The Expressiveness of Generic Process Shape Types

Jan Jakubův  J. B. Wells
Heriot-Watt University  Heriot-Watt University

July 30, 2009

Abstract

META✶ is a generic process calculus that can be instantiated by supplying rewriting rules defining an operational semantics to make numerous process calculi such as the π-calculus, the system of Mobile Ambients, and many of their variants. POLY✶ is a generic type system that makes a sound type system with principal types and a type inference algorithm for any instantiation of META✶. POLY✶ provides a generic notion of shape types which describe behavior of processes by a direct description of allowed syntactic configurations.

This paper evaluates the expressiveness of generic process shape types by comparing POLY✶ with three quite dissimilar type/static analysis systems in the literature. The first comparison is with Turner's type system for the π-calculus without type annotations (which is essentially Milner's system of sorts). The second comparison is with an explicitly typed version of Mobile Ambients by Cardelli and Gordon. Finally, the third comparison is with a static analysis for BioAmbients developed by Nielson, Nielson, Priami, and Rosa. We instantiate META✶ to the process calculi in question and compare expressiveness of POLY✶ shape types with the predicates provided by the three systems. We show that POLY✶ shape types can express much more precise information than the previous type/static analysis systems and can also express essentially the same information as the previous systems.

To do the comparisons, we needed to alter how META✶ handles α-conversion in order to develop a new method for handling name restriction in POLY✶.
## Contents

1 Introduction  
  1.1 Notations and Preliminaries ............................................. 2

2 Metacalculus \textsc{Meta}* and Generic Type System \textsc{Poly}*  
  2.1 General Syntax of Processes ............................................. 3  
  2.2 Instantiations of \textsc{Meta}* ........................................... 5  
  2.3 \textsc{Poly}* Shape Predicates and Types for \textsc{Meta}* ............. 8  
  2.4 \textsc{Poly}* Types and Syntactically Closed Shape Predicates ....... 9

3 Using and Comparing \textsc{Poly}*  
  3.1 General View of Type/Static Analysis Systems ....................... 13  
  3.2 How To Use \textsc{Poly}* ? ................................................ 14  
  3.3 Relating Calculi \textsc{C} and \textsc{C}_\textsc{R} .................................. 14  
  3.4 Comparing Systems \textsc{S}_\textsc{C} and \textsc{S}_\textsc{R} ................. 15

4 \textsc{Poly}* and Types for the \textsc{pi}-calculus  
  4.1 A Polyadic \textsc{pi}-calculus ............................................ 16  
  4.2 Types for the Polyadic \textsc{pi}-calculus (\textsc{Tpi}) .................. 17  
  4.3 Instantiation of \textsc{Meta}* to the \textsc{pi}-calculus .................. 18  
  4.4 Embedding of \textsc{Tpi} in \textsc{Poly}* ................................ 19  
  4.5 Conclusions and Further Discussions ................................ 21

5 \textsc{Poly}* and Types for Mobile Ambients  
  5.1 Mobile Ambients (\textsc{Ma}) ........................................... 21  
  5.2 Types for Mobile Ambients (\textsc{Tma}) .............................. 23  
  5.3 Instantiation of \textsc{Meta}* to \textsc{Ma} ................................ 24  
  5.4 Embedding of \textsc{Tma} in \textsc{Poly}* ................................ 25  
  5.5 Conclusions and Further Possibilities ............................... 29

6 \textsc{Poly}* and Static Analysis of BioAmbients  
  6.1 BioAmbients (\textsc{Ba}) ............................................... 30  
  6.2 Static Analysis of BioAmbients (\textsc{Saba}) ........................ 32  
  6.3 Instantiation of \textsc{Meta}* to BioAmbients .......................... 34  
  6.4 Comparing \textsc{Poly}* Types with \textsc{Saba}'s Results .............. 35  
    6.4.1 Constructing a \textsc{Saba} Predicate from a Shape Type .......... 35  
    6.4.2 Constructing a Shape Type from a \textsc{Saba} Result ............. 36  
  6.5 Conclusions and Further Discussions ................................ 38

7 Conclusions and Future Work .............................................. 39

A Basic Properties of \textsc{Poly}* Shape Types ......................... 41
List of Figures

1 Syntax of META* processes. ............................................. 3
2 Application of a substitution to META* entities. .................... 5
3 Structural equivalence of META*......................................... 5
4 Syntax and semantics of META* reduction rule descriptions. ....... 6
5 Syntax and semantics of POLY* shape predicates. ................... 9
6 Application of a type substitution to POLY* entities. ............... 10
7 Matching of process templates to shape graphs. .................... 11
8 The syntax and semantics of the pi-calculus. ......................... 17
9 Encoding of pi-calculus processes in META*. ........................ 18
10 Property of shape types corresponding to |- of TPI. ................. 20
11 Syntax and semantics of MA. ......................................... 22
12 Syntax of TMA types and typing rules. .............................. 24
13 Encoding of TMA processes in META*. .............................. 26
14 Construction of POLY* type embedding. ............................ 28
15 Syntax and semantics of BA. ......................................... 31
16 SABA analysis of BA* processes. ................................. 33
17 Closure conditions valid for SABA results. ........................ 34
18 Encoding of BA* processes in META*. .............................. 35
19 Construction of a shape graph corresponding to a SABA result. . 37
1 Introduction

Many type systems for many process calculi have been developed to statically guarantee various important properties of processes. Necessary properties of these type systems, such as type soundness, have to be proved for each of them separately. POLY* [MW05, MW04] is a general type system framework which can be instantiated to make type systems for a wide range of process calculi. POLY* works for any process calculus whose semantics can be expressed by rewriting rules which satisfy certain syntactic conditions common for calculi in the literature. Once these conditions are met then only the rewriting rules are necessary to instantiate POLY* to a ready-to-use sound type system. POLY* provides a general concept of polymorphic shape types which can be used to express and verify various properties of processes.

The high level of generality achieved by POLY* inevitably implies a higher complexity of the system and some readers of the earlier paper might find this daunting. Partly to help with this difficulty, and partly to advocate the use of generic process shape types as provided by POLY*, this paper demonstrates how to use the system on concrete examples of well-known process calculi, namely, on the π-calculus [MPW92, Mil99] (Sec. 4), Mobile Ambients [CG98] (Sec. 5), and BioAmbients [RPS/A0] (Sec. 6). We illustrate how POLY* can be used without needing to fully understand all the details of the underlying formalism which is nevertheless presented in Sec. 2. At the same time this paper deals with the important question whether the general notion of process shape types is expressive enough to describe specific properties of processes of various calculi. We show that POLY* can be used to answer the same questions answered by other systems and we back up our claims by formal comparisons with three systems from the literature, namely, with an implicitly typed version of the π-calculus [Tur95], an explicitly typed version of Mobile Ambients [CG99], and a static analysis system for BioAmbients [NNPR07]. The diversity of these systems shows the versatility of possible applications of POLY*. We show on concrete examples polymorphic properties of process shape types which can express polymorphic properties which are not commonly expressible in other systems. The polymorphic power can be further extended using marks [MW04, Sec. 5.6] which are not discussed in this paper.

POLY* is built on the metacalculus META* [MW05, MW04] which is based on the observation that many syntactic constructions have similar semantics in many process calculi found in the literature. Examples of these constructions include parallel composition (“|”), prefixing a process with an action (sometimes called a capability) (“.”), and name restriction (“ν”). Process calculi differ mainly in the set of actions and their meanings. META* collects constructors shared among process calculi and provides a general syntax in which various actions can be encoded. META* is instantiated with a rewriting rule set R that specifies the meanings of actions. META* can be instantiated to many calculi including for example the π-calculus, Mobile Ambients, numerous variations of these, and other systems.

POLY* provides type systems for META*. Instantiating META* by a rewriting
rule set \( \mathcal{R} \) makes a process calculus \( C_{\mathcal{R}} \), and instantiating \( \textsc{Poly} \) by \( \mathcal{R} \) makes a type system \( S_{\mathcal{R}} \) for \( C_{\mathcal{R}} \). \( \textsc{Poly} \) provides \emph{shape predicates} which describe possible syntactic configurations of \( \textsc{Meta} \) processes. The set of shape predicates is shared among all instantiations on \( \textsc{Poly} \). Which shape predicates, however, are considered to be \emph{shape types} in \( S_{\mathcal{R}} \) depends on the set \( \mathcal{R} \). A shape predicate \( \pi \) is a \((\mathcal{R}-)\)type of the system \( S_{\mathcal{R}} \) iff \( \pi \)'s meaning is guaranteed by a simple test to be closed under rewriting with \( \mathcal{R} \). Every instance of \( \textsc{Poly} \) has desirable properties such as subject reduction, the existence of principal typings [Wel02], and an already implemented type inference algorithm.

This paper demonstrates the usage of \( \textsc{Poly} \) and addresses the natural question of how expressive \( \textsc{Poly} \) types are when compared to the predicates of other type/static analysis systems. We use three different systems for three different process calculi to achieve the goal. First in Sec. 4, we show how to instantiate and use \( \textsc{Poly} \) as a type system for the \( \pi \)-calculus, and we compare \( \textsc{Poly} \) with an implicitly typed version of the \( \pi \)-calculus developed by Turner which we name TPI. We show that questions expressible by Turner’s system can be equivalently answered by \( \textsc{Poly} \). Second in Sec. 5, we show how to instantiate \( \textsc{Poly} \) to a type system for Mobile Ambients and we compare this type system with an explicitly typed version of Mobile Ambients (MA) designed by Cardelli and Gordon which we call TMA. Because TMA is one of the first type systems for MA and many other MA type systems are based on it, TMA can viewed as the seminal MA type system. Third in Sec. 6, we demonstrate how to use \( \textsc{Poly} \) as a type system for BioAmbients and compare \( \textsc{Poly} \) with the static analysis system for BioAmbients (SABA) designed by Nielson, Nielson, Priami, and Rosa. We show that \( \textsc{Poly} \) principal typings contain the information provided by SABA results, although they can (and generally will be) more precise. All of the above three sections have the same structure composed of five subsections. These subsections present in turn (1) the calculus \( C \) in question, (2) its previously designed type or static analysis system \( S_C \), (3) an instantiation of \( \textsc{Poly} \) to the type system \( S_{\mathcal{R}} \) for \( C \), (4) basic properties and usage \( S_{\mathcal{R}} \), (5) a formal comparison of \( S_C \) and \( S_{\mathcal{R}} \), and finally (6) conclusions including a discussion of a related work.

This paper also refines \( \alpha \)-conversion in \( \textsc{Meta} \) and uses this to handle name restriction in \( \textsc{Poly} \) more simply than before. This was important for the TMA comparison and simplified the SABA comparison. This change also yields uniform handling of \( \nu \)-bound and input-bound names in \( \textsc{Meta} \). Proofs of the main theorems are found in App. B.

1.1 Notations and Preliminaries

Let \( i, j, k \) range over natural numbers. We shall use one of the following statements (in this case with the same meaning) to express similar claims:

\[
\begin{align*}
\forall i,j,k : & \quad i, j, k \in \text{Nat} := 0 \mid 1 \mid 2 \mid \cdots \\
\forall i,j,k : & \quad i, j, k \in \text{Nat} = \{0, 1, 2, \ldots\}
\end{align*}
\]
\begin{align*}
  \text{a, b} & \in \text{BasicName} := a \mid b \mid \cdots \mid \text{in} \mid \text{out} \mid \text{open} \mid \cdots \mid \emptyset \mid \bullet \mid \cdots \\
  x, y & \in \text{Name} := a^i \\
  F & \in \text{Form} := x_0 \ldots x_k \\
  M & \in \text{Message} := F \mid 0 \mid M_0 : M_1 \\
  E & \in \text{Element} := x \mid (x_1, \ldots, x_k) \mid <M_1, \ldots, M_k> \\
  A & \in \text{Action} := E_0 \ldots E_k \\
  P, Q & \in \text{Process} := 0 \mid A.P \mid (P \mid Q) \mid \nu(x).P \mid !P
\end{align*}

\begin{figure}
\begin{center}
\begin{tabular}{|c|c|}
\hline
\text{a, b} & \in \text{BasicName} := a \mid b \mid \cdots \mid \text{in} \mid \text{out} \mid \text{open} \mid \cdots \mid \emptyset \mid \bullet \mid \cdots \\
\hline
\text{x, y} & \in \text{Name} := a^i \\
\hline
\text{F} & \in \text{Form} := x_0 \ldots x_k \\
\hline
\text{M} & \in \text{Message} := F \mid 0 \mid M_0 : M_1 \\
\hline
\text{E} & \in \text{Element} := x \mid (x_1, \ldots, x_k) \mid <M_1, \ldots, M_k> \\
\hline
\text{A} & \in \text{Action} := E_0 \ldots E_k \\
\hline
\text{P, Q} & \in \text{Process} := 0 \mid A.P \mid (P \mid Q) \mid \nu(x).P \mid !P \\
\hline
\end{tabular}
\end{center}
\caption{Syntax of Meta\textasteriskcentered processes.}
\end{figure}

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

\begin{align*}
  \text{dom}(f) &= \{u : (u \mapsto v) \in f\} & \text{function's domain} \\
  \text{rng}(f) &= \{v : (u \mapsto v) \in f\} & \text{function's range} \\
  f^{-1} &= \{(v, u) : (u \mapsto v) \in f\} & \text{inverse function/relation} \\
  f[u \mapsto v] &= \{(u' \mapsto v') : u \neq u' \} \cup \{u \mapsto v\} & \text{function extension/replacement} \\
  U \rightarrow V &= \{f \in (U \times V) : f \text{ is a function}\} & \text{all functions from } U \text{ to } V \\
  U \rightarrow_{\text{fin}} V &= \{f \in (U \rightarrow V) : f \text{ is finite}\} & \text{all finite functions from } U \text{ to } V
\end{align*}

2 Metacalculus Meta\textasteriskcentered and Generic Type System POLY\textasteriskcentered

This section describes Meta\textasteriskcentered and POLY\textasteriskcentered. This section describes Meta\textasteriskcentered process syntax, instantiating Meta\textasteriskcentered to specific process calculi, and POLY\textasteriskcentered.

Meta\textasteriskcentered [MW05, MW04] has changed: names are now built from basic names and \( \alpha \)-conversion now preserves the underlying basic name (Sec. 2.1). This makes a new simple POLY\textasteriskcentered name restriction rule (Sec. 2.3) do the right thing. The previous handling of name restriction [MW04, Sec. 5.3] was over-complicated and also inadequate for comparisons with other systems.

2.1 General Syntax of Processes

Syntax of processes. Meta\textasteriskcentered process syntax supports simple embeddings of many calculi. Fig. 1 gives the syntax of Meta\textasteriskcentered entities. The metavariable \( Z \) ranges over all Meta\textasteriskcentered entities. A name \( a^i \) is a pair of a basic name \( a \) and a natural number \( i \). The basic part of a name \( x \) is denoted \( x^\flat \), that is, \( a^i = a \). This notation is extended to sets of names structurally, that is, \( X = \{x : x \in X\} \). For a set of basic names \( Y \), we define a similar function \( Y \) which computes the set of all possible names constructed from \( Y \), that is, \( Y = \{a^i : a \in Y \& i \geq 0\} \). When \( \alpha \)-converting, we preserve the basic name and change the number. We write \( a \) instead of \( a^0 \) when no confusion can arise.
Processes are built from the null process "0" by prefixing with an action ("."), by parallel composition ("|"), by name restriction ("\(\nu\)"), and by replication ("\([\cdot]\)". Actions can support prefixes from various calculi such as \(\pi\)-calculus communication actions, MA capabilities, or MA ambient boundaries. Ambient boundaries are further supported by the notation "\(x_1 \ldots x_i [P] y_1 \ldots y_j\)" which abbreviates "\(x_1 \ldots x_i [\cdot] y_1 \ldots y_j . P\)" where "\(\cdot\)" is a (single) special name. For example, "\(x \cdot P\)" stands for "\(x [\cdot]. P\).

Process constructors have standard semantics. "0" is an inactive or finished process, "\(A . P\)" executes the action \(A\) and then continues as \(P\), "\(P \mid Q\)" runs processes \(P\) and \(Q\) in parallel, "\(\nu(x). P\)" behaves as \(P\) with private name \(x\) (i.e., \(x\) in \(P\) differs from all names outside \(P\)), and finally "\(P^i\)" acts as infinitely many copies of \(P\) in parallel ("\(P \mid P \mid \cdots \)\). Let "\(\cdot\)" bind more tightly than "\(\mid\)" and let "\(\nu\)" bind more tightly than "\(\mid\)". These constructors have standard properties, e.g., "\(\mid\)" is commutative, adjacent "\(\nu\)" can be interchanged, etc. In contrast, the semantics of actions is defined by instantiating \(\text{META}\) (see below).

**Free and bound names.** All occurrences of the name \(x\) in "\(\nu(x). P\)" are \(\nu\)-bound. When the action \(A\) contains an element "\((x_1, \ldots, x_k)\)" then all occurrences of the \(x_i\)'s in "\(A . P\)" as well as in \(A\) on its own are called (input-)bound. An occurrence of \(x\) that is not bound is free. The occurrence of \(a\) in \(a^i\) is bound (resp. free) when this occurrence of \(a^i\) is. Bound occurrences of names can be \(\alpha\)-converted which can rename \(a^i\) only to \(a^j\) with \(j\) arbitrary, i.e., preserving the basic name part \(a\). Processes that are \(\alpha\)-convertible are identified. The set of free names of \(P\) is denoted \(fn(P)\) and the set of bound names of the action \(A\) is written \(bn(A)\).

**Well-scopedness.** A process \(P\) is **well scoped** when (W1) its input-bound, \(\nu\)-bound, and free basic names do not overlap, (W2) nested input binders do not bind the same basic name, and (W3) no action contains an input-binding of a basic name more than once. Condition W1 allows simplifications, W2 and W3 are important for type inference, and condition W1 also rules out some processes with an unclear meaning like "\(a^0 (a^0) . a^p . 0\)". Henceforth, we consider only well scoped processes. Well-scopedness can be achieved by renaming if necessary.

**Substitution.** A \(\text{META}\) substitution, denoted \(\sigma\), is a finite function from Name to Message. Fig. 2 defines applying substitutions to \(\text{META}\) entities. This is written postfix as \(M^\sigma\) and \(P^\sigma\) for messages and processes, and as \(Z^\sigma\) for other \(\text{META}\) entities, and binds more tightly than other operators, e.g., \(A . P^\sigma\) stands for \(A . (P^\sigma)\). Let \(fn(\sigma)\) be the names in messages in the range of \(\sigma\). Fig. 2 also defines the message splicing \(M^a P\) which discards empty messages 0 from \(M\) and pushes components of \(M\) from right to left onto \(P\) (for example \(((a . b) . c) a . P = a . b . c . P\).

Substitution replaces names by messages, but non-name messages are \(\text{META}\) syntax errors at some name positions. For example, substituting "in a" for \(b\) in "open b" would yield "open (in a)" which is invalid syntax. In some process calculi, the syntax allows such expressions but they are semantically inert. In
parallel composition, name restriction, and replication. Parallel composition is commutative and associative and has an equivalence relation that satisfies the rules in Fig. 3 and is congruent with the Structural equivalence.

The structural equivalence relation can be extruded from name restriction, parallel composition, and actions when does not vary with instantiation of \( \eta \).

Instead of fixing the semantics of actions, \( \text{META}^* \) provides syntax for specifying rewriting rules that give meaning to actions and also defines how these rules yield a rewriting relation on processes. This is how \( \text{META}^* \) is instantiated to

| Message decomposition operator: |
|---------------------------------|
| \((M_0 . M_1) \sigma P = M_0 \sigma (M_1 \sigma P)\) |
| \(0 \sigma P = P\) |
| \(A_\sigma P = A.P\) |

| Application of a substitution to names, forms, elements, and actions: |
|------------------------|
| \((E_0 \ldots E_k) \sigma = E_0 \sigma \ldots E_k \sigma\) |
| \(\langle x_1, \ldots, x_k \rangle \sigma = \langle x_1, \ldots, x_k \rangle \sigma\) |
| \(\langle M_1, \ldots, M_k \rangle \sigma = \langle M_1 \sigma, \ldots, M_k \sigma \rangle\) |

| Application of a substitution to messages: |
|-------------------------------------------|
| \((M_0 . M_1) \sigma = M_0 \sigma . M_1 \sigma\) |
| \(0 \sigma = 0\) |
| \(F \sigma = \begin{cases} \sigma(F) & \text{if } F = x \in \text{dom}(\sigma) \\ F \sigma & \text{otherwise} \end{cases}\) |

| Application of a substitution to processes: |
|---------------------------------------------|
| \(0 \sigma = 0\) |
| \((P | Q) \sigma = P \sigma | Q \sigma\) |
| \((!P) \sigma = !P \sigma\) |

\[\begin{align*}
\text{Figure 2: Application of a substitution to META* entities.}
\end{align*}\]

\[\begin{align*}
P | Q & \equiv Q | P \quad P | (Q | R) \equiv (P | Q) | R \quad P | 0 \equiv P \\
0 & \equiv !0 \quad \nu(x).\nu(y).P \equiv \nu(y).\nu(x).P \quad !P \equiv P \equiv !P \\
x & \notin \text{fn}(A) \cup \text{bn}(A) \quad x & \notin \text{fn}(P) \\
A.\nu(x).P & \equiv \nu(x).A.P \quad P | \nu(x).Q \equiv \nu(x).(P | Q)
\end{align*}\]

\[\begin{align*}
\text{Figure 3: Structural equivalence of META*.}
\end{align*}\]

\(\text{META*}, \text{ substitution places a special name } “\bullet” \text{ at positions that would otherwise be syntax errors, e.g., the above substitution yields “open } \bullet”.\)

**Structural equivalence.** The structural equivalence relation \( \equiv \) is the smallest equivalence relation that satisfies the rules in Fig. 3 and is congruent with the META* process constructors. This expresses the following standard properties of parallel composition, name restriction, and replication. Parallel composition is commutative and associative and has \( 0 \) as its unit. The scope of name restriction can be extruded from name restriction, parallel composition, and actions when there is no binding conflict. Replication implements repetitive behavior. This basic semantics of operators described by structural equivalence is fixed and does not vary with instantiation of META*.

### 2.2 Instantiations of META*

Instead of fixing the semantics of actions, META* provides syntax for specifying rewriting rules that give meaning to actions and also defines how these rules yield a rewriting relation on processes. This is how META* is instantiated to
make a particular process calculus. Example instantiations are given in Sec. 4.3, 5.3, and 6.3.

Process templates and rewriting rules. Fig. 4 presents the rewriting rule syntax. Process templates are used to describe both left and right-hand sides of rewriting rules. Template syntax resembles process syntax except syntax tree leaves can be variables in addition to names. Variables in templates are replaced during rule instantiation by values of appropriate sorts, i.e., name variables range over names, etc. A substitution application template “\(\{\hat{x}_1 := \hat{s}_1, \ldots, \hat{x}_k := \hat{s}_k\}\) \(\hat{P}\)” describes a substitution to be applied on the right-hand side of some rule. The rewrite rules specify ordinary rewriting rules while active rules describe rewriting contexts, i.e., positions in processes other than at top-level where rewriting rules are to be applied.

An entity instantiation \(\rho\) maps name, message, and process variables respectively to names, messages and processes. Applying \(\rho\) to \(\hat{P}\), written \([\hat{P}]_\rho\), instantiates the template to make a process by filling in values for variables in \(\hat{P}\) as assigned by \(\rho\). We forbid the name “\(\bullet\)” as a value of name variables to prevent distinct earlier error results from being treated as the same name.

Given a rewriting rule set \(\mathcal{R}\), Fig. 4 defines the rewriting relation \(\Rightarrow\). The additional conditions 11 and 12 as well as the set \(\text{fn}(\mathcal{R})\) of all names of \(\mathcal{R}\) are described in the following subsection.

Additional restrictions on rewriting rules. It is desirable to rule out rules and inferences that can capture a free name, release a bound name, unleash a nested input-binders, or that can 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 $\mathcal{A}$). This section describes them.

