

A Second Year Report

The Poly★ System:

from Poly★ v1 to Poly★ v2

Jan Jakubův

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

March 24, 2009

Contents

1 Overview of the Report	2
1.1 Notations and Preliminaries	2
2 Introduction to Process Calculi and Poly*	3
2.1 Concurrent Systems and Process Calculi	3
2.2 Type Systems for Process Calculi	5
2.3 Brief Overview of Poly* v2	7
2.3.1 Syntax of Meta* Processes	8
2.3.2 Specifying Semantics by Instantiating Meta*	10
2.3.3 Emulating Process Calculi in Meta*	12
2.3.4 Poly* Shape Predicates and Types	12
2.3.5 Comparing Poly* with Related Type Systems	14
3 Differences Between Poly* v1 and Poly* v2	15
3.1 Basic Names and α -Renaming	15
3.2 Free Names, Bound Names, and Well-Scopedness	16
3.3 Process Substitution and Structural Equivalence	17
3.4 Rewriting Rules and Rewriting Relation	18
3.5 Poly* Shape Predicates and Name Restriction	18
3.6 Summary of Changes Between Poly* v1 and Poly* v2	19
4 On the Way from Poly* v1 to Poly* v2	20
4.1 Motivating Example	20
4.2 Solution Doubling ν -bound Names	21
4.3 Problem with Sending of ν -bound Names	23
4.4 Solution with Basic Names	23
5 Conclusion	24
6 Future Work	25
A Meta* With α-equivalent Processes Unified	25
B Additional Restrictions on Rewriting Rules	27
C Syntactically Closed Shape Predicates	29
D Draft of the Paper	31

1 Overview of the Report

This report is intended to describe the work done by Jan Jakubův during the second year of his PhD study on Heriot-Watt University. His work is related to the Poly★ system. The aim set by the previous First Year Report [Jak08] was to finish a paper that provides a comparison of Poly★ with a related system. Nevertheless it turned out that the version of Poly★ assumed in the First Year Report is not optimal for this purpose. It turned out that some changes and extensions to Poly★ are necessary to enable more direct and intuitive comparison with other systems. Thus Jan’s work done during the second year can be divided into two parts. At first some work was done on the paper that compares Poly★ and the related system. This paper in its current and not yet finished state is attached to this report in Appendix D. Second part of Jan’s work consists of changes and extensions to the Poly★ system. The necessity of these changes and the nonfulfilment of the aim set by the First Year Report are further advocated in the conclusion in Section 5.

While this report is related to the Poly★ project Section 2 provides brief, but for the reasons of this report complete, introduction to process calculi and Poly★. The version of Poly★ described in Section 2.3 is the version that was developed during the second year. This version is in this report referred as Poly★ v2. The former version which is described in the First Year Report and in the Poly★ papers [MW05, MW04] is in this report referred as Poly★ v1. Differences between Poly★ v1 and Poly★ v2 are described in Section 3. Section 4 tries to provide some insight into motivations of the changes and into circumstances which lead to them.

Section 5 contains the already mentioned conclusion. It summarizes and advocates the work done during the second year and links to other parts of the report which provide detailed explanations. Section 6 provides a description of plans for the future work on the PhD thesis. It includes the work to be finished before the writing of the thesis can begin and an approximate submission date. The rest of this section describes basic notations used in the paper.

1.1 Notations and Preliminaries

Metavariables i, j, k range over the set of natural numbers denoted \mathbf{Nat} . We shall use one of the following statements (in this case with the same meaning) to express similar claims:

$$\begin{aligned} i, j, k \in \mathbf{Nat} & ::= 0 \mid 1 \mid 2 \mid \dots \\ i, j, k \in \mathbf{Nat} & = \{0, 1, 2, \dots\} \end{aligned}$$

$\mathcal{P}_{\text{fin}}(U)$ is the set of all finite subsets of a set U , and ‘\’ denotes set subtraction. A function f is a set of pairs such that $(u, v) \in f$ and $(u, w) \in f$ implies $v = w$. Let $u \mapsto v$ be an alternate notation for pairs used for pairs in functions. Given the function f and the sets U and V we suppose the following definitions:

$\text{dom}(f) = \{u \mid (u \mapsto v) \in f\}$	function domain
$U \rightarrow V = \{f \subseteq (U \times V) \mid f \text{ is a function}\}$	all functions from U to V
$U \xrightarrow{\text{fin}} V = \{f \in (U \rightarrow V) \mid f \text{ is finite}\}$	all finite functions from U to V
$U \triangleleft f = \{u \mapsto v \mid (u \mapsto v) \in f \ \& \ u \in U\}$	domain restriction
$f[u \mapsto v] = ((\text{dom}(f) \setminus \{u\}) \triangleleft f) \cup \{u \mapsto v\}$	function extension/replacement

2 Introduction to Process Calculi and Poly★

2.1 Concurrent Systems and Process Calculi

A concurrent system is any system where several interacting units engage in activity at the same time. Units are called processes or agents. Several formalisms intended to describe concurrent systems in Computer Science were developed. The most important properties that these formalisms are trying to capture are notions of interaction and concurrency. The main point is to describe how processes interact and how their interaction affects the behavior of engaged processes and the rest of the system. Many of the developed formalisms describe interaction of processes in computers and computer networks. Nevertheless there are also formalisms used to describe business, chemical, or biological systems.

Process calculi usually denote applications of rewriting systems intended to describe concurrency. They form one family of approaches to model concurrent systems. Other formalisms include for example Petri nets which models concurrent systems using directed graphs, and process algebras¹ which use algebraic equations to describe the behavior of a system. In the rest of this section we describe a well-known process calculi published by L. Cardelli and A. D. Gordon called Mobile Ambients (MA) [CG98]. We describe it here because it is closely related to the second year’s work of the author.

In MA processes are placed inside bounded locations called ambients. An ambient can contain other (child) ambients and thus ambients form a hierarchical tree of locations. Processes can execute instructions which allow them to change and move in the ambient hierarchy. Additionally, processes inside the same ambient can communicate by sending and receiving sequences of instructions (or their pars) to be executed.

Every ambient has the name associated with it. These names are used to refer to ambients from instructions. Instructions are also called capabilities or expressions. There are three kinds of expressions: `in`, `out`, and `open`. Execution of an expression inside a process always instructs the ambient that surrounds the process to take an appropriate action. The expression ‘`in n` ’ instructs the surrounding ambient to move itself along with its content into the ambient n provided that n is a sibling ambient to the surrounding ambient. Similarly ‘`out n` ’ instructs the surrounding ambient to move out of the parent ambient n and

¹Terminology may vary and sometimes process calculi are used as a synonym for process algebras.

<i>Syntax of MA entities:</i>		
$n, m \in \text{AName}$	$::= a \mid b \mid c \mid d \mid \dots$	
$N \in \text{AExpression}$	$::= \varepsilon \mid n \mid \text{in } N \mid \text{out } N \mid \text{open } N \mid N.N'$	
$B \in \text{AProcess}$	$::= 0 \mid (B_0 \mid B_1) \mid N[B] \mid N.B \mid !B \mid \langle N_1, \dots, N_k \rangle \mid (n_1, \dots, n_k).B \mid (\nu n)B$	
<i>Structural equivalence of MA:</i>		
$!0 \equiv 0$	$(\nu n)0 \equiv 0$	$(N.N').B \equiv N.N'.B$
$\varepsilon.B \equiv B$	$!B \equiv B \mid !B$	$(\nu n)(\nu m)B \equiv (\nu m)(\nu n)B$
$B \mid 0 \equiv B$	$B_0 \mid B_1 \equiv B_1 \mid B_0$	$B_0 \mid (B_1 \mid B_2) \equiv (B_0 \mid B_1) \mid B_2$
$\frac{n \neq m}{(\nu n)(m[B]) \equiv m[(\nu n)B]}$		$\frac{n \notin \text{fn}(B_0)}{B_0 \mid (\nu n)B_1 \equiv (\nu n)(B_0 \mid B_1)}$
<i>Rewriting relation of MA:</i>		
$n[\text{in } m.B_0 \mid B_1 \mid m[B_2]] \rightarrow m[n[B_0 \mid B_1] \mid B_2]$		
$m[n[\text{out } m.B_0 \mid B_1] \mid B_2] \rightarrow n[B_0 \mid B_1] \mid m[B_2]$		
$\text{open } n.B_0 \mid n[B_1] \rightarrow B_0 \mid B_1$		
$(n_1, \dots, n_k).B \mid \langle N_1, \dots, N_k \rangle \rightarrow B\{n_1 \mapsto N_1, \dots, n_k \mapsto N_k\}$		
$\frac{B_0 \rightarrow B_1}{(\nu n)B_0 \rightarrow (\nu n)B_1}$	$\frac{B_0 \rightarrow B_1}{n[B_0] \rightarrow n[B_1]}$	$\frac{B_0 \rightarrow B_1}{B_0 \mid B_2 \rightarrow B_1 \mid B_2}$
$\frac{B'_0 \equiv B_0 \quad B_0 \rightarrow B_1 \quad B_1 \equiv B'_1}{B'_0 \rightarrow B'_1}$		

Figure 1: Syntax and semantics of Mobile Ambients (MA).

become its sibling. The expression ‘open n ’ instructs the surrounding ambient to dissolve the boundary of its child ambient n . This moves the former content of n directly to the surrounding ambient. Expressions can be composed to form sequences where the left most expression is to be executed first. Execution of an expression consumes the expression.

Processes in the same ambient can communicate by exchanging messages. A message is either an ambient name or a (possibly empty) sequence of expressions. Communication in MA is asynchronous which means that a sending process does not engage in activity after sending a message. A receiving process receives a message, substitutes it for designed placeholders, and continues as this newly created process. Communication in MA is polyadic which means that several messages can be sent and received at the same time.

The set **AProcess** of MA processes is generated by the grammar from the top part of Figure 1. The process ‘0’ is a finished inactive process. The operator ‘|’ is used to denote parallel composition of processes, ‘ $N[B]$ ’ denotes the process B running inside the ambient N , ‘.’ denotes sequential composition or prefixing with an expression, and the process $!B$ behaves like infinitely many copies of B in parallel. The process ‘ $\langle N_1, \dots, N_k \rangle$ ’ sends a k -tuple of messages while ‘ $(n_1, \dots, n_k).B$ ’ waits for a k -tuple of messages, receives them, and continues as an appropriately modified B . Finally ‘ $(\nu n)B$ ’ restricts the scope of the name

n to B , that is, makes it different from all other occurrences of n outside of B .

Note that the syntax also allows expressions like ‘in (out a)’ and ‘(in a)[0]’. That is because a substitution in MA substitutes expressions for names and thus this benevolence of the syntax allows application of a substitution to be always defined. This is just a technical simplification and the semantics does not give any meaning to similar expressions (that is to ‘in N ’ and ‘ $N[B]$ ’ where N is not a single name).

Any occurrence of n inside ‘ $(\nu n)B$ ’ is said to be (ν -)bound. Any occurrence of n_i inside ‘ $(n_1, \dots, n_k).B$ ’ is said to be (input-)bound. An occurrence of n inside B which is not bound is said to be free. The set $\text{fn}(B)$ is the set of all names which occur free in B . Bound names are subjects to α -renaming which allows all bound occurrences of some name to be renamed to another name provided no collisions with already present names occur. Processes are identified up to the consistent α -renaming of bound names. The expression ‘ $B\{n \mapsto N\}$ ’ denotes the process B with N substituted for all free occurrences of n .

The structural equivalence relation \equiv of MA is the smallest equivalence relation that is congruent with the process constructors and satisfies the rules from the middle part of Figure 1. It is intended to relate processes which are structurally similar. Basically it expresses that ‘|’ is a commutative and associative operator with ‘0’ as its unit. Additionally it expresses a variable scope of name restriction and a repetitive behavior of replication. The relation \equiv is used to simplify the definition of the rewriting relation.

