Technical Report HW-MACS-TR-0022
School of Mathematical and Computer Sciences
Heriot Watt-University

Instant Polymorphic Type Systems for Mobile Process Calculi: Just Add Reduction Rules and Close*

Henning Makholm
Heriot-Watt University

J. B. Wells
Heriot-Watt University

November 5, 2004

Abstract

Many different *mobile process calculi* have been invented, and for each some number of type systems has been developed. Soundness and other properties must be proved separately for each calculus and type system.

We present the *generic* polymorphic type system Poly✶ which works for a wide range of mobile process calculi. For any calculus satisfying some general syntactic conditions, well-formedness rules for types are derived automatically from the reduction rules and Poly✶ works otherwise unchanged. The derived type system is automatically sound and often more precise than previous type systems for the calculus, due to Poly✶'s *spatial polymorphism*.

We present an implemented type inference algorithm for Poly✶ which automatically constructs a typing given a set of reduction rules and a term to be typed. The generated typings are *principal* with respect to certain natural type shape constraints.

1 Introduction

In the last decade, many calculi that intend to capture the essence of *mobile* and *distributed* computing have been invented. The most well-known of these is probably the *Mobile Ambients* calculus (MA) by Cardelli and Gordon [11], but it has inspired the subsequent development of a wide variety of variants and alternatives, which are variously argued to be easier to program in or reason about, and/or closer to some operational intuition about how programs in a mobile, distributed setting can be implemented. The field stays productive;

*Supported by EC FP5/IST/FET grant IST-2001-33477 "DART", and Sun Microsystems equipment grant EDUD-7826-990410-US.

new calculi are still being proposed and there is not a clear consensus about what should be considered *the* fundamental mobility calculus.

The vast majority of these calculi share the basic architecture of MA: They borrow from π-calculus [22] the syntactic machinery for talking about sets of parallel, communicating processes, plus its primitive ν operator for generating unique names. To this they add some kind of *spatial structure*, usually in the form of a tree of locations where processes can reside. The tree can generally evolve under program control as the processes in it executes; the different calculi provide quite different primitives for mutating it. Mobility calculi also provide for *communication* between processes that are near each other, usually modelled on the communication primitive of the π-calculus, but again with variations and often extended with the possibility to communicate "capabilities", "paths", or other restricted pieces of process syntax, rather than just names.

Most process calculi have an associated *type discipline*, either one that was designed with the calculus from the beginning, or one that was retrofitted later. These type systems are closely tied to a specific calculus and its particular primitives. Once a type system has been designed and its properties (such as soundness or the applicability of a particular type inference algorithm) have been proved, it is in general not trivial to see whether these properties will survive changes to the calculus.

1.1 A generic type system

In contrast, this paper presents the *generic* type system Poly✶ which works for a wide range of mobile process calculi. To use Poly✶, one must instantiate it with the reduction rules that specify the semantics of the target calculus's primitives. From this, a set of provably sound well-formedness rules for types can be *mechanically* produced, guaranteeing that types that satisfy the rules are sound with respect to the reduction rules.

The reduction rules can also be used to guide an automatic *type inference* algorithm for the instantiated type system. The inference algorithm produces a type which is *principal* with respect to certain natural constraints on the shape of types. Our implementation offers several possibilities for tuning the *precision* of the type system it implements, but the use of these is optional — it will always produce a typing even when given only the raw reduction rules of of the target calculus.

For this to work, the target calculus must make one small concession to Poly✶, namely that its *syntax* is sufficiently regular that the implementation can make sense of its terms and reduction rules. We define a **metacalculus** Meta✶ which basically consists of a semantics-less syntax that is both easy to parse and manipulate, and flexible enough that many published calculi can be viewed as restrictions of it without deviating much from their native notations. Meta✶ includes the usual (and non-controversial) semantics of parallelism and name restriction, and also provides a common notion of substitution and a notation for rewriting rules that fits how semantics for process calculi are usually defined.

1.2 Poly✶ and other reasoning principles

A long-term goal of Poly✶ is to make it possible to view many previously existing mobility calculi type systems as restricted instances of Poly✶, at least from the point of view of using the type system to statically verify that certain bad behaviours do not occur. The design we present here does not quite reach that point; there are some features of existing type systems that we have not yet incorporated in Poly✶. We believe it will be particularly important to express something like the *single-threaded* locations introduced by the original type system for Safe Ambients [20]. We hope, though, that we can convince the reader that the possibility of a universal mobility type system is not as insane as it may seem at first sight.

We do not expect actual programming environments based on mobility calculi to use the fully general Poly✶ formalism as their type discipline. Considerations of performance and integration will generally dictate that production environments instead use hand-crafted specialised type systems for the language they support, though *ideas* from Poly✶ may well be employed.

A generic implementation of Poly✶, such as the one we present here, should be a valuable tool for *exploring the design space* for mobility calculi in general. It will make it easy to change some aspect of one's rewriting rules, try to analyse some terms, and see which effect the new rules have on, say, the interference-control properties of one's calculus. At the same time, our Poly✶ implementation makes it easy to experiment with exactly how strong a type system one wants to use in practice, because our implementation supports tuning the precision of types in very small steps.[1]

Like every nontrivial type system with an inference algorithm, Poly✶ can be used as a *control/data flow analysis* to provide the substratum for more specialised automatic *program analyses*.[2] However, we have no pretension of subsuming all other analysis techniques for mobility or process calculi in general. Process calculi have provided the setting for many advanced techniques for reasoning about, say, behavioural equivalence of processes. Poly✶ does not claim to compete with these.

1.3 Spatial polymorphism

The Poly✶ type system descends from (but significantly generalises and enhances) our earlier work [2] on PolyA, a polymorphic type system specific to Mobile Ambients. It inherits from PolyA the difference from most other type systems for mobility calculi that the emphasis is on types for *processes* rather

[1] These fine tuning options are omitted from this paper due to lack of space, but they are described in the implementation's documentation.

[2] Indeed it is well known [24, 4] that the difference between an advanced flow analysis and an advanced type system is often just a question of different perspectives on the same underlying machinery. The presentation of Poly✶ is closer to the data-flow viewpoint than is common for type system, though this of course does not make Poly✶ any less a type system.

than types for (ambient or channel) *names*.[3] In fact, types for names are so deemphasised that they have completely vanished: A name has no intrinsic type of its own, but is distinguished solely by the way it can be used to form processes.

Poly✶ works by approximating the set of terms a given term can possibly evolve to using the given reduction rules. Its central concept is that of a **shape predicate** which is an automaton that describes a set of process terms. Shape predicates that satisfy certain well-formedness rules are called **types**. These rules are derived from the reduction rules of the target calculus and guarantee that the set of terms denoted by a type is closed under the reduction relation, i.e., *subject reduction* holds.

This design gives rise to a new (with PolyA) form of polymorphism that we call **spatial polymorphism**. The type of a process may depend on where in the spatial structure it is found. When the process moves, it may come under influence of another part of the type which allows more reductions. For example, consider a calculus which has the single reduction rule $\mathtt{a[eat\ b \mid P] \mid b[Q]} \hookrightarrow \mathtt{a[P \mid b[Q]]}$. In this calculus, the term $\mathtt{x[eat\ z1 \mid eat\ z2] \mid y1[eat\ x \mid z1[0]] \mid y2[eat\ x \mid z2[0]]}$ has a Poly✶ type (shown on Figure 6) that says that $\mathtt{x[]}$ may contain $\mathtt{z1[]}$ when it it inside $\mathtt{y1[]}$, or $\mathtt{z2[]}$ when it it inside $\mathtt{y2[]}$, but can contain neither when it is found at the top level of the term. Thus Poly✶ can prove that the term satisfies the safety policy that z1 and z2 may never be found side by side. To our knowledge, type systems based on earlier paradigms cannot do this.

With spatial polymorphism, *movement* is what triggers the generation of a polymorphic variant of the original analysis for a piece of code. This is different from, and orthogonal to, the more conventional form of name-parametric polymorphism in the polymorphic π-calculus [28] (or its intersection-based generalisation in [21]), where it is *communication* that causes polymorphic variants to be created. Poly✶ does not support the latter form of polymorphism (and neither does any type system for a mobility calculus with locations that we are aware of); we leave it to future work to try to combine the strengths of these two principles.

1.4 Notation and preliminaries

$\boxed{\mathtt{X}}$, where X is any metavariable symbol, stands for the set that X ranges over.

In grammar rules and other contexts where a sequence of similar objects are indexed with indexes up to k, it is to be understood that k can be any integer ≥ 0. Thus, if the first index is 0, the sequence must have at least one element; on the other hand sequences indexed from 1 to k may be empty.

[3] There are a number of type systems for process calculi *without* an explicit notion of mobility which assign types to processes rather than names, for example [5, 18, 31, 14].

$$
\begin{array}{rl}
\text{Names:} & x, y ::= a \mid b \mid c \mid \ldots \mid [\,] \mid \hat{} \mid ? \mid : \mid * \mid / \mid \ldots \mid \bullet \\
\text{Sub-forms:} & f ::= x_0\ x_1 \ldots x_k \\
\text{Messages:} & M, N ::= f \mid 0 \mid M.N \\
\text{Elements:} & E ::= x \mid (x_1, x_1, \ldots, x_k) \mid \langle M_1, M_2, \ldots, M_k \rangle \\
\text{Forms:} & F ::= E_0\ E_1 \ldots E_k \\
\text{Processes:} & P, Q, R ::= F.P \mid\ !P \mid \nu(x).P \mid 0 \mid (P \mid Q)
\end{array}
$$

Free and bound names in terms are defined thus (the omitted cases being purely structural):

$$
\begin{array}{rlrl}
FN(x) &= \{x\} & BN(x) &= \varnothing \\
FN((\cdots)) &= \varnothing & BN((x_1, \ldots, x_k)) &= \{x_1, \ldots, x_k\} \\
FN(F.P) &= FN(F) \cup (FN(P) \setminus BN(F)) & BN(F.P) &= BN(F) \cup BN(P) \\
FN(\nu(x).P) &= FN(P) \setminus \{x\} & BN(\nu(x).P) &= BN(P)
\end{array}
$$

$$
\frac{}{P \equiv P} \qquad \frac{P \equiv Q}{Q \equiv P} \qquad \frac{P \equiv Q \qquad Q \equiv R}{P \equiv R} \qquad \frac{}{P \mid Q \equiv Q \mid P}
$$

$$
\frac{}{P \mid (Q \mid R) \equiv (P \mid Q) \mid R} \qquad \frac{}{P \mid 0 \equiv P} \qquad \frac{}{!P \equiv P \mid !P}
$$

$$
\frac{}{!0 \equiv 0} \qquad \frac{P \equiv Q}{F.P \equiv F.Q} \qquad \frac{P \equiv Q}{!P \equiv !Q} \qquad \frac{P \equiv Q}{\nu(x).P \equiv \nu(x).Q}
$$

$$
\frac{P \equiv Q}{P \mid R \equiv Q \mid R} \qquad \frac{x \notin FN(F) \qquad x \notin BN(F)}{F.\nu(x).P \equiv \nu(x).F.P} \qquad \frac{x \notin FN(P)}{P \mid \nu(x).Q \equiv \nu(x).(P \mid Q)}
$$

$$
\frac{y \notin FN(P)}{\nu(x).P \equiv \nu(y).[x := y]P} \qquad \frac{}{\nu(x).\nu(y).P \equiv \nu(y).\nu(x).P}
$$

Figure 1: Syntax of Meta✶ plus its structural congruence relation

2 Meta✶: A metacalculus of concurrent processes

The metacalculus Meta✶ defined in this section is the *syntactic* setting for Poly✶. As a first approximation, Meta✶ is a "syntax without a semantics" except that it does fix the semantics of a few very basic constructs such as process replication and substitution.