When the action template $\hat{A}$ contains an element template ‘(x₁, ..., xₖ)’ 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 ‘{$\hat{x}_1 := \hat{s}_1, \ldots, \hat{x}_k := \hat{s}_k$} $\hat{P}$’. Any occurrence of a variable that is not bound is said to be free. The set $\text{fv}(\hat{P})$ of bound variables of the process template $\hat{P}$ is the set of all variables with a bound occurrence. Only a name variable can be bound. The set $\text{fv}(\hat{P})$ of free variables of $\hat{P}$ is the set of all variables with a free occurrence. 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 $\text{fn}(\hat{P})$ is the set of free names of $\hat{P}$ (those element templates $\hat{x}$ that are names $\hat{x}$). For example, given the process template

$\hat{P} = \text{do} (\hat{y}).\hat{A} [.] (\text{out} \hat{b}.\hat{P} \{ \hat{x} := \hat{y} \} \hat{Q})$ we have $\text{bv}(\hat{P}) = \{\hat{x}, \hat{y}\}$, $\text{fv}(\hat{P}) = \{\hat{a}, \hat{b}, \hat{b}, \hat{P}, \hat{Q}\}$, and $\text{fn}(\hat{P}) = \{\text{do}, [], \text{out}\}$. The set $\text{fn}(\mathcal{R})$ of free names of the rule set $\mathcal{R}$ is the union of the sets of free names of all process templates in $\mathcal{R}$.

In this section, we use the metavariable $\hat{z}$ 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 2.1.** We say that an occurrence of $\hat{z}$ in $\hat{P}$ is under the scope of $\hat{x}$ when $\hat{P}$ contains either:

(U1) $\hat{A}, \hat{Q}$ with the given occurrence of $\hat{z}$ in $\hat{Q}$, $\hat{x} \in \text{bv}(\hat{A})$, or

(U2) $\{\ldots \hat{x} := \hat{s} \ldots\} \hat{P}$ with $\hat{P} = \hat{z}$ being the given occurrence of $\hat{z}$.

Write $\hat{P} \vdash_\mathcal{A} \hat{x} \triangleright \hat{z}$ when there is an occurrence of $\hat{z}$ in $\hat{P}$ under the scope of $\hat{x}$. Write $\hat{P} \vdash_\mathcal{V} \hat{x} \triangleleft \hat{z}$ when all occurrences of $\hat{z}$ in $\hat{P}$ are under the scope of $\hat{x}$.

The following defines additional restrictions that applies to left hand side templates in rewriting rules.

**Definition 2.2.** We say that $\hat{P}$ is a well formed lhs-template when $\hat{P}$ satisfies the following properties:

(L1) $\text{fv}(\hat{P}) \cap \text{bv}(\hat{P}) = \emptyset$

(L2) any message and process variable occurs at most once in $\hat{P}$

(L3) $\hat{P}$ does not contain $\{\hat{x}_1 := \hat{s}_1, \ldots, \hat{x}_k := \hat{s}_k\} \hat{P}$

(L4) when $\hat{P}$ contains $\hat{A}$ then every $\hat{x} \in \text{bv}(\hat{A})$ occurs exactly once in $\hat{A}$

(L5) when $\hat{P} \vdash_\mathcal{A} \hat{x} \triangleright \hat{z}$ then $\hat{P} \vdash_\mathcal{V} \hat{x} \triangleright \hat{z}$

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

**Definition 2.3.** We say that $\hat{Q}$ is a well formed rhs-template w.r.t. a well formed lhs-template $\hat{P}$ when $\hat{Q}$ satisfies the following properties:
(R1) $fv(\hat{Q}) \subseteq fv(\hat{P})$
(R2) $bv(\hat{Q}) \subseteq bv(\hat{P})$
(R3) when $\hat{Q}$ contains $\{\hat{x}_1 := \hat{s}_1, \ldots, \hat{x}_k := \hat{s}_k\}$ then $\hat{x}_i$’s are pairwise distinct
(R4) when $\hat{Q}$ contains $\hat{A}$ then every $\hat{x} \in bv(\hat{A})$ occurs exactly once in $\hat{A}$
(R5) when $\hat{Q} \vdash \hat{x} > \hat{z}$ then $\hat{Q} \vdash \hat{x} > \hat{z}$
(R6) for $\hat{z} \in \text{var}(\hat{Q})$ and any $\hat{x}$ holds that $\hat{P} \vdash \hat{x} > \hat{z}$ iff $\hat{Q} \vdash \hat{x} > \hat{z}$

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

**Definition 2.4.** The rule $\text{rewrite}(\{ \hat{P} \leftarrow \hat{Q} \})$ is said to be well formed when $\hat{P}$ is a well formed lhs-template and $\hat{Q}$ is a well formed rhs-template w.r.t. $\hat{P}$. The rule $\text{active}(\hat{p} \in \hat{P})$ is said to be well formed when $\hat{P}$ 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 $\text{RRW}$. Additionally we require the following condition to be satisfied for both $\text{RRW}$ and $\text{RACCT}$ to avoid name captures when picking a name representant for input-binders:

(I1) whenever $\hat{x}, \hat{y} \in bv(\hat{P})$ and $\hat{x} \neq \hat{y}$ then $[\hat{x}]_\rho \neq [\hat{y}]_\rho$, and
(I2) whenever $\hat{x} \in bv(\hat{P})$ then $[\hat{x}]_\rho \notin \text{fn}(\mathcal{R})$.

### 2.3 POLY Shape Predicates and Types for META

**Shape Predicates.** POLY types are built on the notion of shape predicates. A *shape predicate* describes possible structures of process syntax trees. A shape predicate’s meaning is the set of all processes with the given structure. When a rewriting rule from $\mathcal{R}$ is applied to a process, its syntax tree changes, and sometimes the new syntax tree is no longer satisfies the same shape predicates. All POLY (R-)types are shape predicates that describe process sets closed under rewriting using $\mathcal{R}$. For feasibility, types are defined via a syntactic test that enforces rewriting-closedness. Further restrictions are used to ensure the existence of principal typings.

Fig. 5 defines the syntax of shape predicates. Action types describe actions and have corresponding syntax. The main difference between actions and action types 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 describes a set of actions. A shape predicate is a rooted directed finite graph with edges labeled by action types. A process matches a shape predicate $\pi$ when the process’s syntax tree is a “subgraph” of $\pi$. Because a shape predicate can have loops, it can describe syntax trees of arbitrary height.

Fig. 5 also describes matching META entities against types. The rule matching actions against action types also matches forms against form types. Matching processes against shape predicates is independent of rewriting rules, i.e., it
Syntax of \( \text{POLY} \star \) shape predicates:

\[
\begin{align*}
\psi & \in \text{FormType} \quad ::= \ a_0 \ldots a_k \\
\Phi & \in \text{FormTypeSet} \quad ::= \ \mathcal{F}_0(\text{FormType}) \\
\mu & \in \text{MessageType} \quad ::= \ \Phi \star | a \\
\varepsilon & \in \text{ElementType} \quad ::= \ a | (a_1, \ldots, a_k) | \langle \mu_1, \ldots, \mu_k \rangle \\
\alpha & \in \text{ActionType} \quad ::= \ \varepsilon_0 \varepsilon_1 \ldots \varepsilon_k \\
\chi & \in \text{Node} \quad ::= \ X | Y | Z | \cdots \\
\eta & \in \text{Edge} \quad ::= \ \chi_0 \xrightarrow{\alpha} \chi_1 \\
G & \in \text{ShapeGraph} \quad ::= \ \mathcal{F}_0(\text{Edge}) \\
\pi & \in \text{ShapePredicate} \quad ::= \ \langle G, \chi \rangle
\end{align*}
\]

Rules for matching \( \text{META} \star \) entities against shape predicates:

\[
\begin{align*}
\vdash \ a^* : \ a & \quad \vdash \ M : \Phi \quad \text{M \ not BasicName} \\
\vdash F : \phi & \quad \phi \in \Phi \\
\vdash 0 : \Phi & \quad \vdash M_0 : \Phi \quad \vdash M_1 : \Phi \\
\forall i : 0 < i \leq k & \quad \vdash x_i : a_i \\
\vdash \langle x_1, \ldots, x_k \rangle : \langle a_1, \ldots, a_k \rangle & \\
\forall i : 0 < i \leq k & \quad \vdash \langle M_1, \ldots, M_k \rangle : \langle \mu_1, \ldots, \mu_k \rangle \\
\forall i \leq k & \quad \vdash E_0 \ldots E_k : \varepsilon_0 \ldots \varepsilon_k \\
\vdash (\chi_0 \xrightarrow{\alpha} \chi_1) \in G & \quad \vdash A : \alpha \quad \vdash P : \langle G, \chi_0 \rangle \\
\vdash A \cdot P : \langle G, \chi_0 \rangle & \quad \vdash P : \pi \\
\forall \chi : \vdash \nu(\chi).P : \pi & \\
\vdash P : \pi & \\
\vdash \vdash P : \pi
\end{align*}
\]

Figure 5: Syntax and semantics of \( \text{POLY} \star \) shape predicates.

works the same in any \( \text{META} \star \) instantiation. The meaning of the shape predicate \( \pi \), written \( [\pi] \), is the set of all processes matching \( \pi \), namely \( \{ P : \vdash \pi \} \).

2.4 \( \text{POLY} \star \) Types and Syntactically Closed Shape Predicates

This section provides details about syntactically closed shape predicates regarded as \( \text{POLY} \star \) types.

Semantically Closed Shape Predicates. A shape predicate \( \pi \) is semantically closed w.r.t. rewriting rule set \( \mathcal{R} \) when the meaning of \( \pi \) is closed under \( \mathcal{R} \)-rewritings. Formally as follows.

Definition 2.5. Let \( \mathcal{R} \) be a rule set. We call the shape predicate \( \pi \) semantically closed w.r.t. \( \mathcal{R} \), written \( \mathcal{R} \Rightarrow \pi \), iff

\[
\vdash P : \pi \text{ and } P \xrightarrow{\mathcal{R}} Q \quad \text{imply} \quad \vdash Q : \pi
\]
### Application of a type substitution to form types \( \tau \):

\[
(a_0 \ldots a_k)\tau = \begin{cases} 
\Phi & \text{if } k = 0 \& \tau(a_0) = \Phi^* \\
\{(a_0\tau)\ldots (a_k\tau)\} & \text{otherwise}
\end{cases}
\]

### Application of a type substitution to message types \( \tau \):

\[
a\tau = \begin{cases} 
\tau(a) & \text{if } a \in \text{dom}(\tau) \\
a & \text{otherwise}
\end{cases}
\]

\[
(\{\varphi_1, \ldots, \varphi_k\})\tau = (\varphi_1\tau \cup \ldots \cup \varphi_k\tau)^*
\]

### Application of a type substitution to element types and actions:

\[
a\tau = \begin{cases} 
\tau(a) & \text{if } a \in \text{Name} \\
a & \text{if } a \notin \text{dom}(\sigma) \\
\cdot & \text{otherwise}
\end{cases}
\]

\[
(\epsilon_0 \ldots \epsilon_k)\tau = (\epsilon_0\tau \ldots (\epsilon_k\tau)
\]

**Figure 6**: Application of a type substitution to \( \text{POLY}^* \) entities.

Because deciding if a shape predicate is semantically closed w.r.t. an arbitrary \( R \) is nontrivial, we use an easier-to-decide property of shape predicates, namely syntactic closure, which by design is algorithmically verifiable. The following sections lead to the definition of this notion.

**Type Substitutions and Flow Edges.** Shape graphs also contain flow edges, which are used in type inference algorithms and in recognizing syntactic closure. While the action edges shown in Fig. 5 are labeled with action types, flow edges are labeled instead with type substitutions, finite functions from basic names to message types. A type substitution \( \tau \) represents a set of \( \text{META}^* \) substitutions.

**DEFINITION 2.6.** The type substitution \( \tau \) is defined below. We extend the definition of shape predicates from Fig. 5 as follows:

\[
\tau \in \text{TypeSubstitution} = \text{Name} \rightarrow \text{MessageType} \\
\eta \in \text{Edge} ::= \cdot | \chi_0 \rightarrow \chi_1
\]

Application of a type substitution to \( \text{META}^* \) form types, message types, and element types is defined in Fig. 6. The postfix application \( \tau \) maps form types to sets of form types, \( \tau \) maps message types to message types, and the postfix application of \( \tau \) maps element types to element types. Note that as a special case the postfix application of \( \tau \) maps basic names to basic names.

A flow edge \( (\chi \rightarrow \chi') \in G \) expresses the property that whenever the process \( P \) matches \( \langle G, \chi \rangle \) and \( \sigma \) is a substitution represented by \( \tau \), then the process \( P\sigma \) must match \( \langle G, \chi' \rangle \). Flow edges describe possible movements of processes that involve substitution application. Syntactic closure insists on the presence of flow edges implied by rewriting rules or other flow edges. Of course the above property associated with a flow edge is not satisfied automatically.
The Meaning of Flow Edges. A shape graph is flow closed when the intuitive meaning of flow edges described in the previous paragraph is satisfied. Note that while the existence of particular flow edges is involved by the set of rewriting rules \( R \), the notion of flow-closedness itself does not depend on \( R \). Bound basic names of the action type \( \alpha \), denoted \( bn(\alpha) \), are those basic names of \( \alpha \) that appear inside some input element type \( (a_1, \ldots, a_k) \).

**Definition 2.7.** The shape graph \( G \) is said to be flow-closed iff whenever it contains \( \chi \stackrel{\alpha}{\rightarrow} \chi' \) and \( \chi \stackrel{\beta}{\rightarrow} \chi_0 \) such that \( bn(\alpha) \cap \text{dom}(\tau) = \emptyset \) then it holds that

(F1) if \( \tau(\alpha) = \{\varphi_1, \ldots, \varphi_k\} \star \) then \( \{\chi_0 \stackrel{\alpha i}{\rightarrow} \chi_0: 0 < i \leq k\} \cup \{\chi' \stackrel{\beta}{\rightarrow} \chi_0\} \subseteq G \)

(F2) otherwise there is \( \chi'_0 \) such that \( \{\chi' \stackrel{\beta}{\rightarrow} \chi'_0, \chi_0 \stackrel{\alpha r}{\rightarrow} \chi'_0\} \subseteq G \).

We call the shape predicate \( (G, \chi) \) flow-closed iff its \( G \) component is.

Syntactically Closed Shape Predicates. Type instantiations are used to relate process templates and shapes graph just like entity instantiations are used to relate templates and processes. However, the below defined relation between templates and shape graphs is not functional like in the case of templates and processes, where an entity instantiation and a template unambiguously determine a process. Rather, it relates a type instantiation and a template with several shape predicates. The formal definition is as follows.

**Definition 2.8.** A type instantiation \( \theta \) is a finite function mapping NameVar to BasicName\( \setminus \{\ast\} \), MessageVar to MessageType, and ProcessVar to Node. Let \( \bar{\theta} \) denote application of \( \theta \) to META\( \ast \) element templates, form templates, and to substitutes. It maps element templates to element types, action templates to action types, and substitutes to message types.

\[
\begin{align*}
\bar{\theta}(a^i) &= a & \bar{\theta}((\tilde{x}_1, \ldots, \tilde{x}_k)) &= (\theta(\tilde{x}_1), \ldots, \theta(\tilde{x}_k)) \\
\bar{\theta}(\tilde{x}) &= \bar{\theta}(\tilde{x}) & \bar{\theta}(<\tilde{m}_1, \ldots, \tilde{m}_k>) &= <\theta(\tilde{m}_1), \ldots, \theta(\tilde{m}_k)> \\
\bar{\theta}(\tilde{m}) &= \theta(\tilde{m}) & \bar{\theta}(\tilde{E}_0 \ldots \tilde{E}_k) &= \bar{\theta}(\tilde{E}_0) \ldots \bar{\theta}(\tilde{E}_k)
\end{align*}
\]

The relation between type instantiations and process templates is given by the inference system in Fig. 7. As a special exception, \( \bar{\theta} \models s \bar{P} : \pi \) is not considered to hold if \( \theta(\tilde{x}_0) = \theta(\tilde{x}_1) \) for \( \tilde{x}_0 \neq \tilde{x}_1 \) such that \( \tilde{x}_0 \) occurs in \( \bar{P} \) below a form template containing a binding element \( (\ldots, \tilde{x}_1, \ldots) \).

**Figure 7:** 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.
The relation $\trianglerighteq_L$ is used to relate a left-hand-side template with a shape graph. Informally, $\varnothing, \trianglerighteq_L \mathcal{P} : \pi$ says that $\mathcal{P}$ can be instatiated to some process that matches $\pi$ and that $\varnothing$ describes this instatiation. Alternatively, the above statement says that $\mathcal{P}$ can be “attached” to $\pi$ when variables in $\mathcal{P}$ are filled in accordingly to $\varnothing$. Similarly, $\trianglerighteq_R$ is used to relate right-hand-side templates with shape graphs. It differs in that it takes exisitning flow edges into account and it allows the template to contain substitution constructions. Suppose the rewriting rule $\text{rewrite}(\mathcal{P} \leftrightarrow \mathcal{Q})$ is given and that $\varnothing, \trianglerighteq_L \mathcal{P} : \pi$ holds. Then the shape graph $\pi$ has to contain certain additional edges in order for $\varnothing, \trianglerighteq_R \mathcal{Q} : \pi$ to hold. In this way these two relations are used to force the existence of edges important for the meaning of $\pi$ to be closed under the rewriting rule.

The following defines the set of active nodes determined by active rules, that is, the set of nodes where rewritings rules are to be applied.

**Definition 2.9.** 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 $\mathcal{P}$ in $\mathcal{R}$, it holds that $\varnothing, \trianglerighteq_L \mathcal{P} : \langle G, \chi_1 \rangle$ implies $\varnothing, \trianglerighteq_R \mathcal{Q} : \langle G, \chi \rangle$.

A shape predicate is locally closed at some node when application of any of the rewriting rules at the node does not insist the existence of a new flow edge.

**Definition 2.10.** $G$ is locally closed at $\chi$ w.r.t. $\mathcal{R}$ iff whenever $\mathcal{R}$ contains the rule $\text{rewrite}(\mathcal{P} \leftrightarrow \mathcal{Q})$ it holds that $\varnothing, \trianglerighteq_L \mathcal{P} : \langle G, \chi \rangle$ implies $\varnothing, \trianglerighteq_R \mathcal{Q} : \langle G, \chi \rangle$.

Finally, syntactically closed shape predicates are those both flow closed and locally closed at any active node.

**Definition 2.11.** The shape predicate $\pi$ 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 $\pi$ an $\mathcal{R}$-type, denoted by $\mathcal{R} \ni \pi$. When $\mathcal{R} \ni \pi$ and $\vdash \mathcal{P} : \pi$ we say that $\pi$ is an $\mathcal{R}$-type of $\mathcal{P}$.

Subject reduction says that the meaning of a syntactically closed shape predicate is closed under rewritings.

**Theorem 2.12 (Subject Reduction).** For every $\pi \in \text{ShapePredicate}$ and $\mathcal{R} \in \text{RuleSet}$, it holds that $\mathcal{R} \ni \pi$ implies $\mathcal{R} \ni \pi$.

**Poly Principal Typings.** A type $\pi$ of $\mathcal{P}$ is a principal typing of $\mathcal{P}$ when $[\pi_\pi] \subseteq [\pi_0]$ for any other type $\pi_0$ of $\mathcal{P}$. The following section defines two properties of shape graphs called the width and the depth restriction. Among Poly Principal Typings satisfying these restriction the existence of the principal typing for every $\mathcal{P}$ hold.

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

**Definition 2.13.** Write $\alpha_0 \approx \alpha_1$ iff there is $A$ such that $\vdash A : \alpha_0$ and $\vdash A : \alpha_1$.

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

**Definition 2.14.** $G \in \text{ShapeGraph}$ satisfies the width restriction iff whenever there are two edges $(\chi \xrightarrow{\alpha} \chi_0) \in G$ and $(\chi \xrightarrow{\alpha'} \chi_1) \in G$ with $\alpha \approx \alpha'$, then it holds that $\chi_0 = \chi_1$.

**Definition 2.15.** $G \in \text{ShapeGraph}$ satisfies the depth restriction iff whenever there is a path of edges in $G$ like $\chi_0 \xrightarrow{\alpha_1} \chi_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_k} \chi_k$ with $\alpha_1 \approx \alpha_k$, then it holds that $\chi_1 = \chi_k$.

Among syntactically closed shape predicates that satisfy the width and depth restrictions the principal typing property can be proved [MW04]. Moreover, the type inference algorithm is implemented.

### 3 Using and Comparing POLY∗

This section discusses type and static analysis systems, how to use POLY∗ to achieve goals attained by other systems, how to use POLY∗ on its own, how and why to relate META∗ with other calculi, and several possibilities how a formal comparison of POLY∗ with other systems can be made.

#### 3.1 General View of Type/Static Analysis Systems

Different type and static analysis systems are designed for different purposes. A single system is usually intended to statically verify a specific fixed property of processes of a given calculus, for example, that a process does not execute an ill-formed instruction. Commonly, it is easy to verify this property for a specific process. On the other hand, to verify that the property in question is satisfied for a given process and for all of its successor states, is generally much more complicated task. Type and static analysis systems provide an effective solution of this problem.

A typical type or static analysis system $S_C$ for the process calculus $C$ works as follows. Firstly, it defines the set of predicates. Let $\varphi$ range over it. Predicates formally represent properties which the system reasons about and verifies. Secondly, the system defines a binary relation on processes and predicates. Let $B$ range over processes of $C$. Let us write the relation as $\triangleright B : \varphi$. This relation formally represents the statement “$B$ has the property $\varphi$”. The relation is desired to be effectively verifiable. Thirdly, the system has to enjoy the subject reduction property, which states that the relation $\triangleright$ is preserved under rewriter of processes, that is, $\triangleright B_0 : \varphi$ implies $\triangleright B_1 : \varphi$ for any successor state $B_1$ of the process $B_0$. 

3.2 How To Use POLY*?

Consider some type/static analysis system $S_C$ with the above properties (Sec. 3.1). Furthermore suppose that the rewriting rules of the calculus $C$ can be described in $\text{META}*$ by the set of rewriting rules $R$. This gives us the instantiation $C_R$ of $\text{META}*$ and the type system $S_R$ provided by POLY*. Now suppose that we would like to compare expressiveness of $S_C$ and $S_R$. We would like to know whether questions answerable by the relation $\triangleright$ of $S_C$ can be expressed and answered within $S_R$. Being able to do this for several different systems from the literature would lead us to the conclusion that the generic concept of POLY* shape types is at least as expressive as the single-purpose predicates of the compared systems. We shall show how to do this for three systems from the literature. Several approaches to do this are outlined in Sec. 3.4.

Some systems are designed to verify a certain fixed property of processes. Then we can use POLY* directly without referencing $S_C$. Let $P$ denote the property in question. For these systems the following holds: When there exists some $\varphi$ such that $\triangleright B : \varphi$ then $B$ has property $P$. Usually some over approximation is encountered, which means that the opposite implication is not always satisfied. This is a commonly accepted trade-off between preciseness and time complexity of the verification of $P$. Suppose that we can formulate a condition on a POLY* shape type $\pi$ whose fulfillment implies that every process matching $\pi$ has property $P$. Then we can use POLY* to verify $P$ directly. We can also design property $P$ directly without any references to other type or static analysis systems. We shall show how to use POLY* to verify communication safety of the $\pi$-calculus and Mobile Ambients processes. We show that POLY* provides better results, that is, it over approximates communication safety less, than other two systems designed specifically for this purpose. Moreover, we show that POLY* can also exactly recognize whether or not $\triangleright B : \varphi$ holds for a given $B$ and $\varphi$ in these two systems. This is important because the relation $\triangleright B : \varphi$ might be used by some applications of $S_C$ for various purposes and not only to verify property communication safety.

3.3 Relating Calculi $C$ and $C_R$

POLY* requires the set of rewriting rules $R$ to become an instantiated type system. Calculi $C_R$ and $C$ are usually reasonably equivalent. Nevertheless, for the reason of a formal comparison of the systems $S_C$ and $S_R$ a reasonable relationship between the calculus $C$ and $C_R$ has to be proved. It is important to understand, however, that a similar relationship has to be established only for the reasons of formal comparisons, such are those presented in the next sections, and it is not required for a standard use of POLY*.

At first we need an encoding $\{ \}$ which translates processes of $C$ into $\text{META}*$ processes. Because of a benevolent syntax of $\text{META}*$, this encoding is in many cases almost an identity. To avoid technical problems, such as a different handling of $\alpha$-conversion, we suppose that processes of $C$ are build from $\text{META}*$ names, and that $C$ does $\alpha$-conversion as $\text{META}*$. One can easily construct an
equivalent version of $C$ which meets this requirement when necessary. We suppose that the encoding $\{ \} \downarrow$ preserves free and basic bound names. The relationship between the rewriting relation $\rightarrow$ of $C$ and the relation $\overset{\approx}{\rightarrow}$ of Meta* has two parts. The first says that $B_0 \rightarrow B_1$ implies $\{B_0\} \overset{\approx}{\rightarrow} \{B_1\}$. The second ensures that whenever $\{B_0\} \overset{\approx}{\rightarrow} P_1$ then $P_1$ is the translation of some $B_1$ such that $B_0 \rightarrow B_1$. To handle subtle differences in structural congruences of different calculi, we formulate this property modulo structural congruences $\equiv$ of $S_C$ and Meta*. As mentioned, proving Prop. 3.1 is usually easy.