Finally the rewriting relation \rightarrow of MA is presented in the bottom part of Figure 1. The first three axioms describe in turn the execution of in, out, and open expressions. The fourth axiom describes communication inside an ambient. The other inference rules express that rewriting can be also applied under the name restriction operator, inside an ambient, and to a one part of a parallel process. The last inference rule incorporates structural equivalence into rewriting.

2.2 Type Systems for Process Calculi

Type systems in process calculi are used in different ways to achieve various aims. Usually types denote chosen properties of processes. These properties are chosen by the designer of a type system and reflect particular needs. For example common properties for MA are that a process does not perform any communication, or that a specific ambient never dissolves its boundary. The main work of a type system is then to verify that designed properties are preserved under rewriting. This is called the subject reduction property. From another point of view a type corresponds to a set of processes (the set of processes which have the given property associated with the type). Subject reduction then says that every set that corresponds to some type is closed under rewriting.

The rest of this section describes a seminal type system for MA presented by L. Cardelli and A. D. Gordon [CG99]. We refer to this type system as to TMA.

<i>Syntax of TMA types and processes:</i>		
$W \in$ AMessageType	$::= \text{Amb}[T] \mid \text{Cap}[T]$	
$T \in$ AExchangeType	$::= \text{Shh} \mid W_1 \times \dots \times W_k$	
$B \in$ AProcess	$::= \dots \mid (n_1 : W_1, \dots, n_k : W_k).B \mid (\nu n : W)B$	
$E \in$ Environment	$= \text{AName} \xrightarrow{\text{fn}} \text{AMessageType}$	
<hr/> <i>Typing rules of TMA:</i>		
$\frac{E(n) = W}{E \vdash n : W}$	$\frac{}{E \vdash \varepsilon : \text{Cap}[T]}$	$\frac{E \vdash N : \text{Cap}[T] \quad E \vdash N' : \text{Cap}[T]}{E \vdash N.N' : \text{Cap}[T]}$
$\frac{E \vdash N : \text{Amb}[T']}{E \vdash \text{in } N : \text{Cap}[T]}$	$\frac{E \vdash N : \text{Amb}[T']}{E \vdash \text{out } N : \text{Cap}[T]}$	$\frac{E \vdash N : \text{Amb}[T]}{E \vdash \text{open } N : \text{Cap}[T]}$
$\frac{E \vdash B : T}{E \vdash !B : T}$	$\frac{E \vdash N : \text{Cap}[T] \quad E \vdash B : T}{E \vdash N.B : T}$	$\frac{E \vdash N : \text{Amb}[T] \quad E \vdash B : T}{E \vdash N[B] : T'}$
$\frac{E \vdash B_0 : T \quad E \vdash B_1 : T}{E \vdash B_0 \mid B_1 : T}$	$\frac{}{E \vdash 0 : T}$	$\frac{E \vdash N_1 : W_1 \quad \dots \quad E \vdash N_k : W_k}{E \vdash \langle N_1, \dots, N_k \rangle : W_1 \times \dots \times W_k}$
$\frac{E[n_1 \mapsto W_1, \dots, n_k \mapsto W_k] \vdash B : W_1 \times \dots \times W_k}{E \vdash (n_1 : W_1, \dots, n_k : W_k).B : W_1 \times \dots \times W_k}$	$\frac{E[n \mapsto \text{Amb}[T]] \vdash B : T'}{E \vdash (\nu n : \text{Amb}[T])B : T'}$	

Figure 2: Syntax and typing rules of TMA.

The relation between TMA and the author's work is described in Section 4 and Section 5.

TMA is designed to recognize processes which perform communication correctly. Communication in MA can be incorrect basically in two ways. Firstly a sending process can send a compound expression while a receiver expects a single name and contrariwise. Secondly a sender can send a tuple of objects while a receiver expects a tuple of a different arity. TMA is designed to ensure that the above two situations never happen in processes that have some type (well typed processes).

TMA assigns exchange types to a processes. An exchange type describes expressions that are allowed to be send and received. Expressions are described using message types. Every ambient has an exchange type associated with it. This type describes processes allowed inside the ambient. Syntax of exchange and message types is presented in the top part of Figure 2. The exchange type Shh is the type of processes that do not execute any communication at all. The exchange type $W_1 \times \dots \times W_k$ is the type of processes that exchange k -tuples of messages where the i -th message has the message type W_i . The type $\mathbf{1}$ stands for $W_1 \times \dots \times W_k$ with $k = 0$. The message type $\text{Amb}[T]$ describes names of ambients where only processes of the exchange type T can be placed. As mentioned above each ambient in a process has one exchange type T associated with it. Then T is the type of processes allowed inside the ambient while $\text{Amb}[T]$ is the type of the name of the ambient. The message

type ‘ $\text{Cap}[T]$ ’ is the type of all expressions whose execution could bring some process of the type T into the executing ambient. Note that ‘ $\text{Amb}[T]$ ’ is both an exchange type (a tuple exchange type with $k = 1$) and a message type.

The syntax of TMA processes is similar to the syntax of MA processes. The only difference is that explicit type annotations of bound names are added to binders. The structural equivalence of TMA simply adapts to this change of the syntax. There are just two rules where the difference is not straightforward. Rules ‘ $(\nu n)(\nu m)B \equiv (\nu m)(\nu n)B$ ’ and ‘ $(\nu n)0 \equiv 0$ ’ are now as follows:

$$\frac{n \neq m}{(\nu n : W_0)(\nu m : W_1)B \equiv (\nu m : W_1)(\nu n : W_0)B} \quad \frac{}{(\nu n : \text{Amb}[T])0 \equiv 0}$$

There is an additional premise $n \neq m$ in the first rule. This premise is not necessary in MA because when $n = m$ then both processes in the rule’s conclusion are equal. That is not the case in TMA where they differ as long as W_0 differs from W_1 . The second rule says that the process ‘ $(\nu n : \text{Cap}[T])0$ ’ is not structural equivalent to 0. That is because only ν -bound names of some $\text{Amb}[T]$ type can legally occur in well typed processes. Thus when ‘ $(\nu n : \text{Cap}[T])0$ ’ was equivalent to 0 then a non-well typed process would be equivalent to a well typed process. This would cause problems with subject reduction. The rewriting relation of TMA is also similar to the one of MA. The communication rule simply ignores type annotations.

Typing rules of TMA are presented in the bottom part of Figure 2. Message types of free names are assigned using environments. Let \mathbf{a} have the message type $\text{Amb}[T]$. Execution of ‘open \mathbf{a} ’ can dissolve the boundary of \mathbf{a} and bring all processes running inside \mathbf{a} to the executing ambient. Because exchange types have to be preserved under rewriting TMA has to ensure that new processes exchange messages of correct types. Thus ‘open \mathbf{a} ’ can have the type $\text{Cap}[T]$ only when \mathbf{a} has the type $\text{Amb}[T]$. On the other hand ‘in \mathbf{a} ’ and ‘out \mathbf{a} ’ can have the type $\text{Cap}[T']$ for any T' because their execution can not bring any new process to the executing ambient. Expressions in a sequence have to have the same type. Any expression of the type $\text{Cap}[T]$ can be executed only by a process of type T . Communication inside an ambient can not directly effect processes in other ambients. That is why when B has the type T then $\mathbf{a}[B]$ can have an arbitrary type T' .

Note that a single name may also have a $\text{Cap}[T]$ type but only when input-bound (or when it is explicitly stated in an environment). On the other hand a ν -bound name can only have an $\text{Amb}[T]$ type. This is because a single name can meaningfully have a $\text{Cap}[T]$ type only when it is later instantiated by communication to some executable expression. That is not possible when the name is ν -bound. Finally note that ‘(in n) $[B]$ ’, ‘in (out n)’, and similar have no type and thus can never occur in a well typed process.

2.3 Brief Overview of Poly★ v2

The Poly★ system provides a generic type system for a large family of process calculi. It is build on the top of the metacalculus Meta★. Meta★ provides a

<i>Syntax of Meta★ entities:</i>		
$a \in$	BasicName	$::= a \mid b \mid \dots \mid \text{in} \mid \text{out} \mid \text{open} \mid \dots \mid [] \mid \bullet \mid \dots$
$x, y \in$	Name	$::= a^i$
$F \in$	Form	$::= x_0 \dots x_k$
$M \in$	Message	$::= F \mid 0 \mid M_0.M_1$
$E \in$	Element	$::= x \mid (x_1, \dots, x_k) \mid \langle M_1, \dots, M_k \rangle$
$A \in$	Action	$::= E_0 \dots E_k$
$P, Q, R \in$	Process	$::= 0 \mid A.P \mid (P \mid Q) \mid \nu(x).P \mid !P$
<hr/> <i>Meta★ structural equivalence:</i>		
$\frac{}{P \mid Q \equiv Q \mid P}$	(SCOM)	$\frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R}$
$\frac{}{P \mid 0 \equiv P}$	(SNLPAR)	$\frac{x \notin \text{fn}(A) \cup \text{bn}(A)}{A.\nu(x).P \equiv \nu(x).A.P}$
$\frac{x \notin \text{fn}(P)}{P \mid \nu(x).Q \equiv \nu(x).(P \mid Q)}$	(SNUACT)	(SNUPAR)
$\frac{}{\nu(x).\nu(y).P \equiv \nu(y).\nu(x).P}$	(SNUV)	$\frac{}{0 \equiv !0}$
$\frac{}{!P \equiv P \mid !P}$	(SNLREP)	(SBANG)

Figure 3: Syntax and structural equivalence of Meta★.

general syntax of processes which covers common constructions used in process calculi in the literature. Moreover Meta★ provides a way to describe particular rewriting rules and thus can be instantiated to a specific process calculus. Poly★ then provides a type system for every process calculus which Meta★ can be instantiated to. The following sections describe Meta★, Poly★, and their usage in more details.

2.3.1 Syntax of Meta★ Processes

The syntax of Meta★ processes is intended to be general enough to allow straightforward encoding of processes from other calculi. The syntax of Meta★ entities is described in the top part of Figure 3. The metavariable Z shall range over all Meta★ entities. A name can be seen as a pair of a basic name and a natural number. The point is to provide infinitely many variants for each basic name to be used when α -renaming names. Processes are build from the null process ‘0’ by prefixing with an action, by parallel composition ‘|’, by name restriction ‘ ν ’, and by replication ‘!’. The syntax of actions is intended to be general enough to allow us to encode MA capabilities, ambient boundaries, and other prefixes from various calculi.

We use actions to encode ambient boundaries from MA as follows. The construction ‘ $x_0 \dots x_k [P]$ ’ is an abbreviation for ‘ $x_0 \dots x_k [] . P$ ’ where ‘[]’ is a (single) special name. When no confusion can arise we write a instead of a^0 for some Meta★ basic name a . We omit parenthesis in ‘ $(P \mid Q)$ ’ when possible. Message composition and action prefix operations ‘.’ associates to the right (that is ‘ $a.b.c.0$ ’ stands for ‘ $a.(b.(c.0))$ ’).

All occurrences of the name x in ‘ $\nu(x).P$ ’ are said to be (ν -)bound. When the action A contains an element ‘ (x_1, \dots, x_k) ’ then all occurrences of the x_i ’s in ‘ $A.P$ ’ as well as in A on its own are said to be (input-)bound. An occurrence of

x that is not bound is said to be free. An occurrence of a basic name is bound (resp. free) when the occurrence of the corresponding name is bound (resp. free). Bound occurrences of names are subject to α -renaming with the following restriction. The name a^i can be α -renamed only to a^j with j arbitrary, that is, preserving the basic name part a . Processes are identified up to the consistent α -renaming of bound names. Formally, we work with classes of α -equivalence and the formal background is presented in Appendix A.

