($\text{HO}\pi$) [San93, Par01]. In the π -calculus only single names play role of objects of communication. The differences between the π -calculus and $\text{HO}\pi$ is that $\text{HO}\pi$ allows also processes to be communicated. In addition to the syntax of the π -calculus $\text{HO}\pi$ adds a process variables (here denoted as X). Every process variable is a process. Process variables can also be used to form an input commutation prefix. So for example, for arbitrary $\text{HO}\pi$ processes P and Q the following reduction is allowed:

$$a\langle P \rangle.Q \mid a(X).X \rightarrow Q \mid P$$

A process that is to be send by commutation can also contains an input commutation prefix. Let, for example, $R = b(X).(X \mid X)$. Now the following reduction can happen:

$$R \mid b\langle R \rangle.0 \rightarrow R \mid R$$

The expressiveness of $\text{HO}\pi$ was studied by Sangiorgi [San93]. Sangiorgi did show that the ability to send whole processes, surprisingly, does not add expressiveness. $\text{HO}\pi$ can be simulated in the *first order* π -calculus using communication.

A different attempt to add a hierarchy of locations into the π -calculus is the Seal Calculus (SC) [VC99]. SC was firstly published by J. Vitek and G. Castagna in 1999. Like in Mobile Ambients processes are placed inside boundaries, in

SC called *seals*. Communication is channel-based like in the π -calculus. First difference between Mobile Ambients is that there is no way how to dissolve a boundary of a seal. On the other hand there is a possibility to communicate across a boundary. There are three directions of communication: *local* that takes place inside one seal, *child-to-parent* or *down* in which a process in a seal sends data to a process in a parent seal and *parent-to-child* or *up* in which data are send in the opposite direction. In the last case the name of a child seal has to be explicitly specified because one seal can have more then one child. Another difference between Mobile Ambients is that movement in/out capabilities are substituted by a special kind of communication. It works like follows. A name of a seal is send from a process P to a process Q . When the process Q receives the seal name, the seal itself disappear from the location of P and reappear at the location of the process Q . In addition, the process Q can give a new name to the seal. The process Q has to be prepared to receive the seal name and thus the movement can occur only when both participated parts agree.

The main stress in SC lays on security. This is the reason of missing **open** capability because there is a point of view from which this capability is dangerous. It also adds the confirmation of a movement by the second part and the movement itself is invoked from a process that is in parallel with a seal, not like in Mobile Ambients from a process inside the seal. SC can also serve as a framework used to write secure network applications. Approaches based on SC are already used to develop commercial distributed applications.

From variants of Mobile Ambients we will mention Mobile Safe Ambients and Boxed Ambients. Mobile Safe Ambients (SA) [LS00] was published by F. Levi and D. Sangiorgi in 2000. SA addresses the property like the confirmation of a movement in SC. In addition to three Mobile Ambients in/out/open actions, it adds three *co-actions* $\overline{\text{in}}/\overline{\text{out}}/\overline{\text{open}}$. Whenever a process inside an ambient is going to execute a move action, the second participated part has to be ready to execute the corresponding co-action. For example, the out reduction rule in SA looks like:

$$y[x[\text{out } y.P_1 \mid P_2] \mid \overline{\text{out}} y.Q_1 \mid Q_2] \rightarrow x[P_1 \mid P_2] \mid y[Q_1 \mid Q_2]$$

In the case of the out-rule is not clear which process exactly is the second participated part. It can be the process that should allow the ambient y to be expelled (like above) or it can be the process that should allow the ambient y to enter new location, in the above rule some process running in the top-level ambient. Thats why, for example in a paper of Margaria and Zacchi [MZ03] one can find the following variant of the above rule:

$$y[x[\text{out } y.P_1 \mid P_2] \mid Q] \mid \overline{\text{out}} y.R \rightarrow x[P_1 \mid P_2] \mid y[Q] \mid R$$

In cases of in/open actions there is no confusion in which location the second participator resides.

Boxed Ambients (BA) [BCC01]. BA was published by M. Bugliesi, G. Castagna and S. Crafa in 2001. It is a calculus that results from Mobile Ambients by dropping the **open** capability like in SC. Again, in order to allow

communication new primitives which provide communication across an ambient boundary are added. In BA input communication prefixes are written as $(x)^\eta$ and $\langle M \rangle^\eta$, where η can be one of \uparrow , \star or an arbitrary name. It has the following meaning: \uparrow means that communicated value is to be send or read from the parent ambient; single name x means that communicated value is to be send or read from the child ambient x ; and \star means that communication is local to the ambient of executing process. Consider the following example with three nested ambients:

$$n[(x)^p.P \mid p[\langle M \rangle^\star.0 \mid (y)^\star.Q \mid q[\langle N \rangle^\uparrow.0]]]$$

In this example the process $(x)^p.P$ wants to read a value from the child ambient p . The process $(y)^\star.Q$ wants to read a local value. The process $\langle M \rangle^\star.0$ is sending the local value M . The process $\langle N \rangle^\uparrow.0$ is sending the value N to its parent. The semantics of BA allows the process $(y)^\star.Q$ to read both values N and M , while $(x)^p.P$ is allowed to read M only. Of course both processes can not read the value M at one time, because there is only one sending process. This is a difference between the Seal Calculus where semantics of SC allows the process $(y)^\star.Q$ to read only the value M (if communication is done on the same channel). In BA output values that are marked local (\star) are allowed to be read from all three directions, and local input can also read values from child's or parent's ambient that are directed to an ambient in which the reader resides. In contrast, in SC communication can only take place with matching pairs of annotations. This is to point out that although SC and BA use the same annotations (\uparrow, \star, x) their semantics is different. This is reflected by number of rules that are necessary to provide reduction semantics of these calculi. While in BA 5 communication rules are needed in SC is communication described by 3 rules only. It can be seen that in BA the directed output to child or parent captures the notion of *write access* to resource; on the other hand, directed input from child or parent captures the notion of *read access* from resource.

At the end of this section we will mention two process calculi that are intended to model complex biological systems. The first one is the BioAmbients Calculus [RPS⁺04]. It was published by A. Regev, E. M. Panina, W. Silverman, L. Cardelli and E. Shapiro in 2004. This calculus is intended to model biological compartments. Anonymous ambients are used to model compartments, and communication and movement express their interactions. BioAmbients takes cross-boundary communication from BA and coupled capabilities from SA. This calculus will be described in more details in Section 2.4. The last variation of Mobile Ambients that we will briefly discuss is the Brane Calculus [Car04] of L. Cardelli published in 2004. It is a successor of BioAmbients and the main difference here is that in the Brane Calculus also membranes that surrounds biological compartments are modeled by processes. While in BioAmbients membranes are modeled by an ambient boundary and processes are used to describe the behavior of the system surrounded by the membrane, in the Brane Calculus the main computation happens on membranes. Finally just note that the Brane Calculus is not restricted only to describe biological compartments but its scope is larger. It can for example be used to model chemical reactions. That's why 'brane' in

the name can be seen as a generalized version of ‘membrane’.

2.4 BioAmbients

In this section we will describe the BioAmbients calculus in a more detailed manner. Syntax of BioAmbients terms is presented at the top of Figure 3. *Directions* express kinds of communication. There are four kinds of communication in BioAmbients (ranged over by $\$$): local between processes in the same ambient, sibling-to-sibling (s2s) between processes in sibling ambients, parent-to-child (p2c) from process running within one ambient to a second process running in a child ambient of the first one, and child-to-parent (c2p) that is opposite to the previous one. In BioAmbients, only single names are objects of communication. Communication is located on channels like in the π -calculus. There are four different input prefixes for each communication kind $\$$ as well as four output prefixes. Output prefixes are written as ‘ $\$x!\{y\}$ ’ with the meaning that the name y is send over the channel x to a process located accordingly to $\$$. Input prefixes ‘ $\$x?\{y\}$ ’ are similar. There are three pairs of movement capabilities: *enter/accept*, *exit/expel*, *merge+/merge-*. Ambients in BioAmbients are anonymous in the sense that they don’t have any name. In Mobile Ambients as well as in all of its variants presented above, names of ambients are parts of capabilities which control movement. Another approach has to be taken in BioAmbients in order to determine an ambient that is to be moved. In BioAmbients the ambient that executes the action ‘*enter x*’ can enter any ambient that is ready to execute the co-action ‘*accept x*’. Capabilities *exit/expel* play the role of capabilities *out/over* from SA. Execution of *merge* capabilities instructs two sibling ambients to merge their content and continue as a single ambient. The syntax of BioAmbients processes is similar to the syntax of Mobile Ambients. Restriction, the null process, parallel composition and replication are exactly the same. Anonymous ambient are written as ‘ $[P]$ ’ and also prefixing of processes with input/output actions ($\alpha.P$) and with capabilities ($M.P$) follows the same style. Again we present a choice-free variant with replication, this time with synchronous output. Another variants will be discussed later.

There are two kind of binders: restriction and the input communication prefix. In the process ‘ $\$x?\{y\}.P$ ’ all occurrences of the name y are bound. This gives rise to the definition of free names of a term as expected. Then the structural congruence relation is defined in the middle part of Figure 3. There two notable differences between the corresponding definition from Mobile Ambients. At first, it is yet another handling of α -conversion. In this version of BioAmbients α -conversion is build in directly into the structural congruence relation. The approach used in the definition of the π -calculus above is similar, but in the case of the π -calculus a stand alone relation on terms is defined *before* the structural congruence relation. We will discuss all these approaches later. The second thing to note is the rule ‘ $[0] \equiv 0$ ’ that is not presented in Mobile Ambients.

Reduction semantics is given in the bottom part of Figure 3. First three rules which handle movement of ambients are described above. Four rules that handle

| | | | |
|-------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|-----------------------------------------------|
| $x, y, z \in \text{Name}$ | $::= a \mid b \mid \dots \mid z \mid \dots$ | | |
| $\$ \in \text{Direction}$ | $::= \text{local} \mid \text{s2s} \mid \text{p2c} \mid \text{c2p}$ | | |
| $\alpha \in \text{Action}$ | $::= \$x?\{y\} \mid \$x!\{y\}$ | | |
| $M \in \text{Capability}$ | $::= \text{enter } x \mid \text{exit } x \mid \text{merge}^+ x \mid$ $\text{accept } x \mid \text{expel } x \mid \text{merge}^- x$ | | |
| $P, Q, R, S \in \text{Process}$ | $::= (\nu x)P \mid 0 \mid (P \mid Q) \mid !P \mid [P] \mid M.P \mid \alpha.P$ | | |
| <i>Structural equivalence:</i> | | | |
| $\frac{}{P \equiv P}$ | $\frac{P \equiv Q}{Q \equiv P}$ | $\frac{P \equiv Q \quad Q \equiv R}{P \equiv R}$ | $\frac{P \equiv Q}{(\nu x)P \equiv (\nu x)Q}$ |
| $\frac{P \equiv Q}{P \mid R \equiv Q \mid R}$ | $\frac{P \equiv Q}{!P \equiv !Q}$ | $\frac{P \equiv Q}{[P] \equiv [Q]}$ | $\frac{P \equiv Q}{M.P \equiv M.Q}$ |
| $\frac{P \equiv Q}{\alpha.P \equiv \alpha.Q}$ | $\frac{}{P \mid Q \equiv Q \mid P}$ | $\frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R}$ | $\frac{}{P \mid 0 \equiv P}$ |
| $\frac{}{(\nu x)0 \equiv 0}$ | $\frac{}{(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P}$ | $\frac{x \notin \text{fn}(P)}{P \mid (\nu x)Q \equiv (\nu x)(P \mid Q)}$ | |
| $\frac{}{(\nu x)[P] \equiv [(\nu y)P]}$ | $\frac{}{!P \equiv P \mid !P}$ | $\frac{}{!0 \equiv 0}$ | $\frac{}{[0] \equiv 0}$ |
| $\frac{z \notin \text{fn}(P)}{\$x?\{y\}.P \equiv \$x?\{z\}.P\{y \mapsto z\}}$ | | $\frac{z \notin \text{fn}(P)}{(\nu y)P \equiv (\nu x)P\{y \mapsto z\}}$ | |
| <i>Reduction relation:</i> | | | |
| $\frac{}{[\text{enter } x.P \mid Q] \mid [\text{accept } x.R \mid S] \rightarrow [[P \mid Q] \mid R \mid S]}$ | | | |
| $\frac{}{[[\text{exit } x.P \mid Q] \mid \text{expel } x.R \mid S] \rightarrow [P \mid Q] \mid [R \mid S]}$ | | | |
| $\frac{}{[\text{merge}^+ x.P \mid Q] \mid [\text{merge}^- x.R \mid S] \rightarrow [P \mid Q \mid R \mid S]}$ | | | |
| $\frac{}{\text{local } x?\{y\}.P \mid \text{local } x!\{z\}.Q \rightarrow P\{y \mapsto z\} \mid Q}$ | | | |
| $\frac{}{\text{p2c } x?\{y\}.P \mid [Q \mid \text{c2p } x!\{z\}.R] \rightarrow P\{y \mapsto z\} \mid [Q \mid R]}$ | | | |
| $\frac{}{[Q \mid \text{c2p } x?\{y\}.P] \mid \text{pc2 } x!\{z\}.R \rightarrow [Q \mid P\{y \mapsto z\}] \mid R}$ | | | |
| $\frac{}{[R \mid \text{s2s } x?\{y\}.P] \mid [S \mid \text{s2s } x!\{z\}.Q] \rightarrow [R \mid P\{y \mapsto z\}] \mid [S \mid R]}$ | | | |
| $\frac{P \rightarrow Q}{(\nu x)P \rightarrow (\nu x)Q}$ | $\frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R}$ | | |
| $\frac{P \rightarrow Q}{[P] \rightarrow [Q]}$ | $\frac{R \equiv P \quad P \rightarrow Q \quad Q \equiv R'}{R \rightarrow R'}$ | | |

Figure 3: Syntax and semantics of BioAmbients

communication follows. This approach of handling communication is the most similar to that one taken in Seal Calculus. But this time because of absence of ambient names, the parent-to-child and child-to-parent communication has to be described by two different rules. The fourth communication rule handles sibling-to-sibling communication. Rest of rules is similar to those in Mobile Ambients.

Now we will present really simple example motivated by the original BioAmbients paper [RPS⁺04]. More complex example will be presented later. We will model a system in which there is a virus V and a cell C and the virus is allowed to enter the cell. Let the virus contain a molecule M_V and let the cell contain a molecule M_C . Here, we will model the molecules by some unspecified processes. The abstracted (modeled) operation here is a *membrane fusion*. We will model it by using merge capabilities. The system is modeled by the process ‘ $V \mid C$ ’ where:

$$V = [\text{merge- } a.M_V] \quad C = [!\text{merge+ } a.0 \mid M_C]$$

This simple system will evolve as follows:

$$\begin{aligned} V \mid C & \equiv \\ [\text{merge- } a.M_V] \mid [!\text{merge+ } a.0 \mid M_C] & \rightarrow \\ [M_V \mid M_C] & \end{aligned}$$

Now we can see that virus injected its molecule inside the cell and the molecules can possibly interact. The original cell is again prepared to accept another membrane fusion.

3 Type systems for process calculi and related approaches

Type systems were introduced for the π -calculus [KPT96, IK01], Mobile Ambients [CG99, CGG99, CGG00, LS00, AMW04a], the Seal Calculus [CGN01], Safe Ambients [LS00, MZ03] as well as for Boxed Ambients [BCC01]. In this section we discuss properties of type systems for Mobile Ambients in general and then we describe probably the first type system for Mobile Ambients [CG99] in more details. At the end of this section we briefly describe an alternative approach to static analysis of BioAmbients.

Type systems in process calculi are usually used to grant a certain property of process terms. These properties might be of different kinds. For example, a type system may grant that a process does not execute some kind of potentially dangerous action or that a process is not in a particular state. A desired property here is a *subject reduction* which says that once a type system grants a property for one process term then the same property holds for all processes to which can the original process evolve.

Type systems in general provide a class of types and a type judgment relation between processes and types. The type judgment relation for a process P and a type T is usually written as $\vdash P : T$ and is read like ‘*the process P has the*

type T. Each type T somehow encapsulates a desired property which is a type system designed to grant. It is usual to see a type T as a class of processes that have the type T . Type systems for process calculi are useful in situations where some kind of *over approximation* is admissible. For example, when a type T grants a property that a process P does not execute a dangerous action we can be sure that P will not do so whenever $\vdash P : T$ holds. On the other hand, $\not\vdash P : T$ usually does not necessarily mean that P *shall* execute the dangerous action.

Type systems for Mobile Ambients and its variants usually introduce two kind of types. *Message types* for names and capabilities and *process types* for processes. They usually follow the same idea that they assign a message type to every ambient name that occurs in a process. Such an assignment is called an *environment*, denoted E . Types for names that are bound in a process are added to the term itself. Type systems that allow types to be part of process terms are called type systems *à la Church* in the opposite to type systems *à la Curry* that does not contain types inside terms.