2.1 Terms

Figure 1 shows the syntax of process terms in Meta✶. The trivial process 0, parallel composition of processes $P \mid Q$, process replication $!P$, and name restriction $\nu(x).P$ are all well-known from most process calculi, including π-calculus and MA. They are given their usual behaviour by the structural congruence relation

$\equiv$ also defined in Figure 1. [4]

Meta$\star$ amalgamates all other process constructors into the general concept of a **form**. Forms have no intrinsic meaning until a set of reduction rules give them one. Examples of forms include the communication actions "x<y>" and "x(y)" from the π-calculus, the movement capabilities "in x", "out x", and "open x" from Mobile Ambients, and even ambient boundaries themselves, which we write as "x []". We support the traditional syntax "x[P]" for ambients by interpreting "$E_1 \ldots E_k[P]E'_1 \ldots E'_k$," as syntactic sugar for "$E_1 \ldots E_k[]E'_1 \ldots E'_k$.P". Except for this syntactic convention, the symbol [] has no special interpretation in Meta$\star$ and it is a (single) name just like in and out. The process F.0 can be abbreviated as F.

A form consists of a nonempty sequence of **elements**, each of which is either a **name**, a **binding** element, or a **message** element. Names are used to name channels, ambients, and so on, but also work as keywords that distinguish forms with different roles in the calculus. A keyword is simply a free name that is matched explicitly by some reduction rule.

Most non-alphanumeric ASCII characters that do not have any special function in the syntax can also be used as names. This allows us to encode, for example, annotated communication actions like "$\langle M \rangle^*$" or "$(x)^y$" from Boxed Ambients [7] using pseudo-TEX notation as the forms "<M> ^ *" and "(x) ^ y".

Binding elements $(\cdots)$ are used to create forms that bind names in the process they are applied to. The canonical use of this is for constructing receive actions, but again the meaning of the form is specified only by the reduction rules.

Message elements $\langle \cdots \rangle$ allow a form to contain other forms, which — given appropriate reduction rules for communication — can later be substituted into processes. For technical reasons we have to restrict the forms contained in message elements in that they cannot contain message or binding elements themselves. We refer to such restricted forms and their elements as **sub-forms** and **sub-elements**. In future work we hope to be able to permit sub-forms to contain message elements, which will allow us to handle calculi such as the spi-calculus [1] which communicate structured messages. We do not currently anticipate allowing binding elements in sub-forms in the future, because this runs into more fundamental problems related to the risk of variable capture when a message is spliced into a process tree.

A message element consists of **messages** each of which is essentially a list of sub-forms. However, we do not formally treat the M.N operator as associative, nor do we consider the empty message 0 a neutral element. This allows us to distinguish between a message that consists of a single form and one that consists of a list of forms which happens to have length one: The latter contains one or more 0's. These cases behave differently when they are substituted into contexts like a in the form "a b". This allows us to faithfully implement the

[4] Traditionally the structural congruence relation also includes the axiom $\nu(x).0 \equiv 0$. We omit it here because it is technically troublesome (it takes extra special cases to avoid breaking Proposition 3.3) and only really needed for reasoning about behavioural identity, which is orthogonal to the purpose of Poly$\star$.

semantics of various ambient calculi as defined in the literature.

It is not uncommon for calculi to prefer using an explicit recursion construction "$P ::= \mathbf{rec}\ X.P$" to express infinite behaviour rather than the process replication operator "!". There are certain technical problems with supporting this directly in Meta✶ (which may however be approachable by novel techniques involving regular grammars developed by Nielson et al. [23]). In the common case where the target calculus does not allow location boundaries to come between the **rec** X binder and the bound X, it can easily be simulated in Meta✶ by adding the reduction rule $\mathsf{spawn}\ a \mid \mathsf{rec}\ a.P \hookrightarrow P$ and then representing $\mathbf{rec}\ X.\cdots.X$ as $\nu(x).(\mathsf{spawn}\ x \mid\ !\mathsf{rec}\ x.\cdots.\mathsf{spawn}\ x)$.

2.2 Well-scoped terms

Definition 2.1. The process term P is **well scoped** iff it contains no nested binding of the same name and if none of its free names also appear bound somewhere in the term. Formally, it is required that

1. $BN(P)$ and $FN(P)$ are disjoint,
2. Whenever P contains $F.Q$, $BN(F)$ and $BN(Q)$ are disjoint, and
3. Whenever P contains $\nu(x).Q$, $x \notin BN(Q)$. □

We generally require that terms are always well scoped. The reduction rules in an instantiation of Meta✶ must preserve well-scopedness. This significantly simplifies the type analysis because it means that we do not have to support α-conversion of ordinary binding elements.

We must still handle α-conversion of private names, which is built into the $\equiv$ relation, but we will assume that it is not used to create terms that are not well scoped.

2.3 Substitutions

Substitutions in Meta✶ substitute *messages* for *names*. The fact that entire *processes* cannot be substituted is an important technical premise of Poly✶; it means that substitution can preserve well-scopedness. (For comparison, this is not true in the λ-calculus; Poly✶ is intrinsically unsuited for typing λ-calculus terms, for this reason and others.[5]) It is remarkable that mobility calculi in general refrain from substituting processes; calculi such as Seal [29] and $\mathbf{M^3}$ [15] which allow exchange of entire processes do it by local *movement* rather than *substitution*. This probably reflects the intuition that a running process is harder to distribute across a recipient process than a mere name or code sequence.

In Mobile Ambients and its descendant calculi, the value exchanged in a communication operation can be either a name or a (sequence of) capabilities. The former is the case in reduction $\langle b \rangle \mid (a).\mathsf{out}\ a.0 \hookrightarrow \mathsf{out}\ b.0$ and the latter

[5] Boudol [5] gives a simple translation of the call-by-name λ-calculus into a process calculus that can be represented by Meta✶. But we have not investigated how well Poly✶ performs in that setting.

$$\mathcal{S}^{\mathbf{E}}\,x = \begin{cases} x & \text{when } x \notin \mathrm{Dom}\,\mathcal{S} \\ y & \text{when } \mathcal{S}(x) = y \text{ for some } y \\ \bullet & \text{otherwise} \end{cases} \qquad \mathcal{S}^{\mathbf{F}}\,E_0 = \begin{cases} \mathcal{S}(x) & \text{if } E_0 = x \in \mathrm{Dom}\,\mathcal{S} \\ \mathcal{S}^{\mathbf{E}}\,E_0 & \text{otherwise} \end{cases}$$

$$\mathcal{S}^{\mathbf{P}}\,(\nu(x).P) = \nu(x).\mathcal{S}^{\mathbf{P}}\,P \qquad \mathcal{S}^{\mathbf{P}}\,(F.P) = \begin{cases} M_*\mathcal{S}^{\mathbf{P}}\,P & \text{when } \mathcal{S}^{\mathbf{F}}\,F = M \\ F'.\mathcal{S}^{\mathbf{P}}\,P & \text{when } \mathcal{S}^{\mathbf{F}}\,F = F' \end{cases}$$

$$(M.N)_*P = M_*(N_*P) \qquad 0_*P = P \qquad f_*P = f.P$$

Figure 2: The actions of term substitution. The omitted cases simply substitute pointwise into the syntactic element in question. The M_*P helper operator serves to linearise messages once we do not need to keep track of whether they are composite or not. (In other systems, this is often done by the structural congruence relation instead.)

in $\langle \mathsf{in}\,b.\mathsf{in}\,c\rangle \mid (a).x[a.0] \hookrightarrow x[\mathsf{in}\,b.\mathsf{in}\,c.0]$. If the recipient expects a different kind of value than the sender sends, we may instead end up with $\langle \mathsf{in}\,b.\mathsf{in}\,c\rangle \mid (a).\mathsf{out}\,a.0 \hookrightarrow \mathsf{out}\,(\mathsf{in}\,b.\mathsf{in}\,c).0$.

The published formalisms of most ambient-inspired calculi usually regard "out (in b.in c)" as *syntactically* possible but *semantically* meaningless. That this configuration cannot occur is often the most basic soundness property of type systems for such calculi. [6]

In Meta✶ this semantic error becomes a syntactic one: It is simply not possible to use an entire compound message as an element (except for a message element). If, at runtime, a substitution nevertheless tries to do so, we substitute the special name "•", which is to be interpreted as, "an erroneous substitution happened here". Thus, with the MA communication rule, Meta✶ reduces <in b.in c>.0 | (a).out a.0 $\hookrightarrow$ out •.0. This convention is technically convenient because it allows us to bound the nesting depth of forms (using the sub-form restriction). Because most published calculi attach no semantics to forms like "out (in b.in c)", we do not lose any real expressiveness.

A • is only produced when the name to be substituted is not the single element of a form. Thus substituting $[a \mapsto \mathsf{in}\,b.\mathsf{in}\,c]$ in $x[a.0]$ still produces $x[\mathsf{in}\,b.\mathsf{in}\,c.0]$, as it should.

Apart from generating it from certain substitutions, Meta✶ does not attach any special semantics to •. The calculus designer can freely define reduction rules that create •'s in other situations that he wants to consider erroneous. For example, in the polyadic π-calculus, it is usually considered to be a run-time error if someone tries to send an m-tuple on a channel where another process is listening for an n-tuple, with $n \neq m$. By writing explicit rules[7] for such situa-

[6] π-calculus variants such as spi-calculus [1] which allow compound messages typically have a similar problem in that a compound message is syntactically allowed in the position of a channel name but cannot actually be used to handshake a communication event.

[7] E.g., **reduce**{<M1,M2>.P | (x1,x2,x3).Q $\hookrightarrow$ •.0} for $(m, n) = (2, 3)$. Our implementation also provides an extension for writing a single rule that catches all pairs (m, n) at once.

tions, they can be handled in parallel with malformed substitutions. (One cannot straightforwardly write patterns to test for malformed substitutions, which is why the generation of • in this cases is built into Meta✶).

In either case, the Poly✶ type system will conservatively estimate *whether* (and where) a • can occur. Which conclusions to draw from this (e.g., rejecting the input program due to "type error") is up to the designer of the calculus.

Substitution of compound messages into element positions involves one final subtlety. Consider this process:

```
<0>.0 | (a).(<a.b> | (c).foo c.0)
```

After two applications of a simple communication rule, will "foo b.0" or "foo •.0" be left? In our preferred variant of Meta✶, we get the latter. We view the compound-ness of a.b to be an inherent property which does not go away when a gets replaced by the empty message. This is why we do not identify the messages 0.b and b. We choose this semantics because we feel it is closer to our operational intuition about the passing of capability sequences in most mobility calculi, but the opposite choice would pose no fundamental problems either.