The set of free names of P is denoted $\text{fn}(P)$. The set of free basic names of P is denoted $\text{fbn}(P)$. Note that actions are identified up to the α -renaming only inside processes and thus actions ‘ (a^0) ’ and ‘ (a^1) ’ are different while processes ‘ $(a^0).0$ ’ and ‘ $(a^1).0$ ’ are equal. This allows us to define the set of bound names of the action A , written $\text{bn}(A)$. The set $\text{bn}(\cdot)$ is not defined for processes. Nevertheless because basic names are preserved under α -renaming we can define the set of input-bound basic names $\text{ibbn}(P)$ and the set of ν -bound basic names $\text{nbbn}(P)$.

The process P is called **well scoped** when (W1) its input-bound, ν -bound, and free basic names do not overlap, (W2) nested input binders do not bound the same basic name, and (W3) every action contains an input-bound basic name just once. Formally:

- (W1) $\text{fbn}(P)$, $\text{ibbn}(P)$, and $\text{nbbn}(P)$ are pairwise disjoint, and
- (W2) for every subprocess $A.Q$ of P : $\text{ibbn}(A)$ and $\text{ibbn}(Q)$ are disjoint, and
- (W3) for every action A in P and $a \in \text{ibbn}(A)$: a occurs exactly once in A .

Conditions W1 and W2 are necessary mainly to improve expressiveness of **Poly*** types. Condition W3 rules out some processes with unambiguous meaning. From now on we suppose all processes to be well scoped.

The structural equivalence relation \equiv on **Meta*** processes is defined to be the smallest equivalence relation which is congruent with the process constructors and which satisfies the rules from the bottom part of Figure 3. It expresses that parallel composition is commutative, associative, and it has 0 as its unit. Furthermore it expresses a variable scope of name restriction and a repetitive behavior of replication.

A substitution in **Meta***, denoted σ , is a finite function from **Name** to **Message**. Free names of σ , written $\text{fn}(\sigma)$, are free names of messages in its range. The message decomposition operation M_*P serves to remove empty messages 0 from M , and to transform M to a linear shape (for example $((a.b).c)_*P = a.b.c.P$). It is defined in the top part of Figure 4. The remaining three parts of Figure 4 define application of the substitution σ to **Meta*** entities. Application is written postfix as $M\sigma$ and $P\sigma$ for messages and processes, and as $Z\dot{\sigma}$ for other **Meta*** entities. We adopt the convention that a postfix application of substitution operators binds more tightly than other operators and we omit parenthesis when possible. For example $A.P\sigma$ stands for $A.(P\sigma)$.

A substitution substitutes messages for names but some messages are not syntactically allowed at positions of names in **Meta***. For example we can not substitute ‘in a’ for b in ‘open b’ because the resulting expression ‘open (in a)’

<i>Message decomposition operator:</i>		
$(M_0.M_1)_*P = M_0*(M_1*P) \quad 0_*P = P \quad A_*P = A.P$		
<i>Application of a substitution to names, forms, elements, and actions:</i>		
$(E_0 \dots E_k)\dot{\sigma} = E_0\dot{\sigma} \dots E_k\dot{\sigma}$	$x\dot{\sigma} = \begin{cases} \sigma(x) & \text{if } \sigma(x) \in \text{Name} \\ x & \text{if } x \notin \text{dom}(\sigma) \\ \bullet & \text{otherwise} \end{cases}$	
$(x_1, \dots, x_k)\dot{\sigma} = (x_1, \dots, x_k)$		
$\langle M_1, \dots, M_k \rangle \dot{\sigma} = \langle M_1\sigma, \dots, M_k\sigma \rangle$		
<i>Application of a substitution to messages:</i>		
$(M_0.M_1)\sigma = M_0\sigma.M_1\sigma$	$F\sigma = \begin{cases} \sigma(F) & \text{if } F = x \in \text{dom}(\sigma) \\ F\dot{\sigma} & \text{otherwise} \end{cases}$	
$0\sigma = 0$		
<i>Application of a substitution to processes:</i>		
$0\sigma = 0$	$(P \mid Q)\sigma = P\sigma \mid Q\sigma$	$(!P)\sigma = !P\sigma$
$(\nu(x).P)\sigma = \nu(x).P\sigma \quad \text{if } x \notin \text{dom}(\sigma) \cup \text{fn}(\sigma)$		
$(A.P)\sigma = \begin{cases} \sigma(A)_*P\sigma & \text{if } A = x \in \text{dom}(\sigma) \\ A\dot{\sigma}.P\sigma & \text{if } A \notin \text{dom}(\sigma) \ \& \ \text{bn}(A) \cap (\text{dom}(\sigma) \cup \text{fn}(\sigma)) = \emptyset \end{cases}$		

Figure 4: Application of a substitution to Meta★ entities.

is not a valid Meta★ entity. In other process calculi similar expression are allowed by syntax but semantics does not give them any meaning. In Meta★ similar expressions are not syntactically allowed and application of a Meta★ substitution produces a special name ‘•’ at positions of these syntactic errors. Thus for example the result of the above substitution is ‘open •’.

2.3.2 Specifying Semantics by Instantiating Meta★

Meta★ does not provide a fixed rewriting relation. Instead it provides the syntax which can be used to describe variety of rewriting relations. Thus Meta★ is instantiated to a particular calculus by providing the description of its rewriting relation. The syntax used to describe rewriting rules is presented in the top part Figure 5. An important role play process templates which are used to describe left and right hand sides in axioms of the rewriting relation in question. Their syntax imitate the syntax of processes up to that leaves of their syntax trees can contain variables additionally to names. Variables in process templates are intended to range over appropriate values, that is, name variables range over names, and so on. A substitution application operator ‘ $\{\hat{x}_1 := \hat{s}_1, \dots, \hat{x}_k := \hat{s}_k\} \hat{P}$ ’ explicitly describes a substitution to be applied on the right hand side of some rule. Furthermore, **rewrite** rules are used to describe axioms of the rewriting relation and **active** rules are used to describe positions in processes other than the top-level where rewriting rules are to be applied.

When the action template \hat{A} contains an element template ‘ $(\hat{x}_1, \dots, \hat{x}_k)$ ’ then all occurrences of the name variables \hat{x}_i ’s in ‘ $\hat{A}.\hat{P}$ ’ are said to be bound. Also name variables \hat{x}_i ’s are said to be bound in ‘ $\{\hat{x}_1 := \hat{s}_1, \dots, \hat{x}_k := \hat{s}_k\} \hat{P}$ ’. The set of bound variables of the process template \hat{P} is denoted $\text{bv}(\hat{P})$. Only a name variable can be bound. The set of free variables of \hat{P} is denoted $\text{fv}(\hat{P})$.

Syntax of Meta* reduction rules:

$\hat{x}, \hat{y} \in$	NameVar	$::=$	$\hat{a} \mid \hat{b} \mid \hat{c} \mid \dots$
$\hat{m} \in$	MessageVar	$::=$	$\hat{M} \mid \hat{N} \mid \dots$
$\hat{p} \in$	ProcessVar	$::=$	$\hat{P} \mid \hat{Q} \mid \hat{R} \mid \dots$
$\hat{s} \in$	Substitute	$::=$	$\hat{x} \mid \hat{m}$
$\hat{E} \in$	ElementTempl	$::=$	$x \mid \hat{x} \mid (\hat{x}_1, \dots, \hat{x}_k) \mid \langle \hat{m}_1, \dots, \hat{m}_k \rangle$
$\hat{A} \in$	ActionTempl	$::=$	$\hat{E}_0 \hat{E}_1 \dots \hat{E}_k$
$\hat{P}, \hat{Q} \in$	ProcessTempl	$::=$	$\hat{p} \mid \hat{A}.\hat{P} \mid 0 \mid (\hat{P} \mid \hat{Q}) \mid \{\hat{x}_1 := \hat{s}_1, \dots, \hat{x}_k := \hat{s}_k\} \hat{p}$
$\hat{r} \in$	Rule	$::=$	rewrite $\{\hat{P} \leftrightarrow \hat{Q}\} \mid$ active $\{\hat{p} \text{ in } \hat{P}\}$
$\mathcal{R} \in$	RuleSet	$=$	$\mathcal{P}_{\text{fn}}(\text{Rule})$

Semantics of Meta* reduction rules:

$\frac{\text{rewrite}\{\hat{P} \leftrightarrow \hat{Q}\} \in \mathcal{R}}{\llbracket \hat{P} \rrbracket_\rho \xrightarrow{\mathcal{R}} \llbracket \hat{Q} \rrbracket_\rho} \text{ (RRW)}$	$\frac{\text{active}\{\hat{p} \text{ in } \hat{P}\} \in \mathcal{R} \quad P \xrightarrow{\mathcal{R}} Q}{\llbracket \hat{P} \rrbracket_{\rho[\hat{p} \mapsto P]} \xrightarrow{\mathcal{R}} \llbracket \hat{P} \rrbracket_{\rho[\hat{p} \mapsto Q]}} \text{ (RACT)}$	
$\frac{P \xrightarrow{\mathcal{R}} Q}{P \mid R \xrightarrow{\mathcal{R}} Q \mid R} \text{ (RPAR)}$	$\frac{P \xrightarrow{\mathcal{R}} Q \quad x \notin \text{fn}(\mathcal{R})}{\nu(x).P \xrightarrow{\mathcal{R}} \nu(x).Q} \text{ (RNU)}$	$\frac{P \equiv P_0 \quad P \xrightarrow{\mathcal{R}} Q \quad Q \equiv Q_0}{P_0 \xrightarrow{\mathcal{R}} Q_0} \text{ (RSTR)}$

Figure 5: Syntax and semantics of Meta* reduction rule descriptions.

This includes all message and process variables from \hat{P} . The set of all variables of \hat{P} (either free or bound) is denoted $\text{var}(\hat{P})$. The set of free names of \hat{P} (those element templates \hat{E} that are names x) is denoted $\text{fn}(\hat{P})$. For example, given the process template $\hat{P} = \text{do}(\hat{y}).\hat{a}[\] . (\text{out } \hat{b}.\hat{P} \mid \{\hat{x} := \hat{M}\} \hat{Q})$ we have $\text{bv}(\hat{P}) = \{\hat{x}, \hat{y}\}$, $\text{fv}(\hat{P}) = \{\hat{a}, \hat{b}, \hat{M}, \hat{P}, \hat{Q}\}$, and $\text{fn}(\hat{P}) = \{\text{do}, [\], \text{out}\}$. The set of free names of the rule set \mathcal{R} , written $\text{fn}(\mathcal{R})$, is a union of sets of free names of all process templates in \mathcal{R} .

A process template is instantiated to a Meta* process by filling in values for all name, message, and process variables in the template. Entity instantiations serve this purpose. An entity instantiation ρ is a finite mapping that maps name variables to names, message variables to messages, and process variables to processes. Given the entity instantiation ρ and the process template \hat{P} the Meta* process denoted $\llbracket \hat{P} \rrbracket_\rho$ is constructed by applying ρ to \hat{P} component wise and substituting appropriate Meta* entities for template variables. Only exception is the case of the substitution application operator which is instantiated as follows:

$$\llbracket \{\hat{x}_0 := \hat{s}_0, \dots, \hat{x}_k := \hat{s}_k\} \hat{p} \rrbracket_\rho = (\llbracket \hat{p} \rrbracket_\rho) \sigma$$

where $\sigma = \{\llbracket \hat{x}_0 \rrbracket_\rho \mapsto \llbracket \hat{s}_0 \rrbracket_\rho, \dots, \llbracket \hat{x}_k \rrbracket_\rho \mapsto \llbracket \hat{s}_k \rrbracket_\rho\}$

Note that we have identified processes up to α -renaming of bound names but an entity instantiation picks names for bound variables. Thus we need some additional conditions when instantiating a template to avoid name captures. These together with some additional syntactic restrictions on templates in rewriting rules are described in Appendix B.