**Property 3.1.** When $B_0 \rightarrow B_1$ then there are $B'_0$ and $B'_1$ such that $B_0 \equiv B'_0$ & $\{B'_0\} \overset{\approx}{\rightarrow} \{B'_1\}$ & $B'_1 \equiv B_1$. When $\{B_0\} \overset{\approx}{\rightarrow} P_1$ then there is $B_1$ such that $B_0 \rightarrow B_1$ & $\{B_1\} \equiv P_1$.

### 3.4 Comparing Systems $S_C$ and $S_R$

A straightforward way to relate $S_C$ and $S_R$ is to define a translation $\{ \} \downarrow$ of predicates of $S_C$ to Poly* types such that $\triangleright B :: \varphi$ if and only if $\vdash \{B\} :: \{\varphi\}$. This approach is not possible when the relation $\triangleright$ of $S_C$ is preserved under renaming of bound basic names of a process. Recall that we suppose that $C$ builds processes from Meta* (basic) names. Unfortunately, this is the case of majority of the systems in literature, especially of those we work with in this paper. The problem is that bound basic names are used to build Poly* shape types. Thus, when a bound basic name in a process is changed then the new process does not need to match the same types as before.

In order to explain this, let us suppose the encoding $\{ \} \downarrow$ with the above desired property, and let $\triangleright (\forall x)B_0 :: \varphi$ for some $B_0$ and $\varphi$ such that $x \in \text{fn}(B_0)$. Now, because both $\{(\forall x)(B_0)\}$ and $\{\varphi\}$ are finite objects we can take some basic name $a$ which is in none of them. Let us take $B' = (\forall a^0)(B_0\{x \mapsto a^0\})$. Because $\triangleright$ is preserved under renaming of bound basic names of $S_C$'s processes, we have that also $\triangleright B' :: \varphi$. But we can see that $\{B'\}$ can hardly match $\{\varphi\}$ because $\{\varphi\}$ does not contain any $a$ necessary to match occurrences of $a^0$ in $\{B'\}$. Similar argument can be made even when we do not require $\{ \} \downarrow$ to preserve bound basic names.

We investigate another ways to compare $S_C$ and $S_R$ to avoid the above problem. The first one, which we use in Sec. 4 to compare Poly* with a typed version of the $\pi$-calculus, exploits the existence of principal types for processes in $S_R$. We answer the question $\triangleright B :: \varphi$ by performing a simple check on $\tau_B$. Formally we define the relation $\varphi \equiv \pi$, which says that $\varphi$ "agrees" with $\pi$, and we prove that $\triangleright B :: \varphi$ if and only if $\varphi \equiv \pi_B$.

Another approach to compare $S_C$ and $S_R$ is an enhancement of the straightforward comparison outlined above. We equip a translation of $S_C$'s predicate $\varphi$ into a Poly* type with necessary information $I_B$ about bound names of the process $B$. We translate $\varphi$ and $I_B$ into the Poly* type $\{\varphi, I_B\}$ and we prove that $\triangleright B :: \varphi$ if and only if $\vdash \{B\} :: \{\varphi, I_B\}$. We use this approach to compare Poly* with a typed version of Mobile Ambients in Sec. 5.
Yet another style of comparison is used to compare POLY with a static analysis system for BioAmbients in Sec. 6. For a given $B$, we compute a POLY principal type $\pi_B$ of $\{B\}$ and use it to construct a predicate $\varphi_B$ of system $S_C$ such that $B : \varphi_B$. Then we take the actual result $\varphi$ of the analysis of $B$ computed by $S_C$ and we prove that $\varphi_B$ constructed from $\pi_B$ is at least as precise as $\varphi$, let us write it as $\varphi_B \subseteq \varphi$. Our result says that $B : \varphi$ implies $\varphi_B \subseteq \varphi$. The opposite implication would not give a meaningful result in this particular case. We also discuss that the second above approach can be used.

4 POLY and Types for the $\pi$-calculus

This section demonstrates how to use POLY with a polyadic $\pi$-calculus and compares POLY with a type system for the $\pi$-calculus. Sec. 4.1 introduces the $\pi$-calculus, Sec. 4.2 introduces a type system from the literature, Sec. 4.3 describes the type system for the $\pi$-calculus provided by POLY, Sec 4.4 formally compares expressiveness of the above two systems, and finally Sec. 4.5 provides conclusions and discussion of a related work.

4.1 A Polyadic $\pi$-calculus

The $\pi$-calculus [MPW92, Mil99] is a process calculus involving process mobility developed by Milner, Parrow, and Walker. Mobility is abstracted as channel-based communication whose objects are atomic names. The $\pi$-calculus does not distinguish between names and channel labels thus allowing channels to be passed by communication as well. This ability is commonly referred as link passing and it is the most distinctive feature of the $\pi$-calculus from its predecessors. We consider a polyadic version of the $\pi$-calculus which supports communication of tuples of names.

Fig. 9 presents the syntax, structural equivalence, and semantics of the $\pi$-calculus. Processes are built from META names. The process “$c(n_1, \ldots, n_k).B$” waits to receive a k-tuple of names over channel $c$ and then behaves like $B$ with the received values substituted for the occurrences of $n_1, \ldots, n_k$. The output action (input-)binds the names $n_1, \ldots, n_k$ in the process. The process “$c<n_1, \ldots, n_k>B$” sends the k-tuple $n_1, \ldots, n_k$ over channel $c$ and then behaves like $B$. Other constructors have the same meaning as the corresponding META ones described in Sec. 2.1. Analogously, $n$ is (\nu-bound in “(\forall n)B”. The set fn($B$) is the set of free (i.e., not bound) names of $B$, and fn(B) is the corresponding set of free basic names of $B$. As in META, we define the sets of input-bound, and \nu-bound basic names of a process $B$.

Again, processes are identified up to $\alpha$-conversion of bound names which preserves basic names. A substitution in the $\pi$-calculus is a finite function from names to names, and its application to processes is written postfix, e.g., “$B[m_1 \mapsto n_1]$”. A process $B$ is well scoped when (S1) its input-bound, \nu-bound, and free basic names do not overlap, (S2) nested input binders do not bind the same basic name, and (S3) no input action contains the same basic name more then
### Syntax of the \( \pi \)-calculus processes:

\[
\begin{align*}
\text{c, n, m} & \in \text{PiName} \rightarrow \text{Name} \\
N & \in \text{PiAction} ::= \text{c}(n_1, \ldots, n_k) | \text{c}<n_1, \ldots, n_k> \\
B & \in \text{PiProcess} ::= \text{0} | (B_0 | B_1) | \text{N.B} | \text{!B} | (\text{vn})B
\end{align*}
\]

### Structural equivalence of the \( \pi \)-calculus:

\[
\begin{align*}
B & \equiv B | 0 \\
\text{!}B & \equiv B | \text{!B} \\
\text{!}B_0 | (B_1 | B_2) & \equiv (B_0 | B_1) | B_2 \\
(\text{vn})B_0 | B_1 & \equiv (\text{vn})(B_0 | B_1) & \text{if } n \not\in \text{fn}(B_1)
\end{align*}
\]

### Rewriting relation of the \( \pi \)-calculus:

\[
\begin{align*}
\text{c}(n_1, \ldots, n_k).B_0 | \text{c}<m_1, \ldots, m_k>.B_1 \rightarrow \\
B_0[n_1 \rightarrow m_1, \ldots, n_k \rightarrow m_k] | B_1 \\
B_0 & \rightarrow B_1 \Rightarrow (\text{vn})B_0 \rightarrow (\text{vn})B_1 \\
B_0 & \rightarrow B_1 \Rightarrow B_0 | B_2 \rightarrow B_1 | B_2 \\
B_0' & \equiv B_0 \& B_0 \rightarrow B_1 \& B_1 \equiv B_1' \Rightarrow B_0' \rightarrow B_1'
\end{align*}
\]

**Figure 8:** The syntax and semantics of the \( \pi \)-calculus.

Once. Henceforth, we require processes to be well scoped. (Well-scopedness can be achieved by as initial renaming and it is preserved by rewriting.)

**Example 4.1.** Consider the following \( \pi \)-process:

\[
B = \text{!s}(x, y).x<y>.0 | s<a, n>.0 | a(v).v(p).0 | n<o>.0 | s<b, m>.0 | b(w).v(q, r).0 | m<o, o>.0
\]

*Using the rewriting relation \( \rightarrow \) sequentially four times we can obtain (among others) the following process:*

\[
\text{!s}(x, y).x<y>.0 | n(p).0 | n<o>.0 | m(q, r).0 | m<o, o>.0
\]

### 4.2 Types for the Polyadic \( \pi \)-calculus (T\( \pi \))

We compare \textsc{Poly\star} with a simple type system for the polyadic \( \pi \)-calculus [Tur95, Ch. 3] presented by Turner which we name T\( \pi \). This system is essentially similar to Milner’s sorts discipline [Mil99]. In the presence of polyadic communication, an arity mismatch error on channel \( c \) can occur when the lengths of the sent and received tuple do not agree, for example, like in “\( \text{c}(n).0 | \text{c}<m, m>.0 \)”. Processes which can never evolve to a state with a similar situation are called communication safe. T\( \pi \) is designed to check communication safety of \( \pi \)-processes.

The syntax and typing rules of T\( \pi \) are presented in Fig. 9. Recall that \( n \) denotes the basic name of \( n \). Types \( \delta \) are assigned to names. Type variables \( \beta \) are types of names which are not used as channel labels. The channel type \( "[\delta_1, \ldots, \delta_k]" \) describes a channel which can be used to communicate any \( k \)-tuple whose \( i \)-th member is a name of type \( \delta_i \) (for all \( 0 < i \leq k \)). A context
Syntax of TPI types:

$$\begin{align*}
\beta & \in \text{PiTypeVariable} \ ::= \ i | i' | i'' | \cdots \\
\delta & \in \text{PiType} \ ::= \ \beta \uparrow [\delta_1, \ldots, \delta_k] \\
\Delta & \in \text{PiContext} \ ::= \ \text{BasicName} \to [\text{PiType}]
\end{align*}$$

Typing rules of TPI:

$$\begin{align*}
\Delta & \vdash \text{B}_0 \quad \Delta \vdash \text{B}_1 \quad \Delta \vdash \text{B} \quad \Delta[n_1 \mapsto \delta] \vdash \text{B} \\
\Delta & \vdash \text{c}(\text{n}_1, \ldots, \text{n}_k) \cdot \text{B} \\
\Delta(g) & \vdash \uparrow [\delta_1, \ldots, \delta_k] \quad \Delta[\text{n}_1 \mapsto \delta_1, \ldots, \text{n}_k \mapsto \delta_k] \vdash \text{B} \\
\Delta & \vdash \text{c}(<\text{n}_1, \ldots, \text{n}_k>) \cdot \text{B}
\end{align*}$$

Figure 9: Syntax of TPI types and typing rules.

$\Delta$ assigns types to free names of a process (via their basic names$^1$). The relation $\Delta \vdash \text{B}$ then expresses that the actual usage of channels in $\text{B}$ agrees with requirements stated by $\Delta$. The subject reduction property holds for TPI. Moreover, when there is some $\Delta$ such that $\Delta \vdash \text{B}$ then $\text{B}$ is communication safe. The opposite implication does not necessarily hold.

Example 4.2. Given the process $\text{B}$ from Ex. 4.1 we can see that there is no $\Delta$ such that $\Delta \vdash \text{B}$. It is because the parts $<\text{a}, \text{n}>$ and $<\text{a}, \text{m}>$ imply that types of $\text{n}$ and $\text{m}$ must be equal while the parts $<\text{o} \circ \text{o}>$ and $<\text{m} \circ \text{o}, \text{o} \circ \text{o}>$ force them to be different. On the other hand we could see that $\text{B}$ is communication safe. We check this using POLY$\ast$ in Sec 4.3.

4.3 Instantiation of META$\ast$ to the $\pi$-calculus

In order to use META$\ast$ and POLY$\ast$ one needs to choose a syntax of processes which matches the syntax of META$\ast$. It is easy in the case of the $\pi$-calculus because our syntax from the previous section (Fig. 8) already matches META$\ast$. META$\ast$ can be instantiated to this $\pi$-calculus as follows:

$$\mathcal{P} = \bigcup_{k=0}^\infty \{ \text{rewrite} \{ \hat{c} \{ \hat{\text{n}}_1, \ldots, \hat{\text{n}}_k \} \cdot \hat{\text{p}} | \hat{\text{c}}(\hat{\text{a}}_1, \ldots, \hat{\text{a}}_k) \cdot \hat{\text{q}} \mapsto \\
\hat{\text{p}} | \{ \hat{\text{a}}_1 := \hat{\text{n}}_1, \ldots, \hat{\text{a}}_k := \hat{\text{n}}_k \} \cdot \hat{\text{q}} \} \}$$

Each communication prefix length has its own rule; in our implementation, a single rule can uniformly handle all lengths, but the formal META$\ast$ presentation is deliberately simpler.

Providing the set $\mathcal{P}$ is all that needs to be done to instantiate META$\ast$ to the calculus $C_\mathcal{P}$ and POLY$\ast$ to its type system $S_\mathcal{P}$. In Sec. 4.4 we show that $C_\mathcal{P}$ is essentially the same as the polyadic $\pi$-calculus from the previous section.

$^1$Turner’s original system does not use META$\ast$ basic names and assigns types directly to names. This variation allows a simpler presentation of the correspondence with POLY$\ast$. 
The next example shows how to use POLY* to check communication safety of π-processes directly without using TPI.

**Example 4.3.** Let P be a META* equivalent of B from Ex. 4.1. We can use the POLY*'s type inference algorithm to compute a principal type π_P of P which looks as follows:

Here R is its root node. The type π_P contains all computational futures of P in one place. Thus, because there are no two edges from the root node labeled by “a(b_1, . . . , b_j)” and “a<b_1, . . . , b'>” with k ≠ j, we can conclude that P is communication safe which Ex. 4.2 shows TPI can not do. Our type inference implementation can be instructed (using an additional rule) to insert a special error name at the place of communication errors. Then checking communication safety is equivalent to checking the presence of the special error name.

### 4.4 Embedding of TPI in POLY*

Applying the terminology from Sec. 3.1 we have that C is the polyadic π-calculus, S_C is Turner's system TPI, predicates ϕ of S_C are TPI's contexts Δ, and S_C's relation △ B : ϕ is the typing relation Δ ⊲ B. Moreover R is P which was together with C_P and S_P introduced in the previous Sec. 4.3. This section provides a formal comparison between S_P and TPI which shows how to, for a given B and Δ, answer the question Δ ⊲ B using POLY*.

As stated in Sec. 3.3, in order to relate the π-calculus and TPI we need to provide an encoding [·] of π-processes in META*. The encoding is presented in Fig. 10. We can see that it is almost an identity because the syntax of the π-calculus
The set of expected and actual channel types of $G$:

$$\text{chtypes}(\Delta, G) = \{ \{ \Delta(a), \uparrow[\Delta(b_1), \ldots, \Delta(b_k)] \}: (\chi_{\Delta(b_1, \ldots, b_k)}^a, \chi') \in G \wedge (\chi_{\Delta(b_1, \ldots, b_k)}^{<b_1, \ldots, b_k} \chi') \in G \}$$

**Figure 11:** Property of shape types corresponding to $\vdash$ of TPI.

$\pi$-calculus presented in this paper already agrees with Meta$. Thus the main work of $\vdash$ is to change the syntactic category. Prop. 3.1 holds in the above context.

Given $\Delta$, we define a simple property of POLY$\ast$ shape types which holds for the principal type $\pi_B$ of $\{B\}$ iff $\Delta \vdash B$. This property is given by the relation $\Delta \cong \pi$ from Fig. 11. The auxiliary set $\text{chtypes}(\Delta, G)$ is a set of pairs of TPI types extracted from graph $G$. Each pair in this set correspond to an edge of $G$ labeled by a communication action type “$a(b_1, \ldots, b_k)$” or “$a<b_1, \ldots, b_k>$”. The first member of the pair $\Delta(a)$ says how channel $a$ should be used accordingly to $\Delta$. The second member describes the actual usage of $a$ computed from the types of the names $b_i$’s. Note that $\text{chtypes}(\Delta, G)$ is undefined when some value of $\Delta$ needed by the definition is not defined. Bound (basic) names of $B$ are not mentioned by $\Delta$ but are contained in graph parts of types of $B$. For $\text{chtypes}(\Delta, G)$ to be defined, we need to provide types of all names mentioned in $G$. In the definition of $\cong$, the types of bound names are supposed to be given by the context $\Delta'$. The following theorem shows how $\Delta \vdash B$ can be answered by using POLY$\ast$ and $\cong$.

**Theorem 4.4.** Let $B$ be a $\pi$-process such that no two different binders bind the same basic name, $\pi_B$ be a principal (P-)type of $\{B\}$, and $\text{dom}(\Delta) = \text{fbn}(B)$. Then $\Delta \vdash B$ iff $\Delta \cong \pi_B$.

The previous theorem requires that no two different binders bind the same basic name (which can be achieved by renaming). Nevertheless this property is not preserved under rewriting because two same binders can be introduced by replication. But when we start with a process $B_0$ where all bounding basic names differ, then the theorem holds for any successor process $B_1$ even when the requirement on bound names is not met for $B_1$. The main point is to ensure that there are no two bound names to which different types are assigned by the derivation of $\Delta \vdash B$. We use a little bit stronger assumption in the theorem because the exact condition is hard to formulate. Also note that the theorem does not hold for an arbitrary $\pi$-type $\pi$ of $\{B\}$ because $\pi$ could contain some additional edges which are not used when matching $\{B\}$ but can preclude $\Delta \cong \pi$. 
4.5 Conclusions and Further Discussions

We showed the example process (Ex. 4.1) that can not be recognized as communication safe by TPI (Ex. 4.2) but can be proved safe by POLY\textsuperscript{*} (Ex. 4.3). Moreover, Thm. 4.4 implies that every process recognized safe by TPI can be recognized safe by POLY\textsuperscript{*} as well. Thus we conclude that POLY\textsuperscript{*} is strictly better in recognition of communication safety than TPI. Thm. 4.4 also allows us to recognize typability in TPI: B is typable in TPI iff $\emptyset \equiv \pi_B$. This is computable because a POLY\textsuperscript{*} principal type can always be found (in polynomial time), and checking $\equiv$ is easy.

Turner in his thesis [Tur95, Ch. 5] presents also a more sophisticated polymorphic system for the \pi-calculus. This system recognizes B from Ex. 4.1 as safe. However, with respect to our best knowledge, it can not recognize safety of the process “B | $\sigma n, a_0$.0” which can still be proved correct by POLY\textsuperscript{*}. Also, we are not aware of any process that can be recognized safe by Turner's polymorphic system but not by POLY\textsuperscript{*}. On the other hand, there are still processes which are communication safe but not shown to be so by POLY\textsuperscript{*}, for example, “$a(x).a(y, z).0 \mid a<0\cdot a<0, a<0\cdot0$”.

Other type systems for the \pi-calculus are found in the literature. Kobayashi and Igarashi [IK01] present generic types for the \pi-processes with types looking like simplified processes. Their type system can be used verify properties which are hard to express using POLY\textsuperscript{*} shape types (race conditions, deadlock detection) but it does not support polymorphism. Thus one can expect application where POLY\textsuperscript{*} is more expressive as well as contrariwise. It must be emphasized, however, that POLY\textsuperscript{*} shape types can be used for many process calculi, not only for the \pi-calculus.

5 POLY\textsuperscript{*} and Types for Mobile Ambients

This section shows how to instantiate POLY\textsuperscript{*} to make a type system for Mobile Ambients [CG98] (MA). Furthermore it compares this instantiation of POLY\textsuperscript{*} with a type system for MA from the literature. This section describes MA, a type system [CG99] for it which we call TMA, instantiates META\textsuperscript{*} to MA, describes some properties, shows how to embed TMA predicates in POLY\textsuperscript{*} types, and discusses possible extensions of the embedding.

5.1 Mobile Ambients (MA)

Mobile Ambients (MA), introduced by Cardelli and Gordon [CG98], is a process calculus for representing process mobility. Processes are placed inside named bounded locations called \textit{ambients} which form a tree hierarchy. Processes can change the hierarchy. Processes can also send to nearby processes messages containing either ambient names or hierarchy change instructions. In order to simplify the presentation we build processes from META\textsuperscript{*} basic names which are preserved under $\alpha$-renaming as in META\textsuperscript{*}.
Fig. 12 describes MA process syntax. Some names, e.g., “in”, are reserved for the translation. Capabilities are ambient hierarchy change instructions. Executing a capability causes the surrounding ambient to participate in a change. The capability “in n” instructs the surrounding ambient to move itself and its contents into a sibling ambient named n. Similarly, “out n” instructs the surrounding ambient to move out of a parent ambient named n and become its sibling. The capability “open n” instructs the surrounding ambient to dissolve the boundary of a child ambient named n. Although the syntax allows an arbitrary N at the position n so that substituting a capability for a name yields valid syntax, capabilities where N is not a single name are inert and meaningless. In capability sequences, the left-most capability will be executed first. Executing a capability consumes it.

The process constructors “0”, “!”, “.”, “!” and “ν” have standard meanings. The name n is (ν)-bound in (vn : ω)B and comes with an explicit type annotation. Types are described in Sec. 5.2 below. The expression n[B] describes the process B running inside the ambient n. As above, the syntax allows inert
meaningless constructions with $N$ at the position of $n$. Capabilities can be communicated in messages. $\langle N_1, \ldots, N_k \rangle$ is a process that sends a $k$-tuple of messages. $(n_1: \omega_1, \ldots, n_k: \omega_k).B$ is a process that receives a $k$-tuple of messages, substitutes them for appropriate $n$’s in $B$, and continues as this new process. The name $n_i$ is said to be (input-)bound in the process and, again, comes with an explicit type annotation. Bound basic names and the free names of a process are defined like in Meta*. Processes that are $\alpha$-convertible are identified. A substitution in $MA$ is a finite function from names to messages and its application to processes is written postfix. Fig. 12 also describes structural equivalence and semantics of $MA$ processes. The only thing the semantics does with type annotations is copy them around. We require all processes to be well-scoped w.r.t. conditions S1-3 from Sec. 4.1, and the additional condition (S4) that explicit annotations assign the same message type to bound names with the same basic name.

Example 5.1. In this $MA$ process, packet ambient $p$ delivers a synchronization message to destination ambient $d$ by following instructions $x$. As we have not yet properly defined message types, we only suppose $\omega_p = \text{Amb} \left[ \kappa \right]$ for some $\kappa$.

\[
B = \langle \text{in} d \rangle | (\text{vp}: \omega_p)(\text{d}[\text{open} p.0] | (x: \omega_x).p[x.<>]) \rightarrow \\
(\text{vp}: \omega_p)(\text{d}[\text{open} p.0] | p[\text{in} d.<>]) \rightarrow \\
(\text{vp}: \omega_p)(\text{d}[\text{open} p.0 | p[<>]]) \rightarrow \text{d[<>]}
\]

This example is also used in the sections to follow.

5.2 Types for Mobile Ambients (TMA)

$MA$ as presented here involves polyadic communication and thus an arity mismatch error can occur, like in “$\langle a, b \rangle .0 \mid (x).\text{in } x.0$”. A second kind of communication error can be encountered when a sender sends a capability while a receiver expects a single name. For example “$\langle \text{in } a \rangle .0 \mid (x).\text{out } x.0$” could give rise to an inert and meaningless process “$\text{out } \langle \text{in } a \rangle .0$”. A third kind of communication error is encountered when a process is to execute a capability which is a single name, for example “$a.0$”. This situation can be caused by a wrong communication, that is why we count it as a communication error. Processes which can never evolve to a state when any of the above three kinds of communication errors occur are said to be communication safe.

$TMA$, a typed version of $MA$ introduced by Cardelli and Gordon [CG99], is designed to verify communication safety. TMA assigns an allowed communication topic to each ambient location and ensures that communications respect the allowed topic at each location.