3.1 Types for Mobile Ambients (TMA)

In this section we will describe the type system of L. Cardelli and A. D. Gordon [CG99]. There are two type systems presented in this paper. We will describe the first one from Section 3 of the paper [CG99], to which we will refer as TMA. Lot of type systems for Mobile Ambients and its variants are inspired by TMA. TMA is a type system for (choice-free) polyadic synchronous Mobile Ambients with replication. TMA is designed to grant a property that an exchange of values of wrong kind can not occur anywhere in a typable process. The exchange of values of wrong kind can occur in two situations. At first, because of polyadic communication, a sending process can send a value of a different arity than a receiving process expects. Secondly, a sending process can send a capability while a receiving process expects a single name and *vice versa*. The latter can give rise to meaning less terms like in the following situation:

$$(x).\text{open } x.0 \mid \langle \text{out } a \rangle \rightarrow \text{open } (\text{out } a).0$$

Syntax of TMA types and process terms is given in the top part of Figure 4. *Message types* W are types of capabilities including names. *Exchange types* T are types of processes. Exchange types describe communication that is a process allowed to execute. Allowed communication is described by types of values that are allowed to be exchanged. A message type ‘ $\text{Amb}[T]$ ’ is the type of a name of an ambient that allows exchanges of type T . A message type ‘ $\text{Cap}[T]$ ’ is the type of capabilities whose executing can lead to exchanges of type T (by opening an ambient that exchanges T). The exchange type ‘ Shh ’ is the type of processes that does not execute any communication at all. An exchange type ‘ $W_1 \times \dots \times W_k$ ’ describes processes that exchange k -tuple of values, each value of the appropriate message type. For the case of $k = 0$ we write ‘ $\mathbf{1}$ ’. The type ‘ $\mathbf{1}$ ’ describes processes that execute communication on empty tuples that is useful

| | |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| $x, y \in \text{Name}$ | $::= a \mid b \mid \dots \mid z \mid \dots$ |
| $W \in \text{MessageType}$ | $::= \text{Amb}[T] \mid \text{Cap}[T]$ |
| $T \in \text{ExchangeType}$ | $::= \text{Shh} \mid W_1 \times \dots \times W_k$ |
| $E \in \text{Environment}$ | $= \text{Name} \xrightarrow{\text{fn}} \text{MsgType}$ |
| $M \in \text{Capability}$ | $::= x \mid \text{in } M \mid \text{out } M \mid \text{open } M \mid \varepsilon \mid M.M'$ |
| $P, Q \in \text{Process}$ | $::= 0 \mid (P \mid Q) \mid !P \mid M[P] \mid M.P \mid (\nu x : W)P$ $\langle M_1, \dots, M_k \rangle \mid (x_1 : W_1, \dots, x_k : W_k).P$ |

Type judgment relation:

| | | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| $\frac{E(x) = W}{E \vdash x : W}$ | $\frac{}{E \vdash \varepsilon : \text{Cap}[T]}$ | $\frac{E \vdash M : \text{Cap}[T] \quad E \vdash M' : \text{Cap}[T]}{E \vdash M.M' : \text{Cap}[T]}$ |
| $\frac{E \vdash M : \text{Amb}[T_0]}{E \vdash \text{in } M : \text{Cap}[T_1]}$ | $\frac{E \vdash M : \text{Amb}[T_0]}{E \vdash \text{out } M : \text{Cap}[T_1]}$ | $\frac{E \vdash M : \text{Amb}[T]}{E \vdash \text{open } M : \text{Cap}[T]}$ |
| $\frac{E \vdash M : \text{Cap}[T] \quad E \vdash P : T}{E \vdash M.P : T}$ | $\frac{E \vdash M : \text{Amb}[T_0] \quad E \vdash P : T_0}{E \vdash M[P] : T_1}$ | |
| $\frac{}{E \vdash 0 : T}$ | $\frac{E \vdash P : T \quad E \vdash Q : T}{E \vdash P \mid Q : T}$ | $\frac{E \vdash P : T}{E \vdash !P : T}$ |
| $\frac{E[x \mapsto \text{Amb}[T_0]] \vdash P : T_1}{E \vdash (\nu x : \text{Amb}[T_0])P : T_1}$ | $\frac{E \vdash M_1 : W_1 \quad \dots \quad E \vdash M_k : W_k}{E \vdash \langle M_1, \dots, M_k \rangle : W_1 \times \dots \times W_k}$ | |
| $\frac{E[x_1 \mapsto W_1] \dots [x_k \mapsto W_k] \vdash P : W_1 \times \dots \times W_k}{E \vdash (x_1 : W_1, \dots, x_k : W_k).P : W_1 \times \dots \times W_k}$ | | |

Figure 4: Syntax and typing rules of TMA

for synchronization. *Environments* are finite mappings from names to message types. In order for a term to be typable, an environment has to assign a message type to every free name contained in the term. Capabilities are the same as in Section 2.2. Processes differ from an untyped version presented in Figure 2 by polyadic communication and by type annotations of bound names.

Structural congruence and the reduction relation is similar to those of the untyped version. Both structural congruence and the reduction relation simply ignore type annotations. We present different rules anyway. Different structural congruence rules are as follows:

$$\begin{array}{c}
\frac{P \equiv Q}{(\nu x : W)P \equiv (\nu x : W)Q} \qquad \frac{}{(\nu x : W)0 \equiv 0} \\
\\
\frac{}{(\nu x : W_0)(\nu y : W_1)P \equiv (\nu y : W_1)(\nu x : W_0)P} \\
\\
\frac{x \notin \text{fn}(P)}{P \mid (\nu x : W)Q \equiv (\nu x : W)(P \mid Q)} \qquad \frac{x \neq y}{(\nu x : W)(y[P]) \equiv y[(\nu x : W)P]}
\end{array}$$

The only one different reduction rule is the communication one. It differs by type annotations and by polyadic communication. It is as follows:

$$\frac{}{(x_1 : W_1, \dots, x_k : W_k).P \mid \langle M_1, \dots, M_k \rangle \rightarrow P\{x_1 \mapsto M_1, \dots, x_k \mapsto M_k\}}$$

The type judgment relation is defined in the bottom part of Figure 4. It defines statements of the form $E \vdash M : W$ for capabilities M and statements of the form $E \vdash P : T$ for processes P . The only one way how to infer a statement $E \vdash M : \text{Amb}[T]$ for some M and T is the first inference rule. It requires $E(M) = \text{Amb}[T]$ to hold and that's why M has to be a single name. Thus, for example, the capability ‘in (out x)’ can not be typed. When x is a single name then capabilities in $x/\text{out } x$ can have a type $\text{Cap}[T_1]$ for an arbitrary T_1 that does not depend on the type of x . It reflects the fact that a movement of an ambient can not cause a new kind of exchanges, either in an ambient itself or at the destination location. On the other hand, dissolving an ambient boundary *can* bring a communication from the opened ambient to the location of the process that executed the **open** capability. So, a $\text{Cap}[T]$ type of an **open** capability is derived from the type of the ambient name contained in the capability (i.e., some type of the form $\text{Amb}[T]$) from which it inherits the exchange type T .

The rule for typing a process prefixed by a capability says that whenever M is a capability that may unleash an exchange of T (i.e., $E \vdash M : \text{Cap}[T]$) and P is a process exchanging T (i.e., $E \vdash P : T$) then $M.P$ is also a valid process exchanging T . When typing a process P that is running inside an ambient M we need to check that the exchange type of P agrees with the type of exchanges allowed inside M , and if so, the whole ambient with the process running inside can have an arbitrary exchange type. It reflects the fact that a communication enclosed inside an ambient boundary can not affect a computation outside an ambient boundary. Two processes running in parallel have an exchange type T if each of has the type T . When a capability M has a message type $\text{Cap}[T]$ then the process ‘ $\langle M \rangle$ ’ that is sending M has the corresponding exchange type $\text{Cap}[T]$. Here note that, for every message type W there is the exchange type W (that is $W_1 \times \dots \times W_k$ for $k = 1$) that shares the syntax. When a process P has an exchange type $W_1 \times \dots \times W_k$ then the process P prefixed by an input communication prefix can be typed only when a message type of each name that forms the prefix agrees with the corresponding W_i . When types agree then the prefixed process has the type $W_1 \times \dots \times W_k$.

The following proposition formulates a mile stone of every type system: the *subject reduction* property. It says that types are preserved under reductions.

Proposition 3.1 (Subject Reduction) *If $E \vdash P : T$ and $P \rightarrow Q$ hold then also $E \vdash Q : T$ holds.*

Now, we will provide several examples. For $E = \{a \mapsto \text{Amb}[\text{Shh}]\}$ the following holds:

- $E \vdash \text{in } a : \text{Cap}[T]$ for any T

- $E \vdash \text{open } a : \text{Cap}[T] \quad \text{iff } T = \text{Shh}$
- $E \not\vdash \text{in } (\text{out } a) : W \quad \text{for any } W$
- $E \vdash a[\text{in } a.0] : T \quad \text{for any } T$
- $E \not\vdash a[\langle \rangle] : T \quad \text{for any } T$
- $E \vdash \langle a, \text{open } a \rangle : \text{Amb}[\text{Shh}] \times \text{Cap}[\text{Shh}]$
- $E \vdash \langle a \rangle \mid (x : \text{Amb}[\text{Shh}]).\text{in } x.0 : \text{Amb}[\text{Shh}]$
- $E \not\vdash \langle \text{in } a \rangle \mid (x : \text{Amb}[\text{Shh}]).\text{in } x.0 : T \quad \text{for any } T$
- $E \not\vdash \langle \rangle \mid (x : W).0 : T \quad \text{for any } W, T$

From the fact mentioned above, that meaningless capabilities like $\text{in } (\text{out } a)$ can not have any type, and due to the subject reduction property, it is clear that a meaningless capability will never occur inside a typed process. The same holds for ill formed communication, when a capability is sent instead of a single name (or *vice versa*), or when arities of sent/received tuples do not agree. Ill formed communication is illustrated on last two examples above.

In the introduction of this section we have already mentioned over approximations that are common when using type systems. On the presented type system we can describe them in more details. It's clear that an ambient that does not do any communication (i.e., an ambient of the type Shh) can be opened in any other ambient. Unfortunately, TMA allows to open only ambients that have the same type as the process that executes the open action. This can be seen as too restrictive because it causes an over approximation of desired property (i.e., that exchange of values of wrong kind will not occur) in the sense that processes that does not lead to ill formed communication can not be typed. As an example, suppose the following:

$$\begin{aligned} E &= \{a \mapsto \text{Amb}[1], b \mapsto \text{Amb}[\text{Shh}]\} \\ P &= a[\langle \rangle \mid ().\text{open } b.0 \mid b[0]] \end{aligned}$$

In TMA, the process P can not be typed in the environment E although it's clear that no ill formed communication can occur and that communication is as stated in E .

One should solve this particular problem by adding the following type inference rule and check that the subject reduction property still holds:

$$\frac{E \vdash M : \text{Cap}[\text{Shh}] \quad E \vdash P : T}{E \vdash M.P : T}$$

As another example we can take a process ' $P = (x : W).x.0$ ' that can not have any exchange type T in TMA for an arbitrary message type W . In order to have $E \vdash P : T$ for some E the exchange type T is forced to be W by the only one applicable rule. So, one needs $E[x \mapsto W] \vdash x.0 : T$. But it means that $E[x \mapsto W] \vdash x : \text{Cap}[T]$ which is possible only when $T = \text{Cap}[T]$ that is not

possible. Again one can imagine a process like $(x : \text{Cap}[\text{Shh}]).x.0$ that can not lead to ill formed communication. This process should have the exchange type $\text{Cap}[\text{Shh}]$. Again the additional rule provided above can be used to type this process.

There are also situations whose solution is more difficult. One can easily imagine a process that can exchange several kinds of capabilities without mixing them. The main problem here is that TMA assigns to each ambient a single exchange type. In fact this is the basic property of TMA and in order to successfully type mentioned processes while still granting communication well-formedness another approach has to be taken. To describe processes with different exchanges in a single ambient is possible in the PolyA system [AMW04a] that will be described later.

3.2 Flow analysis for BioAmbients (FABA)

In this section we introduce an approach to a flow analysis of BioAmbients that is not based on type systems. Although type systems like PolyA [AMW04a] or Poly \star [MW04a, MW05] can also be used for a flow analysis, in this section we provide a basic description of an alternative approach of F. Nielson, H. R. Nielson, C. Priami and D. Rosa [NNPR07] firstly presented in 2003. We present this alternative approach here mainly to be able to provide a more detailed comparison with mentioned type systems later. We present its basic notations and results while we omit details.

A *flow analysis* (or a *static analysis*) provides a way how to determine final states to which a system can possibly evolve. It allows the set of final states to be over approximated. It means that its result may some times contain a state that can not be reached. On the other hand it allows a result to be computed with lower space and time complexity. It can be seen as some kind of precision for efficiency exchange. Thus, a flow analysis comes in useful in the situation when a size of a modeled system is very large and more precise *dynamic analysis* or *emulations* can not be used. This is the case of modeling biological systems whose sizes are frequently larger than sizes of many computer systems.

In this section we briefly describe a flow analysis for BioAmbients of Nielson et al. (FABA) [NNPR07]. FABA analysis is designed for a monadic synchronous version of BioAmbients that contains the choice operator $+$ to express non-determinism, and the *rec* operator to express recursive behavior. Both of these operators will be described later. The aim of FABA is to keep track of contents of ambients and of bindings of names caused by communication. Although ambients in BioAmbients do not have names, FABA gives some informative names to them in order to describe an ambient hierarchy more comfortably. Names of ambients, denoted μ , are taken from a finite set of ambients identities, denoted **Ambient**. In order to deal with α -conversion, FABA assigns to each name $x \in \text{Name}$ a canonical name $[x] \in \text{CanonName}$ and assume that canonical names are preserved under α -conversion. Again, the set **CanonName** is finite. A set of

all capabilities constructed from canonical names is denoted `CanonCapability`¹.

FABA keeps track of an approximation of contents of ambients in the following set:

$$\mathcal{I} \subseteq \text{Ambient} \times (\text{Ambient} \cup \text{CanonCapability})$$

Whenever $(\mu, u) \in \mathcal{I}$ holds, it means that an ambient with the identity μ may contain the ambient identified by u , or may contain the canonical capability u . In FABA, $u \in \mathcal{I}(\mu)$ stands for $(\mu, u) \in \mathcal{I}$. An approximation of relevant name bindings is stored in the following set:

$$\mathcal{R} \subseteq \text{CanonName} \times \text{CanonName}$$

Here, $(v, v') \in \mathcal{R}$ means that v may take the value v' . It's again abbreviated as $v' \in \mathcal{R}(v)$. Judgments of FABA analysis take the following form:

$$(\mathcal{I}, \mathcal{R}) \models^\mu P$$

It has the following sense. Whenever the process P is enclosed in the ambient with the identity μ then \mathcal{I} and \mathcal{R} correctly captures the behavior of P , i.e., \mathcal{I} is an over approximation of contents of ambients in P and \mathcal{R} over approximates bindings of names which come from communication. The formal definition of the above relation is out of the scope of this report. Correctness of FABA analysis is stated by the following proposition. Here, \star stands for the identity of the top-level ambient.

Proposition 3.2 (Correctness of FABA analysis) *Let $(\mathcal{I}, \mathcal{R}) \models^\star P$, $\forall x \in \text{fn}(P) : [x] \in \mathcal{R}([x])$, and let $P \rightarrow Q$. Then $(\mathcal{I}, \mathcal{R}) \models^\star Q$.*

FABA analysis works in two phases. In the first phase, it describes properties of sets \mathcal{I} and \mathcal{R} that correctly describes an initial process. It describes properties of \mathcal{I} and \mathcal{R} by formulas of Alternation-Free Least Fixed Point Logic (ALFP) [NNS⁺04]. ALFP is the fragment of the First-Order Logic. This fragment allows a polynomial search for a model of a set of formulas, while it preserves an expressiveness to naturally describe specifications of programs for the needs of a flow analysis. Within formulas that describe behavior of the initial process, two binary predicates are used to represent \mathcal{I} and \mathcal{R} , and canonical names are represented by ALFP variables. Then, formulas that describe the reduction semantics of BioAmbients are added to the formulas that describe the initial process. In the second phase, a Succinct Solver [NNS⁺04] is used to find the least model that satisfies those formulas. This model consists of realizations of the two binary predicates that represent \mathcal{I} and \mathcal{R} . This model is also a result of FABA analysis.

¹Names used in original FABA for `Ambient`, `CanonName` and `CanonCapability` are in turn **Ambient**, **Name** and **Cap**. We've adopted the notation to fit the convention of this report. Other notations are unchanged.

4 The Poly★ System

Poly★ [MW04a, MW05] is a generic type system for a large family of process calculi. It is the type system for every process calculus whose syntax can be described in a generic process calculus Meta★, and whose semantics can be described by reduction rules that satisfy some properties that are in common for many process calculi. Meta★ provides a syntax of process terms. This syntax is general enough to cover syntaxes of many process calculi, for example the π -calculus, Mobile Ambients and lot of their variants. Meta★ does not have any fixed semantics but it provides the way to describe reduction semantics of a particular calculus. Once Meta★ terms are equipped with the particular reduction semantics, Poly★ provides a type system for the described calculus. The type system provided by Poly★ enjoys some properties that are not in common for most of type systems for process calculi, namely a *spatial polymorphisms*. Last but not least, there is the implemented type inference algorithm for Poly★.