Given the set of rewriting rules \mathcal{R} the rewriting relation on Meta* processes $\xrightarrow{\mathcal{R}}$ is defined by the rules in the bottom part of Figure 5.

The usage is best demonstrated on an example. The following set of rewriting rules describes the rewriting relation of MA:

$$\mathcal{R} = \left\{ \begin{array}{l} \mathbf{active}\{ \dot{P} \text{ in } \dot{a}[\dot{P}] \}, \\ \mathbf{rewrite}\{ \dot{a}[\text{in } \dot{b}.\dot{P} \mid \dot{Q}] \mid \dot{b}[\dot{R}] \leftrightarrow \dot{b}[\dot{a}[\dot{P} \mid \dot{Q}] \mid \dot{R}] \}, \\ \mathbf{rewrite}\{ \dot{a}[\text{out } \dot{a}.\dot{P} \mid \dot{Q}] \mid \dot{R}] \leftrightarrow \dot{a}[\dot{R}] \mid \dot{b}[\dot{P} \mid \dot{Q}] \}, \\ \mathbf{rewrite}\{ \text{open } \dot{a}.\dot{P} \mid \dot{a}[\dot{R}] \leftrightarrow \dot{P} \mid \dot{R} \}, \\ \mathbf{rewrite}\{ \langle \dot{M} \rangle.\dot{P} \mid (\dot{a}).\dot{Q} \leftrightarrow \dot{P} \mid \{ \dot{a} := \dot{M} \} \dot{Q} \} \end{array} \right\}$$

2.3.3 Emulating Process Calculi in Meta*

Suppose that we would like to emulate some existing process calculus in Meta*. Let us call the emulated calculus PC and its rewriting relation \rightarrow . To emulate PC in Meta* the following two steps have to be done. Firstly a translation of PC processes to Meta* processes has to be defined. Secondly axioms of the PC's rewriting relation has to be described using the Meta* syntax for rewriting rules explained in Section 2.3.2. The first step gives us the translation function $([\cdot])$. The second step gives us the set of Meta* descriptions of rewriting rules \mathcal{R} and the rewriting relation $\xrightarrow{\mathcal{R}}$.

To check that $\xrightarrow{\mathcal{R}}$ faithfully corresponds to the PC's rewriting relation the following two results has to be proved. Firstly that a one step of rewriting on PC processes (using \rightarrow) corresponds to a one step of rewriting on images of the translation (using $\xrightarrow{\mathcal{R}}$). Formally that

$$B_0 \rightarrow B_1 \quad \text{iff} \quad ([B_0]) \xrightarrow{\mathcal{R}} ([B_1])$$

holds for all PC processes B_0 and B_1 . Secondly it has to be verified that the image of a PC process can not be rewritten using $\xrightarrow{\mathcal{R}}$ to a Meta* process that has no counterpart among PC processes. It means that the set of translations of PC processes is closed under rewriting. Formally that it holds that

$$([B_0]) \xrightarrow{\mathcal{R}} P_1 \quad \text{implies} \quad \exists B_1 : P_1 = ([B_1])$$

where B_0, B_1 are PC processes and P_1 is a Meta* process.

It is common that the PC's structural equivalence differs from the Meta*'s one. Then correctness of the translation is not as straightforward as above because it has to reflect these differences. The correspondence of rewriting relations is usually up to structural equivalence. In some cases one step of rewriting in PC corresponds to two steps of rewriting in Meta* or otherwise. Occasionally more steps are necessary.

2.3.4 Poly* Shape Predicates and Types

The concept of Poly* types is build on the notion of **shape predicates**. A shape predicate describes possible structure that a process syntax tree can have. It constitutes the set of a all processes whose syntax trees have the structure in

<i>Syntax of Poly* shape predicates:</i>			
$\varphi \in$	FormType	::=	$a_0 \dots a_k$
$\Phi \in$	FormTypeSet	=	$\mathcal{P}_{\text{fin}}(\text{FormType})$
$\mu \in$	AMessageType	::=	$\Phi^* \mid a$
$\varepsilon \in$	ElementType	::=	$a \mid (a_1, \dots, a_k) \mid \langle \mu_1, \dots, \mu_k \rangle$
$\alpha \in$	ActionType	::=	$\varepsilon_0 \varepsilon_1 \dots \varepsilon_k$
$\chi \in$	Node	::=	$\mathbf{X} \mid \mathbf{Y} \mid \mathbf{Z} \mid \dots$
$\eta \in$	Edge	::=	$\chi_0 \xrightarrow{\alpha} \chi_1$
$G \in$	ShapeGraph	=	$\mathcal{P}_{\text{fin}}(\text{Edge})$
$\pi \in$	ShapePredicate	::=	$\langle G, \chi \rangle$
<hr/> <i>Rules for matching non-process Meta* entities:</i>			
$\frac{}{\vdash a^i : a}$	(MNAME)	$\frac{\vdash M : \Phi \quad M \neq a}{\vdash M : \Phi^*}$	(MMsg)
$\frac{}{\vdash 0 : \Phi}$	(MNL)	$\frac{\vdash F : \varphi \quad \varphi \in \Phi}{\vdash F : \Phi}$	(MFRM)
$\frac{\vdash M_0 : \Phi \quad \vdash M_1 : \Phi}{\vdash M_0.M_1 : \Phi}$	(MComp)	$\frac{\vdash x_i : a_i \quad \forall i : 0 < i \leq k}{\vdash (x_1, \dots, x_k) : (a_1, \dots, a_k)}$	(MIN)
$\frac{\vdash M_i : \mu_i \quad \forall i : 0 < i \leq k}{\vdash \langle M_1, \dots, M_k \rangle : \langle \mu_1, \dots, \mu_k \rangle}$	(MOUT)	$\frac{\vdash E_i : \varepsilon_i \quad \forall i \leq k}{\vdash E_0 \dots E_k : \varepsilon_0 \dots \varepsilon_k}$	(MACT)
<hr/> <i>Rules for matching of Meta* processes:</i>			
$\frac{}{\vdash 0 : \pi}$	(MPNUL)	$\frac{(\chi_0 \xrightarrow{\alpha} \chi_1) \in G \quad \vdash A : \alpha \quad \vdash P : \langle G, \chi_1 \rangle}{\vdash A.P : \langle G, \chi_0 \rangle}$ (MPACT)	
$\frac{\vdash P : \pi \quad \vdash Q : \pi}{\vdash P \mid Q : \pi}$	(MPPAR)	$\frac{\vdash P : \pi}{\vdash \nu(x).P : \pi}$	(MPNU)
		$\frac{\vdash P : \pi}{\vdash !P : \pi}$	(MPREP)

Figure 6: Syntax and semantics of Poly* shape predicates.

question. When a rewriting rule from \mathcal{R} is applied to a process its syntax tree is changed. It can happen that the new syntax tree is no longer described by the same shape predicate as before. Only shape predicates that describe sets of processes closed under rewriting rules from \mathcal{R} are called **Poly*** (\mathcal{R} -)types. The set of types is further slightly tighten to be able to recognize types and to achieve results such as existence of principal types.

The formal syntax of shape predicates is defined in the top part of Figure 6. Action types describe actions and their syntaxes correspond each other. Main differences are that actions are built from basic names instead of names, and that compound messages are described up to commutativity, associativity, and repetitions of their parts. Thus a single action type describe the set of actions. A shape predicate is a rooted directed finite graph with edges labeled by action types. Intuitively a process matches a shape predicate when the process's syntax tree is a "subgraph" of the shape predicate. Shape predicate is a graph and thus a single shape predicate can describe also infinite syntax trees, or better said, syntax trees of an arbitrary depth.

Matching of non-process Meta* entities against appropriate types are de-

scribed in the middle part of Figure 6. Note that rule MACT is used also to match forms against form types when all E_i 's are names and ε_i 's basic names. Finally matching of $\text{Meta}\star$ processes against shape predicates is described in the bottom part of Figure 6. Note that matching of processes against shape predicates does not depend on a set of rewriting rules. It works in the same way in any instantiation of $\text{Meta}\star$. We define the **meaning** $\llbracket \pi \rrbracket$ of the shape predicate π to be the set of all processes matching the shape predicate (that is $\llbracket \pi \rrbracket = \{P : \vdash P : \pi\}$).

Let the set of rewriting rules \mathcal{R} be given. We call the shape predicate π **semantically closed** w.r.t. \mathcal{R} when the meaning of π is closed under \mathcal{R} -rewritings. Formally when $\vdash P : \pi$ and $P \xrightarrow{\mathcal{R}} Q$ imply $\vdash Q : \pi$ for any P and Q . It is not easy to decide whether a shape predicate is semantically closed w.r.t. an arbitrary set of rewriting rules. That is why we define an easier to decide subset of semantically closed shape predicates called **syntactically closed** shape predicates. Syntactical closure is designed to be easily algorithmically decidable. On the other hand the formal definition is somewhat complex and that is we present it in Appendix C for it is not crucial for purposes of this report.

Requiring types to be syntactically closed allows to algorithmically recognize types. Nevertheless it is still not enough to sufficiently easily proof the principal typing property. A principal type of P is the P 's type with the least meaning (w.r.t. inclusion) from all other types of P . The **principal typing** property for $\text{Poly}\star$ says that every process has the principal type. To be able to proof this property we need an additional restriction to the set of types. Thus we defined the width restriction that limits the number distinct successors of nodes in a shape graph, and the depth restriction that limit shape graph's maximal depth.

The meaning $\llbracket \alpha \rrbracket$ of the action type α is the set of all actions matching α . We call two action types α_0 and α_1 similar, written $\alpha_0 \approx \alpha_1$, when there is some action A contained both in $\llbracket \alpha_0 \rrbracket$ and $\llbracket \alpha_1 \rrbracket$. The relation \approx is equality on action types not containing message types of the form $\Phi\star$. A shape graph satisfies the **width restriction** when there are no two edges with the same source and distinct destinations labeled with similar action types. A shape graph satisfies the **depth restriction** when every two edges that lay on the same (oriented) path and are labeled with similar action types have the same destination. Finally by a $\text{Poly}\star$ \mathcal{R} -**type** we mean a syntactically closed shape predicate whose shape graph satisfies the width and the depth restriction.

2.3.5 Comparing $\text{Poly}\star$ with Related Type Systems

Suppose that we would like to compare the type system provided by $\text{Poly}\star$ with some existing type system TS of some existing process calculus PC . At first we need to emulate the PC 's rewriting relation as described above in Section 2.3.3. Otherwise the result is not very interesting. By a TS typing we mean a collection of all information other than the process which are present in type judgment statements. For example type judgments in TMA has the form ' $E \vdash B : T$ ' and thus a typing is a pair (E, T) . Given the TS typing I the meaning of I denoted $\llbracket I \rrbracket$ is the set of all processes which are typable with the typing I . To

faithfully embed I in $\text{Poly}\star$ we need a $\text{Poly}\star$ type (or a set of types) with the same meaning (at the level of images of the translation) as I . Let the process translation function $\llbracket \cdot \rrbracket$ apply to set of processes member-wise. Then we need to translate a TS typing I into the $\text{Poly}\star$ type $\llbracket I \rrbracket$ such that:

$$\llbracket \llbracket I \rrbracket \rrbracket = \llbracket \llbracket I \rrbracket \rrbracket$$

3 Differences Between $\text{Poly}\star$ v1 and $\text{Poly}\star$ v2

Changes done to $\text{Poly}\star$ v1 can be divided into (1) improvements and (2) corrections. Improvements (1) constitute simplifications and extensions of the $\text{Poly}\star$ theory required to achieve a more uniform behavior with other process calculi. Corrections (2) consist of fixes of inconsistencies discovered in the theory.