Fig. 13 describes $TMA$ type syntax. Exchange types are assigned to processes and ambient locations to describe allowed communication topics. The type $\text{Shh}$ indicates silence (no communication). $\omega_1 \otimes \cdots \otimes \omega_k$ indicates communication of $k$-tuples of messages whose $i$-th member has the message type $\omega_i$. When $k = 0$ we write $1$ which describes processes executing only synchronization actions $<>$ and $()$. Message types describe messages (capability sequences and names). $\text{Amb}[\kappa]$ is the type of the name of an ambient where communication described
by \( \kappa \) is allowed. \( \text{Cap}[\kappa] \) describes capabilities whose execution can unleash communication described by \( \kappa \) (by opening some ambient). Environments assign message types to free names (via basic names). Fig. 13 also describes the \( \text{TMA} \) typing rules. A type from the conclusion of a rule which is not mentioned in the assumption is supposed to be arbitrary. For example, the type of \( \text{N}[\text{B}] \) can be arbitrary provided the content \( \text{B} \) is well-typed. It reflects the fact that the communication inside an ambient does not directly interact with the outside of the ambient. Existence of some \( \Delta \) and \( \kappa \) such that \( \Delta \) does not assign a Cap-type to any free name and \( \Delta \vdash \text{B} : \kappa \) holds implies that \( \text{B} \) is communication safe. (For more details see [CG99].)

**Example 5.2.** Consider the process \( \text{B} \) from Ex. 5.1. Let us take

\[
\Delta = \{ d \mapsto \text{Amb}[1] \} \quad \omega_p = \text{Amb}[1] \quad \omega_x = \text{Cap}[1]
\]

We can see that \( \Delta \vdash \text{B} : \text{Cap}[1] \) but, for example, \( \Delta \not\vdash \text{B} : 1 \).

### 5.3 Instantiation of \( \text{META} \ast \) to \( \text{MA} \)

When we omit type annotations, add “0” as a continuation after output actions, and write capability prefixes always in a right associative manner (that is like “\( \text{in a.(out b.(in c.0)))} \)”), we can see that the \( \text{MA} \) syntax is included in the syntax of \( \text{META} \ast \). Thus we can instantiate \( \text{META} \ast \) to \( \text{MA} \) expressing \( \text{MA} \)’s rewriting
relation in the Meta* syntax from Fig. 4 as follows:

\[
\mathcal{A} = \{ \text{active}(\hat{P} \text{ in } \hat{\hat{P}}), \text{rewrite}(\hat{a} [\text{in} \hat{b}.\hat{P} | \hat{Q}] | \hat{\hat{b}}[\hat{\hat{P}} | \hat{Q}] \rightarrow \hat{b}[\hat{\hat{a}}[\hat{\hat{P}} | \hat{Q}] | \hat{\hat{R}}]), \text{rewrite}(\hat{a} [\text{out} \hat{a}.\hat{P} | \hat{Q}] | \hat{\hat{a}}[\hat{\hat{R}}] | \hat{\hat{b}}[\hat{\hat{P}} | \hat{Q}] \rightarrow \hat{\hat{a}}[\hat{\hat{R}}] | \hat{\hat{b}}[\hat{\hat{P}} | \hat{Q}]), \text{rewrite}(\text{open} \hat{a}.\hat{P} | \hat{\hat{a}}[\hat{\hat{R}}] \rightarrow \hat{\hat{P}} | \hat{\hat{R}}] \} \cup \bigcup_{k=0}^{\infty} \{ \text{rewrite}(<\hat{M}_1, \ldots, \hat{M}_k>, \hat{P} | (\hat{a}_1, \ldots, \hat{a}_k).\hat{Q} \rightarrow \hat{\hat{P}} | \{ \hat{a}_1 := \hat{M}_1, \ldots, \hat{a}_k := \hat{M}_k \} \hat{Q} \} \}
\]

This straightforwardly translates TMA’s rules. The active rule lets rewriting be done inside ambients and corresponds to the TMA rule ‘\[B_0 \rightarrow B_1 \Rightarrow \pi(B_0) \rightarrow \pi(B_1)\]’. Each communication prefix length has its own rule as in the case of the π-calculus. The set \(\mathcal{A}\) gives use the calculus \(C_A\) and its type system \(S_A\).

Communication safety of \(P\) can be checked on a POLY* type as follows. Every pair of edges labeled by \((a_1, \ldots, a_k)\) and \(<b_1, \ldots, b_j>\) with the same source node where \(k \neq j\) indicates an arity mismatch error (but only at active positions). Every label which contains the error name “/AD” (introduced by a substitution) indicates that a capability was sent but a single name was expected. Moreover, an edge labeled with a single name \(a\) at active position which is not input-bound in \(P\) indicates that a single name capability could be erroneously executed. A type of \(P\) which does not indicate any error proves its communication safety. Checking the safety this way is easy.

**Example 5.3.** Let us consider process B from Ex. 5.1 whose \(C_A\) equivalent is “\[P = \langle \text{in } d, 0 | \nu(p)(d[p, 0] | (\langle x \rangle . p[x, < >]) \rangle\]”. Its \(S_A\) principal type looks as follows:

\[
\begin{array}{c}
\text{in } d
\end{array}
\]

Its root node is \(R\) and the names of other nodes are omitted. We can easily conclude that \(P\) is communication safe by simply checking the labels of edges as described above. The edge labeled by \(x\) does not constitute an error because \(x\) is input-bound in \(P\).

### 5.4 Embedding of TMA in POLY*

To show that everything TMA can express can also be expressed in POLY*, we construct an embedding. We emphasize that this embedding is primarily for the theoretical comparison and not intended for use in practice. Using the notation from Sec.3.1 we have that \(C\) is MA, \(S_C\) is TMA, predicates \(\varphi\) are pairs \((\Delta, \kappa),\)
and $S_C$’s relation $\triangleright B : \varphi$ is the TMA typing relation $\Delta \vdash B : \kappa$. Moreover $R$ is $A$ which was together with $C_A$ and $S_A$ introduced in the previous Sec. 5.3. This section provides a formal comparison between $S_A$ and TMA which shows how to, for a given $B, \Delta$, and $\kappa$, answer the question $\Delta \vdash B : \kappa$ using POLY*.

Following the general discussion in Sec. 3.3 we need to provide an encoding $\{\cdot\}$ of MA processes in META*. This encoding, presented in Fig. 14, is straightforward due to the flexibility of META* syntax. The encoding $\{\cdot\}$ translates capabilities to META* messages and TMA processes to META* processes. Meaningless expressions allowed by TMA’s syntax are translated using the auxiliary mapping $\gamma$ and the special name “•”. For example “$\{\text{in } a\} = \text{in }$”. Recall that in META* “$x[\cdot]$” is an abbreviation for “$x[\cdot]P$”, and that “$\cdot$” linearizes composed messages (like $(a.b)\cdot P = a.b.P$). The encoding erases type annotations; this is okay because TMA’s rewriting rules only copy type annotations around without any other effect. The type embedding in Sec. 5.4 will recover type information by different means. Prop. 3.1 holds in the context given by MA and $A$.

Because the TMA relation $\vdash$ is preserved under renaming of bound basic names of processes, the discussion from Sec. 3.4 applies. Thus we can not translate $(\Delta, \kappa)$ to a POLY* shape type with an equivalent meaning. Nevertheless this becomes possible when we specify the sets of allowed input- and $\nu$-bound basic names and their types. Because the MA processes used by TMA contain explicit type annotations these sets can be easily extracted from a given process $B$. Let $\Delta_B^\nu$ denote an extended environment which describes the types of all $\nu$-bound basic names of $B$, and let $\Delta_B^\nu$ describe types of input-bound basic names. For a given $\Delta, B$, and $\kappa$ we construct the shape type $\{\Delta \cup \Delta_B^\nu, \Delta_B^\nu, \kappa\}$ such that $\Delta \vdash B : \kappa$ holds in TMA if and only if $\vdash \{B\} : \{\Delta \cup \Delta_B^\nu, \Delta_B^\nu, \kappa\}$ holds in POLY*. We union free and $\nu$-bound basic names because our construction does not need to distinguish among them. In contrast, the construction needs to know which basic names are input-bound. The well-scopedness rules S1-4 ensure there is no ambiguity in using only basic names to refer to typed names in a process.

The construction of bound names environments $\Delta_B^\nu$ and $\Delta_B^\nu$ is given in the top part of Fig. 15. In the case of $\Delta_B^\nu$, we need only ambient types because only they can be types of $\nu$-bound names in typable TMA processes; this is where we enforce this TMA property. The type information I (Fig. 15, 2nd part)
encapsulates what is needed to construct a POLY type. When \( I = (\Delta_0, \Delta_1, \kappa) \) is given we write \( \Delta_1 \) for \( \Delta_0 \cup \Delta_1 \), and \( \Delta^n_{1} \) for \( \Delta_1 \), and \( \kappa_1 \) for \( \kappa \). Thus when \( I \) is constructed from \( \Delta, B, \) and \( \kappa \) as \( (\Delta \cup \Delta^n_B, \Delta^n_B, \kappa) \) then \( \Delta_1 \) describes types of all names in \( \Delta \) and \( B \), and \( \Delta^n_B \) describes types of input-bound names, and \( \kappa_1 \) is simply \( \kappa \).

**Example 5.4.** \( \Delta, B, \) and \( \kappa \) from the previous examples (Ex. 5.1 and Ex. 5.2) gives us \( I = (\Delta \cup \Delta^n_B, \Delta_B^n, \text{Cap}[1]) \) and we have:

\[
\Delta \cup \Delta_B^n = \{d \mapsto \text{Amb}[1], p \mapsto \text{Amb}[1]\} \quad \Delta_B^n = \{x \mapsto \text{Cap}[1]\}
\]

The main idea of the construction of the POLY type from \( I \) is as follows. The constructed shape graph contains exactly one node for every exchange type of some ambient location. It means one node for the top-level type \( \kappa \), and one node for \( \kappa' \) whenever some basic name contained in \( I \) has the type \( \text{Amb}[\kappa'] \). The node corresponding to the top-level type becomes the shape predicate root. Each node corresponding to some \( \kappa \) has self-loops which describe all capabilities and communication actions which a process of the type \( \kappa \) can execute. For example, when \( \Delta_1(d) = \text{Amb}[1] \) then every node would have a self-loop labeled by “in d” because in-capabilities can be executed by any process. On the other hand only the node which corresponds to \( 1 \) would allow “open d” because only processes of type \( 1 \) can legally execute it. Finally, following an edge labeled with “d[]” means entering \( d \). Thus the edge has led to the node that corresponds to the \( d \)'s type. In the above example, the shape graph would contain edges labeled with the ambient action type “d[]” from any node to the node corresponding to \( 1 \).

The construction starts by building the node set of a shape predicate (the middle part of Fig. 15). All the exchange types of ambient locations are gathered in the set \( \text{types}_1 \). These types are put in bijective correspondence (via the two mutually inverse bijections \( \text{nodeof}_1 \) and \( \text{typeof}_1 \)) with the members of \( \text{nodes}_1 \).

**Example 5.5.** Our example gives us \( \text{types}_1 = \{\text{Cap}[1], 1\} \). Let us choose \( \text{nodes}_1 = \{\text{R}, 1\} \) and define the bijections such that \( \text{nodeof}_1(\text{Cap}[1]) = \text{R} \) and \( \text{nodeof}_1(1) = 1 \).

The fourth part of Fig. 15 defines two important auxiliary functions used by the translation: \( \text{namesof}_1 \) and \( \text{allowedin}_1 \). The first one provides for each message type \( \omega \) the set \( \text{namesof}_1(\omega) \) of all basic names declared with the type \( \omega \) by \( I \). The second one provides for the TMA exchange type \( \kappa \) the set of POLY action types \( \text{allowedin}_1(\kappa) \) which describe (translations of) all capabilities and action prefixes which are allowed to be legally executed by a process of the type \( \kappa \). The set \( \text{allowedin}_1(\kappa) \) consists of three parts: \( \text{moves}_1, \text{opens}_1(\kappa), \) and \( \text{comms}_1(\kappa) \). The action types in \( \text{moves}_1 \) describe all in/out capabilities which can be constructed from ambient basic names in \( I \). In other words all in/out capabilities which can be legally executed by a process of some appropriate type from \( I \). The set does not depend on \( \kappa \) because in/out capabilities can be executed by any process. The set \( \text{opens}_1(\kappa) \) describe capabilities which can be executed
Extraction of types of bound names:
\[ \Delta_\kappa^a (a) = \omega \text{ iff } \text{B has a subprocess } (\ldots, a^1 : \omega, \ldots).B_0 \]
\[ \Delta_\kappa^a (a) = \omega \text{ iff } \omega = \text{Amb}[\kappa] \& \text{B has a subprocess } (\forall a^1 : \omega).B_0 \]

Type information:
\[ I \in \text{TypInfo} = \text{AEEnvironment} \times \text{AEEnvironment} \times \text{AEExchangeType} \]

When \( I = (\Delta_0, \Delta_1, \kappa) \) is given we write \( \Delta_1 \) for \( \Delta_0 \cup \Delta_1 \), and \( \Delta_1^n \) for \( \Delta_1 \), and \( \kappa_1 \) for \( \kappa \).

Set of nodes of a shape graph (and correspondence functions):
\[ \text{types}_1 = \{ \kappa_1 \} \cup \{ \kappa : \text{Amb}[\kappa] \in \text{rng}(\Delta_1) \} \]

Let \( \text{nodeo}_i \) be an arbitrary but fixed set of nodes such that there exist the bijection \( \text{typeo}_1 \) from \( \text{nodes}_1 \) into \( \text{types}_1 \).

Action types describing legal capabilities:
\[ \text{namesof}_i (\omega) = \{ \alpha : \Delta_1 (\alpha) = \omega \} \]
\[ \text{allowedin}_1 (\kappa) = \text{moves}_1 \cup \text{opens}_1 (\kappa) \cup \text{comms}_1 (\kappa) \]
\[ \text{moves}_1 = \{ \text{in } a, \text{out } a : \exists \kappa. a \in \text{namesof}_i (\text{Amb}[\kappa]) \} \]
\[ \text{opens}_1 (\kappa) = \{ \text{open } a : a \in \text{namesof}_i (\text{Amb}[\kappa]) \} \]
\[ \text{comms}_1 (\text{Amb}[\kappa]) = \text{namesof}_i (\text{Amb}[\kappa]) \]
\[ \text{comms}_1 (\text{Cap}[\kappa]) = \text{namesof}_i (\text{Cap}[\kappa]) \]
\[ \text{msg}_1 (\text{Amb}[\kappa]) = \text{namesof}_i (\text{Amb}[\kappa]) \]
\[ \text{msg}_1 (\text{Cap}[\kappa]) = \text{namesof}_i (\text{Cap}[\kappa]) \]
\[ \text{comms}_1 (\text{Shh}) = \emptyset \]
\[ \text{comms}_1 (\omega_1 \otimes \cdots \otimes \omega_k) = \{ \chi : \text{msg}_1 (\omega_1) \} \]
\[ \{ (a_1, \ldots, a_k) : \Delta_1^n (a_i) = \omega_i \& (i \neq j \Rightarrow a_i \neq a_j) \} \]

Construction of shape predicates and embedding of type judgments:
\[ \{ I \} = \{ \chi \rightarrow \alpha : \chi \in \text{allowedin}_1 (\text{typeo}_1 (\chi)) \& \chi \in \text{nodes}_1 \} \]
\[ \{ \chi \rightarrow \alpha : \alpha \in \text{namesof}_i (\text{Amb} [\text{typeo}_1 (\chi)]) \& \chi, \chi' \in \text{nodes}_1 \} \]
\[ \{ I \} = \{ I, \text{nodeo}_i (\kappa_1) \} \]

**Figure 15:** Construction of POLY* type embedding.

by a process of the type \( \kappa \). It consists of open-capabilities constructible from ambient names in \( I \) and from those basic names of the type \( \text{Cap}[\kappa] \) by \( I \). The second part of \( \text{opens}_1 (\kappa) \) describes names which are supposed to be instantiated by communication to some executable capabilities. The set \( \text{comms}_1 (\kappa) \) describes communication actions which can be executed by a process of type \( \kappa \). Its first part describes output- and the second one input-communication actions. The first part uses the auxiliary function \( \text{msg}_1 \) which for each TMA message type \( \omega \) defines the set of POLY* message types describing all messages of the type \( \omega \) constructible from names in \( I \).

**Example 5.6.** Relevant sets for our example are:
Finally the bottom part of Fig. 15 describes the construction of the shape graph \( I \) and the shape predicate \( \langle I \rangle \) from \( I \). The first part of the definition of \( \langle I \rangle \) describes self-loops of nodes. These self loops describe actions allowed to be executed by the node’s corresponding type. Edges from the second part of \( \langle I \rangle \) describe transitions among nodes. Any edge labeled by the ambient action \( "a[]" \) always leads to the node which corresponds to the exchange type allowed inside \( a \).

**Example 5.7.** The resulting shape predicate \( \{I\} = \langle G, R \rangle \) in our example has the root \( R \) and its shape graph \( G \) is below. We merge edges with the same source and destination into one using “\( I \)”. 

Correctness of the translation is expressed by Thm. 5.8. The conditions on \( \nu \)-bound names ensure that no \( \nu \)-bound basic name has a different type assigned by \( \Delta \) and that no \( \nu \)-bound basic name \( \text{Cap} \)-type assigned by type annotation.

**Theorem 5.8.** Let \( \text{dom}(\Delta) \cap \text{nbn}(B) = \emptyset \) and \( \text{dom}(\Delta_B^n) = \text{nbn}(B) \) then 

\[
\Delta \vdash B : \kappa \quad \text{iff} \quad \vdash \{B\} : \{(\Delta \cup \Delta_B^n, \Delta_B^n, \kappa)\}
\]

**Theorem 5.9.** \( \{(\Delta \cup \Delta_B^n, \Delta_B^n, \kappa)\} \) is a \( \text{POLY}^* \cdot A \)-type (up to the presence of flow edges).

### 5.5 Conclusions and Further Possibilities

In Sec. 5.4 we have shown how to embed TMA’s typing relation in \( \text{POLY}^* \). In Sec. 5.3 we have shown how to use \( \text{POLY}^* \) to recognize communication safety of processes directly. When \( B \) is typable in TMA then there is some \( \Delta \) and \( \kappa \) such that \( \Delta \vdash B : \kappa \). It easy to see that the type \( \{I\} \) constructed as in Sec. 5.4 can also be used to prove the safety of \( B \). But then, it follows from the properties of principal types, that the safety of \( B \) can be recognized directly from its \( \text{POLY}^* \) principal type as well. That is why any process proved safe by TMA can be proved safe by \( \text{POLY}^* \) on its own.

On the other hand there are processes which can be recognized safe by \( \text{POLY}^* \) but are not typable in TMA. For example, the TMA process \("(x:\omega)\cdot x.0 |\)
"in a" is not typable in TMA but it is trivially safe. A more sophisticated example exploits polymorphic abilities of POLY* types. For example, the C₄ process
\[
! (x, y, m).x[in y.<m>.0] \mid <p, a, c>.0 \mid a[open p] \mid q, b, in a>.0 \mid b[open q]
\]
can be proved safe by POLY* but it constitutes a challenge for TMA-like non-polymorphic type systems. We are not aware of other type systems for MA and its successor that can handle this kind of polymorphism.

The construction of the POLY* type from Sec. 5.4 can be altered in many ways to improve expressiveness. In subsequent work [CGG99], Cardelli, Ghelli, and Gordon define a type system which can ensure that some ambients stay immobile or that their boundaries are never dissolved. We can easily adapt our construction to reflect these needs by removing appropriate capabilities from self loops of nodes. In our construction we have one node for each type of some ambient location. Alternatively we can have a separate node for each ambient. This allows us to express more refined properties of ambients and to express predicates defined by another subsequent work of the above authors concerning ambient groups [CGG00].

We are not forced to use only self loops to describe actions inside ambients. We can use an edge sequence to describe action execution order similarly to “;” sequencing of session types [Hon93]. For example, we can express that the action “out a” is followed by “in b”. Moreover, we can take an advantage of POLY*’s spatial polymorphism to express location-dependent properties of ambients, e.g., that ambient a can be opened only inside ambient b.

6 POLY* and Static Analysis of BioAmbients

We show how to instantiate POLY* to a type system for BioAmbients [RPS +04] and how to use it for static analysis. Moreover we compare results achieved by POLY* with a static analysis system for BioAmbients [NNPR07] from the literature which we call SABA.

6.1 BioAmbients (BA)

BioAmbients, introduced by Regev, Panina, Silverman, Cardelli, and Shapiro [RPS +04], is a process calculus for modeling biomolecular systems. Regev et al. present BioAmbients with the choice operator to express computation options and with replication. We work with a choice-free variant of BioAmbients with replication which we name BA. POLY* can handle choice in a way that achieves the same results as SABA but we omit it to simplify the presentation. A discussion how to extend our approach to handle choice is presented in Sec. 6.5.

BA is similar to MA but it differs in several ways. Ambients are anonymous, that is, are not labeled with names. It implies that capabilities can no longer use names to refer to ambients. Thus capabilities come in require/allow pairs synchronized by names, for example, “enter a/accept a”. Then an appropriate action
Syntax of BA:
\[
\begin{align*}
l & \in \text{BioLabel} \\
n, m & \in \text{BioName} \\
d & \in \text{BioDirection} \\
N & \in \text{BioCapability} \\
B & \in \text{BioProcess}
\end{align*}
\]

Structural equivalence of BA is generated by:
\[
\begin{align*}
B & \equiv B \\
B_0 \mid B_1 & \equiv B_1 \mid B_0 \\
B_0 \mid (B_1 \mid B_2) & \equiv (B_0 \mid B_1) \mid B_2 \\
0 & \equiv 0 \\
!B & \equiv B \mid !B
\end{align*}
\]

Rewriting relation of BA:
\[
\begin{align*}
[\text{enter } n. B_0 \mid B_1]^{n, 0} & \mid [\text{accept } n. B_2 \mid B_3]^{n, 1} \rightarrow [[B_0 \mid B_1]^{n, 0} \mid B_2 \mid B_3]^{n, 1} \\
[\text{exit } n. B_0 \mid B_1]^{n, 0} & \mid \text{expel } n. B_2 \mid B_3]^{n, 1} \rightarrow [B_0 \mid B_1]^{n, 0} \mid [B_2 \mid B_3]^{n, 1} \\
[\text{merge+ } n. B_0 \mid B_1]^{n, 1} & \mid \text{merge- } n. B_2 \mid B_3]^{n, 1} \rightarrow [B_0 \mid B_1 \mid B_2 \mid B_3]^{n, 1} \\
\text{local } n?\{m_{\circ}\}. B_0 & \mid \text{local } n!\{m_{\circ}\}. B_1 \rightarrow B_0 \mid m_{\circ} \rightarrow m_{\circ}\} \mid B_1 \\
p2c\ n?\{m_{\circ}\}. B_0 & \mid [c2p\ n?\{m_{\circ}\}. B_1 \mid B_2]^{n, 1} \rightarrow B_0 \mid m_{\circ} \rightarrow m_{\circ}\} \mid [B_2 \mid B_1]^{n, 1} \\
[c2p\ n?\{m_{\circ}\}. B_0 & \mid B_1]^{n, 1} \mid p2c\ n!\{m_{\circ}\}. B_2 \rightarrow [B_0 \mid m_{\circ} \rightarrow m_{\circ}\} \mid B_1]^{n, 1} \mid B_2 \\
[s2s\ n?\{m_{\circ}\}. B_0 & \mid B_1]^{n, 1} \mid [s2s\ n!\{m_{\circ}\}. B_2 \mid B_3]^{n, 1} \rightarrow \\
[B_0 \mid m_{\circ} \rightarrow m_{\circ}\} \mid B_1]^{n, 1} \mid [B_2 \mid B_3]^{n, 1} \\
B_0 & \rightarrow B_1 \\
B_0 \rightarrow B_1 \\
[B_0]^{l} & \rightarrow [B_1]^{l} \\
B_0 \mid B_2 & \rightarrow B_1 \mid B_2 \\
B_0 & \rightarrow B_1 \mid B_1 & \equiv B_1' \\
B_0 & \rightarrow B_1
\end{align*}
\]

Figure 16: Syntax and semantics of BA.

is performed when two ambients containing corresponding parts are found in a required position. The open capability is replaced by an operation that merges two sibling ambients. Communication is channel-based, that is, both a sender and receiver have to agree on a channel name for communication to happen. Moreover, communication is allowed also across some ambient boundaries, and only single names are exchanged.

Fig. 16 gives the syntax of BA. As in the case of the \(\pi\)-calculus and MA, we build processes from \textsc{Meta}\* names to ease comparison. Some names are reserved for translating BA processes into \textsc{Meta}\*. The capability “enter \(n\)” instructs an ambient to enter a sibling containing a corresponding “accept \(n\)”. “exit \(n\)” instructs an ambient to exit its parent ambient provided it allows it with the “expel \(n\)” capability. Finally, “merge+ \(n\)” instructs an ambient to merge with a sibling containing “merge- \(n\)”. Communication is in four directions: between processes in the same ambient (local), between processes in sibling ambients...
(s2s), from a parent ambient to its child (p2c), and from a child to the parent (c2p). Communication output is “d n \{m\}” where n is the channel name, d is the desired direction, and m is the name being sent. The input prefix “d n ?\{m\}” (input-)binds the name m.