We can now define substitution formally:

Definition 2.2. A (term) **substitution** $\mathcal{S}$ is a finite map from names to messages. Figure 2 defines the action of $\mathcal{S}$ on the various syntactic classes of Meta✶. $\mathcal{S}^{\mathbf{M}}$ is the action on messages, $\mathcal{S}^{\mathbf{E}}$ the action on elements, $\mathcal{S}^{\mathbf{F}}$ on forms, and $\mathcal{S}^{\mathbf{P}}$ on processes. □

Note that $\mathcal{S}^{\mathbf{F}}\,\mathsf{F}$ can be either a message or a form, depending on the shape of F. $\mathcal{S}^{\mathbf{M}}$ is defined completely by pointwise substitution of subforms using $\mathcal{S}^{\mathbf{F}}$ This is syntactically meaningful because $\mathcal{S}^{\mathbf{F}}\,\mathsf{f}$ is either a message or a *sub*-form, and the latter can be construed a message.

The definitions in Figure 2 do not worry about name capture. In general, therefore, $\mathcal{S}^{\mathbf{X}}\,\mathsf{X}$ is only intuitively correct if $\mathrm{BN}(\mathsf{X})$ is disjoint from the names mentioned in $\mathcal{S}$. In practise, this will always follow from the assumption that all terms are well scoped.

2.4 Reduction rules

Figure 3 defines most of the syntax and semantics of reduction rules for Meta✶. As an extra-syntactic restriction, the forms marked (R) in the grammar are allowed only to the *right* of the arrow in a **reduce** rule. In other words, a reduction rule cannot match on the structure of messages. The message can be substituted into a process, and then the *process* can be matched in future reductions. (This is not an intrinsic limitation of our methods, but it saves us the trouble of worrying about whether to treat M.N as associative when matching or not).

With this syntax we can describe, say, Mobile Ambients by the ruleset

Name variables: $\mathring{x} ::= \mathtt{a} \mid \mathtt{b} \mid \mathtt{c} \mid \cdots$

Message variables: $\mathring{m} ::= \mathtt{M} \mid \mathtt{N} \mid \cdots$

Process variables: $\mathring{p} ::= \mathtt{P} \mid \mathtt{Q} \mid \mathtt{R} \mid \cdots$

Substitutions: $\underline{S} ::= \{\mathring{x}_1 := \underline{M}_1, \ldots, \mathring{x}_k := \underline{M}_k\}$

Sub-element templates: $\underline{e} ::= \mathring{x} \mid x$ (R)

Sub-form templates: $\underline{f} ::= \underline{e}_0\ \underline{e}_1 \ldots \underline{e}_k$ (R)

Message templates: $\underline{M} ::= \underline{S}\ \mathring{m}$

$\quad \mid \underline{f} \mid 0 \mid \underline{M}_1.\underline{M}_2$ (R)

Element templates: $\underline{E} ::= \mathring{x} \mid x \mid (\mathring{x}_1, \ldots, \mathring{x}_k) \mid \langle \underline{M}_1, \ldots, \underline{M}_k \rangle$

Forms templates: $\underline{F} ::= \underline{E}_0\ \underline{E}_1 \ldots \underline{E}_k$

Process templates: $\underline{P} ::= \underline{S}\ \mathring{p} \mid \underline{F}.\underline{P} \mid 0 \mid (\underline{P}_1 \mid \underline{P}_2)$

Rules: $\mathcal{R}^1 ::= \mathbf{reduce}\{\underline{P}_1 \hookrightarrow \underline{P}_2\} \mid \mathbf{active}\{\mathring{p} : \underline{P}\}$

Rulesets: $\mathcal{R} \in \mathcal{P}_{\text{fin}}(\boxed{\mathcal{R}^1})$

A trivial $\underline{S}$ (i.e, the $k = 0$ case "{}") can be omitted.

Let an **instantiation** $\mathcal{V}$ map $\boxed{\mathring{x}}$ to $\boxed{x}\backslash\{\bullet\}$, $\boxed{\mathring{m}}$ to $\boxed{M}$, and $\boxed{\mathring{p}}$ to $\boxed{P}$. It applies to templates in this way (in the omitted cases, apply it to the template component pointwise):

$$\mathcal{V}^{\mathbf{S}}\{\ldots, \mathring{x}_i := \underline{M}_i, \ldots\} = \{\ldots, \mathcal{V}(\mathring{x}_i) \mapsto \mathcal{V}^{\mathbf{M}}\,\underline{M}_i, \ldots\}$$

$$\mathcal{V}^{\mathbf{M}}(\underline{S}\ \mathring{m}) = (\mathcal{V}^{\mathbf{S}}\,\underline{S})^{\mathbf{M}}(\mathcal{V}(\mathring{m}))$$

$$\mathcal{V}^{\mathbf{E}}\,\mathring{x} = \mathcal{V}(\mathring{x}) \text{ – but undefined if } \mathcal{V}(\mathring{x}) \text{ is bound by } \mathring{x}' \neq \mathring{x}$$

$$\mathcal{V}^{\mathbf{E}}\,x = x$$

$$\mathcal{V}^{\mathbf{F}}(\underline{E}_0 \ldots \underline{E}_k) = (\mathcal{V}^{\mathbf{E}}\,\underline{E}_0) \ldots (\mathcal{V}^{\mathbf{E}}\,\underline{E}_k)$$

$$\mathcal{V}^{\mathbf{P}}\,\underline{S}\ \mathring{p} = (\mathcal{V}^{\mathbf{S}}\,\underline{S})^{\mathbf{P}}(\mathcal{V}(\mathring{p}))$$

$$\frac{\mathbf{reduce}\{\underline{P}_1 \hookrightarrow \underline{P}_2\} \in \mathcal{R}}{\mathcal{R} \vdash \mathcal{V}^{\mathbf{P}}\,\underline{P}_1 \hookrightarrow \mathcal{V}^{\mathbf{P}}\,\underline{P}_2} \qquad \frac{\mathcal{R} \vdash P \hookrightarrow Q}{\mathcal{R} \vdash \nu(x).P \hookrightarrow \nu(x).Q}$$

$$\frac{\mathbf{active}\{\mathring{p} : \underline{P}\} \in \mathcal{R} \qquad \mathcal{R} \vdash P \hookrightarrow Q}{\mathcal{R} \vdash (\mathcal{V}, \mathring{p} \mapsto P)^{\mathbf{P}}\,\underline{P} \hookrightarrow (\mathcal{V}, \mathring{p} \mapsto Q)^{\mathbf{P}}\,\underline{P}}$$

$$\frac{\mathcal{R} \vdash P \hookrightarrow Q}{\mathcal{R} \vdash P \mid R \hookrightarrow Q \mid R} \qquad \frac{P \equiv Q \qquad \mathcal{R} \vdash Q \hookrightarrow R}{\mathcal{R} \vdash P \hookrightarrow R}$$

Figure 3: Syntax and semantics of reduction rules. The side condition for $\mathcal{V}^{\mathbf{E}}\,\mathring{x}$ prevents, e.g., `(a).c.0 | (b).c.0` *from being instantiated to* `(x).x.0 | (x).x.0`.

$$\mathcal{R}_{\mathrm{MA}} = \{\ \textbf{active}\{\mathtt{P} : \mathtt{a}[\mathtt{P}]\},$$
$$\textbf{reduce}\{\mathtt{a}[\mathsf{in}\ \mathtt{b}.\mathtt{P} \mid \mathtt{Q}] \mid \mathtt{b}[\mathtt{S}] \hookrightarrow \mathtt{b}[\mathtt{a}[\mathtt{P} \mid \mathtt{Q}] \mid \mathtt{S}]\},$$
$$\textbf{reduce}\{\mathtt{a}[\mathtt{b}[\mathsf{out}\ \mathtt{a}.\mathtt{P} \mid \mathtt{Q}] \mid \mathtt{S}] \hookrightarrow \mathtt{a}[\mathtt{S}] \mid \mathtt{b}[\mathtt{P} \mid \mathtt{Q}]\},$$
$$\textbf{reduce}\{\mathsf{open}\ \mathtt{a}.\mathtt{P} \mid \mathtt{a}[\mathtt{R}] \hookrightarrow \mathtt{P} \mid \mathtt{R}\},$$
$$\textbf{reduce}\{\texttt{<M>}.\mathtt{P} \mid \texttt{(a)}.\mathtt{Q} \hookrightarrow \mathtt{P} \mid \{\mathtt{a} := \mathtt{M}\}\mathtt{Q}\}\ \}$$

These five rules are all that is necessary to instantiate Meta✶ to be Mobile Ambients.[8] The four **reduce** rules directly correspond to the reduction axioms of the target calculus. The rule **active**$\{\mathtt{P} : \mathtt{a}[\mathtt{P}]\}$ is the Meta✶ notation for the "evaluation context" rule $\mathtt{P} \hookrightarrow \mathtt{P}' \Longrightarrow \mathtt{a}[\mathtt{P}] \hookrightarrow \mathtt{a}[\mathtt{P}']$. This is, in fact, the only concrete **active** rule that we have so far found necessary for encoding existing calculi. We might just have hard-coded something like this rule into Meta✶, but we find it cleaner not to have any built-in distinction between "action" forms and "process container" forms in the theory.

The semantics of rules is based on the concept an **instantiation map** $\mathcal{V}$ that fills the hole in a template by mapping each $\mathring{\mathtt{x}}$ in it template to a name, each $\mathring{\mathtt{m}}$ to a message, and each $\mathring{\mathtt{p}}$ to a process, and the ordinary process term $\mathcal{V}^{\mathtt{P}}\,\underline{\mathtt{P}}$ arises.

As a special exception a name variable $\mathring{\mathtt{x}}$ cannot be instantiated to $\bullet$. This gives the calculus designer the freedom to specify that forms containing $\bullet$ are completely passive and do not take part in interactions — for example, it is not obvious that two actions that try to send and receive, respectively, on channel $\bullet$ because of two *independent* substitution errors would actually rendezvous. A rule *can* match a verbatim $\bullet$, so it is possible to simulate a $\mathring{\mathtt{x}}$ that does match $\bullet$ by duplicating the rule.

The inference system at the bottom of Figure 3 now describes how to derive a reduction relation between process terms from a ruleset.

Templates have to satisfy some scoping restrictions that are not apparent from the syntax. The restrictions will be satisfied by most rules that are intuitively sensible; as a precise understanding of how the restrictions work is not important for a high-level understanding of Meta✶ we defer a precise definition to Section 5.1. A ruleset that satisfies the restrictions has the property that the reduction relation derived from it automatically preserves well-scopedness.

2.5 Example instantiations

We have checked (using machine-readable rulesets for our type inference implementation for Meta✶/Poly✶) that Meta✶ can handle π-calculus [22]; Mobile Ambients [11]; Safe Ambients [20], and various variants regarding where the $\overline{\mathsf{out}}$ capability must be found, and which name co-capabilities must refer to (variants with anonymous co-capabilities also exist [19]); the Seal calculus [29] in the non-duplicating variation of [13]; Boxed Ambients [7], as well as its "asynchronous" and "Seal-like" variants (the latter being what later papers

[8] The rules are not sufficient to get communication reduction with arbitrary arity. Our implementation provides a syntax for defining arbitrary-arity communication rules, but for reasons of space and clarity we omit it in our formal development.

most often refer to as BA); Channelled Ambients [25]; NBA [8]; Push and Pull Ambient Calculus [26]; and $\mathbf{M^3}$ [15].