3.1 Basic Names and α -Renaming

The main difference and improvement to $\text{Meta}\star$ v1 and $\text{Poly}\star$ v1 is the introduction of basic names. Names in $\text{Meta}\star$ v1 consist of simple identifiers like ‘a’ while in $\text{Meta}\star$ v2 names consist of pairs of the form ‘a⁷’ where ‘a’ is called a basic name and an arbitrary natural number is allowed at the position of the upper index ‘7’. The introduction of basic names thus refers to the introduction of these indexes. This seemingly simple improvement has surprisingly useful influences on other parts of the system. The initial motivation was to improve handling of name restriction at the level of $\text{Poly}\star$ types. This is described in details in Section 4. In this section we describe influences of the introduction of basic names to other parts of the system.

Basic names give us more refine control over α -renaming of bound names. In $\text{Meta}\star$ v2 both input- and ν -bound names are subjects to α -renaming with the restriction that the basic name part of a name is preserved. On the other hand in $\text{Meta}\star$ v1 only ν -bound names are subjects to α -renaming but without any restrictions on the choice of a new name. Thus in $\text{Meta}\star$ v1 input-bound names can not be α -renamed. The main reason for it is that input-bound names are also used to build types in $\text{Meta}\star$ v1. There is a correspondence between an input-bound name in a process and the same name in a type. If the input-bound name in the process was α -renamed this correspondence would be lost. In $\text{Meta}\star$ v2 this correspondence is implemented using basic names and thus is still traceable after α -renaming.

It is a common practice in process calculi to α -rename bound names and usually it is needed to avoid name collisions and captures. $\text{Meta}\star$ v1 builds on the observation that α -renaming of input-bound names can be avoided under certain circumstances. The first important observation is that rewriting rules of most process calculi found in the literature do not inject an input-binder under the scope of another input-binder. Another inspection of the rules tells us that when a substitution of M for x is invoked then M comes from a part of a process outside the scope of x (of x ’s original binder in particular). In this

configuration a name capture can occur only when M contains some name that is simultaneously input-bound under the scope of x . Thus when we ensure that no input-bound name occurs outside the scope of its binder² we can see that no substitution can produce a name capture and thus α -renaming of input-bound names is not needed.

The above solution is possible but it has two drawbacks. Firstly it is a hard and space consuming task to explain it to the reader. Secondly this approach is not applicable to ν -bound names in the presence of replication because a replicated process can contain a ν -binder which can extend its scope to contain another copy of itself. Thus we still need to handle α -renaming of ν -bound names and it is against the reader's intuition that α -renaming of ν -bound and input-bound names are handled in different ways for it is not a common practice in other process calculi.

Thus the introduction of basic names to **Poly★ v2** allows to α -rename input-bound names and to handle input-bound and ν -bound names in the same way as it is in common in other process calculi. This is the first clarification of the theory which can remove unnecessary obstacles for the reader's intuition. Moreover it allows us to identify processes up to the consistent α -renaming of bound names. This also simplifies the formal development of the theory because it allows us to concentrate directly to its important aspects instead of chasing after details in the form of side conditions treating name captures. Another great improvement allowed by basic names can be found on the level of **Poly★ v2** types. Typing of processes containing name restriction is in **Poly★ v2** handled by the same rules as all other processes (see rule **MPNU** in Figure 6). In **Poly★ v1** there is a basic version of shape predicates for processes not containing name restriction and a stand alone machinery for handling of name restriction is build on it. This is described in Section 3.5.

It remains for us to say that in other process calculi there are usually no restrictions on the choice of new names that a bound name can be α -renamed to (up to avoiding name captures). Nevertheless the main intention of α -renaming in process calculi is to be able to always provide a fresh instance of a process. Thus the approach of **Poly★ v2** is satisfactory for there are infinitely many fresh variants of each name.

3.2 Free Names, Bound Names, and Well-Scopedness

In **Meta★ v1** the set of free names $\text{FN}(P)$ and the set of bound names $\text{BN}(P)$ are defined for every process P . The set $\text{FN}(P)$ contains names of P that are neither ν - nor input-bound. The set $\text{BN}(P)$ contains names of P that are input-bound but not ν -bound. Finally ν -bound names of P are in none of these sets. This notation could confuse a reader that is already familiar with other process calculi for in other calculi a set of bound names contains generally both input- and ν -bound names. Moreover this notation has already caused an inconsistency in the

²More precisely we can allow processes like $(x).x.0 \mid (x).x.0$ which do not contain any subprocess with a free and input-bound occurrence of the same name. In fact it turns out to be necessary in the presence of replication.

definition of well-scopedness in **Meta★ v1** as described below in next paragraphs. Reasons for this slightly unconventional definition were different approaches to handling of input- and ν -bound names. These reasons do no longer apply in **Meta★ v2**.

Meta★ v1 contains the notion of well-scopedness that is stated as follows [MW05, Section 2.2]:

The process term P is **well scoped** iff it contains no nested binding of the same name and none of its free names also appear bound in the term. Formally, it is required that (1) $\text{BN}(P)$ and $\text{FN}(P)$ are disjoint, (2) whenever P contains $A.Q$, $\text{BN}(A)$ and $\text{BN}(Q)$ are disjoint, and (3) whenever P contains $\nu(x).Q$, $x \notin \text{BN}(Q)$.

This definition is necessary to avoid the need to α -rename input-bound names. The intention is to ensure that no substitution invoked from a well scoped process can lead to a name capture. Unfortunately this definition accidentally labels the process $(x).\nu(x).x.0$ as well scoped. Recall that ν -bound names are not inside BN and thus condition (2) does not apply. This causes an inconsistency in the subject reduction property because substitution in **Meta★ v1** does not guard against name captures as described in the next Section 3.3. Another weakness of **Meta★ v1** is that it does not rule out processes like $(x,x).0$ or $(x)(x).0$. These processes can lead to application of an undefined substitution and thus produce undesirable results.

Above mentioned problems are solved in **Meta★ v2**. Substitution in **Meta★ v2** guards against name captures and action prefixes which bound the same name more than once are forbidden.

3.3 Process Substitution and Structural Equivalence

Substitution in **Meta★ v1** does not guard against name captures because they are not supposed to occur. Relevant cases are defined as follows [MW05, Figure 3] (where \mathcal{S} denotes a substitution and $\mathcal{S}^{\mathbf{P}}P$ denotes application of \mathcal{S} to process P):

$$\mathcal{S}^{\mathbf{P}}(\nu(x).P) = \nu(x).\mathcal{S}^{\mathbf{P}}P \quad \mathcal{S}^{\mathbf{P}}(x.P) = \begin{cases} \mathcal{S}(x)_*(\mathcal{S}^{\mathbf{P}}P) & \text{when } x \in \text{Dom}(\mathcal{S}) \\ x.(\mathcal{S}^{\mathbf{P}}P) & \text{otherwise} \end{cases}$$

Not presence of side conditions which prevent name captures is explained as follows [MW05, Section 2.3]:

The definitions in Figure 3 do not worry about name capture. In general, therefore, $\mathcal{S}^{\mathbf{X}}X$ is only intuitively correct if $\text{BN}(X)$ is disjoint from the names mentioned in \mathcal{S} . In practice, this will always follow from the assumption that all terms are well scoped.

Unfortunately well-scopedness as defined in **Meta★ v1** is not capable to prevent name capture. As already noted above the process $(x).\nu(x).x.0$ is well

scoped as well as ‘ $\langle a \rangle.0 \mid (x).\nu(x).x.0$ ’. When the MA communication rule is applied to the second process the substitution $\mathcal{S} = \{x \mapsto a\}$ is applied to ‘ $\nu(x).x.0$ ’. Result of this application is ‘ $\nu(x).a.0$ ’ which is wrong because ‘a’ escaped its binder.

Substitution in **Meta★ v2** guards correctly against name captures and application of $\sigma = \{x^0 \mapsto a^0\}$ to ‘ $\nu(x^0).x^0.0$ ’ produces the right result ‘ $\nu(x^0).x^0.0$ ’.

The structural equivalence relation is almost the same in both versions. The only difference is that the one in **Meta★ v2** does not need to contain a rule that handles α -renaming of ν -bound names. There is also a technical problem with this rule in **Meta★ v1** for it uses an undefined notation $[x := y]P$ for substitution (and this substitution is a different one than the standard **Meta★ v1** substitution).

3.4 Rewriting Rules and Rewriting Relation

As described in the above sections it is necessary for the **Meta★** rewriting relation to preserve well-scopedness. It requires ruling out of some rewriting rules by restricting the rule’s syntax. In **Meta★ v1** there are four restrictions on rules left-hand sides (L1-L4) and five restrictions on right-hand sides (R1-R5) [MW04, Section 5.1]. We do not present exact definitions here. These restrictions has two drawbacks. Firstly they are stated using implicitly defined notions like “nested binders” or “scope of a binder”. For example it is not clear whether ‘ $(\hat{a})(b).0$ ’ contains nested binders. Secondly there are several inconsistencies which allow rules to produce a non-well scoped process. Thus they lack the desired properties [MW04, Lemma 5.1, 5.2].

In **Meta★ v2** restrictions on rewriting rules are revised. New and more formal definitions are presented in Appendix B.

Another problem related to the rewriting relation in **Meta★ v1** is that it allows rewriting rules to match on a bound name and remove it from its binder. For an example take the following rule:

$$\mathcal{R} = \{ \text{rewrite}\{ a.0 \hookrightarrow b.0 \} \}$$

Note that ‘a’ and ‘b’ are explicitly given names and not a name variables. Using this rule one can derive ‘ $a.0 \xrightarrow{\mathcal{R}} b.0$ ’ and from this ‘ $\nu(a).a.0 \xrightarrow{\mathcal{R}} \nu(a).b.0$ ’. This is definitely not desirable. Also a non-well scoped process can be produced in this way (for example start with ‘ $a.0 \mid (b).0$ ’). This behavior is not consistent with the subject reduction property of **Poly★ v1**.

In **Meta★ v2** the rewriting relation inference rules are extended with side conditions which prevent rules to match on a bound name by an explicitly specified name. In the case of rule **RNU** it is the condition ‘ $x \notin \text{fn}(\mathcal{R})$ ’. In the case of rule **RACR** it is condition I2 from Appendix B.

3.5 Poly★ Shape Predicates and Name Restriction

As already noted above the introduction of basic names allows simplified handling of name restriction in **Poly★ v2**. Basic version of **Poly★ v1** does not handle

name restriction at all. Shape predicates in **Poly★ v1** can match only on processes not containing the ν operator. Extended version of shape predicates called **guarded shape predicates** intended to handle restriction is defined on the top of the basic **Poly★ v1** theory. This section describe differences between types in **Poly★ v1** and **Poly★ v2**.

A guarded shape predicate is a pair containing a shape predicate and a set of names. The set of names of a guarded shape predicate contains names of the shape predicate that can match on an arbitrary ν -bound name in a process. In **Poly★ v1** names in processes are references directly from types. Thus if a name in a process is α -renamed and the corresponding name in the type is not then this reference is broken. That is why in **Poly★ v1** specific names are allowed to match on an arbitrary name. In **Poly★ v2** names in processes are referenced from types by basic names only. Thus this reference is preserved under α -renaming of names in processes (for basic names are preserved under α -renaming).

Except for the simplified handling of name restriction in the theory this approach also increases expressiveness of types in **Poly★ v2**. For example in **Poly★ v1** every guarded shape predicate that matches the process ' $\nu(a).in\ a.0\ |\ \nu(b).out\ b.0$ ' has to also match the process ' $\nu(a).(in\ a.0\ |\ out\ a.0)$ '. Thus **Poly★ v1** can not distinguish between the above two processes at type level. In **Poly★ v2** this distinction is possible provided that 'a' and 'b' are different basic names. As described in Section 4 this improvement of expressiveness of types was the initial motivation for the introduction of basic names.