4.1 A history of Poly★

Poly★ was presented by H. Makholm and J. B. Wells [MW05] in 2005, previously presented in the technical report [MW04a] in 2004. Poly★ was developed from the previous work of the above authors and T. Amtoft on PolyA [AMW04a, AMW04b]. PolyA is a type system for Mobile Ambients and it is motivated by the previous work of Amtoft and Wells [AW02] and by the previous work of Amtoft, Kfoury, and Pericas-Geertsen [AKPG02, AKPG01].

PolyA does not follow the way of TMA as presented in Section 3.1, i.e., it does not assign a fixed exchange type to each ambient. Instead, it assigns a type to each process that gives upper bounds on (1) a possible ambient hierarchy tree contained in the process, (2) values that may be communicated, and (3) capabilities that may be used. PolyA allows, for example, typing of a messenger ambient that can collect a message of non-predetermined type and deliver it to a non-predetermined location. Thus, PolyA provides kind of polymorphisms called here *spatial polymorphisms*. Spatial polymorphisms in PolyA means that, a type of a process may depend on a location where it is found.

Types in PolyA are *dependent* in the sense that names contained in a process term are also used to form its type. This gives rise to the notion of a *shape predicate* that is intended to represent all possible computational future of a term smashed together in a single place. Some shape predicates are *types*. The basic idea of shape predicates can be described as follows. A shape predicate looks like a process term syntax tree. A process term matches the shape predicate if its syntax tree can be ‘bent into shape’ described by the shape predicate (edges of the shape predicate can be used more than once during the matching). This basic idea comes to troubles because there may be a term that can evolve to a term with an arbitrary deep syntax tree (e.g. ‘!a[!in a.0]’ in Mobile Ambients). It means that, we would have to consider infinite shape predicates. On the other hand, there is a desired property to keep types finite. Thus, PolyA restricts itself to possibly infinite trees with finite representations, in other words, *regular*

trees. Although PolyA defines a linear notion of shape predicates called *shape expression* it mostly works directly with graphs.

Not all shape predicates are called types in PolyA. Because a desired property here is the subject reduction, only those shape predicates that are closed under reductions are called types. A shape predicate is closed under reductions if whenever a process term P matches the shape predicate then the same holds for every process term P' to which P can evolve. Shape predicates that satisfy the above condition are called *semantically closed*. Although, recognition of the semantics closure was found far from easy, another, easier to recognize notion of *syntactically closed* shape predicates is defined in PolyA. However, it holds that the syntactic closure implies the semantics closure. Finally, syntactically closed shape predicates are called types in PolyA.

Although a recognition of the syntactically closed shape predicates is relatively easy to implement, it is still not enough to prove a *principal typing* property [Wel02] for the whole class of PolyA types. In fact, the principal typing property for whole PolyA has never been either proved or disproved. The principal typing property says that, among all possible types of a process term there is a type that is the most general². The principal typing property is important for type inference. For a type system with the principal typing property, the principal typing of a process term P is a desired result of a type inference algorithm for the input P . Without this property (or a similar one) it is not clear which type among all possible types of P should be the result of the type inference algorithm for the input P . That's why, PolyA defines a restricted class of types which satisfy a *discrete* and a *modest restriction*. Among this class of types, the existence of principal typing is proved and the type inference algorithm is implemented [MW04b].

The work on PolyA gives rise to Poly★, a generalization from the type system for Mobile Ambients to the type system for a large family of process calculi. Poly★ takes from PolyA the concept of shape predicates. As already noted above, it provides the way to describe reduction semantics of the process calculus in question. Based on this description, the reduction relation is automatically inferred. Again, there are notions of the semantics and the syntactic closure that follow the same ideas as in PolyA. A notable change is that Poly★ leaves off the discrete and the modest restrictions, and instead, it defines simpler conditions on types called a *width* and a *depth restriction*. The main reasons are that, the discrete and the modest restrictions were very complex and hard to understand. Last but not least, the principal typing property is proved and the type inference algorithm is implemented.

4.2 Syntax of Meta★

Meta★ is a metacalculus intended to capture syntactic properties that are in common for process terms of large variety of process calculi. It contains general

²Formally, the most general means the minimal with respect to the ordering \leq on types defined as: $T \leq T'$ iff each process term that has type T has also type T' .

| | | |
|------------------------------|-------|------------------------------------------------------------------------------------------------------------------|
| $x \in \text{Name}$ | $::=$ | $a \mid b \mid \dots \mid \text{in} \mid \text{out} \mid \text{open} \mid \dots \mid [] \mid \bullet \mid \dots$ |
| $f \in \text{Subform}$ | $::=$ | $x_0 \ x_1 \ \dots \ x_k$ |
| $M \in \text{Message}$ | $::=$ | $f \mid 0 \mid M_0.M_1$ |
| $e \in \text{Element}$ | $::=$ | $x \mid (x_1, x_2, \dots, x_k) \mid \langle M_1, M_2, \dots, M_k \rangle$ |
| $F \in \text{Form}$ | $::=$ | $e_0 \ e_1 \ \dots \ e_k$ |
| $P, Q, R \in \text{Process}$ | $::=$ | $F.P \mid !P \mid \nu(x).P \mid 0 \mid (P_0 \mid P_1)$ |

Figure 5: Syntax of Meta*.

constructions found in process calculi like: the null process ‘0’, parallel composition ‘|’, sequential composition (prefixing) ‘.’, and replication ‘!’. Meta* is not intended to be a standalone process calculus. It is intended to provide a method to express a syntax of large variety of process calculi in a uniform way. Meta* does not provide any semantics to process terms. It provides only structural equivalence in the standard way.

The syntax of Meta* is presented in Figure 4.2. The set **Name** contains all single names. It also contains all strings like ‘in’, ‘open’, or ‘coopen’, and it also contains symbols like ‘[]’, ‘•’, ‘^’, or ‘_’. A *subform* is a non-empty sequence of names used to form messages. Examples of subforms are ‘a’, ‘in a’, ‘a s’, ‘in a c’, ‘a []’. We use spaces to separate different names in a subform whenever necessary. *Messages* can serve as objects of communication. Messages are formed from subforms using the sequencing operator ‘.’. Sequencing of messages is needed in Mobile Ambients but not in the π -calculus. The syntax also defines the empty message 0 (often denoted ‘ ε ’ in other calculi). *Elements* unify notions of action or communication prefixes. We support an input element (i.e., an element that binds names) of the form (x_1, x_2, \dots, x_k) and an output element of the form $\langle M_1, M_2, \dots, M_k \rangle$. Both of the above may be empty (when $k = 0$). As an additional restriction for the input element, we require that $x_i \neq x_j$ whenever $i \neq j$. *Form* is non-empty sequence of elements. Note that every subform is also a form. Another example of a form is ‘ch(x)’ which can be used to express channel-based communication in the π -calculus style. *Processes* are built from forms using sequential composition ‘.’ (prefixing). Processes are also formed by replication ‘!’, by restriction ‘ $\nu(x)$ ’, and by parallel composition ‘|’. As usual, ‘0’ stands for the empty process. We provide a shorthand that allows us to support an ambient-like boundary as follows. We use ‘ $e_0 \dots e_k [P]$ ’ as a shorthand for ‘ $e_0 \dots e_k [] . P$ ’.

4.3 Free and bound names, a substitution

The notions of free names (FN) and names bound by an input element (BN) are defined by the following equations. Not presented cases are defined simply as a union of free or bound names of all their components.

| | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Application of a substitution to names, subforms, elements and forms:</i> | |
| $\begin{aligned} \dot{\sigma}(e_0 \dots e_k) &= \dot{\sigma}(e_0) \dots \dot{\sigma}(e_k) \\ \dot{\sigma}((x_1, \dots, x_k)) &= (x_1, \dots, x_k) \\ \dot{\sigma}\langle M_1, \dots, M_k \rangle &= \langle \dot{\sigma}(M_1), \dots, \dot{\sigma}(M_k) \rangle \end{aligned}$ | $\dot{\sigma}(x) = \begin{cases} \sigma(x) & \text{if } \sigma(x) \in \mathbf{Name} \\ x & \text{if } x \notin \text{Dom } \sigma \\ \bullet & \text{otherwise} \end{cases}$ |
| <i>Application of a substitution to messages:</i> | |
| $\begin{aligned} \ddot{\sigma}(M_1.M_2) &= \ddot{\sigma}(M_1).\ddot{\sigma}(M_2) \\ \ddot{\sigma}(0) &= 0 \end{aligned}$ | $\ddot{\sigma}(f) = \begin{cases} \sigma(x) & \text{if } f = x \in \text{Dom } \sigma \\ \dot{\sigma}(f) & \text{otherwise} \end{cases}$ |
| <i>Application of a substitution to processes:</i> | |
| $\begin{aligned} \bar{\sigma}(Q) &= \overline{\text{FN}(Q) \triangleleft \sigma}(Q) \\ \bar{\sigma}(!Q) &= !\bar{\sigma}(Q) \\ \bar{\sigma}(F.Q) &= \begin{cases} \dot{\sigma}(F).\bar{\sigma}(Q) & \text{if } F \notin \text{Dom } \sigma \ \& \ \text{BN}(F) \cap (\text{Dom } \sigma \cup \text{FN}(\sigma)) = \emptyset \\ \sigma(x)_*\bar{\sigma}(Q) & \text{if } F = x \in \text{Dom } \sigma \end{cases} \end{aligned}$ | $\begin{aligned} \bar{\sigma}(0) &= 0 & \bar{\sigma}(Q_0 \mid Q_1) &= \bar{\sigma}(Q_0) \mid \bar{\sigma}(Q_1) \\ \bar{\sigma}(\nu(x).Q) &= \nu(x).\bar{\sigma}(Q) & \text{if } x \notin (\text{Dom } \sigma \cup \text{FN}(\sigma)) & \end{aligned}$ |

Figure 6: Extensions of a substitution to Meta★ terms.

$$\begin{aligned} \text{FN}(x) &= \{x\} & \text{BN}(x) &= \emptyset \\ \text{FN}((x_1, \dots, x_k)) &= \emptyset & \text{BN}((x_1, \dots, x_k)) &= \{x_1, \dots, x_k\} \\ \text{FN}(F.P) &= \text{FN}(F) \cup (\text{FN}(P) \setminus \text{BN}(F)) & \text{BN}(F.P) &= \text{BN}(F) \cup \text{BN}(P) \\ \text{FN}(\nu(x).P) &= \text{FN}(P) \setminus \{x\} & \text{BN}(\nu(x).P) &= \text{BN}(P) \end{aligned}$$

Meta★ distinguishes between two types of binders. The restriction operator $\nu(x)$ and the input element (x_1, \dots, x_k) . Names bound by the input element inside a process P are contained in $\text{BN}(P)$ but not in $\text{FN}(P)$. Names bound by restriction inside P are contained neither in $\text{BN}(P)$ nor $\text{FN}(P)$.

In order to present the definition a substitution we at first define a helper operation M_*P that serves to remove the null message from M , and to transform a message M to a linear shape using an associativity of the message composition operator. In other calculi, this is usually achieved by structural equivalence.

$$(M_0.M_1)_*P = M_{0*}(M_{1*}P) \quad 0_*P = P \quad f_*P = f.P$$

By a substitution σ in Meta★ we mean a finite mapping $\sigma \in \mathbf{Name} \xrightarrow{\text{fin}} \mathbf{Message}$. Free names of a substitution σ are all free names of terms in its range, formally $\text{FN}(\sigma) = \{x \mid x \in \text{FN}(\sigma(x')) \ \& \ x' \in \text{Dom } \sigma\}$.

Now we define the application of σ to all names, subforms, elements and forms, denoted $\dot{\sigma}$, the application of σ to messages, denoted $\ddot{\sigma}$, and finally the application of σ to all Meta★ processes, denoted $\bar{\sigma}$. Definitions are presented in Figure 6.

Among Meta★ names we choose the name ‘ \bullet ’ to have a special meaning. We suppose that \bullet does not occur in any term given by a user. Whenever \bullet occurs in some term after application of a substitution, it flags a place where a subform that is not a single name was to be substituted for a single name. Thus, for example, when $\sigma = \{x \mapsto \text{in } a\}$ then $\bar{\sigma}(\text{open } x.0) = \text{open } \bullet.0$. This

is because such a substituting is not syntactically possible in **Meta***. Note that the extension of a substitution to processes does not do any α -conversion of bound names, and so, some cases that could lead to undesired results are left undefined by the definition. For example, when $\sigma = \{x \mapsto y\}$ then $\bar{\sigma}((y).x.Q)$ or $\bar{\sigma}(\nu(y).x.Q)$ are undefined because of a possible name capture while $\bar{\sigma}(x(y).Q)$ is undefined because it is not clear how to interpret the result. Also note that, $\bar{\sigma}(\nu(x).x.y.0) = \nu(x).x.y.0$ and $\bar{\sigma}((x).x.y.0) = (x).x.y.0$. This definition differs from the original definition presented in the **Poly*** papers [MW04a, MW05]. The reasons for the new definition and differences between the definitions are discussed in Section 5.2.

4.4 α -equivalence and structural equivalence

When we make a convention that names bound by the input element can not occur anywhere else in a term outside the scope of the binder, then we can observe that in situations like ' $(x).Q \mid \langle M \rangle$ ' can not a substituting of M for x in P leads to a wrong name capture. Because of this, we do not need to recognize α -equivalence of names bound by the input element. This is a significant technical simplification, because for many purposes we can treat the input element as any other element, without needing special machinery for α -equivalence in our formal development. On the other hand, because of a special handling of the restriction operator and its dynamic scope extension we need to handle α -conversion of ν -bound names. Because a common practice in process calculi is to define α -equivalence to respect both kinds of binders, we define here both kinds of α -equivalence in order to ease a comparison with other calculi.

Four binary relations on **Meta*** processes are defined in Figure 7. The relation \sim_ν is α -equivalence with respect to ν -bound names only, whereas $\sim_{\nu c}$ is α -equivalence with respect to both kinds of name binders. Then, we define two structural equivalence relations \equiv_ν and $\equiv_{\nu c}$ which refine in turn the relations \sim_ν and $\sim_{\nu c}$. In the following, we will suppose only *well scoped* terms that do not contain nested bindings of the same name, and none of term's free names appear bound somewhere else in the term. The formal definition is as follows.

Definitions 4.1 *The term P is well scoped when all of the following holds:*

1. $\text{BN}(P)$ and $\text{FN}(P)$ are disjoint.
2. Whenever P contains $F.P'$ then $\text{BN}(F)$ and $\text{BN}(P')$ are disjoint.
3. Whenever P contains $\nu(x).P'$ then $x \notin \text{BN}(P')$.

4.5 **Meta*** reduction rules

While **Meta*** does not provide semantics for **Meta*** terms we need to specify it. At this point, our intention of is not to specify one fixed semantics. Instead, we provide the way how to allow a calculus designer to specify his own one. Thus **Meta*** provides the syntax to specify reduction semantics of a calculus. In this

| | | | | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|-------------------------------------------------------|----------------------------------------------------------------------------------|
| <i>General rules:</i> | | | | |
| $\frac{P_0 \sim_{\alpha} P_1}{P_1 \sim_{\alpha} P_0}$ | $\frac{P_0 \sim_{\alpha} P_1 \quad P_1 \sim_{\alpha} P_2}{P_0 \sim_{\alpha} P_2}$ | $\frac{P_0 \sim_{\alpha} P_1}{F.P_0 \sim_{\alpha} F.P_1}$ | | |
| $\frac{P_0 \sim_{\alpha} P_1}{!P_0 \sim_{\alpha} !P_1}$ | $\frac{P_0 \sim_{\alpha} P_1}{\nu(x).P_0 \sim_{\alpha} \nu(x).P_1}$ | $\frac{P_0 \sim_{\alpha} P_1 \quad P_2 \sim_{\alpha} P_3}{P_0 \mid P_2 \sim_{\alpha} P_1 \mid P_3}$ | | |
| <hr/> <i>General structural equivalence rules:</i> | | | | |
| $\frac{}{P_0 \mid P_1 \equiv_{\alpha} P_1 \mid P_0}$ | $\frac{}{P_0 \mid (P_1 \mid P_2) \equiv_{\alpha} (P_0 \mid P_1) \mid P_2}$ | $\frac{}{P \mid 0 \equiv_{\alpha} P}$ | | |
| $\frac{}{!P \equiv_{\alpha} P \mid !P}$ | $\frac{}{!0 \equiv_{\alpha} 0}$ | $\frac{x \notin \text{FN}(F) \quad x \notin \text{BN}(F)}{F.\nu(x).P \equiv_{\alpha} \nu(x).F.P}$ | | |
| $\frac{x \notin \text{FN}(P_0)}{P_0 \mid \nu(x).P_1 \equiv_{\alpha} \nu(x).(P_0 \mid P_1)}$ | $\frac{}{\nu(x).\nu(x').P \equiv_{\alpha} \nu(x').\nu(x).P}$ | | | |
| <hr/> <i>Specific rules:</i> | | | | |
| $\frac{}{P \sim_{\nu} P}$ | $\frac{P_0 \sim_{\nu} P_1}{P_0 \equiv_{\nu} P_1}$ | $\frac{P_0 \sim_{\nu} P_1}{P_0 \sim_{\nu c} P_1}$ | $\frac{P_0 \sim_{\nu c} P_1}{P_0 \equiv_{\nu c} P_1}$ | $\frac{x' \notin \text{FN}(P)}{\nu(x).P \sim_{\nu} \nu(x').\{x \mapsto x'\}(P)}$ |
| $\frac{x'_1 \notin \text{FN}(P) \dots x'_k \notin \text{FN}(P) \quad \{y_1 \dots y_k\} \cap \text{FN}(F_0 \blacktriangle F_1) = \emptyset}{F_0 \blacktriangle (x_1, \dots, x_k) \blacktriangle F_1.P \sim_{\nu c} F_0 \blacktriangle (x'_1, \dots, x'_k) \blacktriangle F_1.\{x_1 \mapsto x'_1, \dots, x_k \mapsto x'_k\}(P)}$ | | | | |

Figure 7: α - and structure equivalence relations. The metavariable \sim_{α} ranges over relations $\{\sim_{\nu}, \sim_{\nu c}, \equiv_{\nu}, \equiv_{\nu c}\}$ and the metavariable \equiv_{α} ranges over relations $\{\equiv_{\nu}, \equiv_{\nu c}\}$.

report we do not present the formal machinery that is used to specify reduction rules of the designed calculus. Instead, we demonstrate how to instantiate **Meta*** to process calculi from previous sections on examples.