In many of these cases, Meta✶ supports the straightforward way to notate process terms as flat ASCII text, but in some cases the native punctuation of the target calculus must be changed superficially to conform to Meta✶ conventions about how a form looks. For example, the original send action $\overline{y}x$ from [22] is represented as "y<x>" (but, say, "/y x" would also have worked), and "$\overline{\mathsf{enter}}(x, y)$" from [8] becomes "co-enter(x)y", because it binds x in its continuation but uses y to handshake with the entering ambient. The "$n[c_1, \ldots, c_k; P]$" construction in Channelled Ambients [25] can be represented as "$n[cs.(c_1.0 \mid \cdots \mid c_k.0) \mid ps.P]$". In our ruleset for Mobile Ambients with Objective Moves [10], the fact that reduction rules cannot inspect the structure of messages force us to represent the original "go $M.m[P]$" as "go.$M.m[P]$".

The original presentation of the π-calculus [22] included a non-deterministic choice "$P + Q$" which we can represent as $+.(P \mid Q)$ with the associated rule **reduce**$\{+.(P \mid Q) \hookrightarrow P\}$. (The parallel composition of P and Q in $+.(P \mid Q)$ will not cause them to interact as long as no **active** rule explicitly specifies it). Milner et al. [22] describe infinite behaviour by "defined agents" with user-specified recursive unfoldings. In Meta✶ the unfoldings can be given as user-specified **reduce** rules.

The Sealed Boxed Ambients calculus of [9] is not currently supported, primarily because it need multiple tiers of context rules, not just the one kind of **active** rules in Meta✶.

3 Poly✶: Types for Meta✶

3.1 Shape predicates

As described in the introduction, *shape predicates* are the central concept in Poly✶. A shape predicate denotes a set of process terms; certain shape predicates that are provably closed under reduction will be called *types*.

The full language of shape predicates is somewhat complex, so let us introduce it piecewise. To begin with, assume that terms contain neither replication (!P) nor name restriction ($\nu(x).P$). Then we can explain the basic idea of shape predicates simply: *A shape predicate looks like a process term. It matches any process term that can arise by repeatedly duplicating and/or removing sub-terms anywhere in the shape predicate.*

For example, a shape predicate written a[in b | in c] | c[0] would match the terms a[in b | in c] | c[0] (which is identical to the shape predicate) and a[in b] | a[in c] | c[0] (which arises by duplicating $a[\cdots]$ and then removing one of the in subterms in each of the copies). But a[in b] | c[a[0]] does not match, because duplicating subterms cannot make a[] appear below a c[]. Neither does in b | in c | c[0] allowed — when removing the a[] form, the entire subterm below it must be removed, too.

In practise shape predicates cannot be exactly term-shaped, but it pays to keep this naive idea in mind as an intuition about what shape predicates are. When we introduce complications in the rest of this subsection, they should all be understood as "whatever is necessary to make the naive idea work in practise".

Replication (`!P`) is ignored when matching shape predicates. This is sensible because `!P` behaves like an infinite number of P's running in parallel, and any *finite* number of P's in parallel match a shape predicate exactly if a single P does.

We want to represent all possible computational future of each term smashed together in a single shape predicate. This creates problems for the naive idea, because terms such as `!x[eat x]` can evolve to arbitrary deep nestings of $x[\cdots]$. Therefore we need shape predicates to be *infinitely* deep trees. We should, however, restrict ourselves to infinite shape predicates with finite *representations* — in other words, regular trees.

There are several known ways of representing regular trees as linear syntax, but we have found it easier to work directly with *graphs*. In other words, a shape predicate now becomes a directed graph (possibly containing cycles) where each edge is labelled with a form. A term matches the shape predicate if its syntax tree can be "bent into shape" to match a subgraph such that each form in the term lies atop a corresponding edge in the graph (edges may be used more than once), and groups of parallel composition, `!`, and `0` lie within a single node in the graph.

In [2] we defined a linear notation called *shape expressions* for such graphs, which aimed at presenting shape predicates readably. However, our subsequent experience is that except for very small graphs, a two-dimensional graphical representation is usually easier to understand. A linear representation is still useful as an interchange format, but due to space restrictions we will not repeat the definition from [2].

Graphs alone are not enough to guarantee a finite type for every term. For example, the term `<x> | !(y).<y.y>` can (given an appropriate reduction rule) evolve into terms with messages that contain arbitrarily long chains of `x`'s *within* a single form. We need to abstract over messages such that an infinity of forms that look alike except having messages of different length within them can be described by the same shape graph label. This is the job of **message types** μ, which are defined in Figure 4.

The message type $\{f_1, \ldots, f_k\}*$ describes any message built from the any of forms f_i — *except* messages that are single names; such a message is matched by the message type `{x}` instead. When `{x}` is the *only* message type that matches `x`, we can see unambiguously from a message type whether $\bullet$ will result from trying to substitute a message it matches into an element position.

We use **element types** ε and **form types** φ to build form-like structures out of message types and non-message elements. The labels on shape-graph edges are now form types instead of forms.

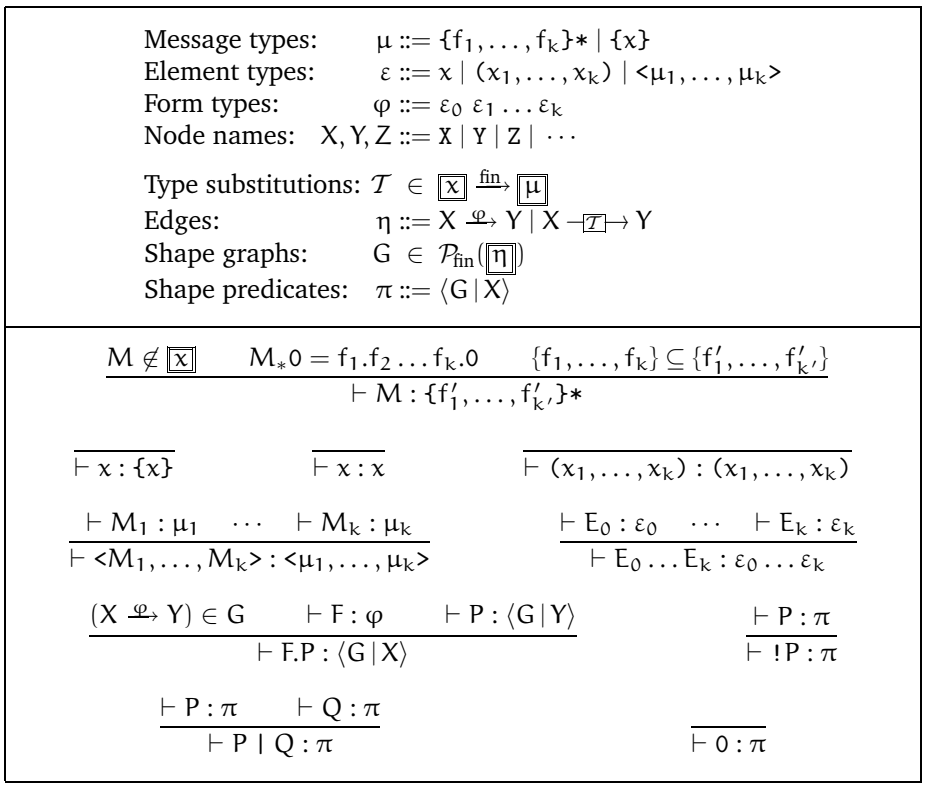


Figure 4: The syntax and semantics of shape predicates. Edges of the form $X \dashv\boxed{\mathcal{T}}\mapsto Y$ *do not influence the semantics of the shape predicate; Sect. 3.2 explains what they are for.*

Definition 3.1. Let $\mu_1 *\mu_2$ be the least message type whose meaning includes $M.N$ for all $M \in [\![\mu_1]\!], N \in [\![\mu_2]\!]$. □

With the language of message types so far (it will be expanded in Section 5.2), $\mu_1 *\mu_2$ always has the form $\{f_1, \ldots, f_k\}*$, where the f_i's are all the sub-forms that appear in either μ_1 or μ_2 in some canonical order (for this purpose the sub-form x is considered to appear in the message type $\{x\}$). The $\mu_1 * \mu_2$ operation is associative.

The syntax and semantics of shape predicates is defined in Figure 4, except for *name restriction* (which we defer to Section 5.3 in order to present the basic theory more clearly), and a third form of message types (which will allow more precise types and is introduced in Section 5.2).

Definition 3.2. Define the **meaning** of message/element/form types and of shape predicates by

$$[\![\mu]\!] = \{ M \mid \vdash M : \mu \} \quad [\![\varepsilon]\!] = \{ E \mid \vdash E : \varepsilon \}$$

$$[\![\varphi]\!] = \{\, F \mid \vdash F : \varphi \,\} \quad [\![\pi]\!] = \{\, P \mid \vdash P : \pi \,\}$$

□

Proposition 3.3. *The meanings of shape predicates respect the structural congruence: If* $P \equiv Q$ *then* $\vdash P : \pi \iff \vdash Q : \pi$ *for all* π. □

3.2 Flow edges and subtyping

The only part of the shape predicate syntax of Figure 4 that has yet not been explained is the **flow edges** $X \multimap\!\boxed{\mathcal{T}}\!\!\rightarrow Y$. They are not used at all in the above definition of the *meaning* of the shape graph, but they will be important for distinguishing between types and non-types. In brief, the flow edge $X \multimap\!\boxed{\mathcal{T}}\!\!\rightarrow Y$ asserts that there may be a reduction where a process described by X is moved to Y and in the process suffers a substitution described by $\mathcal{T}$.

An important special case is when $\mathcal{T} = \varnothing$, where the process moves *without* any substitution. Then $X \multimap\!\boxed{\varnothing}\!\!\rightarrow Y$ can also be viewed as an assertion that $\langle G \mid X\rangle$ is a *subtype* of $\langle G \mid Y\rangle$, or, symbolically, that $[\![\langle G \mid X\rangle]\!] \subseteq [\![\langle G \mid Y\rangle]\!]$. We therefore also speak of $X \multimap\!\boxed{\varnothing}\!\!\rightarrow Y$ as a **subtyping edge**.