Some other parts of the **Poly★** theory underwent, merely cosmetic, changes. New types of forms are introduced (in **Poly★ v1** forms serve directly as types) and message types have new but w.r.t. the meaning equivalent syntax. The rule from **Poly★ v1** used to derive a type of a compound message is now split into more intuitive rules **MMSG**, **MNL**, **MFRM**, and **MCMP**. Rule **MPNU** is added as already noted above. Definition of application of a type substitution to different types (Figure 7) is redefined in order to respect the new syntax of message types. In **Poly★ v1** the application refers to the operation ' $\mu_0 * \mu_1$ ' that is defined to be "the least message type whose meaning includes ' $M_0.M_1$ ' for all $M_0 \in \llbracket \mu_0 \rrbracket$, $M_1 \in \llbracket \mu_1 \rrbracket$ ". Unfortunately no ordering on message types is defined at all. The name of the operator also collides with the message decomposition operator (the top part of Figure 3). This operation on message types is avoided in **Poly★ v2**.

3.6 Summary of Changes Between **Poly★ v1** and **Poly★ v2**

This section briefly summarizes corrections and improvements in **Meta★ v2** and **Poly★ v2** and links to one of the previous sections which provide a more detailed explanation.

Improvements:

- the introduction of basic names (Section 3.1)
- α -renaming of input-bound names (Section 3.1)
- identification of processes up to the consistent α -renaming (Section 3.1)
- simplified handling of name restriction (Section 3.5)

- increased type expressiveness (Section 3.5)

Corrections:

- fixed the definition of well-scopedness (Section 3.2)
- substitution does not produce name captures (Section 3.3)
- names in rewriting rules can not match on bound names (Section 3.4)
- rewriting rules can not produce non-well scoped processes (Section 3.4)
- removed references to undefined notations like $*$ on message types and $[x := y]$ in structural equivalence (Section 3.5, Section 3.3)

4 On the Way from Poly★ v1 to Poly★ v2

This section describes a concrete problem that was encountered when trying to compare Poly★ v1 with TMA. In this way it tries to explain reasons for extensions of the theory. Thus it describes motivations that led to the extension of Poly★ v1 with basic name. It also describes some of the work done on the comparison of Poly★ and TMA during the second year. The terms Meta★ and Poly★ in this section refer to Meta★ v1 and Poly★ v1. Several possible translation of TMA processes to Meta★ are informally discussed below. We use the same notation (\cdot) to denote the translation in question.

4.1 Motivating Example

TMA processes contain explicit type annotations of bound names. On the other hand Meta★ processes does not include types of bound names. Thus we somehow need to handle explicit types when translating TMA processes into Meta★. The simplest solution is to ignore them entirely. This works fine when we want just to emulate TMA rewriting relation by the Meta★ one. It is because TMA rewriting does not take types of bound names into account.

Unfortunately this straightforward solution is not sufficient for embedding of TMA types in Poly★. Suppose the following two TMA processes:

$$\begin{aligned} B_0 &= (\nu a : \text{Amb}[\text{Shh}])a[\langle \rangle] \\ B_1 &= (\nu a : \text{Amb}[\mathbf{1}])a[\langle \rangle] \end{aligned}$$

Using the straightforward translation to Meta★ that ignores explicit types of bound names we obtain

$$\begin{aligned} (B_0) &= \nu(a).a[\langle \rangle.0] \\ (B_1) &= \nu(a).a[\langle \rangle.0] \end{aligned}$$

and we see that both B_0 and B_1 were translated into the same Meta★ process. But now let us realize that B_0 is not typable by any type in TMA while B_1 is typable by the type Shh (for example). Intuitively it is a problem because we see that to emulate TMA in Poly★ we need to somehow distinguish between processes that differ only by types of ν -bound names. The next paragraph

explains the problem in more details. Imagine that we somehow construct a $\text{Poly}\star$ type $\llbracket \text{Shh} \rrbracket$ as a correct embedding of Shh in $\text{Poly}\star$. Now we can see that $\llbracket \llbracket \text{Shh} \rrbracket \rrbracket$ has to contain $\llbracket B_1 \rrbracket$ but not $\llbracket B_0 \rrbracket$ which is not possible because it is the same process.

The above example shows us that to faithfully embed TMA in $\text{Poly}\star$ we can not simply ignore explicit types of bound variables during the translation of TMA processes into $\text{Meta}\star$. Instead, we need to consider these types somehow.

4.2 Solution Doubling ν -bound Names

One possible approach to solve the above problem is to use the benevolence of the $\text{Meta}\star$ syntax and to translate a ν -bound name into a sequence of two names. The first name of a sequence is the ν -bound name itself and the second encapsulates its type. Encapsulating TMA (message) types by $\text{Meta}\star$ names can be done as follows. At first reserve infinitely many names among $\text{Meta}\star$ names that are forbidden to be used in TMA processes. Secondly provide a bijection between these names and TMA types.

Suppose that the bijection maps the TMA type Shh to the $\text{Meta}\star$ name shh and $\mathbf{1}$ to one . Then a translation of the processes B_0 and B_1 from the previous Section would work as follows:

$$\begin{aligned} \llbracket B_0 \rrbracket &= \nu(a).a \text{ shh} [\langle \rangle.0] \\ \llbracket B_1 \rrbracket &= \nu(a).a \text{ one} [\langle \rangle.0] \end{aligned}$$

Note that the binding occurrences of a are not translated into 2-length sequences for $\text{Meta}\star$ syntax does not allow an arbitrary form at binding positions. This translation gives us the desired possibility to distinguish between two processes that differs only by types of some bound names.

A difficulty caused by this approach is that the translation of ν -bound names into 2-length sequence has to be appropriately reflected by a $\text{Meta}\star$ description of the TMA rewriting relation. For example, when the TMA in-rule was previously described as

$$\text{rewrite}\{ \hat{a} [\text{in } \hat{b}.\hat{P} \mid \hat{Q}] \mid \hat{b} [\hat{R}] \leftrightarrow \hat{b} [\hat{a} [\hat{P} \mid \hat{Q}] \mid \hat{R}] \}$$

now it has to be described as

$$\text{rewrite}\{ \hat{a} \hat{u} [\text{in } \hat{b} \hat{v}.\hat{P} \mid \hat{Q}] \mid \hat{b} \hat{v} [\hat{R}] \leftrightarrow \hat{b} \hat{v} [\hat{a} \hat{u} [\hat{P} \mid \hat{Q}] \mid \hat{R}] \}$$

Also we have to keep in mind that only ν -bound names are doubled. Free and input-bound names are translated into single names and thus we need to keep both of the above rewriting rules to allow rewriting with single names too. Additionally we also need to provide two combinations of the above rules. The first one considers a single name at the positions of a 's and a doubled name at the positions of b 's. The second one works the other way round. A possibility to avoid this redundancy in rules is to translate also free names into 2-length sequence using some fixed dummy name, say dum , as its second member. Then

only the second version of the rule that considers doubled names everywhere is needed.

Even when we translate free names into 2-length sequence it is still necessary to translate input-bound names into single names. We can not translate an input-bound name into a 2-length sequence because it would disallow a substitution of some compound message for the name. This is best shown on an example. Suppose the TMA process

$$B_2 = \langle \text{in a.out b} \rangle \mid (x : \text{Shh}).x.0$$

which using the TMA rewriting relation \rightarrow rewrites to ‘in a.out b.0’. Translating B_2 doubling x we obtain:

$$([B_2]) = \langle \text{in a.out b} \rangle \mid (x).x \text{ shh}.0$$

But now using the **Meta*** version of the communication rule we obtain ‘• shh’ instead of the desired ‘in a.out b.0’ (or ‘in a dum.out b dum.0 when doubling also free names using dum). The desired result is obtained by applying TMA rewriting rules to B_2 followed by translation to **Meta***. The problem here is that we are substituting a compound message for a name that is a proper part of an action prefix. The **Meta*** substitution correctly produces the bullet in this case. Recall that a substitution of a compound message for a name in **Meta*** is correct (that is, does not produce a bullet) only when the name is an action prefix on its own.

Thus we can not translate an input-bound name into a 2-length sequence. Fortunately we do not need to do that. We do not need to provide special versions of rewriting rules for the case of input-bound names either. This is because any process under an input binder is intact and rewriting rules can not be applied to it. Moreover, when a process formerly containing input-bound names appears in an active position, then any input-bound name that could be referred from some rewriting rule is already instantiated by communication to some free or a ν -bound name.

Thus our current translation does not need to double input-bound names. It simply ignores types of input-bound names when translation TMA processes into **Meta***. The question arises how we distinguish between two processes that differs only by types of input-bound names. The answer is that we can simply translate input-bound names of different types into a distinct **Meta*** names. Then we can use the ability of **Poly*** to distinguish between two different input-bound names. This approach can not be used to handle ν -bound names. That is because **Poly*** types can not distinguish between two processes that differs only by renaming of ν -bound names. In other words by an example, the process ‘ $\nu(a).a[\text{out a.0}]$ ’ has exactly the same types as ‘ $\nu(b).b[\text{out b.0}]$ ’. On the other hand there exists a type that is a type of ‘ $(a).a[\text{out a.0}]$ ’ but not of ‘ $(b).b[\text{out b.0}]$ ’ (and contrariwise).

4.3 Problem with Sending of ν -bound Names

Unfortunately the translation from the previous section is still not working satisfactorily. A problem appears when a TMA process sends a single name and this name is doubled during the translation to Meta*. Suppose the TMA process

$$B_3 = (\nu a : \text{Amb}[\text{Shh}])(\langle a \rangle \mid (x : \text{Amb}[\text{Shh}]).\text{in } x.0)$$

which is using the translation that doubles names described in the above Section 4.2 translated into

$$([B_3]) = \nu(a).(\langle a \text{ shh} \rangle \mid (x).\text{in } x.0)$$

Then using the communication rule we obtain ' $\nu(a).\text{in } \bullet.0$ ' instead of the desired ' $\nu(a).\text{in } a \text{ shh}.0$ '. The desired result is obtained by applying TMA rewriting rules to B_3 followed by translation to Meta*.

Before the solution described in the next section was developed an attempt was made to make the above translation working. The key idea was as follows. Firstly to translate an (unary) input prefixes bounding a name of some $\text{Amb}[T]$ type into a binary Meta* prefix. The second name in a Meta* prefix was supposed to receive the name that corresponds to the type of the first one. Secondly to translate messages of the form $\langle a \rangle$ which contain only a standalone name into a pair of names of the form $\langle a, t \rangle$. Here t is the name that corresponds to the type of a or dum when a is free. Note that the type of a is known during translation. Thirdly to translate an input-bound name of a $\text{Amb}[T]$ type to a 2-length sequence of Meta* names just like any other ν -bound name. For example the above process B_3 was translated to:

$$([B_3]) = \nu(a).(\langle a, \text{shh} \rangle \mid (x, t).\text{in } x \text{ t}.0)$$

This solution was finally found possible. Nevertheless a formal definition of $([\cdot])$, even for a monadic case, was found very dense and complicated. Later on much more simple and elegant solution was developed. It is described in the next section.

4.4 Solution with Basic Names

This section describes a solution developed to solve the problems with translation of TMA processes into Meta* described in the previous sections. The main idea is to associate a special basic name (alternatively called a canonical name) with each Meta* name and ensure that basic names are preserved under α -renaming of ν -bound names. This allows us to redefine the notion of Poly* type so that it would be possible to distinguish between two Meta* processes that differ only by names of ν -bound names. This distinction is possible when the differing names are associated with different basic names. Finally, this provides us a solution to the problem of handling types of ν -bound names during translation from TMA to Meta*. We will simply translate ν -bound names of different types into Meta* names associated with different basic names.