4.5.1 The π -calculus in **Meta***

We start with the π -calculus from Section 2.1. At first we need to provide a translation $\lceil \cdot \rceil$ from the syntax presented in Figure 1 to **Meta***. This translation is almost straight forward. We translate π -names to **Meta*** names, π -prefixes to **Meta*** elements, and π -processes to **Meta*** processes. The translation $\lceil \cdot \rceil$ is defined as follows.

$$\begin{array}{lll}
\lceil x(y_1, \dots, y_k) \rceil = x(y_1, \dots, y_k) & \lceil x \rceil = x & \lceil [x=y]P \rceil = x=y.\lceil P \rceil \\
\lceil x\langle y_1, \dots, y_k \rangle \rceil = x\langle y_1, \dots, y_k \rangle & \lceil 0 \rceil = 0 & \lceil (\nu x)P \rceil = \nu(x).\lceil P \rceil \\
\lceil \alpha.P \rceil = \lceil \alpha \rceil.\lceil P \rceil & \lceil !P \rceil = !\lceil P \rceil & \lceil (P \mid Q) \rceil = (\lceil P \rceil \mid \lceil Q \rceil)
\end{array}$$

Reduction semantics of the π -calculus specified in the **Meta*** syntax looks like the following.

$$\pi = \left\{ \begin{array}{l} \text{reduce}\{ x(y).P \mid x\langle z \rangle.Q \leftrightarrow \{y := z\}P \mid Q \}, \\ \text{reduce}\{ x=x.P \leftrightarrow P \} \end{array} \right\}$$

The set π contains descriptions of two reduction rules. The first one is the communication rule of the π -calculus. The second one we use to emulate a rule of structural equivalence of the π -calculus that is not part of structural equivalence of **Meta***, in particular ‘ $[x=x]P \equiv P$ ’. Note that the reduction rule described in the set π emulates only one ‘direction’, the opposite one is not emulated. Although the implementation supports polyadic communication we omit it in the case of the formal development. This simplification does not restrain expressiveness. That’s why, the above rule describes a monadic communication only. There is also the possibility to use an infinite set of descriptions of rules to emulate polyadic communication. This will be shown later on the example of Mobile Ambients. From the set π the following reduction relation $\xrightarrow{\pi}$ is automatically inferred.

$$\frac{}{x(y).P \mid x\langle M \rangle.Q \xrightarrow{\pi} \overline{\{y \mapsto M\}}(P) \mid Q} \qquad \frac{}{x=x.P \xrightarrow{\pi} P}$$

$$\frac{P \xrightarrow{\pi} Q}{\nu(x).P \xrightarrow{\pi} \nu(x).Q} \qquad \frac{P \xrightarrow{\pi} Q}{P \mid R \xrightarrow{\pi} Q \mid R} \qquad \frac{R \equiv_{\nu} P \quad P \xrightarrow{\pi} Q \quad Q \equiv_{\nu} R'}{R \xrightarrow{\pi} R'}$$

If we compare this reduction relation with the one presented in Figure 1 we can see that they are almost identical. However, there are four differences. The first one is monadic communication. The second is the reduction rule that emulates the structural equivalence rule discussed above. The third is that **Meta*** messages M also cover messages that are not single names. This does not cause problems because the translation function ‘ $\ulcorner \cdot \urcorner$ ’ can not produce a message that is not a single name. The fourth is that the last rule refers to the relation \equiv_{ν} of **Meta*** instead of to the relation \equiv from Figure 1. These two relations are slightly different. We have already mentioned one difference above. Another difference is in the rule ‘ $(\nu x)0 \equiv 0$ ’ of \equiv , that neither have an equivalent rule in \equiv_{ν} nor is emulated by the reduction rule. Another difference is that structural equivalence of **Meta*** does not provide α -renaming of names bound by the input element (i.e., the input communication prefix of the π -calculus). Last difference is that **Meta*** allows the ν -binder to skip a form which encodes a communication prefix. This is not allowed in \equiv . However, one can still prove the correctness result in the following form. Here, \rightarrow denotes the reduction relation of the π -calculus from Figure 1.

Proposition 4.1 *Let A, B range over π -calculus processes and P over **Meta*** processes.*

- (i) *For all A, B : $A \rightarrow B$ implies $\exists A_0 \equiv A, B_0 \equiv B: \ulcorner A_0 \urcorner \xrightarrow{\pi} \ulcorner B_0 \urcorner$*
- (ii) *For all A, P : $\ulcorner A \urcorner \xrightarrow{\pi} P$ implies $\exists B: (\ulcorner B \urcorner = P) \wedge (A \rightarrow B \vee A \equiv B)$*

4.5.2 Mobile Ambients in **Meta***

Similar instantiation is possible for Mobile Ambients from Section 3.1. At first, we have to define a translation function from terms of TMA to **Meta*** terms. We encode capabilities by **Meta*** subforms. Recall that, every subform is also a

form, and that's why, it can be used to prefix a process. The translation function $\ulcorner \cdot \urcorner$ from TMA capabilities to Meta \star subforms is defined as follows.

$$\begin{aligned} \ulcorner x \urcorner &= x & \ulcorner \text{in } x \urcorner &= \text{in } x & \ulcorner \text{out } x \urcorner &= \text{out } x & \ulcorner \text{open } x \urcorner &= \text{open } x \\ \ulcorner \varepsilon \urcorner &= 0 & \ulcorner M.M' \urcorner &= \ulcorner M \urcorner . \ulcorner M' \urcorner & \ulcorner M \urcorner &= \bullet & \text{otherwise} \end{aligned}$$

Note that the presented translation function rules out meaningless capabilities like 'in (out a)'. This is because the syntax of subforms does not allow these capabilities to be directly translated into subforms. An extension of this translation function to the translation function from TMA processes to Meta \star processes is straight forward. We denote both functions in the same way (i.e., $\ulcorner \cdot \urcorner$).

$$\begin{aligned} \ulcorner 0 \urcorner &= 0 & \ulcorner !P \urcorner &= !\ulcorner P \urcorner \\ \ulcorner M.P \urcorner &= \ulcorner M \urcorner * \ulcorner P \urcorner & \ulcorner (P_0 \mid P_1) \urcorner &= (\ulcorner P_0 \urcorner \mid \ulcorner P_1 \urcorner) \\ \ulcorner \langle M_1, \dots, M_k \rangle \urcorner &= \langle \ulcorner M_1 \urcorner, \dots, \ulcorner M_k \urcorner \rangle . 0 & \ulcorner (\nu x : W) P \urcorner &= \nu(x). \ulcorner P \urcorner \\ \ulcorner (x_1 : W_1, \dots, x_k : W_k).P \urcorner &= (x_1, \dots, x_k). \ulcorner P \urcorner \\ \ulcorner M[P] \urcorner &= \begin{cases} M [] . \ulcorner P \urcorner & \text{if } M \in \text{Name} \\ \bullet . \ulcorner P \urcorner & \text{otherwise} \end{cases} \end{aligned}$$

Here note that the null process is added as a continuation to an output message, and that meaningless ambient names are again ruled out in order to fit the syntax of Meta \star . Rewrite rules for Mobile Ambients can be specified in Meta \star syntax as follows.

$$\begin{aligned} \mathcal{A} = \{ & \text{active}\{ P \text{ in } a[P] \}, \\ & \text{reduce}\{ a[\text{in } b.P \mid Q] \mid b[R] \hookrightarrow b[a[P \mid Q] \mid R] \}, \\ & \text{reduce}\{ a[b[\text{out } a.P \mid Q] \mid R] \hookrightarrow a[R] \mid b[P \mid Q] \}, \\ & \text{reduce}\{ \text{open } a.P \mid a[R] \hookrightarrow P \mid R \} \} \cup \\ \bigcup_{k=0}^{\infty} \{ & \text{reduce}\{ \langle M_1, \dots, M_k \rangle . 0 \mid (x_1, \dots, x_k).Q \\ & \hookrightarrow \{x_1 := M_1, \dots, x_k := M_k\} Q \} \} \end{aligned}$$

Here, we have used an infinite set of descriptions of reduction rules to obtain a correspondence with the polyadic version of Mobile Ambients. Again, it is possible to specify this infinite set as a finite one within our implementation. We have omitted polyadic communication from the formal development because it seems that it unnecessarily complicates the presentation with no new insight. The first member (**active**) of the set \mathcal{A} describes 'active' positions inside terms. Active positions are positions where reduction rules can be applied. The default active position is only the top level of a term. Other active positions has to be specified by **active** rules. Note differences in font typesetting in the above set of descriptions of rules. Names shown in the font like **a**, M or P are supposed to match in turn any Meta \star name, message, or process, while names in the font like in, out are supposed to match one specific Meta \star name only. From this infinite set, a reduction relation that is equivalent to the following one is automatically derived.

$$\begin{array}{c}
\frac{P \xrightarrow{A} Q}{x[P] \xrightarrow{A} x[Q]} \qquad \frac{}{x[\text{in } y.P \mid Q] \mid y[R] \xrightarrow{A} y[x[P \mid Q] \mid R]} \\
\hline
\frac{}{y[x[\text{out } y.P \mid Q] \mid R] \xrightarrow{A} x[P \mid Q] \mid y[R]} \qquad \frac{}{\text{open } x.P \mid x[Q] \xrightarrow{A} P \mid Q} \\
\hline
\frac{}{(x_1, \dots, x_k).P \mid \langle M_1, \dots, M_k \rangle.0 \xrightarrow{A} P\{x_1 \mapsto M_1, \dots, x_k \mapsto M_k\}} \\
\hline
\frac{P \xrightarrow{A} Q}{(\nu x)P \xrightarrow{A} (\nu x)Q} \qquad \frac{P \xrightarrow{A} Q}{P \mid R \xrightarrow{A} Q \mid R} \qquad \frac{R \equiv_\nu P \quad P \xrightarrow{A} Q \quad Q \equiv_\nu R'}{R \xrightarrow{A} R'}
\end{array}$$

The first rule comes from the **active** rule. Next four rules correspond to the rest of rules descriptions from the set \mathcal{A} . Last three rules of the reduction relation \xrightarrow{A} are general, the same for every set of rules descriptions. If we compare the inferred relation with that from Section 3.1 (i.e., the relation from Figure 2 that is extended in Section 3.1 by the polyadic communication rule) we can observe that they are almost identical. Subtle differences that can be observed follow the same reasons as those described in the previous section in the case of the π -calculus. Again, one can prove the following correctness proposition. This one is more straightforward than the one of the π -calculus because we do not emulate any structural equivalence rule by a reduction rule.

Proposition 4.2 *Let A, B range over Mobile Ambients processes and P over Meta \star processes.*

- (i) For all A, B : $A \rightarrow B$ iff $\exists A_0 \equiv A, B_0 \equiv B: \ulcorner A_0 \urcorner \xrightarrow{A} \ulcorner B_0 \urcorner$
- (ii) For all A, P : $\ulcorner A \urcorner \xrightarrow{\pi} P$ implies $\exists B: \ulcorner B \urcorner = P$

4.6 Shape predicates

Now we are ready to present the central concept of Poly \star : the *shape predicates*. A shape predicate denotes a set of process terms. Certain shape predicates that are provably closed under reduction are *types*. In order to present the basic theory more clearly we do not handle *name restriction* here. Handling of name restriction is described in the Poly \star technical report [MW04a] in Section 5.3 (page 24).

The syntax and semantics of shape predicates is defined in Figure 8. *Message types* are types of Meta \star messages. A message type $\{f_1, \dots, f_k\}^*$ describes any message built only from subforms f_i , where the message may contain the subform f_i more than once or not at all. As an exception, the message type above does not match a single name. A single name x is allowed to have only a message type $\{x\}$. This allows us to decide whether the application of a substitution like $\{a \mapsto M\}$ to a subform, say ‘in a ’, will produce \bullet , without knowing anything else about M than its type. *Element types* are types of elements, and *form types* are types of forms (and also subforms). We represent shape predicates as

| | | |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|------------------------------------------------|
| $\mu \in \text{MessageType}$ | $::= \{f_1, \dots, f_k\}^* \mid \{x\}$ | |
| $\varepsilon \in \text{ElementType}$ | $::= x \mid (x_1, \dots, x_k) \mid \langle \mu_1, \dots, \mu_k \rangle$ | |
| $\varphi \in \text{FormType}$ | $::= \varepsilon_0 \varepsilon_1 \dots \varepsilon_k$ | |
| $X, Y \in \text{Node}$ | $::= X \mid Y \mid Z \mid \dots$ | |
| $\eta \in \text{Edge}$ | $::= X_1 \xrightarrow{\eta} X_2$ | |
| <i>aaa</i> | | |
| $G \in \text{ShapeGraph}$ | $= \mathcal{P}_{\text{fin}}(\text{Edge})$ | |
| $\tau \in \text{ShapePredicate}$ | $::= \langle G \mid X \rangle$ | |
| <i>Shape predicate judgment relation:</i> | | |
| $\frac{M \notin \text{Name} \quad M_* 0 = f_1.f_2 \dots f_k.0 \quad \{f_1, \dots, f_k\} \subseteq \{f'_1, \dots, f'_{k'}\}}{\vdash M : \{f'_1, \dots, f'_{k'}\}^*}$ | | |
| $\vdash x : \{x\}$ | $\vdash x : x$ | $\vdash (x_1, \dots, x_k) : (x_1, \dots, x_k)$ |
| $\vdash M_1 : \mu_1 \quad \dots \quad \vdash M_k : \mu_k$ | $\vdash \varepsilon_0 : \varepsilon_0 \quad \dots \quad \vdash \varepsilon_k : \varepsilon_k$ | |
| $\vdash \langle M_1, \dots, M_k \rangle : \langle \mu_1, \dots, \mu_k \rangle$ | $\vdash \varepsilon_0 \dots \varepsilon_k : \varepsilon_0 \dots \varepsilon_k$ | |
| $(X \xrightarrow{\eta} Y) \in G$ | $\vdash F : \varphi$ | $\vdash P : \langle G \mid Y \rangle$ |
| $\vdash F.P : \langle G \mid X \rangle$ | | $\vdash P : \tau$ |
| $\vdash P : \tau$ | | $\vdash !P : \tau$ |
| $\vdash P : \tau$ | $\vdash Q : \tau$ | $\vdash 0 : \tau$ |
| $\vdash P \mid Q : \tau$ | | |

Figure 8: The syntax and semantics of shape predicates.

finite graphs with unlabeled *nodes* and *edges* labeled with form types. A *Shape graph* is a finite set of edges. Finally, a *shape predicate* is a shape graph together with one selected node. The selected node is called the *root* node of a shape predicate.

Poly* defines a binary shape predicate judgment relation, written as $\vdash P : \tau$ for a term P and a shape predicate τ . This is read like ‘*the term P matches the shape predicate τ* ’. This relation is defined in the second part of Figure 8. Informally, a term matches a shape predicate if term’s syntax tree can be ‘bent into shape’ to match the shape graph of the shape predicate starting at the root node, such that each form in the term lies atop the corresponding edge in the graph (edges may be used more than once), and groups of parallel composition, $!$, and 0 lie within a single node in the graph.

Let us demonstrate shape predicates on simple examples from the π -calculus and Mobile Ambients. When writing down shape graphs, we have found it useful to put a label of an edge inside its target node. It improves readability. It is not possible when there are two edges with different labels leading inside the same target. But, this happens rarely and it is not the case here. Suppose the shape predicate (a) from Figure 9, motivated by a part of Example 2.1. It can match all of the following Meta* terms.