Definition 3.4. Write $\vdash \mathcal{S} : \mathcal{T}$ iff $\mathrm{Dom}\ \mathcal{S} = \mathrm{Dom}\ \mathcal{T}$ and $\vdash \mathcal{S}(x) : \mathcal{T}(x)$ for all $x \in \mathrm{Dom}\ \mathcal{S}$. □

Definition 3.5. Define the action of type substitution on message, element, and form types by

$$\mathcal{T}^{\mathbf{M}}\{f_1, \ldots, f_k\}* = \mathcal{T}^{\mathbf{F}} f_1 *\cdots* \mathcal{T}^{\mathbf{F}} f_k \qquad \mathcal{T}^{\mathbf{M}}\{x\} = \begin{cases} \mathcal{T}(x) & \text{if } x \in \mathrm{Dom}\ \mathcal{T} \\ \{x\} & \text{otherwise} \end{cases}$$

$$\mathcal{T}^{\mathbf{E}} x = \begin{cases} y & \text{when } \mathcal{T}^{\mathbf{M}}\{x\} = \{y\} \text{ for some } y \\ \bullet & \text{otherwise} \end{cases}$$

$$\mathcal{T}^{\mathbf{E}} <\mu_1, \ldots, \mu_k> = <\mathcal{T}^{\mathbf{M}}\mu_1, \ldots, \mathcal{T}^{\mathbf{M}}\mu_k> \qquad \mathcal{T}^{\mathbf{E}}(x_1, \ldots, x_k) = (x_1, \ldots, x_k)$$

$$\mathcal{T}^{\mathbf{F}}(E_0 \ldots E_{k+1}) = \mathcal{T}^{\mathbf{E}} E_0 \ldots \mathcal{T}^{\mathbf{E}} E_{k+1} \qquad \mathcal{T}^{\mathbf{F}} x = \mathcal{T}^{\mathbf{M}}\{x\}$$

As for term substitution, the result of substituting in a form type can be either another form type or a message type. □

This definition ensures that $[\![\mathcal{T}^{\mathbf{M}}\mu]\!]$ contains the result of every *term* substitution $\mathcal{S}^{\mathbf{M}} M$ where $\vdash \mathcal{S} : \mathcal{T}$ and $\vdash M : \mu$, and likewise for elements and forms.

Definition 3.6. The shape graph G is **flow closed** iff whenever G contains $X \xrightarrow{\varphi} Y$ and $X \multimap\!\boxed{\mathcal{T}}\!\!\rightarrow Z_0$, such that $\mathrm{BN}(\varphi) \cap \mathrm{Dom}\ \mathcal{T} = \varnothing$, it also contains

1. When $\mathcal{T}^{\mathbf{F}}\varphi = \varphi'$: $Z_0 \xrightarrow{\varphi'} Z_1$ and $Y \multimap\!\boxed{\mathcal{T}}\!\!\rightarrow Z_1$ for some Z_1.
2. When $\mathcal{T}^{\mathbf{F}}\varphi = \{x\}$: $Z_0 \xrightarrow{x} Z_1$ and $Y \multimap\!\boxed{\mathcal{T}}\!\!\rightarrow Z_1$ for some Z_1.
3. When $\mathcal{T}^{\mathbf{F}}\varphi = \{f_1, \ldots, f_k\}*$: $Z_0 \xrightarrow{f_i} Z_0$ for $1 \le i \le k$ and $Y \multimap\!\boxed{\mathcal{T}}\!\!\rightarrow Z_0$

We call a shape predicate $\langle G \mid X\rangle$ flow closed iff its G component is. □

Let a **type instantiation** $\mathcal{U}$ map $\boxed{\mathring{x}}$ to $\boxed{x} \setminus \{\bullet\}$, $\boxed{\mathring{m}}$ to $\boxed{\mu}$, and $\boxed{\mathring{p}}$ to $\boxed{X}$. It applies to templates in this way (in the omitted cases, apply it to the template component pointwise):

$$\mathcal{U}^{\mathbf{S}}\{\ldots, \mathring{x}_i := \underline{M}_i, \ldots\} = \{\ldots, \mathcal{U}(\mathring{x}_i) \mapsto \mathcal{U}^{\mathbf{M}}\,\underline{M}_i, \ldots\}$$

$$\mathcal{U}^{\mathbf{M}}(\underline{S}\ \mathring{m}) = (\mathcal{U}^{\mathbf{S}}\,\underline{S})^{\mathbf{M}}(\mathcal{U}(\mathring{m}))$$

$$\mathcal{U}^{\mathbf{M}}\,\underline{f} = \begin{cases} \{x\} & \text{if } \underline{f} = x \\ \mathcal{U}^{\mathbf{F}}\,\underline{f} & \text{if } \underline{f} \notin \boxed{x} \end{cases}$$

$$\mathcal{U}^{\mathbf{E}}\,\mathring{x} = \mathcal{U}(\mathring{x}) \text{ – but undefined if } \mathcal{V}(\mathring{x}) \text{ is bound by } \mathring{x}' \neq \mathring{x}$$

$$\mathcal{U}^{\mathbf{E}}\,x = x$$

$$\mathcal{U}^{\mathbf{E}}\,(\mathring{x}_1, \ldots, \mathring{x}_k) = (\mathcal{U}(\mathring{x}_1), \ldots, \mathcal{U}(\mathring{x}_k))$$

$$\frac{X = \mathcal{U}(\mathring{p})}{\mathcal{U} \vDash_{\mathsf{L}} \{\}\,\mathring{p}.P : \langle G \mid X\rangle} \qquad \frac{(\mathcal{U}(\mathring{p}) \multimap\!\boxed{\mathcal{U}^{s}\,\underline{S}}\!\to X) \in G}{\mathcal{U} \vDash_{\mathsf{R}} \underline{S}\,\mathring{p}.P : \langle G \mid X\rangle} \qquad \frac{}{\mathcal{U} \vDash_{s} 0 : \pi}$$

$$\frac{\mathcal{U} \vDash_{s} \underline{P}_1 : \pi \qquad \mathcal{U} \vDash_{s} \underline{P}_2 : \pi}{\mathcal{U} \vDash_{s} \underline{P}_1 \mid \underline{P}_2 : \pi} \qquad \frac{(X \xrightarrow{\mathcal{U}^{F}\,\underline{F}} Y) \in G \qquad \mathcal{U} \vDash_{s} \underline{P} : \langle G \mid Y\rangle}{\mathcal{U} \vDash_{s} \underline{F}.\underline{P} : \langle G \mid X\rangle}$$

Figure 5: Matching of reduction rules to shape graphs. The rules for template processes have an L *variant and and* R *variant; the variable letter* s *ranges over* L *and* R*.*

In flow-closed graphs, flow edges give valid information about node meanings:

Proposition 3.7. *Let* G *be flow closed and contain* $X \multimap\!\boxed{\mathcal{T}}\!\to Y$*. Assume that* $\vdash \mathcal{S} : \mathcal{T}$ *and that* $\mathrm{BN}(P) \cap \mathrm{Dom}\,\mathcal{T} = \varnothing$*. Then* $\vdash P : \langle G \mid X\rangle$ *implies* $\vdash \mathcal{S}^{\mathbf{P}}\,P : \langle G \mid Y\rangle$ □

Proof. By induction on the derivation of $\vdash P : \langle G \mid X\rangle$. □

The assumption that $\mathrm{BN}(P) \cap \mathrm{Dom}\,\mathcal{T} = \varnothing$ will be true in applications because we are assuming that all terms are well-scoped.

3.3 Semantic and syntactic closure; types

Definition 3.8. Call the shape predicate π **semantically closed** with respect to $\mathcal{R}$ iff $\vdash P : \pi$ and $\mathcal{R} \vdash P \hookrightarrow Q$ imply $\vdash Q : \pi$. □

As described above, we want *types* to be semantically closed shape predicates. But it is not easy to recognise semantic closure.[9] We therefore define

[9] E.g., let $G = \{\mathtt{Y1} \xleftarrow{\mathtt{c}} \mathtt{Y0} \underset{\mathtt{b}}{\overset{\mathtt{b}}{\rightleftarrows}} \mathtt{X0} \xrightarrow{\mathtt{(a)<\{b\}*>}} \mathtt{X1} \xrightarrow{\mathtt{a}} \mathtt{X2} \xrightarrow{\mathtt{b}} \mathtt{X3} \xrightarrow{\mathtt{a}} \mathtt{X4} \xrightarrow{\mathtt{c}} \mathtt{X5}\}$. Then $\langle G \mid \mathtt{X0}\rangle$ happens to be semantically closed with respect to $\{\mathbf{reduce}\{\mathtt{(a)<M>.P} \hookrightarrow \{\mathtt{a} := \mathtt{M}\}\mathtt{P}\}\}$, but it is not trivial to see this, and we can think of no systematic way to capture all examples of this kind.

a restricted, but easier to decide, class of *syntactically* closed shape predicates, which will become types.

Figure 5 defines a way to match process templates directly to type graphs without going via the process semantics from Figure 3. For example, let G_0 be the graph $\left\{ \mathtt{X} \xleftarrow{\text{in v}} \mathtt{Y} \underset{(\mathtt{v})}{\overset{\mathtt{v}\mapsto\{\mathtt{b},\mathtt{c}\}*}{\rightleftarrows}} \mathtt{Z} \xrightarrow{<\{\mathtt{b},\mathtt{c}\}*>} \mathtt{W} \right\}$ and let $\mathcal{U} = \left[\mathtt{P} \mapsto \mathtt{W}, \mathtt{Q} \mapsto \mathtt{Y}, \mathtt{a} \mapsto \mathtt{v}, \mathtt{M} \mapsto \{\mathtt{a},\mathtt{c}\}*\right]$. Then $\mathcal{U} \models_{\mathsf{L}}$ `<M>.P | (a).Q` $: \langle G_0 \,|\, \mathtt{Z} \rangle$ holds. But $\mathcal{U} \models_{\mathsf{R}} \{\}\mathtt{P} \mid \{\mathtt{a} := \mathtt{M}\}\mathtt{Q} : \langle G_0 \,|\, \mathtt{Z} \rangle$ does not hold, because the subtyping edge $\mathtt{W} \multimap \mathtt{Z}$ $(= \mathcal{U}(\mathtt{P}))$ is missing.

Definition 3.9. Let the shape predicate $\pi = \langle G \,|\, X \rangle$ be given. The set of **active** nodes for $\mathcal{R}$, written $\mathsf{active}_{\mathcal{R}}(\pi)$, is the least set A of nodes which contains X and such that for all $Y \in A$ and all **active**$\{\mathring{p} : \underline{P}\} \in \mathcal{R}$, it holds that $\mathcal{U} \models_{\mathsf{L}} \underline{P} : \langle G \,|\, Y \rangle$ implies $\mathcal{U}(\mathring{p}) \in A$. □

Definition 3.10. G is **locally closed** at X with respect to $\mathcal{R}$ iff whenever $\mathcal{R}$ contains **reduce**$\{\underline{P_1} \hookrightarrow \underline{P_2}\}$ it holds that $\mathcal{U} \models_{\mathsf{L}} \underline{P_1} : \langle G \,|\, X \rangle$ implies $\mathcal{U} \models_{\mathsf{R}} \underline{P_2} : \langle G \,|\, X \rangle$. □

To continue the above example, G_0 is not locally closed at `Z` with respect to the ruleset $\left\{\textbf{reduce}\{\mathtt{<M>.P} \mid \mathtt{(a).Q} \hookrightarrow \{\}\mathtt{P} \mid \{\mathtt{a} := \mathtt{M}\}\mathtt{Q}\}\right\}$, but $G_1 = G_0 \cup \{\mathtt{W} \multimap \mathtt{Z}\}$ is. (However, G_1 is not flow closed – for that $\mathtt{Z} \xrightarrow{\text{in}\bullet} Z'$ is also required for some Z').

Definition 3.11. The shape predicate $\pi = \langle G \,|\, X \rangle$ is **syntactically closed** with respect to $\mathcal{R}$ iff G is flow closed and also locally closed at every $X \in \mathsf{active}_{\mathcal{R}}(\pi)$. When this holds, we call π an ($\mathcal{R}$-)**type**. □

Checking that a purported type is really syntactically closed is algorithmically easy; see Section 5.4 for details.