The current realization of the above idea is as follows. This is where **Poly★ v1** evolves to **Poly★ v2**. We build **Meta★** processes not from atomic names but from pairs consisting of an atomic name and a natural number. To minimize changes necessary to **Poly★ v1** and to achieve some consistency with terminology of other process calculi we denote these pairs as names while former atomic names are called basic names. We use superscripts to write pair names. The grammar of names is now as follows.

$$\begin{aligned} a \in \text{BasicName} & ::= a \mid b \mid \dots \mid \text{in} \mid \text{out} \mid \text{open} \mid \dots \mid \square \mid \bullet \mid \dots \\ x, y \in \text{Name} & ::= a^i \end{aligned}$$

This allows us to use the rest of **Meta★** processes grammar without changes:

$$\begin{aligned} F \in \text{Form} & ::= x_0 \dots x_k \\ M \in \text{Message} & ::= F \mid 0 \mid M_0.M_1 \\ E \in \text{Element} & ::= x \mid (x_1, \dots, x_k) \mid \langle M_1, \dots, M_k \rangle \\ A \in \text{Action} & ::= E_0 \dots E_k \\ P, Q, R \in \text{Process} & ::= 0 \mid A.P \mid \nu(x).P \mid (P \mid Q) \mid !P \end{aligned}$$

We could continue and adapt the structural equivalence rules to preserve the basic name part of a name while α -renaming ν -bound names and to adapt the notion of a **Poly★** shape predicate appropriately. We tried to extend **Poly★** in this way and during this work another options to change and mainly to simplify the theory were discovered. The final **Meta★ v2** and **Poly★ v2** are described above in Section 2.3. The changes between **Poly★ v1** and **Poly★ v2** are in details described above in Section 3.

5 Conclusion

This report describes the work of Jan Jakubův done during the second year of his PhD study on Heriot-Watt University. The main aim set by the First Year Report [Jak08] was to finish the paper that provides a comparison of **Poly★** and TMA. During the work on the paper it turned out that **Poly★ v1** is not expressive enough to allow a sufficiently straightforward comparison. This is described in details in Section 4. The extension of **Poly★ v1** with basic names was proposed in order to extend its expressiveness in an appropriate way. This is discussed in Section 3.1 and Section 4.4. After this extension was formalized it was necessary to verify main results including subject reduction and existence of principal typings. During this verification several inconsistencies in the former **Poly★ v1** theory were discovered. It was necessary to fix them for they applied to **Poly★ v2** as well. This is discussed in Section 3 and Section 3.6 provides a quick summary.

Thus the comparison paper is not finished yet for two unexpected necessities appeared: (1) the need to extend the expressiveness, and (2) the need to fix inconsistencies. The current status of **Poly★ v2** is that all inconsistencies described in Section 3 are fixed. Nevertheless some amount of work and maybe

some additional changes could still be necessary to ensure that main results hold and that other inconsistencies are not present.

Besides the two topics above some work was done on the comparison paper too. This is partially described in Section 4 and the current version of the paper is attached to this report in Appendix D. Moreover additional amount work was done on an extension of **Meta*** and **Poly*** with a more expressive support for recursive processes. This is not described in this report because it is in an early stage and results are still to be verified. The future of this and of the comparison paper is discussed in the next Section 6.

6 Future Work

The future work includes several topics. The first aim is to finish the paper that provides a comparison of **Poly*** v2 and TMA. It includes the following two subgoals. Firstly verify that **Poly*** v2 is already expressive enough to allow an embedding of TMA. Secondly proof and verify main results of **Poly*** v2 namely subject reduction and existence of principal types. Eventual problems discovered while fulfilling these aims could lead to additional extensions and corrections of the **Poly*** v2 theory.

The second aim is to extend **Poly*** v2 with some more expressive support for recursion. This includes extensions to both **Meta*** v2 and **Poly*** v2. Once this extension is developed main formal results have to be verified again. A comparison of expressiveness with some other calculi already which support recursion is also possible. Moreover the type inference algorithm has to be reimplemented to support new extensions.

The final aim is then to write and submit the PhD thesis. It should contain (1) extensions done to **Meta*** and **Poly*** including proofs of main results, (2) the comparison of **Poly*** with TMA and possibly some other systems, and (3) the implementation of a type inference algorithm. An approximate time plan is as follows:

now	-	Apr 2009	finish the comparison paper
Apr 2009	-	Oct 2009	extend Poly* with recursion
Oct 2009	-	May 2010	write the PhD thesis
		May 2010	submit the PhD thesis

A **Meta*** With α -equivalent Processes Unified

In this section we provide a formal background for handling of α -equivalence. It can be seen as assigning of a new meaning to members of **Process**. Their meaning shifts from syntax trees generated by the grammar from Figure 3 to classes of α -equivalence on these syntax trees.

We shall use the following syntax to refer to raw process syntax trees (called simply raw processes):

$$P, Q \in \text{RawProcess} ::= \mathbf{0} \mid A.\mathbf{P} \mid \nu(x).\mathbf{P} \mid (P \mid Q) \mid !P$$

The following defines notions of free and bound names, and some sets of names associated with entities.

Definition A.1. All occurrences of the name x in ' $\nu(x).\mathbf{P}$ ' are said to be (ν -)**bound**. When the action A contains an element ' (x_1, \dots, x_k) ' then all occurrences of the x_i 's in ' $A.\mathbf{P}$ ' as well as in A on its own are said to be (input-) **bound**. An occurrence of x that is not bound is said to be **free**. An occurrence of the basic name a is said to be bound (resp. free) when the corresponding occurrence of the name a^i is bound (resp. free).

The following sets are defined for actions and raw processes (ranged over by Z here):

$\text{fn}(Z)$: all names with a free occurrence in Z

$\text{bn}(Z)$: all names with a bound occurrence in Z

$\text{fbn}(Z)$: all basic names with a free occurrence in Z

$\text{ibbn}(Z)$: all basic names with an input-bound occurrence in Z

$\text{nbbn}(Z)$: all basic names with a ν -bound occurrence in Z ■

The following defines the notion of a well scoped raw process. It rules out some processes with an unambiguous meaning and some processes that would cause troubles with Poly★ type inference.

Definition A.2. The raw process \mathbf{P} is called **well scoped**, written $\text{wsp}(\mathbf{P})$, when all of the followings hold:

(W1) $\text{fbn}(\mathbf{P})$, $\text{ibbn}(\mathbf{P})$, and $\text{nbbn}(\mathbf{P})$ are pairwise disjoint, and

(W2) for every subprocess $A.\mathbf{Q}$ of \mathbf{P} : $\text{ibbn}(A)$ and $\text{ibbn}(\mathbf{Q})$ are disjoint, and

(W3) for every action A in \mathbf{P} and $a \in \text{ibbn}(A)$: a occurs exactly once in A . ■

The next defines the α -equivalence on raw process. It is defined using the name swapping operation.

Definition A.3. Let $(x \leftrightarrow y)\mathbf{P}$ denote \mathbf{P} with all occurrences (either free or bound) of names x and y swapped. The α -**equivalence** relation \sim is defined to be the smallest equivalence relation on RawProcess congruent with the raw process construction operators that satisfies the rule:

$$\frac{a^i \notin \text{fn}(\mathbf{P}) \quad a^j \notin \text{fn}(\mathbf{P})}{\mathbf{P} \sim (a^i \leftrightarrow a^j)\mathbf{P}} \text{ (SWAPNAME)}$$

■

Remark A.4. Note that without the α -equivalence being defined to be congruent with the raw process construction operators the following could not be proved:

$$(\nu(a^0).\mathbf{0} \mid \nu(a^0).\mathbf{0}) \sim (\nu(a^0).\mathbf{0} \mid \nu(a^1).\mathbf{0})$$

The following defines **Meta★** processes. Note that only well scoped raw processes are considered.

Definition A.5. *The α -equivalence class $[P]_\alpha$ of the raw process P is:*

$$[P]_\alpha = \{Q \in \text{RawProcess} : Q \sim P\}$$

Finally, by a **Meta★ process** we mean a class of α -equivalence of a well scoped raw process:

$$P, Q, R \in \text{Process} = \{[P]_\alpha : P \in \text{RawProcess} \ \& \ \text{wsp}(P)\}$$

■

The formal meaning for process constructors used throughout the paper is as follows.

Definition A.6. *We define the following abbreviations for **Meta★** processes:*

$$\begin{aligned} 0 &= \{\mathbf{0}\} \\ A.P &= [A.\mathbf{P}]_\alpha && \text{with } \mathbf{P} \in P \text{ being arbitrary such } \text{wsp}(A.\mathbf{P}) \\ (P \mid Q) &= [(P \mid Q)]_\alpha && \text{with } \mathbf{P} \in P, \mathbf{Q} \in Q \text{ being arbitrary} \\ \nu(x).P &= [\nu(x).\mathbf{P}]_\alpha && \text{with } \mathbf{P} \in P \text{ being arbitrary} \\ !P &= [!P]_\alpha && \text{with } \mathbf{P} \in P \text{ being arbitrary} \end{aligned}$$

We define the set of free names of P as $\text{fn}(P) = \text{fn}(\mathbf{P})$ for $\mathbf{P} \in P$ arbitrary. Similarly we define the sets $\text{fbn}(P)$, $\text{ibbn}(P)$, $\text{nbbn}(P)$ but not $\text{bn}(P)$. ■

Lemma A.7. *Definition A.6 does not depend on the choice of representants.* ■

It is easy to see that every equivalence class $[P]_\alpha$ from **Process** can be described using the above abbreviations only.

The following two lemmas state that the definitions of application of substitutions and process substitutions to processes are correct.

Lemma A.8. *There is exactly one total function (\cdot) on **Meta★** processes and substitutions satisfying the equations in the bottom part of Figure 4.* ■

B Additional Restrictions on Rewriting Rules

It is desirable to rule out rules and inferences that allows a free name capture, release of a bound name, unleash a nested input-binders, or that introduce a nesting of previously not nested input-binders. To ensure that the aboves do not happen we need additional syntactic restrictions on rewriting rules, and additional conditions that apply to inference rules of $\xrightarrow{\mathcal{R}}$. This section describes them.

In this section, we use the metavariable \dot{z} to range over all template variables, that is name, message, and process variables. The following definition defines the notion of the scope of a bound name variable and some useful notations.

Definition B.1. We say that an occurrence of \dot{z} in \dot{P} is **under the scope** of \dot{x} when \dot{P} contains either:

- (U1) $\dot{A}.\dot{Q}$ with the given occurrence of \dot{z} in \dot{Q} , $\dot{x} \in \text{bv}(\dot{A})$, or
- (U2) $\{\dots \dot{x} := \dot{s} \dots\} \dot{p}$ with $\dot{p} = \dot{z}$ being the given occurrence of \dot{z} .

Write $\dot{P} \vdash_{\exists} \dot{x} > \dot{z}$ when there is an occurrence of \dot{z} in \dot{P} under the scope of \dot{x} .
Write $\dot{P} \vdash_{\forall} \dot{x} > \dot{z}$ when all occurrences of \dot{z} in \dot{P} are under the scope of \dot{x} . ■

The following defines additional restrictions that applies to the left hand side template in a rewriting rules.

Definition B.2. We say that \dot{P} is a **well formed lhs-template** when \dot{P} satisfies the following properties:

- (L1) $\text{fv}(\dot{P}) \cap \text{bv}(\dot{P}) = \emptyset$
- (L2) any message and process variable occurs at most once in \dot{P}
- (L3) \dot{P} does not contain $\{\dot{x}_1 := \dot{s}_1, \dots, \dot{x}_k := \dot{s}_k\} \dot{p}$
- (L4) when \dot{P} contains \dot{A} then every $\dot{x} \in \text{bv}(\dot{A})$ occurs exactly once in \dot{A}
- (L5) when $\dot{P} \vdash_{\exists} \dot{x} > \dot{z}$ then $\dot{P} \vdash_{\forall} \dot{x} > \dot{z}$ ■

Similarly the following restrictions apply to the right hand side templates in a rewriting rule.