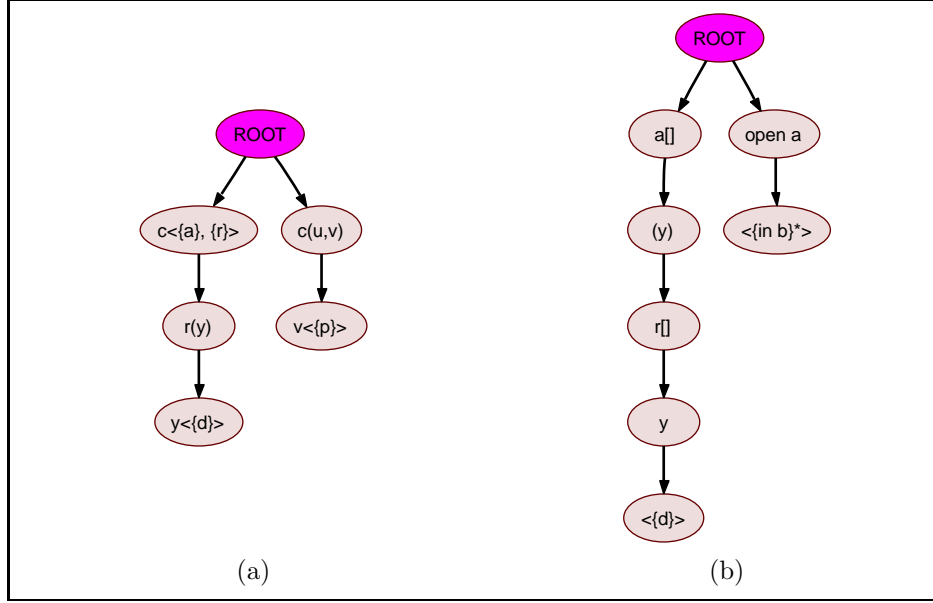


Figure 9: Examples of shape predicates.

$$\begin{aligned}
& c\langle a, r \rangle . r(y) . y\langle d \rangle . 0 \mid c(u, v) . v\langle p \rangle . 0 \\
& !c\langle a, r \rangle . r(y) . y\langle d \rangle . 0 \mid c(u, v) . v\langle p \rangle . 0 \\
& !c(u, v) . !v\langle p \rangle . 0
\end{aligned}$$

On the other hand it can not match the term ‘ $r(y) . y\langle d \rangle . 0 \mid r\langle p \rangle . 0$ ’, that is a term obtained by application of the π -calculus reduction rule to the first term in the above list. The shape predicate (b) from Figure 9 is motivated by Mobile Ambients Example 2.2. It can match, for example, all of the following.

$$\begin{aligned}
& a[(y) . r[y.\langle d \rangle . 0]] \mid \text{open } a . \langle \text{in } b \rangle . 0 \\
& a[0] \mid a[0] \mid \text{open } a . 0
\end{aligned}$$

From these, take the first Meta* term and suppose that the reduction rules of Mobile Ambients are to be applied to it. It may reduce in one step to ‘ $(y) . r[y.\langle d \rangle . 0] \mid \langle \text{in } b \rangle . 0$ ’, and then again in one step to ‘ $r[\text{in } b . \langle d \rangle . 0] \mid 0$ ’. Note that none of these terms can be matched by the shape predicate (b).

In order to prove the subject reduction property, we want types to be shape predicates for which it holds that, a set of all terms that match the shape predicates is closed under reductions. This is formally expressed by the following *semantics closure* property.

Definitions 4.2 Let a set of reduction rules descriptions \mathcal{R} be given. Call the shape predicate τ semantically closed w.r.t. \mathcal{R} , denoted by $\mathcal{R} \Rightarrow \tau$, iff:

$$\vdash P : \tau \text{ and } P \xrightarrow{\mathcal{R}} Q \text{ imply } \vdash Q : \tau$$

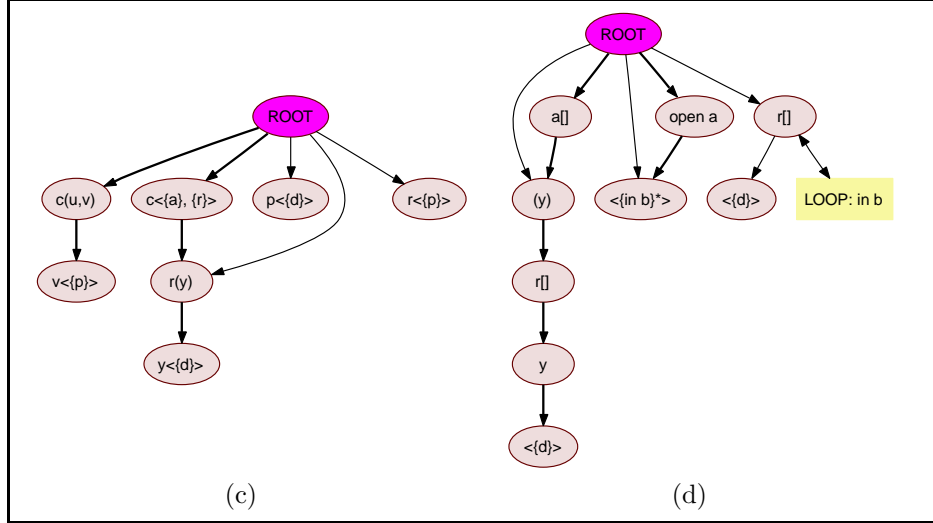


Figure 10: Examples of semantically closed shape predicates (w.r.t. π and \mathcal{A} , in turn). The yellow square denotes a self-loop of a node where it is placed labeled with the form that follows after the string ‘LOOP:’.

So, in the light of the definitions of the sets of rules descriptions from Section 4.5, we can state that the shape predicate (a) is not semantically closed w.r.t. the set π , and that the shape predicate (b) is not semantically closed w.r.t. the set \mathcal{A} . On the other hand, in Figure 10 we can see shape predicates (c) and (d) that are semantically closed w.r.t. π and \mathcal{A} in turn, i.e., $\pi \Leftrightarrow (c)$ as well as $\mathcal{A} \Leftrightarrow (d)$ hold. Moreover, every term that matches (a) also matches (c) and every term that matches (b) also matches (d). It’s not easy to recognize if a given shape predicate τ is semantically closed with respect to an arbitrary set of rules descriptions. We therefore define a restricted, but easier to decide, class of shape predicates, which we call *syntactically* closed shape predicates. **Poly*** types are syntactically closed shape predicates.

4.7 Poly* types

For a shape predicate τ and a set of rules descriptions \mathcal{R} we use the notation $\mathcal{R} \bullet \rightarrow \tau$ to denote that τ is syntactically closed w.r.t. \mathcal{R} . The desired property here is that $\mathcal{R} \bullet \rightarrow \tau$ implies $\mathcal{R} \Leftrightarrow \tau$. Because the formal definition of the notion of a syntactically closed shape predicate requires three pages of raw definitions we will omit it in this report. Instead, we provide the definitions of the *width* and the *depth* restrictions, mentioned in **Poly*** history overview Section 4.1, that leads to the principal typing property. Here we just note, that shape predicates that are semantically closed but not syntactically closed are usually only those that contain some message type of the form ‘{...}*’.

At first we define the binary relation \approx on form types as follows.

Definitions 4.3 Write $\varphi_0 \approx \varphi_1$ iff

$$\{F \in \text{Form} \mid \vdash F : \varphi_0\} \cap \{F \in \text{Form} \mid \vdash F : \varphi_1\} \neq \emptyset$$

The \approx relation is close to being the equality on form types. The only way for non-identical F 's to be related by \approx is when one of them contains a message type $\{\dots\}^*$. It is relatively safe to image \approx to be $=$, at least to the first approximation. It is necessary to take this relation instead of $=$ in two definitions below in order to achieve the principal typing property. Definitions of the width and depth restriction on shape graphs follow. A shape predicate is said to satisfy one of these properties if its shape graph component satisfies the property.

Definitions 4.4 $G \in \text{ShapeGraph}$ satisfies the width restriction iff whenever there are two edges $(X \xrightarrow{\varphi} Y) \in G$ and $(X \xrightarrow{\varphi'} Y') \in G$ with $\varphi \approx \varphi'$, then it holds that $Y = Y'$.

Definitions 4.5 $G \in \text{ShapeGraph}$ satisfies the depth restriction iff whenever there is a path of edges in G like $X_0 \xrightarrow{\varphi_1} X_1 \xrightarrow{\varphi_2} \dots \xrightarrow{\varphi_k} X_k$ with $\varphi_1 \approx \varphi_k$, then it holds that $X_1 = X_k$.

Among syntactically closed shape predicates that satisfy the width and the depth restrictions the principal typing property can be proved [MW04a]. Moreover, the implemented type inference algorithm produces only syntactically closed shape predicates that satisfy these two properties. The width restriction says that, it is not allowed for a shape graph to contain two edges with the ‘same’ label outgoing from the same node. Thus, it restricts the rank of each node to be less or equal than the number of different form types contained in the shape graph. The depth restriction forbids infinite (node disjunctive) paths that contain only finite number of ‘distinct’ form types to be contained in a shape graph. Thus, it restricts the number of nodes of the shape graph when an upper bound on the number of form types in the shape graph is given.

5 Work done in the first year

This section is intended to describe the work of the author of this report done during the first year of his PhD study on Heriot-Watt University. He spent lot of time studying various process calculi, their type systems, and related approaches, as well as by studying the current state of art of the Poly \star system. Beside that, the work can be divided into four groups.

1. Comparing of approaches of different process calculi to various aspects, like α -equivalence or recursion, and finding out ways how to adapt the Poly \star system to handle these different approaches in a uniform way. This is described in Section 5.2, Section 5.3, and Section 5.4.
2. Comparing of several type systems and related approaches with the Poly \star system, mainly with TMA and FABA. This is described in Section 5.6 and Section 5.7.

3. Implementation issues described in Section 5.5.
4. Expressing of various process calculi and their variants in the **Poly*** system, described in Section 5.8.

5.1 Contributions in the previous sections

Although a description of the work of the author of this report is mainly concentrated in the following sections, there are also some contributions that were already presented in the previous sections. In order to be able to distinguish between new contributions presented for the first time in this report from results taken from other papers, we present this section to make the distinction clear. The list of new contributions developed during the first year follows.

- In Section 4.3, the definition of the application of a substitution to **Meta*** terms in Figure 6 and the related discussion.
- In Section 4.5.1, the translation encoding $\ulcorner \cdot \urcorner$, the discussion of differences between reduction relations, and the correctness result Proposition 4.1.
- In Section 4.5.2, the translation encoding $\ulcorner \cdot \urcorner$, the discussion of differences between reduction relations, and the correctness result Proposition 4.2.

5.2 Handling of α -equivalence in process calculi

In previous sections, several approaches to handle α -equivalence and substitution were already described. The first one, in the π -calculus from Section 2.1, uses the standalone α equivalence relation \sim_α , and a substitution that α -renames bound variables in order to avoid a name capture. In order to do this, a choice function or a similar mechanism has to be used, as already discussed in Section 2.1. The π -calculus (the one from Section 2.1) does not identify processes up to α -equivalence.

The same approach was used in Mobile Ambients in Section 2.2. But, we have already mentioned that in the original Mobile Ambients paper [CG98] another approach is used. In that paper, terms are identified up to α -equivalence. Then, one can imagine processes as classes of α -equivalence on terms syntax trees. A substitution is then defined on these classes of equivalence. This is the second approach to α -equivalence described here. The third approach is taken in BioAmbients from Section 2.4. Again, a substitution is supposed to rename bound names in order to avoid a name capture, but there is no standalone definition of α -equivalence. Instead, α -equivalence is built into structural equivalence. Processes are not identified up to α -equivalence.

In **PolyA**, processes are identified up to α -renaming of ν -bound names, but not up to α -renaming of input-bound names. This can be seen as the fourth approach to α -conversion. The fifth approach is taken in **Meta***, as presented in Section 4.4. A substitution does not rename bound names, but instead, it leaves bad cases undefined. In **Meta***, two α -equivalence relations \sim_ν and $\sim_{\nu c}$

are defined. The former is α -equivalence with respect to ν -binder only, and the latter is α -equivalence with respect to both kinds of binders. The relation $\sim_{\nu c}$ is defined in order to be able to provide a more detailed comparison with process calculi that use the notion of α -equivalence with respect to both kinds of binders. In the original Poly \star papers [MW04a, MW05] the relation \sim_{ν} was built into structural equivalence³. In original Meta \star , the definition of structural equivalence precedes the definition of substitution. Thus, in order to define structural equivalence that contains α -equivalence the second notion of substitution was used. This substitution substitutes only names for names, in contrast with the former that substitutes messages for names. The second substitution was expected to guard against wrong name capture, but the formal definition was not presented in details. The former substitution does not guard against wrong name capture in the case of the input element. Instead of that, it relies on the convention of Definition 4.1 that bad cases can never occur. Another assumption was that α -renaming of ν -bound names does not produce processes that are not well scoped.

In order to be able to present formal comparison of Poly \star and TMA one has to somehow deal with different approaches to α -equivalence. Because we need to translate terms of TMA to Meta \star terms, one has to explicitly present all machinery that handles relations between syntax trees and classes of α -equivalence on them. This is orthogonal to the reasons of introducing classes of α -equivalence instead of syntax trees, i.e., to abstract from low-level details, to simplify presentation. That's why, the new version of substitution was developed. This substitution works on syntax trees and it guards against wrong name capture in the sense that wrong cases are left undefined. There are no longer two notions of substitution. Only one substitution is introduced together with its full formal definition. This allows us to state formal relations between TMA and Poly \star . Comparison with TMA is also discussed in Section 5.6.

5.3 Handling of recursive behavior

In order to express recursive behavior several approaches are used in the literature. Probably the most common, the easiest to deal with, and also the least expressive, is to use the replication operator ‘!’, that expresses repetitive behavior, to emulate recursion. A replicated process, written as ‘!P’, behaves like any finite number of copies of P in parallel, i.e., like the process ‘P | P | ... | P’. Together with other constructions provided by a particular calculus this can be used to emulate recursive behavior.

Another common approach that can be found in the literature is the recursion operator ‘rec’. The rec operator is used as follows. Syntax of a calculus is extended to contain a set of *process variables* X, a particular process variable

³Under the light of the latest research it seems that there will be a way how to avoid defining relations $\sim_{\nu c}$ and $\equiv_{\nu c}$ for reasons of comparison with other calculi. Probably, the way like in Proposition 4.1 and Proposition 4.2 will be possible. These propositions do not refer either to $\sim_{\nu c}$ or $\equiv_{\nu c}$. But, further research is needed in order to validate that these correctness results are satisfactory for needs of comparison described in Section 5.6.

is here denoted X . Every process variable is syntactically a process. One should also use ordinary names as process variables, but they are usually distinguished in the syntax. The syntax of processes P is further extended to contain a construction of the form ‘ $\text{rec } X.P$ ’. As an example of a process term that contains the rec operator we can take ‘ $\text{rec } X.\text{in } a.X$ ’, in the Mobile Ambients like syntax. The semantics of the rec operator is as follows. A process ‘ $\text{rec } X.P$ ’ behaves like the process P with the process ‘ $\text{rec } X.P$ ’ substituted for every occurrence of the process variable X inside P . This semantics can be expressed by a structural equivalence rule or by a standalone reduction rule.

One can easily emulate replication by the rec operator by taking ‘ $\text{rec } X.(P \mid X)$ ’ instead of ‘ $!P$ ’. Whether the rec operator can be emulated by replication depends on a specific process calculus. For example, it is possible in the π -calculus [Par01]. But, the same idea can not be used in Mobile Ambients and in other process calculi that contain ambient-like boundaries. When using a process calculus without process boundaries in Poly★ the following emulation [MW04a, MW05] can be used. The set of reduction rules descriptions is extended with the rule description

$$\text{reduce}\{ \text{spawn } a.0 \mid \text{rec } a.P \leftrightarrow P \}$$

and a process ‘ $\text{rec } X. \dots X$ ’ to be emulated is represented as

$$\nu(x).(\text{spawn } x.0 \mid !\text{rec } x. \dots \text{spawn } x.0)$$

The position of X is not limited to the last position in the emulated process. Every occurrence of X in the emulated process is to be replaced by ‘ $\text{spawn } x.0$ ’.