Theorem 3.12 (Subject reduction). *If π is syntactically closed with respect to $\mathcal{R}$, then it is also semantically closed with respect to $\mathcal{R}$.* □

3.4 What to do with types

Once we have a type, what can we use it for? An obvious possibility is to check whether the term may "go wrong". The user (or, more likely, the designer of a programming environment that uses Poly✶) specifies what "going wrong" means. It is a clear indication of error if • turns up in an active position, but opinions may differ about how bad it is that a • is produced at a place in process tree that never becomes active.

One can also imagine very application-specific properties to check for, for example "this process can never evolve to a configuration where an ambient named a is inside one named b. This is easy to check for in the shape graph. Alternatively, one may want to write this as a rule, to have Poly✶ do the checking: **reduce**$\{\mathtt{b[a[P] \mid Q]} \hookrightarrow \bullet.0\}$. The ability to write such rules is one of the reasons why Meta✶ does not distinguish strictly between "names" and "keywords".

Poly✶ makes it fairly easy to check *safety properties* like "no unauthorised ambients (e.g., a) inside secure ambients (e.g., b)", but there are also questions of safety that Poly✶ cannot help determine. This includes properties that depend on the *order* in which things happen, such as the "correspondence assertions" often used to specify properties of communication protocols. There are type systems for process calculi that can reason about such temporal properties (for example, [16] for the π-calculus), but we are aware of none that also handle locations and ambient-style mobility.

4 Type inference for Poly✶

Assume now that we are given a process term P and a ruleset $\mathcal{R}$; we want to produce an $\mathcal{R}$-type for P. It is trivial to construct *some* type for P – one with a single node and a lot of edges in the shape graph. However, such a type may need to contain •'s and thus not prove that P "cannot go wrong". In this section we discuss how to automatically infer more informative types.

We do not know how to do type inference that is complete for the full Poly✶ type system; it allows too many types. Therefore we begin by defining a set of *restricted* types, for which we *can* have complete type inference.

Definition 4.1. Write $\varphi_1 \approx \varphi_2$ iff $[\![\varphi_1]\!] \cap [\![\varphi_2]\!] \neq \varnothing$. □

The $\approx$ relation is close to being equality. The only way for two non-identical φ's to be related by $\approx$ is if at least one of them contains a message type of the shape $\{\cdots\}*$. It is relatively safe to imagine the $\approx$ is just a fancy way to write $=$, at least to a first approximation.

Indeed, the type inference *algorithm* works perfectly well if we take $\approx$ to be $=$ in the following restrictions. It even produces more precise types, but one loses the nice principality result in Theorem 4.4 below.

Definition 4.2. G satisfies the **width restriction** iff whenever it contains $X \xrightarrow{\varphi} Y$ and $X \xrightarrow{\varphi'} Y'$ with $\varphi \approx \varphi'$, it holds that $Y = Y'$. □

Definition 4.3. G satisfies the **depth restriction** iff whenever it contains a chain $X_0 \xrightarrow{\varphi_1} X_1 \xrightarrow{\varphi_2} \cdots \xrightarrow{\varphi_k} X_k$ with $\varphi_1 \approx \varphi_k$, it holds that $X_1 = X_k$. □

Our type inference algorithm only produces types that satisfy both restrictions. In Section 5.6 we describe a feature of our implementation which allows it to loosen the two restrictions by tracking the origin of each φ in the type graph.

4.1 The algorithm

The type inference proceeds in two phases. First we construct a minimal shape predicate which the term matches. Then we **close** the shape predicate — that is, rewrite its shape graph as necessary to make it syntactically closed.

The initial phase is simple. Because shape predicates "look like terms", we can just convert the abstract syntax tree of the term to a tree-shaped shape graph. This graph may or may not satisfy the width and depth restrictions. If it does not, unify the nodes that must be equal. That may cause further violations of the two restrictions; continue unifying nodes as necessary until the restrictions are satisfied.

(In our implementation of the inference algorithm, some of the work of enforcing the restrictions happen incrementally while the graph is being built, but that does not alter the overall understanding.)

The closing of the shape graph is where the real work of type inference happens. It happens in a series of steps. In each step it is checked whether the shape graph is syntactically closed, as described in Section 5.4. If it is, the algorithm ends. Otherwise, the lack of closure can only be because edges already in the graph implies that some other edges *ought* to exist (by Definitions 3.6 or 3.10) but do not. In that case, add the new nodes and edges required by the failing rule, and do another round of unifications to enforce the width and depth restrictions. (Again, in practise some of the unifications happen incrementally).

The width and depth restriction together guarantee that the closure phase terminates. By design, there are only a limited number d of different φ's that may ever be needed to appear in the graph. The width restriction limits the out-degree of the graph to d, and the depth restriction limits the length of any repetition-free path starting at the root node to d. Therefore, there will always be less than d^{d+1} different nodes in the graph, and every non-final closure step adds at least one edge to it which was not there previously. Sooner or later we will run out of possible edges to add, at which point the algorithm must terminate.

Of course, this analysis translates to a completely hopeless worst-case complexity bound for the inference algorithm. We do not know how bad it can get in practise; instead our implementation allows further restrictions on types to be applied in order to quench blow-ups one might observe with particular calculi and example terms. The tightest restrictions will enforce polynomial complexity, at the cost of losing the possibility of spatial polymorphism. Thus restricted, Poly$\ast$ has a strength roughly comparable to current non-polymorphic type systems for ambient-derived calculi.

Theorem 4.4. *The result of the type inference π is a principal typing [30] for the input term P: For every π' such that $\vdash P : \pi'$ it holds that $[\![\pi']\!] \supseteq [\![\pi]\!]$.* □

4.2 Implementation

We have implemented our type inference algorithm. Our implementation is available at the URL ⟨`http://www.macs.hw.ac.uk/DART/software/PolyStar/`⟩, as both a source download and an interactive web interface.

In addition to the features we have described, the implementation allows fine-tuning of the precision of the analysis, which influences the speed of the

```
active{ P : x[P] }
reduce{ a["eat" b|P] | b[Q] --> a[P|b[Q]] }
reduce{ "z1"[P] | "z2"[Q] --> "error" }
analyse{ x[eat z1 | eat z2]
       | y1[eat x | z1[] ]
       | y2[eat x | z2[] ] }
```

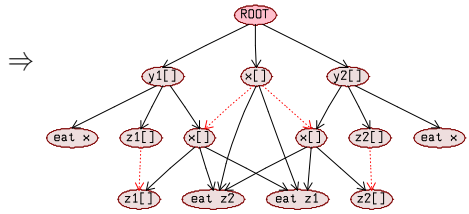


Figure 6: Input and output from the implementation in an example showing spatial polymorphism

inference as well as the size of the inferred type. The user can specify restrictions such as: "For each x there shall be a single node in the shape graph that is the endpoint of all edges labelled `x[]`. For all other form types, except ones that match `x(y,...)`, the edges shall have the same origin and target, forming a length-1 cycle (and these edges are exempted from the depth restriction). Marks shall not be used".

Figure 6 shows how our implementation (with default settings) analyses the example of spatial polymorphism from Section 1.3. The figure shows the exact input to the type inference, and its output as rendered by the VCG graph layout tool [27].

The labels on the ordinary shape graph edges are shown inside their target nodes; this is only possible when all edges leading to a particular node have the same label, but greatly improves the readability of the graph. Dotted arrows represent subtyping edges.

Figure 7 shows a Mobile Ambient example analysed with settings that force all non-ambient, non-communication edges to be length-one loops. For readability, this figure does not show the flow-edges. This example again shows spatial polymorphism in action; the message `<out a>` in the first `t[]` ambient does not interfere with the communication inside `b[]`.

5 Leftover details

Having presented the basic theory of Meta✶, Poly✶ and its inference algorithm, it is now time to put in the features we abstracted away for clarity of presentation.

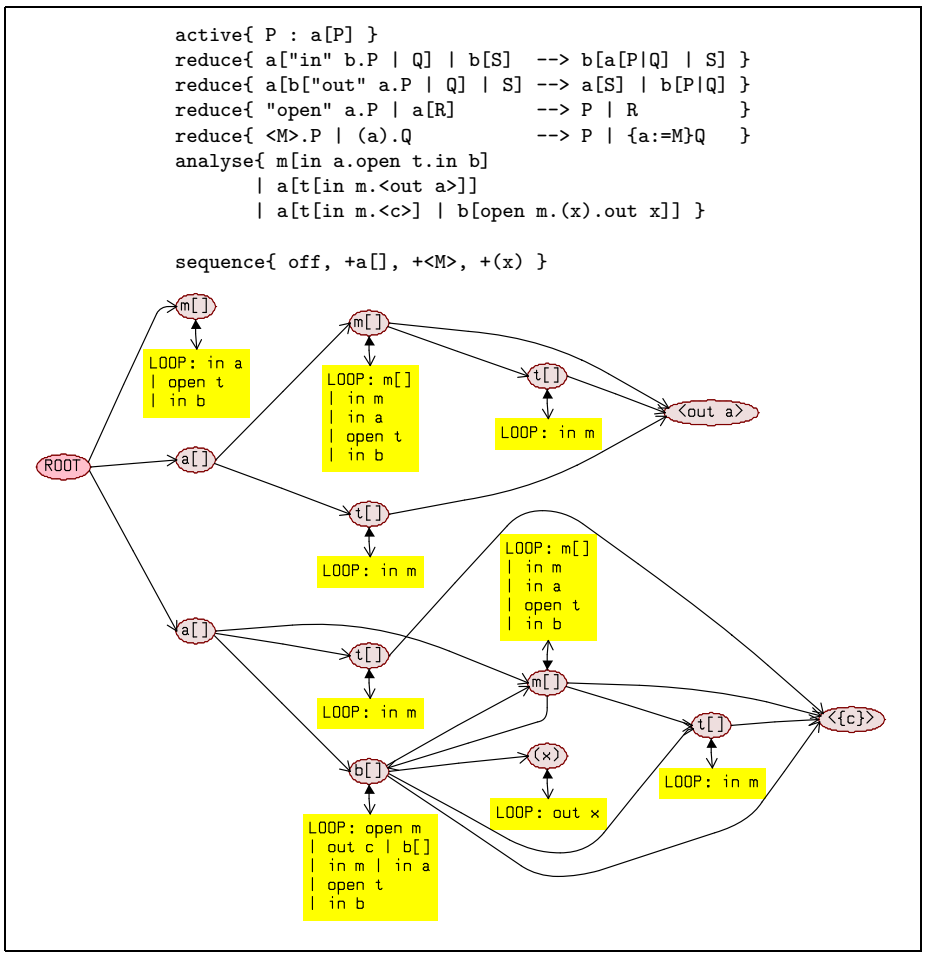


Figure 7: A Mobile Ambients eaxample that shows spatial polymorphism

5.1 Scoping restrictions on reduction rules

5.1.1 Left-hand sides

These restrictions apply to the left-hand side of **reduce** rules, and to the process templates in **active** rules:

L1. A variable of the kind m̊ or p̊ can appear at most once in the entire template. In other words, a reduction rule cannot depend of the fact that two entire messages or processes are identical. It *can* depend on the same *name* appearing in several places.