Definition B.3. We say that \dot{Q} is a **well formed rhs-template** w.r.t. a well formed lhs-template \dot{P} when \dot{Q} satisfies the following properties:

- (R1) $\text{fv}(\dot{Q}) \subseteq \text{fv}(\dot{P})$
- (R2) $\text{bv}(\dot{Q}) \subseteq \text{bv}(\dot{P})$
- (R3) when \dot{Q} contains $\{\dot{x}_1 := \dot{s}_1, \dots, \dot{x}_k := \dot{s}_k\}$ then \dot{x}_i 's are pairwise distinct
- (R4) when \dot{Q} contains \dot{A} then every $\dot{x} \in \text{bv}(\dot{A})$ occurs exactly once in \dot{A}
- (R5) when $\dot{Q} \vdash_{\exists} \dot{x} > \dot{z}$ then $\dot{Q} \vdash_{\forall} \dot{x} > \dot{z}$
- (R6) for $\dot{z} \in \text{var}(\dot{Q})$ and any \dot{x} holds that $\dot{P} \vdash_{\forall} \dot{x} > \dot{z}$ iff $\dot{Q} \vdash_{\forall} \dot{x} > \dot{z}$ ■

The following introduces the notion of a well formed rewriting rule.

Definition B.4. The rule **rewrite** $\{\dot{P} \leftrightarrow \dot{Q}\}$ is said to be **well formed** when \dot{P} is a well formed lhs-template and \dot{Q} is a well formed rhs-template w.r.t. \dot{P} . The rule **active** $\{\dot{p} \text{ in } \dot{P}\}$ is said to be **well formed** when \dot{P} is a well formed lhs-template. The rule set \mathcal{R} is called a **well formed rule set**, written $\text{wf}(\mathcal{R})$, when all of its rules are well formed. ■

From now on we suppose only well formed rule sets. Alternatively we could add the condition $\text{wf}(\mathcal{R})$ to the premise of rule RRW. Additionally we require the following condition to be satisfied for both RRW and RACT to avoid name captures when picking a name representant for input-binders:

- (I1) whenever $\dot{x}, \dot{y} \in \text{bv}(\dot{P})$ and $\dot{x} \neq \dot{y}$ then $\llbracket \dot{x} \rrbracket_{\rho} \neq \llbracket \dot{y} \rrbracket_{\rho}$, and
- (I2) whenever $\dot{x} \in \text{bv}(\dot{P})$ then $\llbracket \dot{x} \rrbracket_{\rho} \notin \text{fn}(\mathcal{R})$.

<i>Application of a type substitution to form types $\dot{\tau}$:</i>	
$\dot{\tau}(a_0 \dots a_k) = \begin{cases} \Phi & \text{if } k = 0 \ \& \ \tau(a_0) = \Phi* \\ \{\bar{\tau}(a_0) \dots \bar{\tau}(a_k)\} & \text{otherwise} \end{cases}$	
<i>Application of a type substitution to message types $\ddot{\tau}$:</i>	
$\ddot{\tau}(a) = \begin{cases} \tau(a) & \text{if } a \in \text{dom}(\tau) \\ a & \text{otherwise} \end{cases}$ $\ddot{\tau}(\{\varphi_1, \dots, \varphi_k\}*) = (\dot{\tau}(\varphi_1) \cup \dots \cup \dot{\tau}(\varphi_k))*$	
<i>Application of a type substitution to element types $\bar{\tau}$:</i>	
$\bar{\tau}(a) = \begin{cases} \tau(a) & \text{if } \tau(a) \in \text{Name} \\ a & \text{if } a \notin \text{dom}(\tau) \\ \bullet & \text{otherwise} \end{cases}$ $\bar{\tau}(\langle \mu_1, \dots, \mu_k \rangle) = \langle \ddot{\tau}(\mu_1), \dots, \ddot{\tau}(\mu_k) \rangle$ $\bar{\tau}(\langle a_1, \dots, a_k \rangle) = \langle a_1, \dots, a_k \rangle$	

Figure 7: Application of a type substitution to form types, message types, and element types.

C Syntactically Closed Shape Predicates

Definition C.1. Let \mathcal{R} be a rule set. We call a shape predicate π **semantically closed** w.r.t. \mathcal{R} , written $\mathcal{R} \bullet \Rightarrow \pi$, iff

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

Definition C.2. A **type substitution** τ is defined as follows. Extend the definition of shape predicates from Figure 6 as follows:

$$\begin{array}{ll} \tau \in \text{TypeSubstitution} & = \text{Name} \xrightarrow{\text{fin}} \text{AMessageType} \\ \eta \in \text{Edge} & ::= \dots \mid \chi_0 \dashrightarrow \chi_1 \end{array}$$

■

Application of a type substitution to **Meta*** form types, message types, and element types is defined in Fig. 7. Function $\dot{\tau}$ maps form types to sets of form types, $\ddot{\tau}$ maps message types to message types, and $\bar{\tau}$ maps element types to element types. Note that as a special case $\bar{\tau}$ maps names to names.

Definition C.3. The shape graph G is said to be **flow-closed** iff whenever it contains $\chi \xrightarrow{\alpha} \chi'$ and $\chi \dashrightarrow \chi_0$ such that $\text{bn}(\alpha) \cap \text{dom}(\tau) = \emptyset$ then it holds that

- (F1) if $\tau(\alpha) = \{\varphi_1, \dots, \varphi_k\}*$ then $\{\chi_0 \xrightarrow{\varphi_i} \chi_0 : 0 < i \leq k\} \cup \{\chi' \dashrightarrow \chi_0\} \subseteq G$
- (F2) otherwise there is χ'_0 such that $\{\chi' \dashrightarrow \chi'_0, \chi_0 \xrightarrow{\bar{\tau}(\varepsilon_0) \dots \bar{\tau}(\varepsilon_k)} \chi'_0\} \subseteq G$.

We call a shape predicate $\langle G, \chi \rangle$ **flow-closed** iff its G component is. ■

$$\begin{array}{c}
\frac{\chi = \vartheta(\dot{p})}{\vartheta \models_{\mathbb{L}} \dot{p} : \langle G, \chi \rangle} \text{ (CVAR)} \qquad \frac{(\vartheta(\dot{p}) \xrightarrow{\text{[0]}} \chi) \in G}{\vartheta \models_{\mathbb{R}} \dot{p} : \langle G, \chi \rangle} \text{ (CFLOW)} \\
\frac{(\vartheta(\dot{p}) \xrightarrow{\text{[}\dots, \vartheta(\dot{x}_i) \rightarrow \vartheta(\dot{s}_i), \dots\text{]}} \chi) \in G}{\vartheta \models_{\mathbb{R}} \{\dots, \dot{x}_i := \dot{s}_i, \dots\} \dot{p} : \langle G, \chi \rangle} \text{ (CSUB)} \qquad \frac{}{\vartheta \models_s \mathbf{0} : \pi} \text{ (CNUL)} \\
\frac{\vartheta \models_s \dot{P}_0 : \pi \quad \vartheta \models_s \dot{P}_1 : \pi}{\vartheta \models_s \dot{P}_0 \mid \dot{P}_1 : \pi} \text{ (CPAR)} \qquad \frac{(\chi_0 \xrightarrow{\vartheta(\dot{A})} \chi_1) \in G \quad \vartheta \models_s \dot{P} : \langle G, \chi_1 \rangle}{\vartheta \models_s \dot{A}.\dot{P} : \langle G, \chi_0 \rangle} \text{ (CACT)}
\end{array}$$

Figure 8: Matching of process templates to shape graphs. The rules for template processes have an L variant and an R variant; the variable letter s ranges over L and R.

Definition C.4. A **type instantiation** ϑ is a finite function mapping **NameVar** to **BasicName**\{\bullet\}, **MessageVar** to **AMessageType**, and **ProcessVar** to **Node**. Let $\bar{\vartheta}$ denote application of ϑ to **Meta*** element templates, form templates, and to substitutes. It maps element templates to element types, action templates to action types, and substitutes to message types.

$$\begin{array}{ll}
\bar{\vartheta}(a) = a & \bar{\vartheta}(\langle \dot{x}_1, \dots, \dot{x}_k \rangle) = \langle \vartheta(\dot{x}_1), \dots, \vartheta(\dot{x}_k) \rangle \\
\bar{\vartheta}(\dot{x}) = \vartheta(\dot{x}) & \bar{\vartheta}(\langle \dot{m}_1, \dots, \dot{m}_k \rangle) = \langle \vartheta(\dot{m}_1), \dots, \vartheta(\dot{m}_k) \rangle \\
\bar{\vartheta}(\dot{m}) = \vartheta(\dot{m}) & \bar{\vartheta}(\dot{E}_0 \dots \dot{E}_k) = \bar{\vartheta}(\dot{E}_0) \dots \bar{\vartheta}(\dot{E}_k)
\end{array}$$

The relation between type instantiations and process templates is given by the inference system in Figure 8. As a special exception, $\vartheta \models_s \dot{P} : \pi$ is not considered to hold if $\vartheta(\dot{x}_0) = \vartheta(\dot{x}_1)$ for $\dot{x}_0 \neq \dot{x}_1$ such that \dot{x}_0 occurs in \dot{P} below a form template containing a binding element $(\dots, \dot{x}_1, \dots)$. ■

Definition C.5. Let the shape predicate $\pi = \langle G, \chi_0 \rangle$ be given. The set of **active nodes** for \mathcal{R} , written $\text{active}(\pi, \mathcal{R})$, is the least set A of nodes which contains X and such that for all $Y \in A$ and all **active**\{\dot{p} in \dot{P} \} $\in \mathcal{R}$, it holds that $\vartheta \models_{\mathbb{L}} \dot{P} : \langle G, \chi_1 \rangle$ implies $\vartheta(\dot{p}) \in A$. ■

Definition C.6. G is **locally closed** at χ w.r.t. \mathcal{R} iff whenever \mathcal{R} contains **rewrite**\{\dot{P}_0 \leftrightarrow \dot{P}_1\} it holds that $\vartheta \models_{\mathbb{L}} \dot{P}_0 : \langle G, \chi \rangle$ implies $\vartheta \models_{\mathbb{R}} \dot{P}_1 : \langle G, \chi \rangle$. ■

Definition C.7. The shape predicate π is **syntactically closed** w.r.t. \mathcal{R} iff G is flow-closed and also locally closed w.r.t. \mathcal{R} at every $\chi \in \text{active}(\pi, \mathcal{R})$. When this holds, we call π an **\mathcal{R} -type**, denoted by $\mathcal{R} \bullet \rightarrow \pi$. When $\mathcal{R} \bullet \rightarrow \pi$ and $\vdash P : \pi$ we say that π is an **\mathcal{R} -type** of P . ■

Theorem C.8 (Subject reduction). For every $\pi \in \text{ShapePredicate}$ and $\mathcal{R} \in \text{RuleSet}$, it holds that $\mathcal{R} \bullet \rightarrow \pi$ implies $\mathcal{R} \bullet \Rightarrow \pi$. ■

References

- [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.
- [Jak08] Jan Jakubův. *A First Year Report: Process Calculi and the Poly★ System*. Heriot-Watt Univ., School of Math. & Comput. Sci., 2008.
- [MW04] Henning Makhholm 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].
- [MW05] Henning Makhholm 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 [MW04].

D Draft of the Paper

The following document is a draft of a paper that describe the comparison of Poly★ with the seminal type system for Mobile Ambients (TMA). The paper is not yet finished and contains the following known mistakes. Firstly the Poly★ theory described in it is not sound and contains several inconstancies. This is described in details in Section 3 of the report. Secondly the translation of TMA processes to Meta★ is not working as expected. This is described in details in Section 4 of the report. Section 5 of the report provides some reflections on the future of the paper.