An alternative to the rec operator with the same expressiveness are *recursive let expressions* or just *let expressions*. In the case of let expressions, the syntax of a calculus is again extended to contain a set of process variables X , and the syntax of processes is extended to express that every process variable is a process, and to contain processes constructed by the following construction ‘ $\text{let } X = P \text{ in } Q$ ’. The correspondence between rec and let is straightforward and we will demonstrate it on examples of processes that have the same semantics. In the following table, each rec process on the left hand side corresponds to the let process on the right hand side in the same row.

| | | |
|------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|-----------------------------|
| $\text{rec } X.P$ | $\text{let } X = P \text{ in } X$ | |
| $(\text{rec } X.P) \mid Q$ | $\text{let } X = P \text{ in } (X \mid Q)$ | (Q doesn’t contain X) |
| $a[\text{rec } X.b[X]]$ | $\text{let } X = b[X] \text{ in } a[X]$ | |
| $\text{rec } X.\nu(z).\text{rec } Y.(X \mid Y \mid \langle z \rangle)$ | $\text{let } X = (\nu(z).\text{let } Y = (X \mid Y \mid \langle z \rangle) \text{ in } Y) \text{ in } X$ | |

In a let expression of the form ‘ $\text{let } X = P \text{ in let } Y = Q \text{ in } R$ ’, the process Q can contain both X and Y but, the process P can not refer to Y because it is not in its scope. An extension of let expressions that allow this can be introduced by the following construction.

$$\text{let } X_0 = P_0, X_1 = P_1, \dots, X_k = P_k \text{ in } P$$

Here the scope of each X_i is the whole expression and thus, each P_j can refer to every X_i .

Another way to represent recursive behavior in process calculi is by using of *constant definitions*. It is common in the π -calculus and in CCS. Again, the syntax is extended to contain a set of process variables X called *constants*, and every constant is a process. Together with a process term, on which reductions are to be applied, comes a set of global definitions of constants, each definition of the form ' $X = P$ '. Whenever some constant appears on the top level of the process, then it is unfolded accordingly to the corresponding definition in the set of definitions. Intuitively, a process with constant definitions can be seen from the point of view of the late version of let expressions like a process of the form

$$\text{let } X_0 = P_0, X_1 = P_1, \dots, X_k = P_k \text{ in } P$$

where P doesn't contain any let expression, and ' $X_i = P_i$ ' are all definitions from the set of definitions.

A limited support for constant definitions can be achieved for an arbitrary calculus in **Meta*** by introducing additional reduction rules of the following form, emulating a set of constant definitions.

$$\text{reduce}\{ A.0 \leftrightarrow P \}$$

Then, one can use the process ' $A.0$ ' in an arbitrary process position, emulating a constant. Syntactic restrictions on **Meta*** reduction rules forbid reduction rules to invent bound names, and thus, the process P above can not be an arbitrary process. But still, this provides a generic way for any process calculus representable within **Meta***. This approach was used to model some recursive BioAmbients terms.

The last approach used to capture recursive behavior found in the literature that we mention here are *parametric definitions*. This is an extension of constant definitions in the sense that constants can have name parameters. The class of constants contains constants of the form ' $X(x_0, \dots, x_k)$ ', where x_i 's are names. Every constant is a process, and again, a process term comes with a global set of definitions. Each definition has the form ' $X(x_0, \dots, x_k) = P$ ' where P may contain names x_i . Whenever a constant ' $X(a_0, \dots, a_k)$ ' is to be unfolded, names a_0, \dots, a_k are substituted for x_0, \dots, x_k in P , and this new instance of P replaces the constant. Again, this approach is common in the π -calculus and in CCS.

Expressiveness of different approaches has been studied in different process calculi [PV05, BZ04]. Expressiveness relations among different approaches depend on a particular process calculus. For example, in the π -calculus all mentioned approaches have the same expressive power. The same does not hold for Mobile Ambients. **Meta*** and **Poly*** handle replication only. So, that is not satisfactory to handle, for example, Mobile Ambients with the **rec** operator or BioAmbients with **rec**. The **rec** operator is especially frequently used when modeling biological systems. A testing version of the type inference algorithm

that handles the `rec` operator has been implemented. This is discussed in Section 5.5. For the future work, we plan to choose one of approaches mentioned in this section and extend also the formal theory of `Poly*` in order to achieve a better support for recursive behavior.

5.4 Handling of nondeterminism

Another aspect that can be found in process calculi is handling of nondeterminism. For this purposes the choice operator, usually written as `+`, is used. A construction `P + Q` represents a process that can behave like a process `P` or like a process `Q`, but only one of them. There are two main approaches to an introducing of the choice operator. The first one allows `+` to be applied to arbitrary processes `P` and `Q`. The second allows `+` to be applied only to processes prefixed by some action prefix, i.e., especially not to processes of the form `!P` and `ν(x).P`. But this difference is purely syntactical and in the former case, the processes where the choice operator is applied to some non-action prefixed processes are usually inert. In some calculi, restriction is allowed to extend or restrict its scope crossing `+` just like in the case of `|`. Thus, the choice operator is used only to choose among processes that are ready to execute some action, i.e., communication action in the case of the π -calculus, or movement/communication action in Mobile Ambients. The choice operator is usually presented as a n -ary sum written as $\sum_{i \in I} P_i$, for some finite set of indexes I . The semantics of the choice operator is usually expressed both by structural equivalence as well as by reduction rules. In the case of binary choice, structural equivalence rules express that choice is a commutative and an associative operator with `0` as its unit. Reduction rules are then used to describe desired nondeterministic behavior of choice, best illustrated on an example of the π -calculus reduction rule as follows.

$$(R_0 + x(y).P) \mid (R_1 + x\langle z \rangle.Q) \rightarrow P\{y \mapsto z\} \mid Q$$

Alternative variants R_0 and R_1 are ‘forgotten’ in the reduct.

The choice operator in `Poly*` can be emulated as follows. One can take a special form that consists of a single name, say `ch` like choice. Then, a process of the form `P0 + P1 + ... + Pk` can be encoded as follows.

$$\text{ch.}(P_0 \mid P_1 \mid \dots \mid P_k)$$

It respects commutativity and associativity because of the corresponding properties of `|`. A subtle difference is that the unit is written as `ch.0`. One can provide a reduction rule to reduce this process to `0` when really needed. Reduction rules are described in order to respect this translation correctly. For example, the description of the communication rule of the π -calculus is as follows.

$$\text{reduce}\{ \text{ch.}(R_0 \mid x(y).P) \mid \text{ch.}(R_1 \mid x\langle z \rangle.Q) \mapsto \{y:=z\}P \mid Q \}$$

This encoding is correct with respect to reduction semantics, but unfortunately, it is not so satisfactory for reasons of the type inference. One of problems here is the width restriction. Imagine the following two **Meta*** terms with different semantics with respect to choice, $P = \text{ch}.\text{(a.0} \mid \text{b.0)}$ and $Q = (\text{ch.a.0} \mid \text{ch.b.0})$. Shape predicates τ_P and τ_Q that corresponds to syntax trees of P and Q are as follows.



Note that every term that matches τ_Q also matches τ_P but, for example, P does not match τ_Q . The shape predicates τ_Q does not satisfy the width property because there are two different edges from the root node labeled with same form type ch . In order to obtain a shape predicate from τ_Q that satisfies the width restriction, these two different edges has to be unified. This unification of edges transforms τ_Q into τ_P . This exactly happens during the type inference. Although the type inference is still correct, it leads to less precise types, to over approximation. In most cases (especially in the most common case where the choice operator can be used only to compose processes prefixed with an action prefix), the result of the type inference is the same if we had simulated choice ‘+’ by ‘|’ directly. In order to achieve more precise types for terms that contain choice, another approach has to be taken. One possibility is to take ‘+’ as a new **Meta*** built-in primitive and adapt the width restriction and the type inference to respect special behavior of ‘+’. This is a subject of the future research.

5.5 Implementation issues

In this section we describe the implementation work done in the first year. This includes changes made in the implementation of the **Poly*** inference algorithm, as well as new tools developed.

A testing extension of the implementation that handles the **rec** operator was developed, as already mentioned in Section 5.3. The main idea can be described as follows. The parser was extended in order to allow parsing of **Meta*** terms that contain **rec**. The original implementation creates a shape predicate that corresponds to a syntax tree of the source **Meta*** term. In this shape predicate, ‘**rec X**’ as well as ‘**X**’ are handled as **Poly*** form types. The following translation is then performed in order to transform this shape predicate to the new one that does not contain edges labeled with form types of the shape ‘**rec X**’ and ‘**X**’. The shape predicate is traversed starting from the root node. Every occurrence of an edge labeled with ‘**rec X**’ is removed, and all edges outgoing from the destination

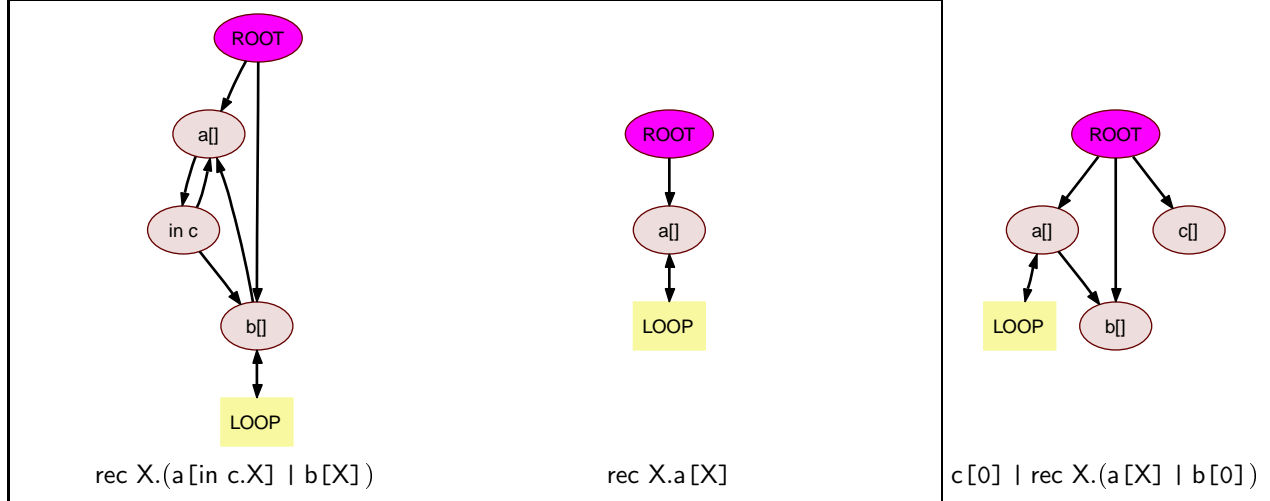


Figure 11: Results of the elimination of the `rec` operator on examples. Each yellow square denotes a self-loop of a node where it is placed, labeled with the form displayed in the node where placed.

node of the `rec` edge are resourced to the source node of the `rec` edge. The original position of the `rec` edge and all resourced edges are remembered. Then, the traversing of the shape predicate continues up to the point where an edge labeled with the form type ‘X’ is reached. This edge is always a leaf. This edge is removed, and a number of edges outgoing from the source node of the removed edge are added to the shape graph. Each new edge corresponds to one of the edges that were resourced and remembered during removing of the ‘`rec X`’ edge. The new edge has the same label and the same destination node as the remembered edge to which it corresponds. On this new shape predicate, the original type inference algorithm is ran. The transformation above is illustrated on simple examples in Figure 11. It is easy to see that this transformation respects the semantics of the `rec` operator.

Besides that, two other tools were developed. The first one (written in ML), is intended to translate a BioAmbients term in the `Meta*` syntax into an input for the Succinct solver, accordingly to the FABAs analysis described in Section 3.2 and Section 5.7. The second one (written in Python), translates an output of the Succinct solver to its equivalent graph representation. Both of these tools were used for a comparison between the FABAs analysis and the `Poly*` system, as well as to produce examples in Appendix A, see Section 5.7.

5.6 Comparison of `Poly*` and TMA

Effort was made to develop a formal comparison between the TMA type system and `Poly*` instantiated by reduction rules of Mobile Ambients. This gives rise to a paper that is now in the state of a draft. In this paper, two different

comparison are introduced. The first one is between $\text{Poly}\star$ and a restriction of TMA, referred as TMA^- . TMA^- is the version of TMA without name restriction and with communication allowed only on empty tuples (i.e., communication can neither send or receive values). Communication on empty can be useful for synchronization. The second comparison is between the full version of TMA and $\text{Poly}\star$.

For the reasons of comparison between $\text{Poly}\star$ and TMA^- , a translation function $\ulcorner \cdot \urcorner$ from TMA^- terms to $\text{Meta}\star$ terms is defined. This function is similar to the one from Section 4.5. For each double that consists of a TMA^- environment E and an exchange type T , a shape predicate $\tau_{E,T}$ is defined. This shape predicate is intended to describe the meaning of (E, T) in terms of $\text{Poly}\star$ shape predicates. Then, the following formal result is stated.

Theorem 5.1 *For every environment E , an exchange type T , and a TMA^- process P holds that:*

$$E \vdash P : T \quad \text{iff} \quad \ulcorner P \urcorner : \tau_{E,T}$$

This theorem declares that, the shape predicate $\tau_{E,T}$ correctly captures the meaning of the (E, T) double of TMA^- . The meaning of a shape predicate τ in $\text{Poly}\star$ is defined as $\llbracket \tau \rrbracket = \{P \mid \vdash P : \tau\}$, where P ranges over $\text{Meta}\star$ processes. We can define the meaning of (E, T) as the set of all TMA^- processes Q such that $E \vdash Q : T$, formally $\llbracket E, T \rrbracket = \{Q \mid E \vdash Q : T\}$. Then, the previous theorem says that for every E and T the following holds.

$$\llbracket \tau_{E,T} \rrbracket = \{\ulcorner Q \urcorner \mid Q \in \llbracket E, T \rrbracket\}$$

In the case of the full version, the main result is different. We did not present syntax and semantics of *guarded shape predicates* which serve as types of $\text{Meta}\star$ terms that contain name restriction. It can be found in the $\text{Poly}\star$ technical report [MW04a] in Section 5.3 (page 24). Guarded shape predicates are denoted Π , and a judgment relation denoted in the same way like in the case of shape predicates is used, i.e., $\vdash P : \Pi$, for a $\text{Meta}\star$ process P that possibly contains name restriction. The main problem is that a processes of TMA contains type information about bound names inside a process terms. That's why, the resulting guarded shape predicate that captures the meaning of (E, T) depends also on a term P , because its construction needs to know types of bound names inside P . It means that, we do not obtain so powerful result. But still, it correctly captures the behavior of TMA's type judgment relation. It is as follows.

Theorem 5.2 *For every environment E , an exchange type T , and a TMA process P holds that:*

$$E \vdash P : T \quad \text{iff} \quad \ulcorner P \urcorner : \Pi_{E,T,P}$$

In the current state of the draft, we were trying to compare a version of TMA that identity processes up to α -equivalence of bound names (both ν -bound and

bound by an input communication action) with a version of Poly★ that identity processes up to α -equivalence of ν -bound names. This was caused by a desire to compare Poly★ with exactly the same version of TMA from the paper [CG99]. But unfortunately, it was found far from easy. For example, in order to provide a translation from processes of TMA to Meta★ one has to somehow pick new names for the input element. Another problem is how to refer to ν -bound names because referring by names can not be used. For this purposes we did define our translation on syntax trees, and then, we did lift it to work also on classes of equivalence. But this approach requires lot of unnecessary notations, definitions, and proofs.

Recently, we have decided to compare a version of Poly★ that does not identity processes up to α -equivalence with a version of TMA that also does not do so. A relationship between two different versions of TMA can be described as follows. Here, P and Q range over classes of α -equivalence of TMA syntax trees, A and B over syntax trees, $\xrightarrow{\alpha}$ is a reduction relation on classes of equivalence (from the paper [CG99]), and \rightarrow is a reduction relation on syntax trees (from Section 3.1).

Proposition 5.1 *For each P, Q it holds that:*

$$P \xrightarrow{\alpha} Q \quad \text{iff} \quad \text{for all } A \in P, B \in Q: A \rightarrow B$$

5.7 BioAmbients in Poly★ and comparison with FAB A

In this section we present a result of comparison between the FAB A analysis [NNPR07] and Poly★. We compare results of FAB A analysis with results of the Poly★ type inference algorithm for the same inputs. For this reasons, an encoding of BioAmbients terms in Meta★, and an instantiation of Meta★ by BioAmbients reduction rules is described. We describe relations between a result of FAB A analysis and a Poly★ type. In general, the Poly★ type inference algorithm provides more precious information than a result of the FAB A analysis. Moreover, behavior of the Poly★ type inference algorithm depends on number of parameters, and there exists a configuration of the type inference algorithm that produces exactly the same results as the FAB A analysis.