L2. A variable of the kind x̊ can appear at most once in a binding element. If it does, then all of its other appearances must be in the template process below the form that the binding element is part of.

L3. Every $\underline{\mathsf{S}}$ must be trivial, i.e. have the form $\{\}$. Such trivial substitutions can be omitted when writing down the rule.
L4. The grammatical forms marked (R) in Figure 3 cannot be used.

5.1.2 Right-hand sides

On the right-hand side of a **reduce** rule, the following restrictions apply:

R1. Every variable appearing on the right-hand side must also appear on the left-hand side: Reductions are not allowed to invent names, messages, or processes out of thin air.
R2. A $\mathring{\mathsf{x}}$ can appear in binding elements only if it was also bound in the left-hand side. If so, it cannot appear *free* on the left-hand side.
R3. Bindings for $\mathring{\mathsf{x}}_1$ and $\mathring{\mathsf{x}}_2$ can be nested only if the corresponding bindings on the left-hand side were also nested. (Otherwise, the rule might match with a $\mathcal{V}$ that mapped $\mathring{\mathsf{x}}_1$ and $\mathring{\mathsf{x}}_2$ to the same concrete name, creating a nested binding when the RHS template is instantiated.)
R4. An $\mathring{\mathfrak{m}}$ or $\mathring{\mathsf{p}}$ that appeared within the scope of a binding element on the left-hand side can only appear within the scope of bindings for the same $\mathring{\mathsf{x}}$ on the right-hand side. For this purpose, a $\underline{\mathsf{S}}$ counts as establishing bindings. This rule prevents a rule from removing names from their bindings.
R5. A $\mathring{\mathsf{p}}$ may not appear within the right-hand-side scope of a $\mathring{\mathsf{x}}$ that was not bound at the appearance of $\mathring{\mathsf{x}}$ at the left-hand side. (Otherwise, $\mathcal{V}(\mathring{\mathsf{p}})$ might contain another binding of $\mathcal{V}(\mathring{\mathsf{x}})$, creating a nested binding).

It is allowed to mention the same $\mathring{\mathsf{p}}$ more than once in the RHS, though such a rule does not blend well with Meta$\star$'s default semantics for name restrictions. For example, the following rule

$$\mathcal{R}^1 = \mathbf{reduce}\{\mathsf{bang.P} \hookrightarrow (\mathsf{P} \mid \mathsf{bang.P})\}$$

results in both of the reductions

$$\{\mathcal{R}^1\} \vdash \mathsf{bang.}\nu(\mathsf{a}).\mathsf{foo\ a} \hookrightarrow \nu(\mathsf{a}).\mathsf{foo\ a} \mid \mathsf{bang.}\nu(\mathsf{a}).\mathsf{foo\ a}$$
$$\{\mathcal{R}^1\} \vdash \mathsf{bang.}\nu(\mathsf{a}).\mathsf{foo\ a} \hookrightarrow \nu(\mathsf{a}).(\mathsf{foo\ a} \mid \mathsf{bang.foo\ a})$$

because the structural congruence relation in Meta$\star$ allows scopes to extrude beyond arbitrary forms. A true solution to this problem awaits further work, but for the time being Poly$\star$ can be used *as if* the latter reduction never happened. This may lead to a slight overapproximation of the set of possible futures for a term, but the types will still be *sound* with respect to a more precise semantics for the target calculus.

5.1.3 Formal properties of the restrictions

Lemma 5.1. *Assume that all rules in $\mathcal{R}$ satisfy the conditions given above. If P is well scoped and $\mathcal{R} \vdash \mathsf{P} \hookrightarrow \mathsf{Q}$, then Q is also well scoped.* □

Lemma 5.2. *Assume that $\mathcal{R}$ satisfies the conditions given above and that every message in the image of $\mathcal{S}$ consists of a single name. If P and $\mathcal{S}^{\mathbf{P}}\,\mathsf{P}$ are both well scoped, then $\mathcal{R} \vdash \mathsf{P} \hookrightarrow \mathsf{Q}$ implies $\mathcal{R} \vdash \mathcal{S}^{\mathbf{P}}\,\mathsf{P} \hookrightarrow \mathcal{S}^{\mathbf{P}}\,\mathsf{Q}$.* □

5.2 Sequenced message types

The only forms of message types introduced in Section 3.1 were $\{x\}$ and $\{f_1, \ldots, f_k\}*$. Our system also supports a third kind of message type

$$\begin{array}{rl} \text{Message types: } \mu ::= & \cdots \\ & \mid \ f_1.f_2.\cdots.f_k \end{array}$$

which describes messages that contain exactly the given sub-forms in the given order (except that like $\{\cdots\}*$ it does not describe a lone name). We will call it a **sequenced message type**.

Sequenced message types are more precise than $\{\cdots\}*$ and tend to appear much more often in the typings of example terms. We have postponed them until now because their interaction with type substitution is technically complex; defining it properly in Section 3.2 would have made understanding more difficult than it needs to be.

Part of the difficulty is that the type inference algorithm in Section 4 depends on the property that only finitely many different form types can be constructed given a finite set of names and an upper bound on the number of elements in a form. In order to provide this, we need to add an extra syntactic restriction:

A sequenced message type must not contain two identical sub-forms.

The rule for matching a sequenced message type is simple:

$$\frac{M \notin \boxed{x} \qquad M_* 0 = f_1.f_2 \ldots f_k.0}{\vdash M : f_1.\cdots.f_k}$$

and it is also easy to add a case for sequenced message types to Definition 3.6 (of a flow-closed shape graph):

- When $\mathcal{T}^{\mathbf{F}}\,\varphi = f_1.\cdots.f_k$, the graph must contain $Z_0 \xrightarrow{f_1} Z_1 \xrightarrow{f_2} \cdots \xrightarrow{f_k} Z_k$ and $Y \dashrightarrow\!\!\!\!\boxed{\mathcal{T}}\!\!\!\!\rightarrow Z_k$ for some Z_1 to Z_k.

The complex part is that the computation of $\mu *_* \mu$ and $\mathcal{T}^{\mathbf{M}}\,\mu$ needs to be redefined. Here it is not enough simply to add cases for sequenced messages in the case analyses – for example $\{a\}*\{b\}$ must now be the sequenced type $a.b$ in order to satisfy the universal property in Definition 3.1 (previously $\{a\}*\{b\}$ was $\{a, b\}*$).

Definition 5.3. Define the message-type **promotions** $\uparrow\mu$ and $\Uparrow\mu$ by

$$\begin{aligned} \uparrow\{f_1, \ldots, f_k\}* &= \{f_1, \ldots, f_k\}* \\ \uparrow(f_1.\cdots.f_k) &= f_1.\cdots.f_k \\ \uparrow\{x\} &= x \end{aligned}$$

$$\begin{aligned} \Uparrow\{f_1, \ldots, f_k\}* &= \{f_1, \ldots, f_k\}* \\ \Uparrow(f_1.\cdots.f_k) &= \{f_1, \ldots, f_k \text{ in a canonical order}\}* \\ \Uparrow\{x\} &= \{x\}* \end{aligned}$$

□

Lemma 5.4. *These are the properties of the promotions:* $\uparrow\mu$ *is the least message type whose meaning includes* $\{\, 0.M \mid M \in [\![\mu]\!] \,\}$ *and* $\Uparrow\mu$ *is the least message type* ***of the form*** $\{\cdots\}*$ *whose meaning includes* $\{\, 0.M \mid M \in [\![\mu]\!] \,\}$. □

Here is the new computation of $\mu_1 *\mu_2$, which satisfies Definition 3.1:

If $\uparrow\mu_1$ and $\uparrow\mu_2$ both have the shape $f_1 \dots f_k$ and there are no duplicated sub-forms, then $\mu_1 *\mu_2$ is simply the concatenation of $\uparrow\mu_1$ and $\uparrow\mu_2$. Otherwise it is $\{\vec{f}\}*$ where $\vec{f}$ is the canonical order of the sub-forms in $\Uparrow\mu_1$ and $\Uparrow\mu_2$.

The new definition of $\mathcal{T}^{\mathbf{M}}\,\mu$ is:

$$\begin{aligned}
\mathcal{T}^{\mathbf{M}}\{f_1,\dots,f_k\}* &= \Uparrow\big(\mathcal{T}^{\mathbf{F}}\,f_1 *\cdots *\mathcal{T}^{\mathbf{F}}\,f_k\big)\\
\mathcal{T}^{\mathbf{M}}\,(f_1.\cdots.f_k) &= \uparrow\big(\mathcal{T}^{\mathbf{F}}\,f_1 *\cdots *\mathcal{T}^{\mathbf{F}}\,f_k\big)\\
\mathcal{T}^{\mathbf{M}}\{x\} &= \begin{cases}\mathcal{T}(x) & \text{if } x \in \operatorname{Dom}\mathcal{T}\\ \{x\} & \text{otherwise}\end{cases}
\end{aligned}$$

Finally, beware that in the presence of sequenced message types, the $\approx$ relation from Definition 4.1 is not transitive. For example, <a.b> $\approx$ <{a, b}*> $\approx$ <b.a> but <a.b> $\not\approx$ <b.a>. The inference algorithm does not depend on transitivity of $\approx$.

5.3 Name restriction

The main problem with name restriction is that its position in the process term is not fixed, so we cannot treat it like just another form. Furthermore, a name restriction can be replicated by a ! and extrude to encompass its own original, by the equivalence

$$\begin{aligned}
!\nu(x).P &\equiv \nu(x).P \mid\ !\nu(x).P\\
&\equiv \nu(y).[x \mapsto y]^{\mathbf{P}}\,P \mid\ !\nu(x).P\\
&\equiv \nu(y).([x \mapsto y]^{\mathbf{P}}\,P \mid\ !\nu(x).P)
\end{aligned}$$

This shows that we need to support name restriction in such a way that a shape predicate does not place an upper limit on the number of different ν-bound names that can be in scope in any part of the term — otherwise shape predicates would not respect the structural congruence relation. Therefore, the *same* name in the shape predicate must be able to match several different names in the actual term if they are bound by (perhaps nested) name restrictions.

A first sketch of a matching rule that achieves this is:

$$\frac{\vdash [x \mapsto y]^{\mathbf{P}}\,P : \pi}{\vdash \nu(x).P : \pi}$$

where y is arbitrary! This lets us choose the same y for several ν bindings in the term, such that they can match the same part of the shape predicate. However, for many practical purposes, this turns out to be too liberal — it leaves too little trace of what happened. Instead we require that y is chosen from a set B that comes with the shape predicate. We define a syntax for this:

$$\begin{aligned}
&\text{Name sets:} && B \in \mathcal{P}_{\text{fin}}(\boxed{x})\\
&\text{Guarded shape predicates:} && \Pi ::= \pi/B
\end{aligned}$$

and also an auxiliary predicate $P \overset{B}{\curvearrowright} Q$ which says that Q arises by replacing each private name in P by a member of the set of names B:

$$\frac{y \in B \qquad [x \mapsto y]^{\mathbf{P}}\, P \overset{B}{\curvearrowright} Q}{\nu(x).P \overset{B}{\curvearrowright} Q} \qquad\qquad \frac{P \overset{B}{\curvearrowright} Q \qquad P' \overset{B}{\curvearrowright} Q'}{P \mid P' \overset{B}{\curvearrowright} Q \mid Q'}$$

$$\frac{}{0 \overset{B}{\curvearrowright} 0} \qquad\qquad \frac{P \overset{B}{\curvearrowright} Q}{!P \overset{B}{\curvearrowright} !Q} \qquad\qquad \frac{P \overset{B}{\curvearrowright} Q}{F.P \overset{B}{\curvearrowright} F.Q}$$

The meaning of a Π is now given by the following rule:[10]

$$\frac{(FN(P) \cup BN(P)) \cap B = \varnothing \qquad P \overset{B}{\curvearrowright} Q \qquad \vdash Q : \pi}{\vdash P : \pi/B}$$

With this design we know that if a process term contains a form that matches the form type $x\ y$ in the shape predicate and x and y are different names, then the two names in the term are genuinely different. If $x = y$ then the names in the term may or may not be the same; only if $x = y$ and is not a member of the B component of the shape predicate can we be sure that they *must* be the same.