Static analysis must refer to ambients to track changes, so following the approach of SABA, our syntax introduces ambient labels with no influence on the semantics. We translate these labels as META basic names and write \[B\] for an ambient labeled l. We identify \(\alpha\)-convertible processes. We require all processes to satisfy well-scopedness rules S1-2 from Sec. 5.1.

Example 6.1. Consider the following simple BA process:

\[
B = [\text{enter } n. \text{accept } x.0 \mid \text{enter } m. \text{merge} \ y.0]^{a} \mid \\
[\text{accept } n.0]^{b} \mid [\text{accept } m.0]^{c}
\]

The following two different rewritings can be proved:

\[
B \rightarrow [([\text{accept } x.0 \mid \text{enter } m. \text{merge} \ y.0])^{b} \mid [\text{accept } m.0]^{c} \\
B \rightarrow [\text{accept } n.0]^{b} \mid ([\text{enter } n. \text{accept } x.0 \mid \text{merge} \ y.0])^{c}
\]

6.2 Static Analysis of BioAmbients (SABA)

Nielsen, Nielson, Priami, and Rosa [NNPR07] designed a static analysis system for BioAmbients (hereafter SABA) which conservatively over-approximates the states that a system can evolve to. The original SABA works for a version of BA with the rec operator\(^2\) instead of replication. Here we suppose only a restricted usage of rec which can be expressed by replication\(^3\) because META does not support rec at the current moment.

The original SABA does \(\alpha\)-conversion similarly to META. It assigns a canonical name to every name that is preserved by \(\alpha\)-conversion. We identify these canonical names with META basic names. Canonical names are used in canonical capabilities and communication prefixes, which we map into POLY action types.

SABA takes a BA process as an input and its output collects information about possible contents of ambients in any process that the input process can evolve to. A result of SABA analysis is a pair \((S, N)\) where \(S \subseteq \text{BasicName} \times \text{ActionType}\), and \(N \subseteq \text{BasicName} \times \text{BasicName}\). For every ambient, \(S\) collects information about possible child ambients, capabilities, and communication prefixes contained in it. For example \((a, b[1]) \in S\) says\(^4\) that the ambient (with the label) a can have a child ambient b, while \((a, \text{enter } n) \in S\) says that an ambient with the label a can possibly contain (and execute) the capability “enter \(n_i\)” for any i. Note that members of \(S\) are built from basic names. In order to match the syntax

\(^2\)Process calculi with rec additionally introduce process variables, say X, and processes of the form \(\text{rec } X.P\) which behave like P with rec \(X.P\) substituted for X.

\(^3\)We can define replication using rec as “\(\text{rec } X.A = \text{rec } X.\{B \mid X\}\)”.

\(^4\)In the original paper [NNPR07] the set \(S\) contains \((a, b)\) instead of \((a, b[1])\). This technical change we make allows easier formulation of our comparison.
of action types we write “d a(b)” instead of “d a?(b)”, and “d a<b>” instead of “d a!(b)”. Input-bound names are handled in a special way. Capabilities built from input-bound names are not contained in \( S \). Instead, \( S \) contains all their actual instantiations introduced by communication. For example, for the input process “local a?(x).enter x.0 | local a!(b).0”, the \( S \) part of the result contains “enter b” but not “enter x”. The set \( \mathcal{N} \) describes possible name instantiations invoked by communication. For example \((x, b) \in \mathcal{N}\) says that communication can instantiate \( x \) to \( b \).

SABA defines the predicate \((S, \mathcal{N}) \models^1 B\) meaning that \( B \) matches the structure allowed by \((S, \mathcal{N})\) inside the ambient \( l \). The name “*” is used to refer to the top level location. SABA starts by computing conditions the result has to satisfy to correctly describe the input \( B \). For example, \((S, \mathcal{N}) \models^* \text{enter} a^1.B\) holds iff \((S, \mathcal{N}) \models^* B\) and \(S(\text{enter} b, *)\) for all \( b \) such that \( \mathcal{N}(b, a) \). Then closure conditions reflecting BA semantics are added. For example from the local communication rule, \( S(\text{local} a?\{b\}, *) \) and \( S(\text{local} a!(b'), *) \) has to imply \( \mathcal{N}(b', b) \). Finally the smallest pair \((S, \mathcal{N})\) which satisfies all the conditions is the result of SABA for \( B \). SABA ensures that the structure described by a valid result is closed under rewrites.

Fig. 17 defines the relation \((S, \mathcal{N}) \models^1 B\). When an input process \( B \) is given, this figure gives us the set of conditions on \((S, \mathcal{N})\) that has to be satisfied for \((S, \mathcal{N}) \models^* B\) to hold. To this conditions additionally closure conditions from Fig. 18 are added. These conditions directly correspond to the BA rewriting rules. The result of SABA for \( B \) is the smallest pair \((S, \mathcal{N})\) such that all the conditions are satisfied and that \( \mathcal{N}(a, a) \) holds for all \( a \in \text{fbn}(B) \).
∀l, l₁, l₂, a :  S(l₁, enter a) & S(l₁, l₁) & S(l₂, accept a) & S(l₁, l₂)
⇒ S(l₁, l₂)
∀l, l₁, l₂, a :  S(l₁, exit a) & S(l₁, l₁) & S(accept a, l₁) & S(l₁, l₁)
⇒ S(l₂, l₁)
∀l, l₁, l₂, a :  S(l₁, merge⁺ a) & S(l₁, l₁) & S(l₂, merge⁻ a) & S(l₂, l₂)
⇒ (∀α : S(l₁, α) ⇒ S(l₁, α))
∀l, a, b, b′ :  S(local a(b), l) & S(local a(b′), l)
⇒ N(b, b′)
∀l₀, l₁, a, b, b′ :  S(l₀, p₂c a(b′)) & S(l₀, l₁) & S(l₀, c₂p a(b))
⇒ N(b, b′)
∀l₀, l₁, a, b, b′ :  S(l₀, p₂c a(b)) & S(l₀, l₁) & S(l₀, c₂p a(b′))
⇒ N(b, b′)
∀l₀, l₁, a, b, b′ :  S(l₀, s₂s a(b)) & S(l₀, l₁) & S(l₁, s₂s a(b′)) & S(l₁, l₁)
⇒ N(b, b′)

Figure 18: Closure conditions valid for SABA results.

Example 6.2. For the process B from Ex. 6.1 SABA computes:

\[ N = \{ (n, n), (m, m), (x, x), (y, y) \} \]
\[ S = \{ (\ast, a[]), (a, enter n), (a, enter m), (a, merge- y), (\ast, b[]), (b, a[]), (b, accept n), (\ast, c[]), (c, a[]), (c, accept m) \} \]

6.3 Instantiation of Meta* to BioAmbients

We can express BA prefixes “d n ! {m}” and “d n ? {m}” as 3-length Meta* actions “d n< m>” and “d n< m>” respectively. Ambient labels can be translated using an ambient syntactic sugar as in MA, that is “[0]"" as “l₂ [0]". Then the syntax of BA matches the syntax of Meta*. The set B of Meta* rewriting rules looks as follows:

\[ B = \{ \text{active}(\hat{P} \text{ in } \hat{A}[\hat{P}]), \text{rewrite}(\hat{A}[\text{enter } \hat{n} \hat{P} | \hat{Q}] | \hat{b}[\text{accept } \hat{n} \hat{R} | \hat{S}] \rightarrow \hat{b}[\hat{A}[\hat{P} | \hat{Q}] | \hat{R} | \hat{S}]), \text{rewrite}(\hat{b}[\text{exit } \hat{n} \hat{P} | \hat{Q}] | \hat{b}[\text{expel } \hat{n} \hat{R} | \hat{S}] \rightarrow \hat{b}[\hat{P} | \hat{Q}] | \hat{R} | \hat{S}]), \text{rewrite}(\hat{A}[\text{merge+ } \hat{n} \hat{P} | \hat{Q}] | \hat{b}[\text{merge- } \hat{n} \hat{R} | \hat{S}] \rightarrow \hat{b}[\hat{P} | \hat{Q}] | \hat{R} | \hat{S}]), \text{rewrite}(\hat{A}[\text{local } \hat{n}(\hat{x}) \hat{P} | \hat{Q}] | \hat{b}[\text{catch } \hat{n} \hat{x} | \hat{Q}] \rightarrow \hat{b}[\hat{x} = \hat{m} \hat{P} | \hat{Q}]), \text{rewrite}(p₂c \hat{a}(\hat{x}) \hat{P} | \hat{Q}] | \hat{a}[c₂p \hat{a}(\hat{x} | \hat{R}] \rightarrow \hat{a}[\hat{x} = \hat{m} \hat{P} | \hat{Q} | \hat{R}]), \text{rewrite}(\hat{a}[c₂p \hat{a}(\hat{x}) \hat{P} | \hat{Q}] | \hat{Q} | \hat{P} | \hat{b}[\text{catch } \hat{n} \hat{x} | \hat{Q}] \rightarrow \hat{b}[\hat{x} = \hat{m} \hat{P} | \hat{Q} | \hat{R}]) \} \]

The set B gives us the calculus C_B and its type system S_B.

Example 6.3. POLY* principal type \( \pi_B \) for a Meta* equivalent of B from Ex. 6.1
Figure 19: Encoding of \( \text{BA}^* \) processes in \( \text{META}^* \).

\[
\begin{align*}
\text{local} & \rightarrow \text{local} \quad \text{p2c} \rightarrow \text{p2c} \\
\text{c2p} & \rightarrow \text{c2p} \quad \text{s2s} \rightarrow \text{s2s} \\
\text{enter n} & \rightarrow \text{enter n} \quad \text{accept n} \rightarrow \text{accept n} \\
\text{exit n} & \rightarrow \text{exit n} \quad \text{expel n} \rightarrow \text{expel n} \\
\text{merge+ n} & \rightarrow \text{merge+ n} \quad \text{merge- n} \rightarrow \text{merge- n}
\end{align*}
\]


Contents of ambients can be easily read from it. It also shows \( \text{POLY}^* \)'s spatial polymorphism in action: ambient \( a \) can execute “accept x” only when contained inside ambient \( b \), and similarly for “merge- y” and \( c \).

### 6.4 Comparing \( \text{POLY}^* \) Types with \( \text{SABA} \)’s Results

Using the notation from Sec. 3.1 we have that \( C = \text{BA} \), \( S_C = \text{SABA} \), predicates \( \phi \) are pairs \( \langle S, \mathcal{N} \rangle \), and \( S_C \)'s relation \( \triangleright : B : \phi \) is \( \langle S, \mathcal{N} \rangle \models B \). Moreover, \( B, C_B, \) and \( S_B \) were discussed in Sec. 6.3. This section shows that \( \text{POLY}^* \) can provide the same information as \( \text{SABA} \) and can do better.

#### 6.4.1 Constructing a \( \text{SABA} \) Predicate from a Shape Type.

As in the case of the previous sections we use an encoding of \( \text{BA} \) processes in \( \text{META}^* \) written \( \{ \} \). The encoding is even more straightforward than in the case of \( \text{MA} \) and Sec. 6.3 already discusses how to express \( \text{BA} \) syntax in \( \text{META}^* \). We leave details of \( \{ \} \) to the technical report because the idea has been already demonstrated. Prop. 3.1 holds.

Information provided by \( \text{SABA} \) results are contained in \( \text{POLY}^* \) principal typings as well. For example when the shape graph contains “\( \chi_0 \xrightarrow{\ell_0} \chi_1 \xrightarrow{\ell_1} \chi_2 \)” then it means that an ambient \( \ell_0 \) can possibly contain some ambient \( \ell_1 \). The above two edges can be possibly separated by other edges. We use the following two predicates to extract relevant information from the shape predicate \( \pi = \langle G, \chi \rangle \). The action type \( \alpha_k \) is said to be under the root of \( \pi \), written \( \text{inroot}_\pi(\alpha_k) \), when \( G \) contains a path of edges starting at \( \chi \) labeled with
\(\alpha_0, \ldots, \alpha_k\) where none of \(\alpha_i\)'s preceding \(\alpha_k\) is of the shape \(\l_1\) \(\l\). The condition on the shape of \(\alpha_i\)'s expresses that \(\alpha_k\) is not inside any ambient. Similarly, the predicate \(\text{inamb}_{\pi}(l, \alpha_k)\) holds when \(\alpha_k\) is contained directly inside the ambient \(l_0\) in \(G\). That is when \(G\) contains the edge path \(\l_0 \l \ldots, \alpha_0 \ldots, \alpha_k\) starting this time at any node. Again none of the \(\alpha_i\)'s preceding \(\alpha_k\) can have the shape \(\l_1\) \(\l\). We write \(\text{inamb}_{\pi}(\l, \alpha)\) for \(\text{inroot}_{\pi}(\alpha)\).

The following predicate is used to recognize non-instantiated capabilities, that is, those that contain basic names which are bound in some other action types of the shape graph. Let \(\text{bn}(G)\) be the set of all basic names which appear as one of \(\alpha_i\)'s in some \((\alpha_1, \ldots, \alpha_k)\) in \(G\). Write \(\text{instantiated}_{\pi}(\alpha)\) when \(\alpha\) labels some edge in the shape graph \(G\) of \(\pi\) and \(\text{fn}(\alpha) \cap \text{bn}(G) = \emptyset\). Then a SABA-like result is constructed from a shape predicate \(\pi = (G, \chi)\) as follows:

\[
S_\pi = \{ (l, \alpha) : \text{inamb}_{\pi}(l, \alpha) \& \text{instantiated}_{\pi}(\alpha) \}
\]

\[
N_\pi = \{ (a, b) : [\chi_0 \l_a \l_b \l \chi_1] \in G \} \cup
\{ (a, a) : a \in \text{fn}(\alpha) \cap \text{BioName} \& \text{instantiated}_{\pi}(\alpha) \}
\]

The set \(N_\pi\) is constructed from POLY* flow-edges (Sec. 2.3). Recall that BioName stands for the set of all basic names that BioName is built from, that is, \([n : n \in \text{BioName}]\). Thm. 6.4 describes the relation between native SABA results and those constructed from POLY*: POLY* principal types contain the information provided by SABA. When \((l, \alpha)\) is in \(S\) but not in \(S_\pi\), then subject reduction of POLY* ensures the situation predicted by SABA can never happen, in which case POLY* is more precise.

**Theorem 6.4.** Let \(B\) be a BA process, let \((S, N)\) be the result of SABA analysis of \(B\), and let \(\pi\) be the POLY* principal typing of \([B]\). Then \(S_\pi \subseteq S\), and \(N_\pi \subseteq N\), and \((S_\pi, N_\pi) \vdash^* B\).

**Example 6.5.** The sets \(S_\pi\) and \(N_\pi\) constructed for process \(B\) (Ex. 6.1) from the shape type \(\pi_B\) (Ex. 6.3) gives exactly the same result as SABA (Ex. 6.2) because of the simplicity of our example. However, Ex. 6.3 shows how POLY* can express more detailed information not contained in SABA results.

### 6.4.2 Constructing a Shape Type from a SABA Result.

This section shows how to construct a POLY* shape type which exactly correspond to a given SABA result. To be able to do this we need an upper bound on input-bound names allowed in the examined process. The reasons for this limitation are discussed in Sec. 3.4.

We don’t need to be able to construct a shape predicate for every possible SABA predicate but only for those which are valid SABA results. We could require this directly but it useful to explicitly state a specific condition on an input SABA predicate. This condition is required for our construction to be correct and it is satisfied for all valid SABA results. The condition on \((S, N)\) is as follows.

**Definition 6.6.** We say that \(S\) is closed w.r.t. \(N\) when all of the following hold for an arbitrary \(l, a, a', b, b', d:\)

1. \(S(l, \text{enter } a') \& N(a, a') \rightarrow (\forall a'' : N(a, a'') \rightarrow S(l, \text{enter } a''))\)
Sets of labels and nodes; bijections between them:
- \( \text{labels}_S = \text{dom}(S) \cup (\text{rng}(S) \cap \text{BioLabel}) \cup \{\}\) 
- \( \text{nodes}_S \) = arbitrary but fixed nodes set of the same size as \( \text{labels}_S \) 
- \( \text{nodesof}_S = \text{labelof}_S^{-1} \ldots \) bijections from \( \text{labels}_S \) into \( \text{nodes}_S \) and reversely

Sets of action types describing legal actions:
- \( \text{activecaps}_{S, N, Z}(l) \) = \{ \( d \ a < b \) : \( S(l, d \ a' < b') \) & \( N(a, a') \) & \( N(b, b') \) \} \cup 
  \{ \alpha [a' \rightarrow a] : \( S(l, a) \) & \( a' \in \text{fbn}(\alpha) \) & \( N(a, a') \) & \( \alpha \notin \text{labels}_S \) \}
- \( \text{inertcaps}_{S, N, Z}(l) \) = \{ \( d \ a (b) : d \in \text{BioDirection} \) & \( a \in Z \setminus \text{dom}(N) \) & \( b \in Z \) \} \cup 
  \{ d a < b, a a < b : d \in \text{BioDirection} \) & \( a \in Z \setminus \text{dom}(N) \) & \( b \in Z \) \} \cup 
  \{ \text{enter} a, \text{accept} a, \text{exit} a, \text{expel} a, \text{merge}+ a, \text{merge} \rightarrow a : a \in Z \setminus \text{dom}(N) \} \cup 
  \{ \text{allowedin}_{S, N, Z}(l) = \text{activecaps}_{S, N, Z}(l) \cup \text{inertcaps}_{S, N, Z}(l) \}

Construction of a shape graph:
- \( G_{(S, N, Z)} = \{ \text{nodelof}_S(l) \xrightarrow{1} \text{nodelof}_S(l_0) : (l, l_0) \in S \) & \( l_0 \in \text{labels}_S \} \cup \{ \chi \xrightarrow{\Delta} \chi : \chi \in \text{nodes}_S \} \)

Figure 20: Construction of a shape graph corresponding to a SABA result.

2-6. as case (1) but for accept, ..., merge-
7. \( S(l, d \ a'(b)) \) & \( N(a, a') \rightarrow (\forall a'' : N(a, a'') \rightarrow S(l, d \ a''(b))) \)
8. \( S(l, d \ a' < b') \) & \( N(a, a') \) & \( N(b, b') \rightarrow \)
   \( (\forall a'', b' : N(a, a'') \) & \( N(b, b') \rightarrow S(l, d \ a'' < b'')) \)

The condition above has eight parts, one for each possible action prefix. It reflects how input-bound names are handled in SABA. Let us describe the case for enter. It says that when an ambient labeled by l can contain “enter a” and some a can be instantiated to a’ by communication then l can also contain all other instantiations of “enter a”. Other cases are similar. Note that when \((S, N)\) is a valid SABA result for B than the following two claims hold. (1) When \(a \in \text{fbn}(B) \cap \text{nbn}(B)\) then \(N(a, a')\) and for any \(a'\) such that \(N(a, a')\) it holds \(a' = a\). (2) When \(a \in \text{fnb}(B)\) then \(a \notin \text{rng}(N)\).

A construction of a shape type which correspond to a SABA predicate \((S, N)\) is presented in Fig. 20. The set \(\text{labels}_S\) is the set of labels contained in \(S\). The set \(\text{nodes}_S\) is a set of nodes with the same number of members like \(\text{labels}_S\), and two mutually inverse bijections on these two sets are introduced. The set \(\text{activecaps}_{S, N, Z}(l)\) describes all action prefixes allowed in an ambient labeled by l. Note that we have to construct also original prefixes from their instantiations. As already noted the construction requires an upper bound on input-bound names allowed in a BA process. This is given by the set of basic names \(Z\). The set \(\text{inertcaps}_{S, N, Z}(l)\) describes all action prefixes constructed from those input-bound names which are never instantiated by communication to any actual value, that is, from communication inert input-bound names. Such actions are not contained in SABA results but a shape type needs to describe them. For example, for the SABA result \(S = \{*, s_2 s a(b)\}\) and \(N = \{(a, a)\}\) it holds that \((S, N) \models ^* s_2 s a?\{b\}.\text{enter} b.0\). But note that “enter b” is not contained in \(S\). In fact an arbitrary number of actions constructed from b can be present un-
der s2s a?{b} and the process is still correctly describe by \((S,N)\) (as long as \(b \notin \text{dom}(N')\)). The list of inert actions is added to \(\text{activecaps}_{(S,N')}(l)\) to form the set \(\text{allowedin}_{(N',S,Z)}(l)\). The shape graph \(G_{(S,N',Z)}\) connects the nodes from nodes\(_S\) accordingly to the ambient hierarchy described by \(N\). Finally the action types from \(\text{allowedin}_{(N',S,Z)}(l)\) are added as labels of loops of the node which correspond to \(l\).

**Example 6.7.** Let us demonstrate the construction on the process \(B\) from Ex. 6.1 and the SABA result for \(B\) from Ex. 6.2. We have \(\text{labels}_S = \{\star, a, b, c\}\). Let us take nodes\(_S = \{R, A, B, C\}\) and nodeof\(_S = \{\star\} = R\), nodeof\(_S(a) = A\), nodeof\(_S(b) = B\), and nodeof\(_S(c) = C\). The situation with input-bound names is simple because we know that \(\text{ibn}(B) = \emptyset\) and thus we can take \(Z = \emptyset\). Thus inertcaps\(_{(N',Z)} = \emptyset\). We have that

\[
\begin{align*}
\text{activecaps}_{(S,N')}(\star) &= \emptyset \\
\text{activecaps}_{(S,N')}(a) &= \{\text{enter } n, \text{accept } x, \text{enter } m, \text{merge- } m\} \\
\text{activecaps}_{(S,N')}(b) &= \{\text{merge- } n\} \\
\text{activecaps}_{(S,N')}(c) &= \{\text{accept } m\}
\end{align*}
\]

The shape graph looks as follows:

![Shape Graph Diagram]

Labels of multiple loop edges of node \(A\) are merged together by “\(|\)”. We can see that this graph is slightly less precise that the graph of the principal type presented in Ex. 6.3.

The correctness of the construction is expressed by the following theorem. The root node of a constructed shape graph is of course the node nodeof\(_S(\star)\). We see that the theorem allows us to exactly emulate SABA relation \((S,N') \models^* B\).

**Theorem 6.8.** Let \(S\) be closed w.r.t. \(N\). When \(\text{ibn}(B) \subseteq Z\) and \(\forall a \in \text{fbn}(B) \cup \text{nbn}(B)\) it holds that \(N(a, a)\) then:

\[
(S,N') \models^* B \iff \vdash [B] : \langle G_{(S,N,Z)}, \text{nodeof}_{S}(\star) \rangle
\]

### 6.5 Conclusions and Further Discussions

We showed how to use POLY\(\star\) for static analysis of BA. Thm. 6.4 says that POLY\(\star\) provides at least the same precision of information as a static analysis system SABA from the literature. We have not shown in this paper, however, how to exactly emulate the SABA’s relation \((S,N') \models^* B\) in POLY\(\star\) which can be
potentially used by some application of SABA. In our technical report we develop a construction of a shape type from a SABA result \((S,N)\) which faithfully describe the result (given an upper bound on input-bound names) and allows us to decide SABA’s relation exactly in POLY*. We leave it to the technical report because its presentation requires lengthy technical definitions.

The original SABA works with a version of BA containing the choice operator (“\(+/\)”) used to express computation options. The process “\(B_0 + B_1\)” behaves like \(B_0\) or \(B_1\) but only one of them. The original SABA just over approximates “\(B_0 + B_1\)” by analyzing it as though it were “\(B_0 \mid B_1\)”. We can either do the same or we can translate “\(B_0 + B_1\)” using some reserved name “\(ch\)” as “\(ch.(\{B_0\} \mid \{B_1\})\)” and alter the rules in \(B\) to work with “\(ch\)”. Both possibilities gives us the same result in static analysis of BA but the later can also be used for other calculi (\(\pi\)-calculus for example). A good handling of choice in POLY* is left for further research.

7 Conclusions and Future Work

This paper makes these contributions. (1) We give Meta* and POLY* a new approach to \(\alpha\)-conversion and name restriction which better supports comparison with other systems. (2) We show how to easily use POLY* as a type system for the \(\pi\)-calculus, MA, and BA. (3) We show the range and expressiveness of POLY* by comparing with three very different type/static analysis systems: TPI, TMA, and SABA. Conclusions for each calculus and comparison were given separately (Sec. 4.5, 5.5, 6.5). (4) Finally, and maybe the most importantly, we advocate a uniform notion of process shape types and show that it can express properties of processes of various calculi and that the expressiveness of shape types is comparable with predicates of other calculi. We also demonstrate polymorphic properties of process shape types. The concept of shape types is not bound to be used exclusively with POLY*. Another system, which can either refine the formalism used by POLY* or which can guarantee subject reduction by fundamentally different means, can make use of this concept.