The encoding $\ulcorner \cdot \urcorner$ of BioAmbients terms in Meta★ is as follows. BioAmbients names and directions are translated to Meta★ names, actions and capabilities to Meta★ forms, and BioAmbients processes to Meta★ processes.

$$\begin{array}{lll} \ulcorner \text{local} \urcorner = \text{local} & \ulcorner \text{s2s} \urcorner = \text{s2s} & \ulcorner \text{p2c} \urcorner = \text{p2c} \\ \ulcorner \text{c2p} \urcorner = \text{c2p} & \ulcorner \$x?\{y\} \urcorner = \ulcorner \$x(y) \urcorner & \ulcorner \$x!\{y\} \urcorner = \ulcorner \$x\langle y \rangle \urcorner \\ \ulcorner \text{enter } x \urcorner = \text{enter } x & \ulcorner \text{exit } x \urcorner = \text{exit } x & \ulcorner \text{merge+ } x \urcorner = \text{merge+ } x \\ \ulcorner \text{accept } x \urcorner = \text{accept } x & \ulcorner \text{expel } x \urcorner = \text{expel } x & \ulcorner \text{merge- } x \urcorner = \text{merge- } x \\ \ulcorner x \urcorner = x & \ulcorner 0 \urcorner = 0 & \ulcorner (\nu x)P \urcorner = \nu(x).\ulcorner P \urcorner & \ulcorner P \mid Q \urcorner = \ulcorner P \urcorner \mid \ulcorner Q \urcorner \\ \ulcorner !P \urcorner = !\ulcorner P \urcorner & \ulcorner [P] \urcorner = [].\ulcorner P \urcorner & \ulcorner M.P \urcorner = \ulcorner M \urcorner.\ulcorner P \urcorner & \ulcorner \alpha.P \urcorner = \ulcorner \alpha \urcorner.\ulcorner P \urcorner \end{array}$$

A set of descriptions of BioAmbients rules, denoted \mathcal{B} , is obtain by simple rewriting of the original BioAmbients rules in the Poly★ syntax. Only communication and movement rules has to be described. These are exactly those

rules from Figure 3 that have no premise. We don't present the exact definition here because it is straightforward. It, however, can be found in Appendix A in Section A.1, written in the syntax of input files of the implementation. The version in Section A.1 also handles the choice operator, as described below. From the set of descriptions \mathcal{B} , the reduction relation $\xrightarrow{\mathcal{B}}$ is derived. Again, this relation exactly corresponds to those from Figure 3, the only one difference is that it refers to the **Meta*** structural equivalence \equiv_{ν} instead of to the structural equivalence of BioAmbients \equiv . Correctness of the translation can be expressed as follows.

Proposition 5.2 *Let A, B range over BioAmbients processes and P over **Meta*** processes.*

- (i) *For all A, B : $A \rightarrow B$ implies $\exists A_0 \equiv A, B_0 \equiv B: \ulcorner A_0 \urcorner \xrightarrow{\mathcal{B}} \ulcorner B_0 \urcorner$*
- (ii) *For all A, P : $\ulcorner A \urcorner \xrightarrow{\mathcal{B}} P$ implies $\exists B: \ulcorner B \urcorner = P$*

In Section 3.2 we have already mentioned that for reasons of analysis, FABA assigns ambient identities $\mu \in \text{Ambient}$ to ambients. In FABA, the assignment of the identity μ to the ambient is written as $[P]^{\mu}$. We encode ambients identities as **Meta*** names, and translate ambients with identities to the usual syntax of Mobile Ambients as follows.

$$\ulcorner [P]^{\mu} \urcorner = \ulcorner \mu \urcorner [] . \ulcorner P \urcorner$$

FABA works with a version of BioAmbients that supports choice and the `rec` operator. The choice operator is translated as proposed in Section 5.4.

$$\ulcorner P_0 + P_1 + \dots + P_k \urcorner = \text{ch}.(P_0 \mid P_1 \mid \dots \mid P_k)$$

In Section 5.4 we have already noted that this translation leads to over approximation, but still, it is good enough for reasons of comparison with FABA. This is because FABA handles '+' in the same way as '|'. Although we haven't yet incorporated the `rec` operator into **Poly***, we can informally provide a translation that handles '`rec X`' as a **Meta*** form, and a process variable X as a **Meta*** name as follows.

$$\ulcorner \text{rec } X.P \urcorner = \text{rec } X. \ulcorner P \urcorner \quad \ulcorner X \urcorner = X.0$$

A principal typing for a **Meta*** process given by this translation, and by BioAmbients reduction rules, provides in general more precious information than a result of FABA analysis. Nevertheless, there exist parameters of the **Poly*** type inference algorithm with which it produces exactly the same result. They are as follows, written in the syntax of an input file of the type inference algorithm implementation.

```
option { showed no }
option { usemarks no }
option { smash yes }
option { sequence no, sequence yes x[] }
```

The parameter ‘`showred no`’ turns off displaying of *subtyping edges* (called *red edges*) that have no counterpart in a FABA result. The parameter ‘`smash yes`’ turns on *smashing* of edges. This forces the following property to hold: whenever a shape predicate contains two edges labeled with the same form type⁴, then they have the same destination node. The last parameter turns off *sequencing* for all edges that are not labeled by an ambient form type, i.e., a form type of the shape ‘ $x []$ ’. It means that all edges that are not labeled by an ambient form type are forced to be self-loops, i.e., to have the same source and destination node. A full description of parameters of the type inference algorithm can be found in a description of the syntax of its input files.

A result of the FABA analysis for an input term P is a pair $(\mathcal{I}, \mathcal{R})$, as already described in Section 3.2. A relation between \mathcal{I} of FABA and a Poly \star typing τ inferred by the type inference algorithm for P is as follows. It holds that, $u \in \mathcal{I}(\star)$ if and only if there is an edge from the root node of τ , labeled with u in the case when u is a capability, or labeled with $u []$ for the case when u is an ambient. Also, $u \in \mathcal{I}(\mu)$ if and only if there is an edge in τ of the form $X \xrightarrow{\mu[]} Y$, and an edge from Y labeled with u or by $u []$ respectively. Informally it says that, when we draw \mathcal{I} as a graph with nodes labeled with u ’s, and if we draw τ as a graph where a label of each edge is displayed in its destination node, then we obtain the same graphs.

This is illustrated on an example of ‘transcriptional regulation by positive feedback’ from the FABA paper [NNPR07] in Appendix A. Appendix A contains a full input file for the implementation including a term to be analyzed and descriptions of reduction rules (Section A.1), the set \mathcal{R} from the result of FABA (Section A.2), the Poly \star type inferred by the inference algorithm (Figure 12), and a graph representation of \mathcal{I} from the result of FABA (Figure 13). Two graphs in Figure 12 and Figure 13 differs in the way that in the case of Poly \star type, all leaves are merged together into yellow boxes labeled with ‘LOOP’. Yellow boxes also contain the form type ‘ch’ used to encode the choice operator. These can be ignored. Another difference is that the Poly \star type contains green *flow edges*. These were not described in this report. These are described in the Poly \star technical report [MW04a] in Section 3.2 (page 15). Nevertheless, we present them in the graph because they correspond to the set \mathcal{R} of the FABA result. Whenever $v' \in \mathcal{R}(v)$ then either $v = v'$ or there is a green box labeled with ‘ $v := \{v'\}$ ’. The label in a green box corresponds to a flow edge labeled with a type substitution. When $v' \in \mathcal{R}(v)$, it means that, the name v may be instantiated with v' as a result of communication. The Poly \star type also provides information where this communication may happened (by placement of the green label). This information is not expressed in the FABA result.

We have not yet developed a formal proposition that describes the relations described above. This is mainly for two reasons. At first, there are technical difficulties in comparison, caused by FABA’s handling of canonical capabilities. Secondly, our aim was to show that Poly \star can be used to flow analysis that is comparable with FABA, rather than to develop formalisms that allow formal

⁴In particular, labeled with form types that are related by \approx from Section 4.7.

description of relations between them. Still, the formal description is left as the future work.

5.8 Other work on process calculi in Poly★

The other work on process calculi in Poly★ covers capturing of properties described by another Mobile Ambients type system [CGG99] directly in Poly★. This type system can be used to check the property that some particular ambient will stay immobile, i.e., no in/out capability will be executed inside an immobile ambient. It can also be used to check the property that some ambient can never be opened. An alternative way to describe these properties were developed directly in Poly★, by extending Mobile Ambients reduction rules. Last but not least, an encoding of the Brane Calculus [Car04] in Poly★ was proposed.

6 Conclusions and future work

In this report we have presented some of the work of the author of this report done during the first year of his PhD study at Heriot-Watt University. Section 2, Section 3, and Section 4 provide general overview of a problematic studied during the first year. Section 5 describes specific issues and results studied and developed during the first year. Overview of the work and results can be found in the introduction of Section 5. Section 5.2, Section 5.3, and Section 5.4 describe the work done on generic properties of process calculi. This work should allow future extensions of the Poly★ system. Section 5.5 describes the changes done in the implementation of the Poly★ type inference algorithm, and other tools developed for reasons of other research. Section 5.6 and Section 5.7 describe specific results of comparison of Poly★ to related systems. Finally, Section 5.8 briefly describes the other work not presented in this report.

The rest of this section provides a list of possible topics of the future work. Some of them were already mentioned in previous sections. Section 6.1 provides a time plan of the future work.

- The first aim is to finish the comparison described in Section 5.6. This work should culminate in a conference paper that presents the results.
- The related work is to finish the comparison with FABA from Section 5.7 and present the results in a formal way. There also exists a successor of the FABA analysis [NNP04] that has a better support for the choice operator, and thus, in some cases, it provides better results than Poly★. A better support for choice in Poly★ is mentioned in the next item.
- Another topic of the future work is to incorporate the `rec` operator into the Poly★ formal theory. This was already mentioned in Section 5.3. This is also necessary in order to provide better results in the previous point. Another thing that should provide better results with respect to the comparison with FABA, and with respect to flow analysis in general, is incorporating of the choice operator, as already discussed in Section 5.4.

- One of aspects that has not been discussed yet is a presentation of results. Although it is not the primary aim of the Poly★ project, it seems that it is necessary to present results of the type inference in well arranged and easy to take in form. In particular, results of flow analysis for huge biological systems is very hard to read in the current version of Poly★. For example, the full Poly★ analysis of the term from Appendix A has to be printed on three A4 pages in order to recognize edges in details. For large graphs, several layout algorithms can be tried. For comparison, we present a graph for the example from Appendix A, produced by one of alternative layout systems, in Figure 14. This is the same graph as in Figure 12, but generated by the tool *circo* instead of the standard *dot* that we use. Both tools *circo* and *dot* comes from the graph visualization package Graphviz [EGK⁺01]. It seems that, the layout of the graph in Figure 14 presents a topology of the graph in a better arranged way. Thus it seems that, it should be useful to spent some time by an investigation of possibilities.

Another related thing is a user-friendly web interface. Poly★ already has one working web interface. But, this is not suitable for a presentation of more complex types. For the future, we would like to produce a web interface with more features related to the presentation of results.

- The Poly★ type inference algorithm comes with many parameters that can be used to adapt its expressiveness for specific needs. Some of them have been already mentioned in Section 5.7. Some of them are not presented in the Poly★ theory, and their properties has not been studied from the theoretical point of view. Incorporate them into the Poly★ theory is left for the future.
- The last point mentioned in this section are ways of increasing Poly★ expressiveness and ways of improving of preciseness of a type inference. Expressiveness can be increased by, for example, some form of higher-order communication, similar to the one found in HO π mentioned in Section 2.3. Preciseness of the type inference can be improved, for example, by better handling of the ν operator, or by some kind of *counting*. Counting should allow Poly★ types to express, for example, the property that ‘there is no more than one copy of an ambient *a* at the top-level’. In the current state of art, Poly★ types can just grant a property of the form ‘there is no ambient *a* at the top-level’.

Last but not least, there is also the aim to find out ways how to present the theory in more simple, more compact, and more readable ways.

6.1 Time plan

Based on the list of possible topics of the future work above, we provide the following research steps to be taken during the remaining PhD.

- **(now - May 2008)** Finish the comparison of Poly★ and TMA.

- (May 2008 - Feb 2009) Extend Poly* with recursion (and/or with choice) and find out ways to better present results of type inference.
- (Feb 2009 - Sep 2009) Choose one way to extend Poly* expressiveness or to improve type inference preciseness as mentioned above, incorporate it into the Poly* theory, and implement it.
- (Sep 2009 - Feb 2010) Write and submit a PhD thesis.

A Analysis: Transcriptional regulation by positive feedback

A.1 A term to be analyzed

```

active { P in a[P] }

reduce {
  a["ch".(T|"enter" n.P) | Q] | b["ch".(U|"accept" n.R) | S]
  --> b[a[P|Q]|R|S] }
reduce {
  a[b["ch".(T|"exit" n.P) | Q] | "ch".(U|"expel" n.R) | S]
  --> b[P|Q] | a[R|S] }
reduce {
  a["ch".(T|"mergеп" n.P) | Q] | b["ch".(U|"mergem" n.R) | S]
  --> a[P|Q|R|S] }
reduce {
  "ch".(T|"p2p" n<m...>.P) | "ch".(U|"p2p" n(p...).Q)
  --> (P{|p...:=m...}Q) }
reduce {
  "ch".(T|"p2c" n<m...>.P) | a["ch".(U|"c2p" n(p...).Q) | R]
  --> P | a[{|p...:=m...}Q|R] }
reduce {
  a[R|"ch".(T|"c2p" n<m...>.P)] | "ch".(U|"p2c" n(p...).Q)
  --> a[R|P] | {|p...:=m...}Q }
reduce {
  a[R|"ch".(T|"s2s" n<m...>.P)] | b["ch".(U|"s2s" n(p...).Q)|S]
  --> a[R|P] | b[{|p...:=m...}Q|S] }

option { showred no }
option { usemarks no }
option { smash yes }
option { sequence no, sequence yes x[] }

term {
  GeneA[
    rec X1.ch.(

```

```

s2s basal(x2).ch.expel a.X1 |
s2s pa (x1).ch.expel a.X1 )
|
RNAa[
  rec X2.ch.exit a.ch.(
    s2s utr(x4).ch.expel b.X2 |
    s2s degm(x3).0 )
  |
  ProteinA[
    ch.exit b.rec X3.ch.accept tf.ch.(
      p2c bb1<d>.(ch.(expel g.X3) | X3) |
      s2s degp(x6).ch.p2c bb3<d>.ch.p2c bb3<d>. 0 |
      s2s degp(x7).ch.p2c bb3<d>. 0 )
    |
    Kinase[
      rec X4.ch.(
        s2s bb2<d>.X4 |
        c2p bb3(x5).0 )
      ]
    ]
  ]
]
|
GeneTF[
  rec X5.ch.(
    s2s basal(y2).ch.expel c.X5 |
    s2s pa(y1).ch.expel c.X5 )
  |
  RNAatf[
    rec X6.ch.exit c.ch.(
      s2s utr(y4).ch.expel e.X6 |
      s2s degm(y3).0 )
    |
    ProteinTF[
      ch.exit e.ch.enter tf.ch.expel atf.
      ch.accept atf.0
      |
      BoundTF[
        ch.exit atf.ch.(
          c2p bb1(y9).ch.enter atf.0 |
          c2p bb3(y8).0 |
          s2s bb2(y7).ch.(
            c2p bb1(y6).ch.expel f.0 |
            c2p bb3(y5).0 ) )
        |
        ActiveTF[

```

```

ch.exit f.ch.exit g.rec X7.ch.(
  s2s ptail<d>.X7 |
  s2s degp(y10).0 )
]
]
]
]
|
Transcr[ rec X8.ch.(
  s2s basal<d>.X8 |
  s2s ptail(z1).ch.s2s pa<d>.X8)
]
|
Transl[ rec X9.ch.s2s utr<d>.X9 ]
|
RNAdeg[ rec X10.ch.s2s degm<d>.X10 ]
|
ProteinDeg[ rec X11.ch.s2s degp<d>.X11 ]
}

```

A.2 A set \mathcal{R} of the FABA analysis

$$\mathcal{R} = \{ \begin{array}{l} (x_1, d), (x_2, d), (x_3, d), (x_4, d), (x_5, d), (x_6, d), (x_7, d), \\ (y_1, d), (y_2, d), (y_3, d), (y_4, d), (y_5, d), (y_6, d), (y_7, d), (y_8, d), (y_9, d), \\ (y_{10}, d), (z_1, d), \\ (ptail, ptail), (atf, atf), (tf, tf), (degp, degp), (utr, utr), (pa, pa), \\ (basal, basal), (bb3, bb3), (bb2, bb2), (bb1, bb1), (g, g), (f, f), (e, e), (d, d), \\ (c, c), (b, b), (a, a) \end{array} \}$$

References

- [AKPG00] Torben Amtoft, Assaf J. Kfoury, and Santiago M. Pericas-Geertsen. What are polymorphically-typed ambients? Technical Report BUCS-TR-2000-021, Comp. Sci. Dept., Boston Univ., December 2000.
- [AKPG01] Torben Amtoft, Assaf J. Kfoury, and Santiago M. Pericas-Geertsen. What are polymorphically-typed ambients? In David Sands, editor, *ESOP 2001, Genova*, volume 2028 of *LNCS*, pages 206–220. Springer-Verlag, April 2001. An extended version appears as [AKPG00].