Future work includes further extensions of Meta* and POLY* and related comparisons. For extensions, priorities are better handling of choice (e.g., because of its use in biological system modeling), and handling of \(rec\) because in many calculi \(rec\) is more expressive than replication and better supports describing some recursive behaviors. For comparison with other systems we would like to compare POLY* with (1) other systems which like POLY* use graphs to represent types [Yos96, Kön99], and (2) to study the relationship between POLY* types and session types [Hon93].

References

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Proc. FoSSaCS ‘98, volume 1378 of LNCS, pages 140–155. SV, 1998.
[CG99] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *Conf. Rec. POPL ’99: 26th ACM Symp. Princ. Program. Lang.*, pages 79–92, 1999.

[CGG99] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In Jiri Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *ICALP’99*, volume 1644 of *LNCS*, pages 230–239. SV, July 1999.

[CGG00] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In *IFIP Int’l Conf. on Theoret. Comput. Sci. (IFIP TCS2000)*, Tohoku University, Sendai, Japan, volume 1872 of *LNCS*, pages 333–347. SV, August 2000.

[Hon93] Kohei Honda. Types for dyadic interaction. In *CONCUR 1993*, volume 715 of *LNCS*, pages 509–523, 1993.

[IK01] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Conf. Rec. POPL ’01: 28th ACM Symp. Princ. Program. Lang.*, pages 128–141, 2001.

[Kön99] Barbara König. Generating type systems for process graphs. In *CONCUR 1999*, volume 1664 of *LNCS*, pages 352–367. SV, 1999.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge Press, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Inform. & Comput.*, 100(1):1–77, September 1992.

[MW04] Henning Makholm and J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. Technical Report HW-MACS-TR-0022, Heriot-Watt Univ., MACS., November 2004.

[MW05] Henning Makholm and J. B. Wells. Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. In *Prog. Languages & Systems, 14th European Symp. Prog.*, volume 3444 of *LNCS*, pages 389–407. SV, 2005.

[NNPR07] Flemming Nielson, Hanne Riis Nielson, Corrado Priami, and Debora Rosa. Control flow analysis for bioambients. *ENTCS*, 180(3):65–79, 2007.

[RPS+04] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Shapiro. Bioambients: An abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167, September 2004.
References

[1] Tur95] David N. Turner. The Polymorphic Pi-Calculus: Theory and Implementation. PhD thesis, Uni. of Edinburgh, 1995. Rep. ECS-LFCS-96-345.

[2] Wel02] J. B. Wells. The essence of principal typings. In Proc. 29th Int'l Coll. Automata, Languages, and Prog., volume 2380 of LNCS, pages 913–925. SV, 2002.

[3] Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. In Found. of Soft. Tech. and Theoret. Comp. Sci., volume 1180 of LNCS, pages 371–386. SV, 1996.

A Basic Properties of POLY* Shape Types

Definition A.1. Let μ be a message type and let a be a basic name. Define a message type μ\a, pronounced as μ without a, as follows:

\[ μ\a = \begin{cases} (\Phi\{a\})^* & \text{if } μ = Φ^* \\ μ & \text{otherwise} \end{cases} \]

Let ε be an element type and let a be a basic name. Define a message type ε\a, pronounced as ε without a, as follows:

\[ ε\a = \begin{cases} \langle (μ_1\a), \ldots, (μ_k\a) \rangle & \text{if } ε = Φ^* \\ ε & \text{otherwise} \end{cases} \]

Let G be a shape graph and a be a basic name. Define a shape graph G\a, pronounced as G without a, as follows:

\[ G\a = \{(\chi_0)^*\cdots(\chi_k)^*, \chi_1) \in G: \forall i \in \{0, \ldots, k\}: ε_i \neq a \& ε_i \neq (\ldots, a, \ldots) \& ε_i \neq <\ldots, a, \ldots> \} \]

Let π = ⟨G, χ⟩ be a shape predicate. Write π\a, pronounced as π without a, for ⟨G\a, χ⟩.

Lemma A.2 (Weakening). Let P be a Meta processes and let a ∉ fbn(P) ∪ ibn(P) ∪ nbn(P). Then for any π it holds that ⊢ P : π\a implies ⊢ P : π.

Lemma A.3 (Strengthening). Let P be a Meta processes and let a ∉ fbn(P) ∪ ibn(P) ∪ nbn(P). Then for any π it holds that ⊢ P : π implies ⊢ P : π\a.

B Proofs of Main Theorems

This section provides proofs and sketches encapsulating main ideas of proofs of the theorems in the paper.
B.1 The Proof of Prop. 3.1

Here we provide proof of Prop. 3.1 only for the case of MA. Proof of Prop. 3.1 for the \( \pi \)-calculus and BA are analogous. The proof for MA is the most complicated because the encoding of MA processes in Meta\( ^* \) is the less straightforward. All important ideas of the proofs for the \( \pi \)-calculus and BA are shown on the proof for MA.

We use the following names for the rewriting rules of MA which are assigned to the rules from Fig. 12 in the left-right and top-down order: \( \text{AIn, AOut, AOpen, ACom, AAmb, ANu, APar, and AStr} \). Similarly we name the Meta\( ^* \) rewriting rules from Fig.4 as follows: RRew, RNu, RStr, RPar, RAct.

**Lemma B.1.** It holds that

\[
\{ B[n_1 \mapsto N_1, \ldots, n_k \mapsto N_k] \} = \{ B\} \{ n_1 \mapsto \{ N_1 \}, \ldots, n_k \mapsto \{ N_k \} \}
\]

The following proposition is the left-to-right implication of Prop. 3.1 for MA. Additionally, we let \( C \) range over MA processes AProcess in the proof.

**Proposition B.2.** Let \( B_0 \rightarrow B_1 \). Then there exist \( B'_0, B'_1 \) such that

\[
B_0 \equiv B'_0 \land \{ B'_0\} \rightarrow \{ B'_1\} \land B'_1 \equiv B_1
\]

**Proof.** By induction on the derivation of \( B_0 \rightarrow B_1 \). Let it be derived by

(AIn): Then, for some \( n, m, C_0, C_1, \) and \( C \) we have \( B_0 = n[m[C_0 \mid C_1]\mid m[C_1] \rangle \) and \( B_1 = m[n[C_0 \mid C_1]\mid C_1] \rangle \). Take the instantiation \( \rho = \{ \hat{\alpha} \mapsto n, \hat{\beta} \mapsto m, \hat{\rho} \mapsto \{ C_0\}, \hat{\psi} \mapsto \{ C_1\} \rangle \). Now, we know that rewrite\( \{ \hat{\alpha} \mapsto n, \hat{\beta} \mapsto m, \hat{\rho} \mapsto \{ C_0\}, \hat{\psi} \mapsto \{ C_1\} \rangle \in A \). Moreover it is easy to see that \( \{ B_0\} = \{ \hat{\alpha} \mapsto n, \hat{\beta} \mapsto m, \hat{\rho} \mapsto \{ C_0\}, \hat{\psi} \mapsto \{ C_1\} \rangle \) and \( \{ B_1\} = \{ \hat{\alpha} \mapsto n, \hat{\beta} \mapsto m, \hat{\rho} \mapsto \{ C_0\}, \hat{\psi} \mapsto \{ C_1\} \rangle \).

Take directly \( B'_0 = B_0 \) and \( B'_1 = B_1 \) and we have \( \{ B'_0\} \rightarrow \{ B'_1\} \) by RRew.

(AOut): Like in case AIn.

(AOpen): Like in case AIn.

(ACom): Similarly to case AIn but with the following changes:

\[
B_0 = (n_1 : \omega_1, \ldots, n_k : \omega_k).C \mid \langle N_1, \ldots, N_k \rangle
B_1 = C[n_1 \mapsto \{ N_1 \}, \ldots, n_k \mapsto \{ N_k \}]
\]

\[
\rho = \langle \hat{\alpha} \mapsto n_1, \ldots, \hat{\beta} \mapsto n_k, \hat{\rho} \mapsto \{ N_1 \}, \ldots, \hat{\beta} \mapsto \{ N_k \}, \hat{\rho} \mapsto 0, \hat{\psi} \mapsto \{ C \} \rangle
\]

\[
P_0 = \{ \langle \hat{\alpha}_1, \ldots, \hat{\beta}_k, \hat{\rho} \rangle \mid \langle N_1, \ldots, N_k \rangle, \hat{\rho} \}_{\rho} \rightarrow (n_1, \ldots, n_k).C \mid \langle N_1, \ldots, N_k \rangle, \rightarrow 0 \}
\]

\[
P_1 = \{ \langle \hat{\alpha}_1 := \hat{\beta}_1, \ldots, \hat{\beta}_k := \hat{\beta}_k \rangle \}_{\rho} \rightarrow 0 \mid \{ C \} \mid n_1 \mapsto \{ N_1 \}, \ldots, n_k \mapsto \{ N_k \}
\]

\[
B'_0 = B_0
B'_1 = 0 \mid B_1
\]

We have \( \{ B'_0\} = P_0 \) directly and \( \{ B'_1\} = P_1 \) by Lemma B.1. By TMA structure equivalence we have \( B_0 \equiv B'_0 \) and \( B_1 \equiv B'_1 \). Thus \( \{ B'_0\} \rightarrow \{ B'_1\} \) by RRew.

(AAmb): Here simply use the induction hypothesis and then instantiate the rule active\( \{ \hat{\rho} \} \) in RAct by \( \rho = \{ \hat{\alpha} \mapsto n \} \) where \( n \) is the ambient name obtained from the assumptions. Here we have to verify that \( n \neq * \) which is true for \( * \) is forbidden to be used by TMA processes.
(ANu): Again use the induction hypothesis and verify that the $\nu$-bound name $\mathcal{A}$ is not in $\text{fn}(\mathcal{A}) = \{\text{in}, \text{out}, \text{open}, \emptyset\}$. This is satisfied for these names are excluded from AName. Then use RNu to prove the claim.

(APar): Use the induction hypothesis and RPar to prove the claim.

(AStr): Use the induction hypothesis and RStr to prove the claim.

The following proposition is the right-to-left implication of Prop. 3.1 for TMA.

**Proposition B.3.** Let $\{B_0\} \rightleftharpoons P_1$. Then there exists some $B_1$ such that $\{B_1\} \equiv P_1$ and $B_0 \rightarrow B_1$.

**Proof.** By induction on the derivation of $\{B_0\} \rightleftharpoons P_1$. Let it be derived by

(RRew): using the rule

1. **Rewrite** $\{\hat{P} \mapsto \hat{Q}\} = \text{rewrite}(\hat{A}((\hat{\mathcal{B}} \mathcal{P} | \mathcal{Q}) | \hat{\mathcal{A}}) \rightarrow \hat{D}(\hat{A}(\hat{\mathcal{P}} | \hat{\mathcal{Q}}) | \hat{\mathcal{A}}))$.

   We also know that there is some instantiation $\rho$ with all the variables mentioned by the rule in its range. We define $x = [\hat{A}]_\rho$, $y = [\hat{\mathcal{B}}]_\rho$, $P'_0 = [\hat{\mathcal{P}}]_\rho$, $P'_1 = [\hat{\mathcal{P}}]_\rho$, $P'_2 = [\hat{\mathcal{P}}]_\rho$. Now we can deduce that $\{B_0\} = x[y, P'_0 | P'_1] y[P'_2]$ and $P_1 = y[x[P'_0 | P'_1] | P'_2]$. Now there have to be $B'_0, B'_1, B'_2$ such that $\{B'_0\} = P'_0$, $\{B'_1\} = P'_1$, $\{B'_2\} = P'_2$, and $B_0 \equiv x[y, B'_0 | B'_1] y[B'_2]$. It holds that both $x$ and $y$ are in AName because (1) $\rho$ can not map a name variable to $\bullet$ and (2) in, out, open, $\emptyset$ can not appear in $B_0$. Now we just take $B_1 = y[x[B'_0 | B'_1] | B'_2]$ and thus we have $\{B_1\} = P_1$. Finally we proof $B_0 \rightarrow B_1$ by AIn and AStr.

2. **Proof for the other three rules** (out, open, and the communication one) is similar as case 1.

(RAct): using the rule $\text{active}(\hat{P}) \mapsto \hat{\mathcal{A}}(\hat{\mathcal{P}})$. Denote $x = [\hat{A}]_\rho$. In this case we have that there are some $P$ and $Q$ such that $P \rightleftharpoons Q$. We also have that $\{B_0\} = x[P]$ and $P_1 = x[Q]$. Thus we see that there is some $B'_0$ such that $\{B'_0\} = P$ and $B_0 = x[B'_0]$. It also implies that $x \in$ AName. Thus we obtain $\{B'_0\} \rightleftharpoons Q$ and by the induction hypothesis we have that there exists $B'_1$ such that $\{B'_1\} \equiv P$ and $B'_0 \rightarrow B'_1$. Take $B_1 = x[B'_1]$. We have $\{B_1\} = x[(B'_1)] \equiv x[Q] = P_1$. Finally $B_0 \rightarrow B_1$ by AAMB.

(RNu): Thus there are $x$, $\mathcal{P}$, and $Q$, such that $\{B_0\} = \nu(x).P$ and $P_1 = \nu(x).Q$, and $P \rightleftharpoons Q$. Here we see that $x \in$ AName and thus $x \notin \text{fn}(\mathcal{A})$. From $\{B_0\} = \nu(x).P$ we can conclude that there are some $\omega$ and $B'_0$ such that $B_0 = (\nu \omega : B'_0)$ and $\{B'_0\} = P$. Thus we have $\{B'_0\} \rightleftharpoons Q$ and by the induction hypothesis we obtain that there exists $B'_1$ such that $\{B'_1\} = Q$ and $B'_0 \rightarrow B'_1$. Let us take $B_1 = (\nu \omega : B'_1)$. Now $\{B_1\} = \nu(x).[B'_1] \equiv \nu(x).Q = P_1$. Finally $B_0 \rightarrow B_1$ by ANu.

(RPar): Proof is similar to case RNu.

(RStr): The problem to deal with in this case is the difference in structural equivalences of Meta* and TMA, in particular, the Meta* rule present which allows a $\nu$-binder to skip an arbitrary action. Against that, TMA allows $\nu$-binders to skip ambient boundaries only. For example for $B_0 = (\nu \omega : A \rightarrow \emptyset)$. \(\nu \mathcal{A}\):
\[ \omega \) in \( a.0 \) and \( B_1 = (\nu a: \omega)() \) in \( a.0 \) we have \( \{B_0\} \equiv \{B_1\} \) in \( \text{META}^\star \) but not \( B_0 \equiv B_1 \) in \( \text{TMA} \). The key observation here is that whenever in \( \text{META}^\star \) some rewriting is inferred by \( \text{RSTR} \) using \( \text{META}^\star \) structural equivalence in a way that is not allowed in \( \text{TMA} \), then the same rewriting statement can be inferred in \( \text{META}^\star \) using a derivation that uses structural equivalence only in a \( \text{TMA} \)-compatible way. Then rest of the proof is a simple application of the induction hypothesis.

### B.2 The Proof of Thm. 4.4

This section provides the proof of Thm. 4.4. Def. B.4 extends the definition from Fig. 11 with some additional notations. Prop. B.5 is the left-to-right implication of Thm. 4.4 and Prop. B.6 is its right-to-left implication. The proofs use a standard weakening and strengthening lemmas [Tur95, Lem. 3.8-9].

**Definition B.4.** Write \( \Delta \cong G \) when \( \Delta \cong (G, \chi) \) for an arbitrary \( \chi \). Moreover, we say that \( \Delta \cong G \) via \( \Delta' \) when \( \text{dom}(\Delta) \cap \text{dom}(\Delta') = \emptyset \) and \( \text{chtype}(\Delta \cup \Delta', G) \) is defined and an identity relation.

**Proposition B.5.** Let

1. \( B \) be a \( \pi \)-process such that no two different binders bind the same name,
2. \( \pi_B \) be a principal \( P \)-type of \( \{B\} \),
3. \( \text{dom}(\Delta) \subseteq \text{fbn}(B) \), and
4. \( \Delta \vdash B \).

Then \( \Delta \cong \pi_B \).

**Proof.** By induction on the structure of \( B \). Let \( \pi = (G, \chi) = \pi_B \). Let

\[ B = 0; \text{Then } G \cong \Delta \text{ holds trivially because } G = \emptyset. \]

\[ B = B_0 \rightarrow B_1: \text{Let } \pi_0 = (G_0, \chi_0) \text{ be a principal type of } \{B_0\} \text{ and } \pi_1 = (G_1, \chi_1) \text{ be a principal type of } \{B_1\}. \text{ Take} \]

\[
\begin{align*}
\Delta_0 &= \{(a \mapsto \delta): (a \mapsto \delta) \in \Delta \land a \in \text{dom}(B_0)\} \\
\Delta_1 &= \{(a \mapsto \delta): (a \mapsto \delta) \in \Delta \land a \in \text{dom}(B_1)\}
\end{align*}
\]

Let us verify the assumptions of the induction step for \( B_0, \pi_0, \) and \( \Delta_0 \):

1. Clear.
2. Clear.
3. \( \text{dom}(\Delta_0) = \text{dom}(\Delta) \cap \text{dom}(B_0) \subseteq \text{dom}(B_0) \).
4. Here \( \Delta_0 \vdash B_0 \) follows from \( \Delta \vdash B_0 \) by strengthening.

Similarly the assumptions are satisfied for \( B_1, \pi_1, \) and \( \Delta_1 \). Thus by the induction hypothesis we have that \( \Delta_0 \cong \pi_0 \) and \( \Delta_1 \cong \pi_1 \). Let \( \Delta_0 \cong G_0 \) via \( \Delta'_0 \) and let \( \Delta_1 \cong G_1 \) via \( \Delta'_1 \). Because (1) we can suppose that \( \text{dom}(\Delta'_0) \cap \text{dom}(\Delta'_1) = \emptyset \). Take \( \Delta' = \Delta'_0 \cup \Delta'_1 \). We shall proof that \( \Delta \cong G \) via \( \Delta_0 \). Denote
\[ \Delta^* = \Delta \cup \Delta'. \] Although \( G \) can contain some additional edges not contained in \( G_0 \) and \( G_1 \) it can not introduce new basic names. When \( G \) contains a type substitution \( \tau \) such that \( \tau(a) = b \) then we can observe that \( \Delta^*(a) = \Delta^*(b) \). Thus all additional members in \( \text{chtypes}(\Delta^*, G) \) are identities because they are constructed by application of type substitutions in \( G \). Thus the claim.

\[ B = c(n_1, \ldots, n_k).B_0: \] Let \( a = c \) and \( b_i = n_i \) for \( 0 < i \leq k \). Let \( \pi_0 = \langle G_0, \chi_0 \rangle \) be a principal type of \( \{B_0\} \). There are some \( \delta_1, \ldots, \delta_k \) such that \( \Delta(a) = \uparrow[\delta_1, \ldots, \delta_k] \). Take \( \Delta_0 \) which does not contain \( b \)'s not mentioned in \( B_0 \) as follows:

\[ \Delta_0 = \Delta|_{\{b_1 \mapsto \delta_1, \ldots, b_k \mapsto \delta_k\} \setminus \{b_i \mapsto \delta_i : 0 < i \leq k \text{ and } b_i \notin \text{dom}(B_0)\}} \]

Now verify the assumptions of the induction step for \( B_0, \pi_0, \) and \( \Delta_0 \):

1. Clear.
2. Clear.
3. \( \text{dom}(\Delta_0) = \text{dom}(\Delta) \cap \{\text{dom}(B_0) \setminus \text{dom}(B_0) \subseteq \text{dom}(B_0)\} \).
4. The assumption \( \Delta \vdash B \) implies \( \Delta|_{\{b_1 \mapsto \delta_1, \ldots, b_k \mapsto \delta_k\}} \vdash B_0 \) and thus \( \Delta_0 \vdash B_0 \) by strengthening.

Thus by the induction hypothesis we have that \( \Delta_0 \cong \pi_0 \). Let \( \Delta_0 \cong G_0 \) via \( \Delta'_0 \). We can find \( \Delta' \) such that \( \Delta \cup \Delta' = \Delta_0 \cup \Delta'_0 \) and thus also \( \Delta \cong \pi_0 \). Denote \( \Delta^* = \Delta \cup \Delta' \). It is easy to see that the principal type \( \pi \) of \( B \) simply directly corresponds to the syntax tree of \( B \) because no rewriting rule can be applied to \( B \) standing alone. Also all action types contained in \( G \), up to \( \alpha = a(b_1, \ldots, b_k) \) which labels the only edge coming out of the root node \( \chi \), are contained in \( G_0 \). The member of \( \text{chtypes}(\Delta^*, G) \) corresponding to the edge labeled by \( \alpha \) is identity because \( \Delta(a) = \uparrow[\delta_0, \ldots, \delta_k] \) follows directly from the assumption \( \Delta \vdash B \). All other members of \( \text{chtypes}(\Delta^*, G) \) are present also in \( \text{chtypes}(\Delta_0 \cup \Delta'_0, G_0) \) and thus \( \text{chtypes}(\Delta^*, G) \) is defined and identity. Thus \( \Delta \cong \pi \).

\[ B = c(n_1, \ldots, n_k).B_0: \] Let \( a = c \) and \( b_i = n_i \) for \( 0 < i \leq k \). Let \( \pi_0 = \langle G_0, \chi_0 \rangle \) be a principal type of \( \{B_0\} \). Take \( \Delta_0 = \{(a' \mapsto \delta) \in \Delta : a' \in \text{fbn}(B_0)\} \). Let us verify the assumptions of the induction step for \( B_0, \pi_0, \) and \( \Delta_0 \):

1. Clear.
2. Clear.
3. \( \text{dom}(\Delta_0) = \text{dom}(\Delta) \cap \text{dom}(B_0) \subseteq \text{dom}(B_0) \).
4. Here \( \Delta_0 \vdash B_0 \) follows from \( \Delta \vdash B_0 \) by strengthening.

By the induction hypothesis we obtain that \( \Delta_0 \cong \pi_0 \). Let \( \Delta_0 \cong G_0 \) via \( \Delta'_0 \). Denote \( \Delta^* = \Delta_0 \cup \Delta'_0 \). Now let us proof that \( \Delta \cong \pi \). It is easy to see that the principal type \( \pi \) of \( B \) simply directly corresponds to the syntax tree of \( B \) because no rewriting rule can be applied to \( B \). Also, it is easy to see that all action types contained in \( G \), up to \( \alpha = a(b_1, \ldots, b_k) \) which labels the only edge coming out of the root node \( \chi \), are contained in \( G_0 \). The member
of chtypes(Δ*, G) corresponding to the edge labeled by α is identity because Δ(a) = ⊢Δ0, ..., Δ(n)) follows directly from the assumption Δ ⊢ P.

All other members of chtypes(Δ*, G) are present also in chtypes(Δ*, G0) and thus chtypes(Δ*, G) is defined and identity. Thus the claim.

B = !B0: Trivial.

B = (∀n)B0: Let a = n. We know that Δ ⊢ B. Condition (3) implies a ∈ dom(Δ).

Take

Δ0 = \begin{cases} 
Δ[a → δ] & \text{when } a ∈ \text{dom}(B_0) \\
Δ & \text{when } a ∉ \text{dom}(B_0) 
\end{cases}

We see that π is a principal type of [B0] as well. Let us verify the assumptions of the induction step for B0, π, and Δ0.

1. Clear.
2. Clear.
3. When a ∈ dom(B0) then dom(Δ0) = dom(Δ) ∪ a ⊆ dom(B) ∪ a = dom(B0). When a ∉ dom(B0) then dom(Δ0) = dom(Δ) ⊆ dom(Δ) = dom(B0).
4. When a ∈ dom(B0) then Δ0 ⊢ B0 follows from Δ ⊢ P. When a ∉ dom(B0) then Δ ⊢ P implies Δ[a → δ] ⊢ B0 and thus Δ0 ⊢ B0 by strengthening.

Thus by the induction hypothesis we have that Δ0 ∪ π. Let Δ0 ∪ π via Δ0. It is easy to see that we can take Δ′ such that Δ ∪ Δ′ = Δ0 ∪ Δ′ and thus the claim Δ ∪ π holds.

PROPOSITION B.6. Let B be a π-process and let

1. πB be a principal P-type of [B],
2. fbn(B) ⊆ dom(Δ), and
3. Δ ∪ πB.