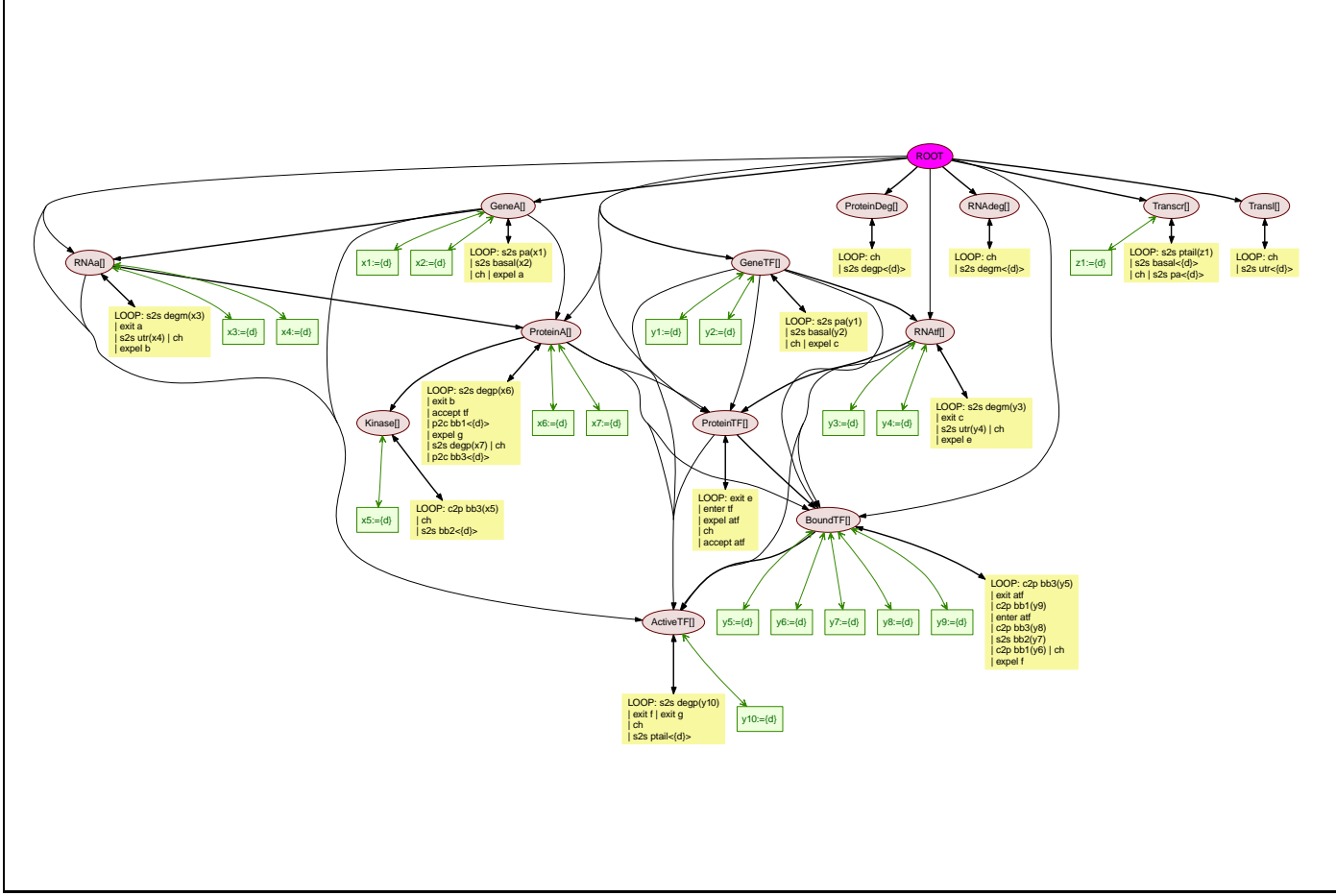


Figure 12: Poly★ shape predicate for the term from Section A.1 (rendered by Graphviz dot).

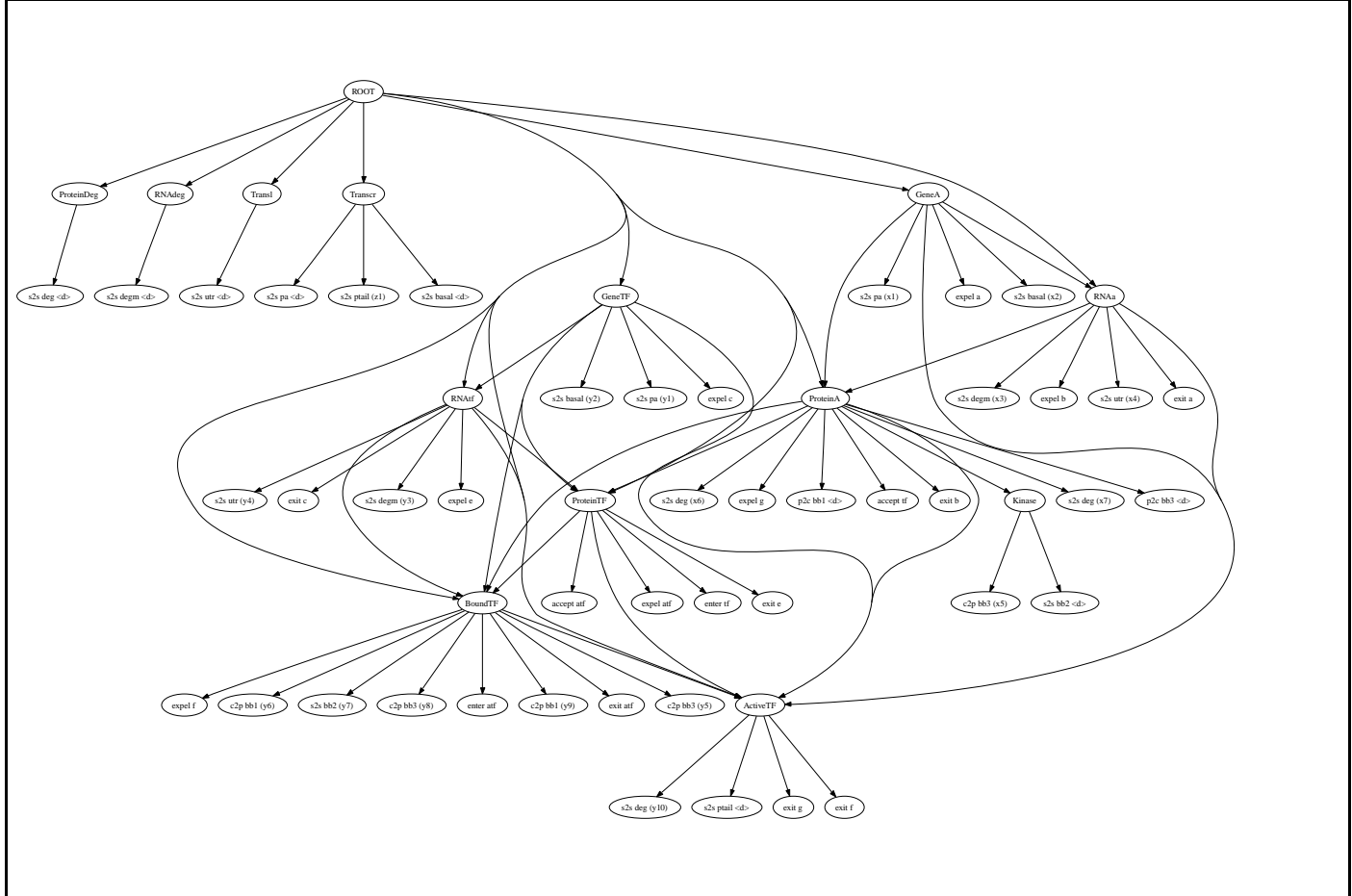


Figure 13: A set \mathcal{I} from the result of the FABa analysis, presented as a graph.

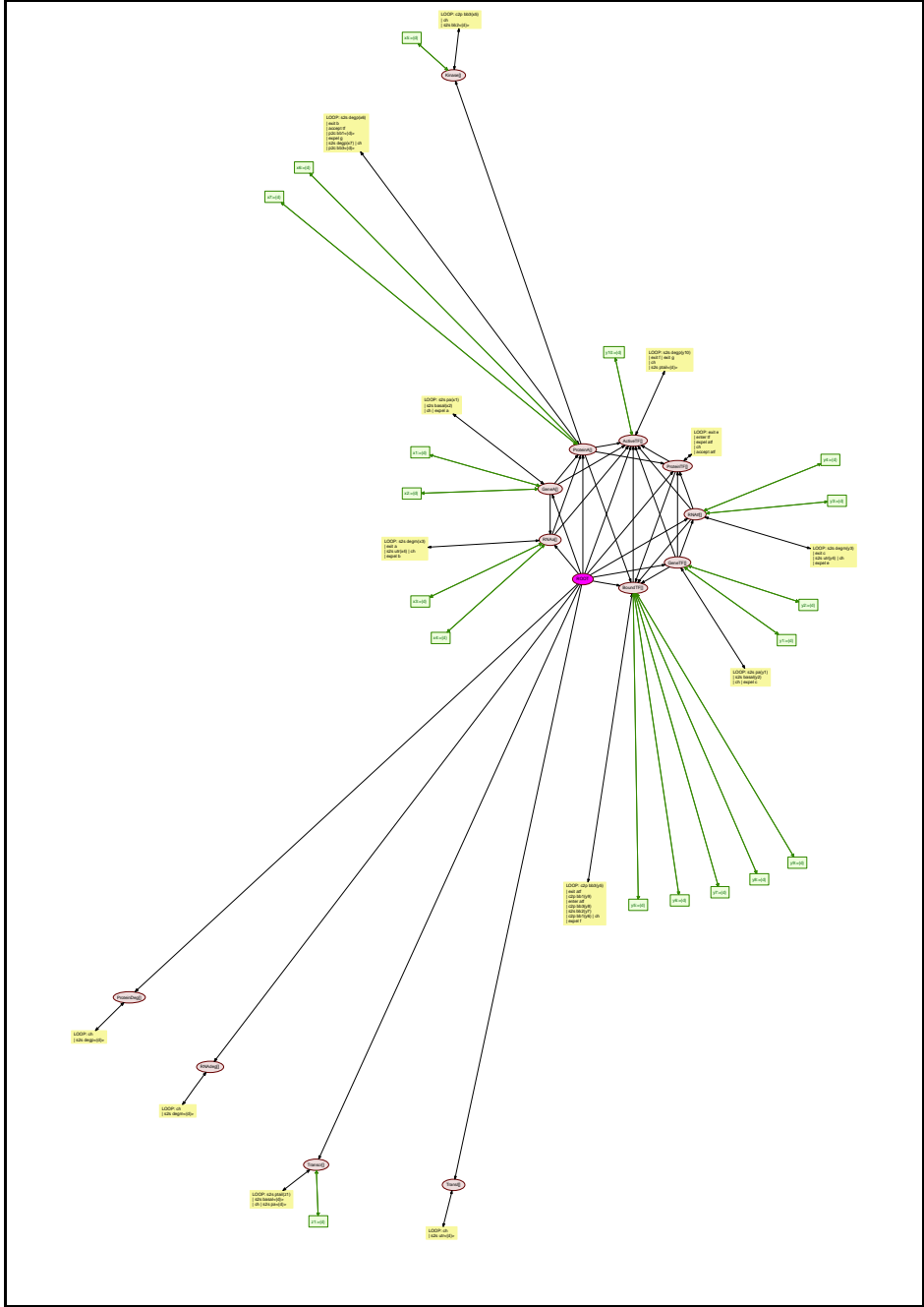


Figure 14: Poly* shape predicate for the term from Section A.1, i.e., the same as in Figure 12 (rendered by Graphviz circo).

- [AKPG02] Torben Amtoft, Assaf J. Kfoury, and Santiago M. Pericas-Geertsen. Orderly communication in the ambient calculus. *Computer Languages*, 28:29–60, 2002.
- [AMW04a] Torben Amtoft, Henning Makhholm, and J. B. Wells. PolyA: True type polymorphism for Mobile Ambients. Technical Report HW-MACS-TR-0015, Heriot-Watt Univ., School of Math. & Comput. Sci., February 2004. A shorter successor is [AMW04b].
- [AMW04b] Torben Amtoft, Henning Makhholm, and J. B. Wells. PolyA: True type polymorphism for Mobile Ambients. In *IFIP TC1 3rd Int'l Conf. Theoret. Comput. Sci. (TCS '04)*, pages 591–604. Kluwer Academic Publishers, 2004. A more detailed predecessor is [AMW04a].
- [AW02] Torben Amtoft and J. B. Wells. Mobile processes with dependent communication types and singleton types for names and capabilities. Technical Report 2002-3, Kansas State University, Department of Computing and Information Sciences, December 2002.
- [Bae05] Jos C. M. Baeten. A brief history of process algebra. *Theoret. Comput. Sci.*, 335(2-3):131–146, 2005.
- [BCC01] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Boxed ambients. In *4th International Conference on Theoretical Aspects of Computer Science (TACS'01)*, volume 2215 of *LNCS*, pages 38–63. Springer-Verlag, 2001.
- [Bek84] Hans Bekič. Towards a mathematical theory of processes. In Cliff B. Jones, editor, *Programming Languages and Their Definition*, volume 177 of *LNCS*. Springer-Verlag, 1984.
- [BK84] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.
- [BZ04] Nadia Busi and Gianluigi Zavattaro. On the expressive power of movement and restriction in pure mobile ambients. *Theoret. Comput. Sci.*, 322(3):477–515, 2004.
- [Car04] Luca Cardelli. Brane calculi. In Vincent Danos and Vincent Schächter, editors, *CMSB*, volume 3082 of *LNCS*, pages 257–278. Springer-Verlag, 2004.
- [CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proc. FoSSaCS '98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

- [CG99] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pages 79–92, 1999.
- [CGG99] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In Jiri Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *ICALP'99*, volume 1644 of *LNCS*, pages 230–239. Springer-Verlag, July 1999. Extended version appears as Microsoft Research Technical Report MSR-TR-99-32, 1999.
- [CGG00] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In *IFIP International Conference on Theoretical Computer Science (IFIP TCS2000)*, Tohoku University, Sendai, Japan, volume 1872 of *LNCS*, pages 333–347. Springer-Verlag, August 2000.
- [CGN01] Giuseppe Castagna, Giorgio Ghelli, and Francesco Zappa Nardelli. Typing mobility in the Seal calculus. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR*, volume 2154 of *LNCS*, pages 81–101. Springer-Verlag, 2001.
- [EGK⁺01] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2001.
- [Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
- [IK01] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Conf. Rec. POPL '01: 28th ACM Symp. Princ. of Prog. Langs.*, pages 128–141, 2001.
- [KPT96] N. Kobayashi, Benjamin C. Pierce, and D.N. Turner. Linearity and the pi-calculus. In *POPL'96, St. Petersburg Beach, Florida*, pages 358–371. ACM Press, January 1996.
- [LS00] Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *POPL'00, Boston, Massachusetts*, pages 352–364. ACM Press, January 2000.
- [Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.
- [Mil99] Robin Milner. *Communicating and Mobile Systems: The π -Calculus*. Cambridge Press, 1999.
- [MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part i & ii. *Inform. & Comput.*, 100(1):1–77, 1992.

- [MW04a] Henning Makholm and J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. Technical Report HW-MACS-TR-0022, Heriot-Watt Univ., School of Math. & Comput. Sci., November 2004. A shorter successor is [MW05].
- [MW04b] Henning Makholm and J. B. Wells. Type inference for PolyA. Technical Report HW-MACS-TR-0013, Heriot-Watt Univ., School of Math. & Comput. Sci., January 2004.
- [MW05] Henning Makholm and J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. In *Programming Languages & Systems, 14th European Symp. Programming*, volume 3444 of *LNCS*, pages 389–407. Springer-Verlag, 2005. A more detailed predecessor is [MW04a].
- [MZ03] Ines Margaria and Maddalena Zacchi. A filter model for safe ambients. In Furio Honsell, Marina Lenisa, and Marino Miculan, editors, *Proc. Final Workshop Project COMETA*, ENTCS, Udine, Italy, December 2003. Elsevier/Forum.
- [NNP04] Hanne Riis Nielson, Flemming Nielson, and Henrik Pilegaard. Spatial analysis of BioAmbients. In Roberto Giacobazzi, editor, *Static Analysis: 11th Int'l Symp.*, volume 3148 of *LNCS*, pages 69–83, Verona, Italy, August 26–28 2004. Springer-Verlag.
- [NNPR07] Flemming Nielson, Hanne Riis Nielson, Corrado Priami, and Debora Rosa. Control flow analysis for bioambients. *ENTCS*, 180(3):65–79, 2007. A preliminary version appeared at Bio-CONCUR 2003.
- [NNS⁺04] Flemming Nielson, Hanne Riis Nielson, Hongyan Sun, Mikael Buchholtz, René Rydhof Hansen, Henrik Pilegaard, and Helmut Seidl. The succinct solver suite. In *TACAS*, volume 2988 of *LNCS*, pages 251–265. Springer-Verlag, 2004.
- [Par01] Joachim Parrow. An introduction to the π -calculus. In *Handbook of Process Algebra*, pages 479–543. North-Holland, Amsterdam, 2001.
- [Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.
- [PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report, IU, 1997.
- [PV05] Catuscia Palamidessi and Frank D. Valencia. Recursion vs replication in process calculi: Expressiveness. *Bulletin of the EATCS*, 87:105–125, 2005.

- [RPS⁺04] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Shapiro. Bioambients: An abstraction for biological compartments. *Theoret. Comput. Sci.*, 325(1):141–167, September 2004.
- [San93] Davide Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT*, volume 668 of *LNCS*, pages 151–166. Springer-Verlag, 1993.
- [VC99] Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, volume 1686 of *LNCS*. Springer-Verlag, 1999.
- [Wel02] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.