Then Δ ⊢ B.

PROOF. By induction on the structure of B. Let π = (G, χ) = πB. Let

B = 0: Trivial.

B = B0 | B1: Let π0 = (G0, χ0) be a principal type of [B0] and let π1 = (G1, χ1) be a principal type of [B1]. It has to hold that any action type α contained in G0 (respectively G1) as an edge label is also contained in G. Thus we have Δ ∪ π0 and Δ ∪ π1. It shows that the assumptions of the induction step are satisfied and by the induction hypothesis we have that Δ ⊢ B0 and Δ ⊢ B1 which proof the claim Δ ⊢ B.

B = c(n1, ..., nk).B0: Let a = c and b1 = n1 for 0 < i ≤ k. Let Δ ∪ G via Δ′. Denote Δ* = Δ ∪ Δ′. Take

Δ0 = Δ[b1 → Δ′(b1), ..., bk → Δ′(bk)]
It is easy to see that $\Delta_0$ is defined. Let $\pi_0 = \langle G_0, \chi_0 \rangle$ be a principal type of $\{B_0\}$. Now let us verify the assumptions of the induction step for $B_0$, $\pi_0$, and $\Delta_0$.

1. Clear.
2. $\text{dom}(B_0) \subseteq \text{dom}(P) \cup \{b_1, \ldots, b_k\} \subseteq \text{dom}(\Delta) \cup b_1, \ldots, b_k = \text{dom}(\Delta_0)$.
3. We can take $\Delta'_0$ such that $\Delta' = \Delta \cup \Delta' = \Delta_0 \cup \Delta'_0$. Note that $G_0$ could contain additional edges which are not in $G$. Those are edges introduced by the closure algorithm to make the shape graph flow closed. But we can observe that whenever $G_0$ contains some flow edge labeled with type substitution $\tau$ such that $\tau(a') = b'$, then it has to hold that $\Delta'(a') = \Delta'(b')$. Thus $\Delta_0 \cong \pi_0$.

By the induction hypothesis we obtain $\Delta_0 \vdash B_0$. We see that $\Delta(a) = \uparrow[\Delta'(b_1), \ldots, \Delta'(b_k)]$ and thus the claim.

$B = \langle n \rangle_{B_0}$: Let $a = a_i$ and $b_i = n_i$, for $0 < i \leq k$. Let $\Delta \cong G$ via $\Delta'$. Denote $\Delta' = \Delta \cup \Delta'$. Let $\pi_0 = \langle G_0, \chi_0 \rangle$ be a principal type of $\{B_0\}$. Now let us verify the assumptions of the induction step for $B_0$, $\pi_0$, and $\Delta$.

1. Clear.
2. $\text{dom}(B_0) \subseteq \text{dom}(B) \subseteq \text{dom}(\Delta)$.
3. To prove $\Delta \cong \pi_0$ use the same argument as in the proof of assumption 3 of the previous case concerning an input-binder.

By the induction hypothesis we obtain $\Delta \vdash B_0$. We see that $\Delta \cong \pi$ implies $\Delta(a) = \uparrow[\Delta(b_1), \ldots, \Delta(b_k)]$ and thus the claim.

$B = \langle \forall \rangle_{B_0}$: Let $a = \forall$. We can suppose that $a \in \text{dom}(B_0)$ because otherwise we can directly use the induction hypothesis (for $B_0$, $\pi$, and $\Delta$) and weakening to proof the claim. Let $\Delta \cong G$ via $\Delta'$. Because $a \in \text{dom}(B_0)$ we have that there is $\delta = (\Delta \cup \Delta')(a)$. Take $\Delta_0 = \Delta[a \mapsto \delta]$. Let us verify the assumptions of the induction step:

1. $\pi$ is a principal type of $\{B_0\}$ as well
2. $\text{dom}(B_0) = \text{dom}(P) \cup \{a\} \subseteq \text{dom}(\Delta) \cup \{a\} = \text{dom}(\Delta_0)$
3. When $a \notin \text{dom}(\Delta)$ then $\Delta_0 \cong G$ via $\Delta'\{a \mapsto \delta\}$. When $a \in \text{dom}(\Delta)$ then $\Delta = \Delta_0$ and thus obvious.

By the induction hypothesis we have that $\Delta_0 \vdash B_0$ and thus the claim.

### B.3 The Proof of Thm. 5.8

Prop. B.7 is the left-to-right implication of Thm. 5.8 and Prop. B.8 is the right-to-left implication. The assumption of the propositions that $\Delta(a) = \Delta'_0(a)$ for every $a \in \text{dom}(\Delta) \cap \text{fn}(B)$ follows from the assumption of the theorem that $\text{dom}(\Delta) \cap \text{fn}(B) = \emptyset$. Note that the part of well-scopedness condition S1, that free basic names and $v$-bound basic names do not overlap, is not preserved for subprocesses. Thus we can not suppose it in structural induction proofs.
Proposition B.7. Let $\Delta(a) = \Delta_0^\gamma(a)$ for every $a \in \text{dom}(\Delta) \cap \text{nbn}(B)$. Then

$$\Delta \vdash B : \kappa \implies \vdash \{B\} : \{(\Delta \cup \Delta_0^\gamma, \Delta_0^\nu, \kappa)\}$$

Proof. Let $\Delta \vdash B : \kappa$. Let $I = (\Delta \cup \Delta_0^\gamma, \Delta_0^\nu, \kappa)$ and $\pi = \{I\}$. Prove $\vdash \{B\} : \pi$ by induction on the structure of $B$. Let

$B = 0$: Clear.

$B = (B_0 \mid B_1)$: We know that $\Delta \vdash B_0 : \kappa$. We see that $\text{nbn}(B_0) \subseteq \text{nbn}(B)$. Thus the assumption of the induction step for $B_0$ is satisfied. Let $\pi_0 = \{(\Delta \cup \Delta_0^\gamma, \Delta_0^\nu, \kappa)\}$. By the induction hypothesis we obtain $\vdash \{B_0\} : \pi_0$. Because $\text{iBN}(B_0) \subseteq \text{iBN}(B)$ and $\text{nbn}(B_0) \subseteq \text{nbn}(B)$ we see that $\pi$ contains all the edges of $\pi_0$. Thus also $\vdash \{B_0\} : \pi$ by Lemma A.2. Similarly we obtain $\vdash \{B_1\} : \pi$. Thus the claim.

$B = N[B_0]$: We know that there is some $\kappa'$ such that $\Delta \vdash N : \text{Amb}[\kappa']$ and $\Delta \vdash B_0 : \kappa'$. Thus it is clear that there is some $n$ such that $N = n$. Let $a = n$. We have $\text{nbn}(B_0) = \text{nbn}(B)$ and thus the assumption of the induction step for $B_0$ and $\kappa'$ is satisfied. Let $\pi_0 = \{(\Delta \cup \Delta_0^\gamma, \Delta_0^\nu, \kappa')\}$. By the induction hypothesis we obtain $\vdash \{B_0\} : \pi_0$. We see that $\Delta_0^\nu = \Delta_0^\nu$ and $\Delta_0^\gamma = \Delta_0^\gamma$ and thus $\pi_0$ and $\pi$ differ only in the root nodes. Let $\chi$ be the root node of $\pi$ and let $\chi_0$ be the root node of $\pi_0$. It is clear that $\chi = \text{nodef_1}(\kappa)$ and $\chi_0 = \text{nodef_1}(\kappa')$. (Note that for $I_0 = (\Delta \cup \Delta_0^\gamma, \Delta_0^\nu, \kappa')$ it does not need to hold that $\kappa \in \text{types}_{i_0}$.) We can see that $\Delta(a) = \text{Amb}[\kappa']$ and thus $a \in \text{namesf_1}(\text{Amb}[\kappa'])$. Thus $(\chi \xrightarrow{a} \chi_0) \in G$. Hence the claim $\vdash n.[\{B_0\}] : \pi$ because $\vdash n[\{I\} : a.]$.

$B = N*B_0$: We see that $\Delta \vdash N : \text{Cap}[\kappa]$ and $\Delta \vdash B_0 : \kappa$. The assumption of the induction step for $B_0$ is satisfied because $\text{nbn}(B_0) = \text{nbn}(B)$. Let $\pi_0 = \{(\Delta \cup \Delta_0^\gamma, \Delta_0^\nu, \kappa)\}$. By the induction hypothesis we obtain $\vdash \{B_0\} : \pi_0$. Clearly $\pi$ contains all the edges as $\pi_0$ and thus also $\vdash \{B_0\} : \pi$ by Lemma A.2. Let $\chi = \text{nodef_1}(\kappa)$ be the root node of $\pi$. We see that $\chi$ is the root of $\pi_0$ as well. Let us prove the claim by induction on the structure of $N$. (We are proving that $\vdash \{B_0\} : \pi$ and $\Delta \vdash N : \kappa$ implies $\vdash \{\text{N.B_0}\} : \pi$.)

$N = n$: Let $a = n$. Then it is clear that $\Delta(a) = \text{Cap}[\kappa]$. Thus $a \in \text{namesf_1}(\text{Cap}[\kappa])$ and $a \in \text{opensf_1}(\kappa)$. Now we see that $\pi$ contains the edge $\chi \xrightarrow{a} \chi$ and because $\vdash n : a$ we see that $\vdash n.\{B_0\} : \pi$. Thus the claim.

$N = \text{in N}$: We see that there is some $\kappa'$ such that $\Delta \vdash \text{Amb}[\kappa']$ and thus there is some $n$ such that $N = n$. Let $a = n$. We have $\Delta(a) = \text{Amb}[\kappa']$ and thus $a \in \text{namesf_1}(\text{Cap}[\kappa'])$ and in $a \in \text{movesf_1}$. Now we see that $\pi$ contains the edge $\chi \xrightarrow{a} \chi$ and because $\vdash n : \text{in} \in a$ we see that $\vdash n.\{B_0\} : \pi$. Thus the claim.

$N = \text{out N}$: As in the case for “in N”.

$N = \text{open N}$: As in the case for “in N” but here $\kappa' = \kappa$ and open $a \in \text{opensf_1}(\kappa)$.
\( N = N_0.N_1: \) We have that \( \Delta \vdash N_0 : \kappa \) and \( \Delta \vdash N_1 : \kappa \). By the induction hypothesis for \( N_1 \) and \( B_0 \) we obtain that \( \vdash \{ N_1, B_0 \} : \pi \). By the induction hypothesis for \( N_0 \) and \( N_1, B_0 \) (which is still structurally smaller than \( N \)) we obtain that \( \vdash \{ N_0, (N_1, B_0) \} : \pi \). Now we see that \( \{ N_0, (N_1, B_0) \} = \{ (N_0, N_1), B_0 \} = \{ B \} \). Hence the claim.

**B = \{B_0\}:** We know that \( \Delta \vdash B : \pi \). The assumption of the induction step is clearly satisfied. Thus the claim follows from the induction hypothesis because \( \Delta^{n}_{\kappa} = \Delta^{n}_{\kappa} \) and \( \Delta^{m}_{\kappa} = \Delta^{m}_{\kappa} \).

**B = (\forall n:\omega)B_0:** Let \( a = n \). We know that there is some \( \kappa' \) such that \( \omega = \text{Amb}[\kappa'] \). Thus \( \Delta^{\kappa'}_{B_0}(a) = \omega \). Let \( \Delta_0 = \Delta[a \mapsto \omega] \). We know that \( \Delta_0 \vdash B_0 : \kappa \). Let \( b \in \text{dom}(\Delta_0) \cap \text{nbn}(B_0) \). When \( b \neq a \) then obviously \( b \in \text{dom}(\Delta) \cap \text{nbn}(B) \) and thus \( \Delta_0(b) = \Delta^{\kappa'}_{B_0}(b) \). When \( b = a \) we have that \( \Delta_0(b) = \omega = \Delta^{\kappa}_{B}(b) \). Now because \( b \in \text{nbn}(B_0) \) we have that \( \Delta^{\kappa}_{B_0}(b) = \omega \) by well-scopedness condition S4. Thus the assumption of the induction step for \( \Delta_0 \) and \( B_0 \) is satisfied.

Let \( \pi_0 = \{ (\Delta_0 \cup \Delta^{\kappa}_{B_0}, \Delta^{m}_{\kappa}) \} \). By the induction hypothesis we obtain that \( \vdash \{ B_0 \} : \pi_0 \). By the same arguments used to prove the assumption of the induction step we can prove that \( \Delta_0 \cup \Delta^{\kappa}_{B_0} = \Delta \cup \Delta^{\kappa}_{B} \). (When \( a \in \text{dom}(\Delta) \) then \( \Delta(a) = \Delta^{\kappa}_{B_0}(a) = \omega = \Delta_0(a) \).) Obviously \( \Delta^{m}_{\kappa} = \Delta^{m}_{B_0} \) and thus \( \pi_0 = \pi \). Hence the claim.

**B = <N_1, \ldots , N_k>:** We know that \( \kappa = \omega_1 \otimes \cdots \otimes \omega_k \) and \( \Delta \vdash N_i : \omega_i \) for all \( 0 < i \leq k \). Let us prove, for any \( i \), by induction on the structure of \( N_i \) that there is some \( \mu_i \in \text{msg}_{\kappa}(\omega_i) \) such that \( \vdash \{ N_i \} : \mu_i \). Let

\[ N_i = n: \] Let \( a = n \). Take \( \mu_i = a \). It is clear that \( a \in \text{names}_i(\omega_i) \) and thus \( a \in \text{msg}_{\kappa}(\omega_i) \). Hence the claim.

\[ N_i = nN: \] It is clear that there is some \( n \) such that \( N = n \). Let \( a = n \). We can see that \( \omega = \text{Amb}[\kappa'] \) for some \( \kappa' \) and \( \Delta(a) = \text{Amb}[\kappa'] \). Take \( \mu_i = a \). Thus \( \mu_i \in \text{msg}_{\kappa}(\omega_i) \). Hence the claim.

\[ N_i = nN: \] As in the case for “in N”.

\[ N_i = nN: \] As in the case for “in N” but here \( \kappa' = \kappa \) and \( \mu_i = \text{open} a \in \text{open}_{\kappa}(\kappa) \).

\[ N_i = nN' \] : It is clear that \( \omega = \text{Cap}[\kappa'] \) for some \( \kappa' \). From the induction hypothesis we have \( \mu \) and \( \mu' \) such that \( \vdash \{ N \} : \mu \) and \( \vdash \{ N' \} : \mu' \). When both \( \mu \) and \( \mu' \) are message types of the form \( \Phi \star \) then \( \mu = \mu' \) and thus \( \vdash \{ N_i \} : \mu \). When both \( \mu \) and \( \mu' \) are basic names then we have \( N, N' \in \text{names}_i(\omega) \). But we know that \( \text{msg}_{\kappa}(\omega) \) contains exactly one message type of the shape \( \Phi \star \) and because \( \text{names}_i(\omega) = \text{names}_i(\text{Cap}[\kappa']) \subseteq \text{open}_{\kappa}(\kappa') \) we have that \( \mu, \mu' \in \Phi \). Thus \( \vdash \{ N_i \} : \Phi \star \). A similar situation is when only one of \( \mu \) and \( \mu' \) is a basic name. Then the second one is the same \( \Phi \star \) as above and the first basic name is in \( \Phi \). Thus the claim.

Now let \( \alpha = <\mu_1, \ldots , \mu_k> \). Let \( \chi = \text{node}_{\kappa}(\kappa) \) be the root node of \( \pi \). We see that \( \pi \) contains \( \chi \xrightarrow{\star} \chi \) because \( \alpha \in \text{allowed}_i(\omega) \). Thus we can prove that \( \vdash <\{ N_1 \}, \ldots , [N_k]> : \pi \). Hence the claim.
\[ B = (n_1; \omega_1, \ldots, n_k; \omega_k).B_0; \] Let \( a_i = n_i \) for \( 0 < i \leq k \). Let \( \Delta_0 = \Delta[n_1 \mapsto \omega_1, \ldots, n_k \mapsto \omega_k] \). We know that \( \kappa = \omega_1 \otimes \cdots \otimes \omega_k \) and \( \Delta_0 \vdash B_0 : \kappa \). By the well-scoped condition S1 (the part that \( \nu \) - and input-bound basic names do not intersect) we have for any \( i \) that \( a_i \notin \text{nbn}(B_0) \). Thus \( \text{dom}(\Delta_0) \cap \text{nbn}(B_0) = \text{dom}(\Delta) \cap \text{nbn}(B) \) and the assumption of the induction step is satisfied.

Let \( \pi_0 = \{ (\Delta_0 \cup \Delta_{B_0}^\nu, \Delta_{B_0}^\nu, \kappa) \} \). By the induction hypothesis we obtain that \( \vdash \{ B_0 \} : \pi_0 \). Let \( \alpha = (a_1, \ldots, a_k) \) and let \( \chi = \text{nodeof}_1(\kappa) \) be the root node of \( \pi \). We can see that \( \chi \) is the root of \( \pi_0 \) as well. Moreover we can see that \( \pi \) contains all the edges of \( \pi_0 \) by Lemma A.2. Thus also \( \vdash \{ B_0 \} : \pi \). For any \( i \) we have \( \Delta_{B_0}^\nu (a_i) = \omega_i \) and thus \( \alpha \in \text{allowed}_1(\omega_1 \otimes \cdots \otimes \omega_k) \). Thus the shape graph of \( \pi \) additionally contains the edge \( \langle \chi \rightarrow \omega_i \rangle \). Thus \( \vdash (n_1, \ldots, n_k).\{ B_0 \} : \pi \). Hence the claim.

**Proposition B.8.** Let \( \text{nbn}(B) = \text{dom}(\Delta_B^\nu) \). Let \( \Delta(a) = \Delta_B^\nu (a) \) for every \( a \in \text{dom}(\Delta) \cap \text{nbn}(B) \). Then

\[ \vdash \{ B \} : \{ (\Delta \cup \Delta_B^\nu, \Delta_B^\nu, \kappa) \} \quad \text{implies} \quad \Delta \vdash B : \kappa. \]

**Proof.** Let \( I = (\Delta \cup \Delta_B^\nu, \Delta_B^\nu, \kappa) \) and \( \pi = \{ I \} \) and \( (G, \chi) = \pi \). Thus we have \( \chi = \text{nodeof}_1(\kappa) \). Let \( \vdash \{ B \} : \pi \). Prove \( \Delta \vdash B : \kappa \) by induction on the structure of \( B \).

Let

- \( B = 0 \): Clear.
- \( B = (B_0 \mid B_1) \): We know \( \vdash \{ B_0 \} : \pi \). Let \( \pi_0 = \{ (\Delta \cup \Delta_B^\nu, \Delta_B^\nu, \kappa) \} \). Now \( \pi \) contains additional edges which are not present in \( \pi_0 \). These come from basic names present in \( B \) but not in \( B_0 \). Thus we can prove \( \vdash \{ B_0 \} : \pi_0 \) applying Lemma A.3 for each of the above basic name not present in \( B_0 \). The other two assumptions of the induction step for \( B_0 \) are clearly satisfied. By the induction hypothesis we obtain \( \Delta \vdash B_0 : \kappa \). Similarly we obtain \( \Delta \vdash B_1 : \kappa \). Thus the claim.
- \( B = N\{ B_0 \} \): We have \( \vdash \{ N\{ B_0 \} \} : \pi \) and thus there is some \( n \) such that \( n = N \) and \( n \neq * \) (for * is not in \( \pi \)). Thus \( \{ n\{ B_0 \} \} = n \{ \} \{ B_0 \} \). Let \( a = n \). There are some \( \alpha \) and \( \chi_0 \) such that \( \vdash n\{ \} : \alpha \) and \( \langle \chi_0 \rightarrow \chi_0 \rangle \in G \), and moreover \( \vdash \{ B_0 \} : (G, \chi_0) \). Thus \( \alpha = a\{ \} \). Let \( \kappa' = \text{typeof}_1(\chi_0) \). Take \( \pi_0 = \{ (\Delta \cup \Delta_B^\nu, \Delta_B^\nu, \kappa') \} \). Now \( \pi \) and \( \pi_0 \) differ only in the root node and \( \pi \) can contain one additional node (its root \( \chi \)). But we can observe that all paths of \( \pi \) which start at \( \chi_0 \) are also present in \( \pi_0 \). Thus \( \vdash \{ B_0 \} : \pi_0 \). The assumptions of the induction step are satisfied because \( \Delta_B^\nu = \Delta_B^\nu \) and \( \Delta_B^\nu = \Delta_B^\nu \). By the induction hypothesis we obtain \( \Delta \vdash B_0 : \kappa' \).

Let us prove \( \Delta \vdash n : \text{Amb}[\kappa'] \). We know that \( \alpha \in G \) and thus \( \alpha \in \text{namesof}_1(\text{Amb}[\text{typeof}_1(\chi_0)]) \). Because \( \alpha \in \text{fbn}(B) \) it has to be the case that \( \Delta(a) = \text{Amb}[\kappa'] \). Hence the claim.

- \( B = N.B_0 \): Take \( N'.B_0' \equiv N.B_0 \) such that \( N' \) is not a composed message, that is, it is either \( n_i \) in \( N_0 \), out \( N_0 \), or open \( N_0 \). We can see that \( \vdash \{ N'.B_0' \} : \pi \).
because ⊢ \{N.B_0\} : \pi. \text{ We have that } \{N'.B'_0\} = \{N'\}_a \{B'_0\} = \{N\}.\{B'_0\}.
Thus there are some \(\alpha\) and \(\chi_0\) such that \(\vdash \{N'\} : \alpha\), and \((\chi_0 \overset{\Delta}{\rightarrow} \chi_0) \in G, and moreover \(\vdash \{B'_0\} : \langle G, \chi_0 \rangle\). It is clear that \(\chi_0 = \chi\) because \(\alpha\) is not of the shape \(b[\ldots]\). Thus \(\vdash \{B'_0\} : \pi\). Take \(\pi_0' = \{(\Delta \cup \Delta_{\chi_0}', \Delta'_{B_0}, \kappa)\}\). Obviously \(\Delta_{\chi_0}' = \Delta_{\chi_0}\) and \(\Delta'_{B_0} = \Delta''_{B_0}\) and thus \(\pi = \pi'_0\). The other two assumptions of the induction step for \(B_0\) are clearly satisfied. By the induction hypothesis we obtain \(\Delta \vdash B'_0 : \kappa\).

Now let us prove \(\Delta \vdash N' : \text{Cap}[\kappa]\). Distinguish the following cases:

\(N' = \text{n}\): Let \(\alpha = \text{n}\). We know \(\vdash \{N'\} : \alpha\) and thus \(\alpha = \text{a}\). Also we know that \(\alpha \in \text{allowedin}_1(\kappa)\). It has to be the case that \(\alpha \in \text{namesof}_1(\text{Cap}[\kappa])\). Because \(\alpha \in \text{fnb}(B)\) it has to be the case that \(\Delta(\alpha) = \text{Cap}[\kappa]\). Thus the claim \(\Delta \vdash \text{n} : \text{Cap}[\kappa]\) holds.

\(N' = \text{in} N_0\): Because \(\vdash \{N'\} : \alpha\) we can see that there has to be some \(n\) (\(n \neq \bullet\)) such that \(n = N_0\). Let \(\alpha = \text{n}\). Thus it has to be \(\alpha = \text{a}\). Also we know that \(\alpha \in \text{allowedin}_1(\kappa)\). It has to be the case that \(\alpha \in \text{moves}_1\). Thus there is some \(\kappa'\) such that \(\alpha \in \text{namesof}_1(\text{Amb}[\kappa'])\). Because \(\alpha \in \text{fnb}(B)\) it has to be the case that \(\Delta(\alpha) = \text{Amb}[\kappa']\). Thus \(\Delta \vdash n : \text{Amb}[\kappa']\) and the claim \(\Delta \vdash n : \text{Cap}[\kappa]\) holds.

\(N' = \text{out} N_0\): As in the case for "in \(N\". \(N' = \text{open} N_0\): As in the case for "in \(N\" but here \(\kappa' = \kappa\) and open \(\alpha \in \text{open}_1(\kappa)\).

otherwise: Other possibilities are not allowed by the choice of \(N'\).

Thus we have \(\Delta \vdash N'.B'_0 : \kappa\). Hence the claim \(\Delta \vdash N.B_0 : \kappa\) because \(N'.B'_0 = N.B_0\).

\(B = \{\forall \nu : \omega\}B_0\): Let \(\alpha = \text{n}\). We see \(\alpha \in \text{fnb}(B)\) and thus \(\Delta'_{\chi_0}(\alpha) = \omega\). That is why \(\omega = \text{Amb}[\kappa']\) for some \(\kappa'\). Let \(\Delta_0 = \Delta[\alpha \mapsto \omega]\). Let \(b \in \text{dom}(\Delta_0) \cap \text{fnb}(B_0)\). When \(b \neq \alpha\) then obviously \(b \in \text{dom}(\Delta) \cap \text{fnb}(B)\) and thus \(\Delta_0(b) = \Delta'_{\chi_0}(b)\). When \(b = \alpha\) we have that \(\Delta_0(b) = \omega = \Delta''_{B_0}(b)\). Now because \(b \in \text{fnb}(B_0)\) we have that \(\Delta''_{B_0}(b) = \omega\) by well-scopedness condition S.4.

We have that \(\vdash \{B_0\} : \pi\). Let \(\pi_0 = \{(\Delta_0 \cup \Delta''_{B_0}, \Delta'_{B_0}, \kappa)\}\). Now we can see that \(\Delta_0 \cup \Delta''_{B_0} = \Delta_0 \cup \Delta''_{B_0}\). (When \(a \in \text{dom}(\Delta)\) then \(\Delta(a) = \Delta''(a) = \omega = \Delta_0(a)\).) Thus \(\pi_0 = \pi\) and \(\vdash \{B_0\} : \pi_0\). Moreover we see that \(\text{fnb}(B_0) = \text{dom}(\Delta''_{B_0})\) is satisfied as well. Thus the assumptions of the induction step for \(\pi_0\), \(\Delta_0\), and \(B_0\) is satisfied. By the induction hypothesis we obtain that \(\Delta_0 \vdash B_0 : \kappa\). Hence the claim because we have already shown that \(\omega = \text{Amb}[\kappa']\) for some \(\kappa'\).

\(B = \langle N_1, \ldots, N_k \rangle\): Let \(A = \langle N_1, \ldots, N_k \rangle\). We see \(\{B\} = A.0\). Now because know that \(\vdash A.0 : \pi\) we have that there are \(\alpha\) and \(\chi_0\) such that \(\vdash A : \alpha\) and \((\chi \overset{\Delta}{\rightarrow} \chi_0) \in G\). Thus there are some \(\mu_1, \ldots, \mu_k\) such that \(\alpha = \langle \mu_1, \ldots, \mu_k \rangle\) and \(\vdash \{N_i\} : \mu_i\). Also clearly \(\alpha \in \text{allowedin}_1(\kappa)\) and \(\alpha \in \text{comms}_1(\kappa)\). It
implies that there are some \( \omega_1, \ldots, \omega_k \) such that \( \kappa = \omega_1 \otimes \cdots \otimes \omega_k \) and \( \mu_i \in \text{msgs}_i(\omega_i) \) for all \( i \).

Let us prove \( \Delta \vdash N_i : \omega_i \) for all \( i \). When \( \mu_i = a \) for some \( a \) (we know \( a \neq \bullet \)) then \( \vdash \{ N_i \} : a \) implies that there is some \( n \) such that \( n = N_i \) and \( a = n \). Now \( \mu_i \in \text{msgs}_i(\omega_i) \) implies \( a \in \text{names}_i(\omega_i) \). We see \( a \in \text{fbn}(B) \) and thus it has to be the case that \( \Delta(a) = \omega_i \). Hence \( \Delta \vdash N_i : \omega_i \). When \( \mu_i = \Phi \bullet \) we know that \( \omega_i = \text{Cap}[\kappa'] \) for some \( \kappa' \) and also we see that \( \Phi = \text{moves}_i \cup \text{open}_i(\kappa') \). Let us prove the claim \( \Delta \vdash N_i : \omega_i \) by the induction of the structure of \( N_i \). Let

\[ N_i = n : \quad \text{We have} \vdash n : \Phi \bullet \text{ and thus } a \in \Phi. \text{ Thus we see } \]

\[ a \in \text{names}_i(\text{Cap}[\kappa']). \text{ Now } a \in \text{fbn}(B) \text{ implies that } \Delta(a) = \text{Cap}[\kappa']. \]

Hence the claim \( \Delta \vdash n : \omega_i \).

\[ N_i = \text{in } N': \quad \text{Because } \vdash \{ N_i \} : \mu_i \text{ and } \mu_i \text{ does not contain } \bullet \text{ we know that there is some } n \text{ such that } n = N'. \text{ Let } a = n. \text{ Thus } \{ M_i \} = \text{in } n \text{ and thus in } a \Phi. \text{ It implies that } a \in \text{moves}_i \text{ and thus there is some } \kappa'' \text{ such that } a \in \text{names}_i(\text{Amb}[\kappa'']). \text{ Because } a \in \text{fbn}(B) \text{ we see that it must be the case } \Delta(a) = \text{Amb}[\kappa'']. \text{ Hence } \Delta \vdash \text{in } n : \text{Cap}[\kappa']. \]

\[ N_i = \text{out } N': \quad \text{As in the case for } \text{“in } N'\text{”}. \]

\[ N_i = \text{open } N': \quad \text{As in the case for } \text{“in } N'\text{” but here } \kappa' = \kappa'' \text{ and open } a \in \text{open}_i(\kappa'). \]

\[ N_i = N', N'': \quad \text{By the induction hypothesis we have } \Delta \vdash N' : \text{Cap}[\kappa'] \text{ and } \Delta \vdash N'' : \text{Cap}[\kappa']. \text{ Hence the claim.} \]

Hence the claim \( \Delta \vdash B : \omega_1 \otimes \cdots \otimes \omega_k \) holds.

\[ B = (n_1 : \omega_1, \ldots, n_k : \omega_k).B_0: \quad \text{We know that it holds } \vdash \{ B \} : \pi \text{ and we have } \{ B \} = (n_1, \ldots, n_k).B_0. \text{ Thus there are some } \alpha \text{ and } \chi_0 \text{ such that } \vdash \{ n_1, \ldots, n_k \} : \alpha, \text{ and } (\chi \overset{\alpha}{\rightarrow} \chi_0) \in G, \text{ and } \vdash \{ B_0 \} : \langle G, \chi_0 \rangle. \text{ We see } \chi_0 = \chi. \text{ Thus } \vdash \{ B_0 \} : \pi. \text{ Take } \Delta_0 = \Delta[n_1 \mapsto \omega_1, \ldots, n_k \mapsto \omega_k]. \text{ Let } \pi_0 = \{ (\Delta_0 \cup \Delta_0^\pi, \Delta_0^\pi, \kappa) \}. \text{ We can see that } \pi_0 \text{ contains all the edges of } \pi \text{ but the edge } (\chi \overset{\alpha}{\rightarrow} \chi_0) \text{ This is because all the names } n_i \text{ from } \Delta_0^\pi \text{ have just moved to } \Delta_0. \text{ Also by well-scopedness condition S1 we know that } n_i \notin \text{dom}(\Delta) \text{ for any } i. \text{ By well-scopedness condition S2 we have that no } n_i \notin \text{fbn}(B_0) \text{ for any } i \text{ and thus the above edge } (\chi \overset{\alpha}{\rightarrow} \chi_0) \text{ is not used when matching } \{ B_0 \} \text{ against } \pi. \text{ Thus also } \vdash \{ B_0 \} : \pi_0. \text{ The assumptions of the induction step are satisfied. By the induction hypothesis we have that } \Delta_0 \vdash B_0 : \kappa \text{ Hence the claim.} \]

\[ \text{B.4 The Proof of Thm. 6.4} \]

In the proof of Thm. 6.4 we refer to some notions defined in the POLY* technical report [MW04, App. B], namely, the notion of pre-principality of the shape predicate \( \pi \) for the process \( P \).

\[ \text{Definition B.9. For } \chi, \alpha, \text{ and the shape predicate } \pi = \langle G, \chi' \rangle \text{ write} \]

\[ \bullet \text{ isanodeunder}_\pi(\chi, *) \text{ when } (\chi' \overset{\alpha}{\rightarrow} \cdots \overset{\alpha_k}{\rightarrow} \chi) \in G \text{ and no } \alpha_i \text{ contains } [] \]
The proof sketch of Thm. 6.4 follows.

Theorem B.10. Let B be a BA process, \((S, N)\) the result of SABA analysis of B, and let \(\pi\) be the POLY principal typing of \([B]\). Then \(S_\pi \subseteq S\) and \(N_\pi \subseteq N\).

Proof. Suppose we have the following derivation of the principal typing of \([B]\), that is, that we have the sequence of shape predicates \(\pi_0, \ldots, \pi_k\) such that

- every \(\pi_i\) is pre-principal for \([B]\),
- \(\pi_0\) corresponds directly to the syntax tree of \([B]\),
- \(\pi_{i+1}\) is the same as \(\pi_i\) except one edge (either a flow or normal edge) and that \(\pi_{i+1}\) has at most one more node than \(\pi_i\), and
- \(\pi_k = \pi\).

Then by induction on the derivation of the principal typing proof that for every \(\pi_i = (G_i, \chi_i)\) all the following hold:

1. \(\text{inamb}_{\pi_i}(a, \alpha)\) implies \((a, \alpha) \in S\),
2. \((\chi_0 \quad \text{a} \quad \text{b} \quad \text{c} \quad \text{d}) \in G_i\) implies \((a, b) \in N\), and
3. for all \(a', b'\) such that \((\chi_0 \quad \text{a} \quad \text{b} \quad \text{c} \quad \text{d}) \in G_i\) and \(\text{isanodeunder}_{\pi_i}(\chi_0, a')\) and \(\text{isanodeunder}_{\pi_i}(\chi_1, b')\) it holds that \((\forall \alpha : (a', \alpha) \in S \Rightarrow (b', \alpha \tau) \in S)\) where \(\tau = \{a \mapsto b\}\).

Properties 1 and 2 when applied to \(\pi\) prove the claim.

B.5 The Proof of Thm. 6.8

The left-to-right implication is Prop. B.11 and the right-to-left implication is Prop. B.12. The assumption \(\text{fbn}(B) \cup \text{nbn}(B) \subseteq \text{dom}(N) \cup Z\) is easily satisfied because \(\text{fbn}(B) \cup \text{nbn}(B) \subseteq \text{dom}(N)\). Also \(\ast \in \text{labels}_S\) by the definition.

Proposition B.11. Let

1. \(\text{ibn}(B) \subseteq Z\)
2. \(\text{fbn}(B) \cup \text{nbn}(B) \subseteq \text{dom}(\tilde{N}) \cup Z\)
3. \(l \in \text{labels}_S\)

Then \((S, N) \models^1 B\) implies \(\vdash [B] : (G_{(S, N, Z)}, \text{nodeof}_S(\ast))\).

Proof. Let \(G = G_{(S, N, Z)}\) and \(\chi = \text{nodeof}_S(\ast)\), and \(\pi = (G, \chi)\). Let us prove the claim \(\vdash [B] : \pi\) by induction on the structure of B. Let

\(B = 0\): Clear.

\(B = B_0 \mid B_1\): From \((S, N) \models^1 B\) it follows that \((S, N) \models^1 B_0\) and \((S, N) \models^1 B_1\). The assumptions of the induction step are clearly satisfied for both \(B_0\) and \(B_1\). Thus by the induction hypothesis we have that \(\vdash [B_0] : \pi\) and \(\vdash [B_1] : \pi\). Thus the claim.
\[ B = [B_0]^\text{t}_0 : \text{From } (S, \mathcal{N}) \models_1 B \text{ it follows that } S(l, l_0 [\emptyset]) \text{ and } (S, \mathcal{N}) \models_1 B_0. \]

Thus we see that the assumptions of the induction step for \( B_0 \) and \( l_0 \) are clearly satisfied. Let \( \chi_0 = \text{nodeof}_S(l_0) \). Thus by the induction hypothesis we have that \( \vdash \{ B \} : \langle G, \chi_0 \rangle \). From the construction of \( G(S, \mathcal{N}, Z) \) it follows that \( (\chi \mid \chi_0, \chi_0) \in G \) and thus the claim because \( \{ B \} = l_0 \{ \{ B_0 \} \} \).

\[ B = \text{N}.B_0 : \text{Suppose, for example, } N = \text{enter } n. \text{ The proof for other capabilities (communication actions are handled separately) is analogous. Let } a = \text{N}. \text{ Furthermore let } A = \text{enter } n \text{ and } \alpha = \text{enter } a. \text{ We see that } \{ B \} = A \{ B_0 \} \text{ and } \vdash A : \alpha. \text{ Now } (S, \mathcal{N}) \models_1 B \text{ implies } (S, \mathcal{N}) \models_1 B_0. \text{ The assumptions of the induction step for } B_0 \text{ are clearly satisfied. Thus by the induction hypothesis we have } \vdash \{ B_0 \} : \pi. \]

To prove the claim it is enough to prove that \( \alpha \in \text{allowedin}_{(S, \mathcal{N}, Z)}(l) \). From (2) we know that either \( a \in \text{dom}(\mathcal{N}) \) or \( a \in Z \). Suppose \( a \in \text{dom}(\mathcal{N}) : \text{Then we have some } a' \text{ such that } \mathcal{N}(a, a'). \text{ It follows from } (S, \mathcal{N}) \models_1 \text{ enter } n \text{ that } S(l, \text{enter } a'). \text{ But now it is easy to see that } \alpha = \text{enter } a' \} \{ a' \mapsto a \} \in \text{allowedin}_{(S, \mathcal{N}, Z)}(l) \). Thus the claim.

\[ a \in Z \backslash \text{dom}(\mathcal{N}) : \text{Then we have that } \alpha \in \text{inertcaps}_{(S, \mathcal{N}, Z)}. \text{ Thus the claim.} \]

\[ B = \lnot B_0 : \text{Simply apply the induction hypothesis.} \]

\[ B = (\forall n)B_0 : \text{Let } a = \text{N}. \text{ Now } (S, \mathcal{N}) \models_1 B \text{ implies } (S, \mathcal{N}) \models_1 B_0. \text{ The assumption of the induction step for } B_0 \text{ are clearly satisfied. Thus by the induction hypothesis we have } \vdash \{ B_0 \} : \pi \text{ and thus the claim because } \{ B \} = \forall (n) \{ B_0 \}. \]

\[ B = \text{d n?}\{ m \}.B_0 : \text{Let } a = \text{N} \text{ and } b = m. \text{ Furthermore let } A = \text{d n}(m) \text{ and } \alpha = \text{d a}(b). \text{ We see that } \{ B \} = A \{ B_0 \} \text{ and } \vdash A : \alpha. \text{ From } (S, \mathcal{N}) \models_1 B \text{ we have that } (S, \mathcal{N}) \models_1 B_0. \text{ The assumptions of the induction step for } B_0 \text{ are satisfied (number (3)) because } b \in Z \text{ follows from } b \in \text{ibn}(B) \text{ and (1)). Thus by the induction hypothesis we have that } \vdash B_0 : \pi. \]

To prove the claim it is enough to prove that \( \alpha \in \text{allowedin}_{(S, \mathcal{N}, Z)}(l) \). From (2) we know that either \( a \in \text{dom}(\mathcal{N}) \) or \( a \in Z \). Suppose \( a \in \text{dom}(\mathcal{N}) : \text{Then we have some } a' \text{ such that } \mathcal{N}(a, a'). \text{ It follows from } (S, \mathcal{N}) \models_1 \text{ d n?}\{ m \} \text{ that } S(l, \text{d a'}(b)). \text{ But now it is easy to see that } \alpha = \text{d a'}(b) \} \{ a' \mapsto a \} \in \text{allowedin}_{(S, \mathcal{N}, Z)}(l) \). Thus the claim.

\[ a \in Z \backslash \text{dom}(\mathcal{N}) : \text{Then we have that } \alpha \in \text{inertcaps}_{(S, \mathcal{N}, Z)}. \text{ Thus the claim.} \]

\[ B = \text{d n!}\{ m \}.B_0 : \text{Let } a = \text{N} \text{ and } b = m. \text{ Furthermore let } A = \text{d n}<m> \text{ and } \alpha = \text{d a}\triangleleft b>. \text{ We see that } \{ B \} = A \{ B_0 \} \text{ and } \vdash A : \alpha. \text{ From } (S, \mathcal{N}) \models_1 B \text{ we have that } (S, \mathcal{N}) \models_1 B_0. \text{ The assumptions of the induction step for } B_0 \text{ are satisfied. Thus by the induction hypothesis we have that } \vdash B_0 : \pi. \]

To prove the claim it is enough to prove that \( \alpha \in \text{allowedin}_{(S, \mathcal{N}, Z)}(l) \). From (2) we know that either \( a \in \text{dom}(\mathcal{N}) \) or \( a \in Z \) and the same for \( b \). Suppose \( a \in \text{dom}(\mathcal{N}) \text{ and } b \in \text{dom}(\mathcal{N}) : \text{Then we have } a' \text{ and } b' \text{ such that } \mathcal{N}(a, a') \text{ and } \mathcal{N}(b, b'). \text{ It follows from } (S, \mathcal{N}) \models_1 \text{ d n?}\{ m \} \text{ that } S(l, \text{d a'}<b'>). \text{ But now it is easy to see that } \alpha \in \text{allowedin}_{(S, \mathcal{N}, Z)}(l) \). Thus the claim.
\[ a \in Z \setminus \text{dom}(N) \lor b \in Z \setminus \text{dom}(N) : \text{Then we have that } \alpha \in \text{inertcaps}_{(N,Z)} \backslash \text{activecaps}_{(N,Z)} \]

Thus the claim.

**Proposition B.12.** Let

1. \( S \) be closed w.r.t. \( N \)
2. \( \text{ibn}(B) \subseteq Z \)
3. \( \text{fbn}(B) \cup \text{nbn}(B) \subseteq \text{dom}(N) \cup Z \)
4. \( \forall a \in \text{nbn}(B) : N(a,a) \)
5. \( l \in \text{labels} \)

Then \( \vdash \{ B \} : G(S,N,Z), \text{nodeof}_S(l) \) implies \( (S,N) \models^1 B \).

**Proof.** Let \( G = G(S,N,Z), \) and \( \chi = \text{nodeof}_S(l), \) and \( \pi = \langle G, \chi \rangle. \) Let us prove the claim \( (S,N) \models^1 B \) by induction on the structure of \( B. \) Let \( B = 0: \) Clear.\n
\[ B = B_0 \mid B_1 : \text{It is easy to see that } \vdash \{ B_0 \} : \pi \text{ and } \vdash \{ B_1 \} : \pi. \text{ The assumptions of the induction step are clearly satisfies. Thus the claim follows directly from the induction hypothesis.} \]

\[ B = \{ B_0 \}_{l_0} : \text{We know that } \{ \{ B_0 \}_{l_0} \} = l_0 \{ \{ B_0 \} \} \text{ and } \vdash \{ B_0 \} : \langle G, \chi \rangle. \text{ Thus } S(l, l_0 \alpha) \text{ and } l_0 \in \text{labels}_S. \text{ Let } \chi_0 = \text{nodeof}_S(l_0). \text{ Thus we have } \vdash \{ B_0 \} : \langle G, \chi_0 \rangle. \text{ The assumptions of the induction step for } B_0 \text{ and } l_0 \text{ are clearly satisfied. Thus by the induction hypothesis we have that } (S,N) \models^1 B_0 \text{ which together with } S(l, l_0 \alpha) \text{ proves the claim.} \]

\[ B = \text{N.B}_0 : \text{Suppose, for example, } \text{N} = \text{enter } n. \text{ The proof for other capabilities (communication actions are handled separately) is analogous. Let } a = n. \text{ We know } \vdash \text{enter } n. \{ B_0 \} : \pi \text{ and thus there is some } \alpha \text{ such that } \vdash \text{enter } n : \alpha. \text{ From the construction of } G(S,N,Z), \text{ it follows that } \alpha = \text{enter } a, \text{ and } \alpha \in \text{allowedin}_{(S,N,Z)}(l), \text{ and also } \vdash \{ B_0 \} : \pi. \text{ The assumptions of the induction step for } B_0 \text{ are clearly satisfied. Thus by the induction hypothesis we have that } (S,N) \models^1 B_0. \]

To prove the claim it is enough to prove the goal \( (S,N) \models^1 \text{enter } a, \) that is, \( \forall a': N(a,a') \Rightarrow S(1, \text{enter } a'). \) The goal holds trivially when there is no \( a' \) such that \( N(a,a'). \) So, let \( a' \) be such that \( N(a,a'). \) This means that \( a \in \text{dom}(N) \) and thus \( \alpha \notin \text{inertcaps}_{(N,Z)} \). Hence we know that \( \alpha \in \text{activecaps}_{(N,Z)}(l). \) Thus there are some \( \alpha_0, a_0, \) and \( a_0' \) such that \( S(l, \alpha_0), \) and \( N(a_0,a_0') \), and \( a_0' \in \text{fbn}(a_0), \) and also \( \alpha_0(a_0' \mapsto a_0) = \alpha = \text{enter } a. \)

It is clear that \( \alpha_0 = \text{enter } a_0' \) and \( a_0 = a. \) We have \( S(l, \text{enter } a_0') \) and \( N(a, a_0'). \) Thus the goal follows from assumption (1) by point (1) of Def. 6.6.

\[ B = \text{N.B}_0 : \text{Simply apply the induction hypothesis.} \]

\[ B = (\forall n)B_0 : \text{Let } a = n. \text{ We have } \{ B \} = (\forall n)\{ B_0 \}. \text{ Thus it is clear that it holds } \vdash \{ B_0 \} : \pi. \text{ The assumptions of the induction step for } B_0 \text{ are clearly satisfied. Thus by the induction hypothesis we have that } (S,N) \models^1 B_0. \text{ To proof the claim it is enough to prove that } N(a,a) \text{ which holds by assumption (4).} \]
\[ B = d\] and \[ B = m. \]

We know \( \vdash d \{m\} : \pi \) and thus there is some \( \alpha \) such that \( \vdash d \{m\} : \alpha \). From the construction of \( G_{(S, N, Z)} \), it follows that \( \alpha = d a(b) \), and \( \alpha \in \text{allowedin}_{(S, N, Z)}(l) \), and also \( \vdash \{B_0\} : \pi \).

The assumptions of the induction step for \( B_0 \) are clearly satisfied. Thus by the induction hypothesis we have that \( (S, N) \models B_0 \).

To prove the claim it is enough to prove the goal \( (S, N) \models^1 d a(b) \), that is, \( \forall a' : N(a, a') \Rightarrow S(l, d a'(b)) \). The goal holds trivially when there is no \( a' \) such that \( N(a, a') \). So, let \( a' \) be such that \( N(a, a') \). This means that \( a \in \text{dom}(N) \) and thus \( a \not\in \text{inertcaps}_{(N, Z)} \).

Hence we know that \( \alpha \in \text{activecaps}_{(S, N)}(l) \). Thus there are some \( \alpha_0, a_0, a_0' \) such that \( S(l, \alpha, \alpha_0) \), and \( N(a_0, a_0') \), and \( a_0' \in \text{fbn}(\alpha_0) \), and also \( \alpha_0 \{a_0' \leftarrow a_0\} = \alpha = d a(b) \).

It is clear that \( \alpha_0 = d a_0'(b) \) and \( a_0 = a \). We have \( S(l, d a_0'(b)) \) and \( N(a, a_0) \). Thus the goal follows from assumption (1) by point (7) of Def. 6.6.

\[ B = d\] and \[ B = m. \]

We know \( \vdash d \{m\} : \pi \) and thus there is some \( \alpha \) such that \( \vdash d \{m\} : \alpha \). From the construction of \( G_{(S, N, Z)} \), it follows that \( \alpha = d a(b) \), and \( \alpha \in \text{allowedin}_{(S, N, Z)}(l) \), and also \( \vdash \{B_0\} : \pi \).

The assumptions of the induction step for \( B_0 \) are clearly satisfied. Thus by the induction hypothesis we have that \( (S, N) \models B_0 \).

To prove the claim it is enough to prove the goal \( (S, N) \models^1 d a(b) \), that is, \( \forall a', b' : N(a, a') \& N(b, b') \Rightarrow S(l, d a'(b')) \). The goal holds trivially when there is no \( a' \) such that \( N(a, a') \) or there is no \( b' \) such that \( N(b, b') \).

So, let \( a' \) and \( b' \) be such that \( N(a, a') \) and \( N(b, b') \). This means that \( a \in \text{dom}(N) \) and \( b \in \text{dom}(N) \), and thus \( a \not\in \text{inertcaps}_{(N, Z)} \).

Hence we know that \( \alpha \in \text{activecaps}_{(S, N)}(l) \). Thus there are some \( a_0' \) and \( b_0' \) such that \( N(a, a_0') \) and \( N(b, b_0') \) and \( S(l, d a_0' \leftarrow b_0') \). Thus the goal follows from assumption (1) by point (8) of Def. 6.6.