Subject reduction for Poly$\star$ with name restriction reduces to subject reduction in the ν-free system:

Proposition 5.5. *If π is semantically closed with respect to a ruleset $\mathcal{R}$, then $[\![\pi/B]\!]$ is closed under $\mathcal{R}$-reductions, too.* □

5.3.1 Type inference for name restriction

Type inference for terms with name restriction in them turns out to be easy: One can just ignore the restriction operators, provided that one first α renames them to all bind different names:

Theorem 5.6. *Let Q be produced by P by first α-renaming all name restrictions in P such that each of them bind a unique name and then erasing all of the $\nu(x)$ operators. Let B_0 be the set of unique names bound by the erased ν's. Now, whenever $\vdash P : \langle G \mid X\rangle/B$, there is a G' such that $\vdash Q : \langle G' \mid X\rangle$ and $P \in [\![\langle G' \mid X\rangle/B_0]\!] \subseteq [\![\langle G \mid X\rangle/B]\!]$. Furthermore, if G is flow closed and syntactically closed, then G' will be, too.* □

This theorem reduces our task to finding types for the ν-free term Q: No type for P is more precise than those that can be found by typing Q and then adding B_0 to the type. We omit the proof, which is completely analogous to the corresponding property for PolyA [3, Prop. E.5].

5.4 How to recognise types

Given $\mathcal{U}_0$, π, and a $\underline{P}$ that satisfies the left-hand-side restrictions from Section 5.1.1, it is easy to compute all the $\mathcal{U}$ that extend $\mathcal{U}_0$ such that $\mathcal{U} \models_L \underline{P} : \pi$. (We

[10] The requirement that none of the names in B appear in P is necessary for principal typings to exist.

are not interested in extensions where $\mathrm{Dom}\,\mathcal{U} \setminus \mathrm{Dom}\,\mathcal{U}_0$ contains a variable that does not occur in $\underline{P}$ at all). The computation is by recursion over the structure of $\underline{P}$; in the case of $\underline{P}_1 \mid \underline{P}_2$ each output of the matching of $\underline{P}_1$ is used as $\mathcal{U}_0$ in the matching of $\underline{P}_2$. The other cases are straightforward.

If we set $\mathcal{U}_0 = \varnothing$, this matching operation allows us to compute $\mathsf{active}_{\mathcal{R}}(\pi)$ for any π and $\mathcal{R}$. To check that G is locally closed at an active node X, we use the same matching operation, still with $\mathcal{U}_0 = \varnothing$, on each **reduce** LHS in $\mathcal{R}$ in turn, and then check that the RHS is satisfied. The latter is straightforward since the entire judgement $\mathcal{U} \vDash_{\mathsf{R}} \underline{P} : \pi$ is known now.

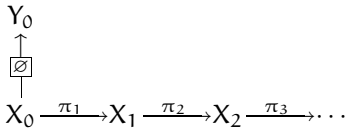
In the case for $\underline{F}.\underline{P}$ there may be more than one Y that applies. Then they must all be checked in sequence until one of them matches, which may cause a small combinatorial explosion. But real-world reduction rules are usually shallow enough that this is not a problem. If the type satisfies the width restriction, there *is* only one Y to check, eliminating the problem completely.

It is equally straightforward to test whether G is flow closed given that we already know how to compute $\mathcal{T}^{\mathbf{F}}\,\varphi$.

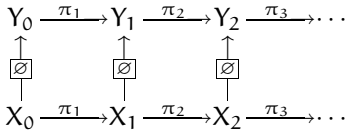
Thus, we can algorithmically recognise whether π is a type when its components are given explicitly.

5.5 An optimisation: Target borrowing

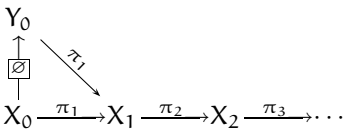
A special optimisation in the type inference algorithm applies to subtyping edges (i.e., flow edges labelled with the trivial substitutions $\varnothing$). Consider the shape graph fragment



The usual procedure for flow closing it would result in

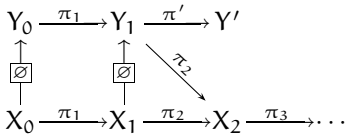


because $\varnothing^{\mathbf{F}}\,\pi_i = \pi_i$. This is wasteful because, all other things being equal, Y_1 will have exactly the same meaning as X_1 and so forth. Therefore the implementation by default produces



We call the edge $Y_0 \xrightarrow{\pi_1} X_1$ a **borrowing** edge, because it borrows its target from the underlying $X_0 \xrightarrow{\pi_1} X_1$. The implementation remembers which edges

are borrowing, because in certain circumstances they may need to be made into real edges with their own targets. This happens, for example, if a closure rule firing at Y_0 wants the path $Y_0 \xrightarrow{\pi_1} \cdot \xrightarrow{\pi'} \cdot$ to exist but there is not already a π' edge from X_1. In that case the borrowing edge must be lifted to produce



because a principal typing should not contain $X_0 \xrightarrow{\pi_1} \cdot \xrightarrow{\pi'} \cdot$. Similar care must be taken if a closure rule applies at X_2, but X_2 is only active because of a borrowing edge. (It takes quite unusual **active** rules to achieve that, but it can happen). In that case enough borrowing edges must be lifted that the closure rule can apply at Y_2 and X_2 can be left alone.

The principal typings property (Thm. 4.4) can be extended to deal with borrowing edges, if each "$\sigma(Y)$" in Defn. B.1 is replaced by "some Z_k such that G' contains $\sigma(Y) = Z_1 \multimap \cdots \multimap Z_k$", but we omit the details.

5.6 Recovering precision with marks

The width and depth restrictions are essential to the termination of the type inference, but they also limit the precision of the analysis in sometimes unwanted ways. This is most often encountered for the width restriction. For example, when the Mobile Ambients term

```
a[b[<in c>]] | b[in a | (x).out x]
```

is analysed, the width restriction will force the contents of the two b ambients inside a to be described by the same node in a shape graph. This will lead the analysis to conclude that a communication takes place and produces out •, though it is entirely clear to a *human* observer that the two b's are and will stay separate.

Our implementation includes a feature designed to mitigate this. Inspired (via [6]) by the flow labels of [17], we annotate each form (and sub-form) type in the shape graph with an (invisible by default) **mark**. The $\approx$ relation now relates form types only if they have the same mark. Thus, with marks, the width restriction does allow two different $\xrightarrow{\texttt{b[]}}$ edges from the same node, as long as they have different marks. Before the initial shape graph is closed, all forms in it are given different marks. No new marks are invented during closure; therefore the mark-aware width and depth restrictions will still be able to guarantee termination in finite time.

The matching of templates in **active** rules and left-hand sides of **reduce** rules ignores the marks. On the other hand, since fresh marks cannot be invented during the closure computation, the marks on the edges requested by the right-hand side of a **reduce** rule must be taken from the left-hand side.

When a reduction rule is first parsed, the implementation permanently decides which template form on the LHS will supply the mark for each template form on the RHS. This decision uses a **mark transfer** heuristics that favours identical template forms, giving the illusion that the mark stays attached to a moving form. If there is no identical form to take the mark from, the heuristics attempts to find a form that at least shares a $\mathring{x}$ or a keyword with the one being built. It is a subject for future work to identify good heuristics for selecting marks in reducts; for the time being we provide syntax to overwrite the default heuristics if it gives unwanted results.

The beauty of this scheme is that if the marks are removed from the shape graph, a perfectly good ordinary Poly$\ast$ type remains,[11] and our ordinary subject reduction result applies. It is not necessary to build a separate theory about marked *terms* and how they reduce. Such a theory could, of course, be built (it would not differ much from Meta$\ast$ with an extra mark pseudo-element added to each form), and might be used as a justification for applications that distinguish between marks in the types – although it is not evident that one can learn more from the marks than from the flow edges also contained in the type.

The reasoning behind Theorem 4.4 can be applied in the marked setting and shows that the marked types produced by the implementation are principal among all marked types that satisfy the marked width and height restriction, provided those other types are locally closed according to the same mark transfer rule as the implementation uses.

6 Conclusion

6.1 Summary of contributions

This paper makes novel and significant contributions beyond what is in the previous literature, as follows:

1. Meta$\ast$ is a syntactic framework that can be instantiated into a large family of mobile process calculi by supplying reduction rules.
2. The *generic type system* Poly$\ast$ works for any instantiation of Meta$\ast$. We have checked that it works for π-calculus, a large number of ambient calculi, and a version of the Seal calculus.
3. Poly$\ast$ has *subject reduction*. Also, given a process term P and a shape predicate π, one can decide by checking purely local conditions whether π is a *type*, and it is similarly decidable whether P *matches* π. Thus, it is decidable whether a process belongs to a specific type.
4. Poly$\ast$ supports a notion of *spatial polymorphism* that achieves what Cardelli and Wegner [12] called "the purest form of polymorphism: the same object or function can be used uniformly in different type context without

[11] One exception is if the marked type contained a message type of the form $f_1.\cdots.f_k$ where two of the f_i's were identical except for their mark. In that case the mark-less version will not be a valid Poly$\ast$ message type at all. We provide an option for purists that makes the implementation ignore marks when deciding whether a message type is syntactically valid.

changes, coercions or any kind of run-time tests or special encodings of representations".

5. The types of Poly✶ are sufficiently precise that many interesting *safety/ security properties* can be checked, especially those that can be formulated as questions on the possible configurations that can arise at run-time.
6. For the subsystem of Poly✶ satisfying the *width* and *depth* restrictions, there is a type inference algorithm (which we have implemented) that always successfully infers a *principal type* for any process term. This means that Poly✶ has the potential for *compositional analysis* where the pieces of a system can be analysed independently and the analysis results for the pieces can be combined without ever needing to reinspect the pieces.
7. Our approach to type inference contains new techniques such as *target borrowing* for making implementations of this new kind of type inference practical.
8. Our approach to type inference uses *marks* to go beyond what the width and depth restriction would otherwise allow and regain part of the full power of Poly✶.

6.2 Related work

Another generic type system for process calculi was constructed by Igarashi and Kobayashi [18]. Like the shape predicates in Poly✶, their types look like process terms and stand for sets of structurally similar processes. Beyond that, however, their focus is different from ours. Their system is specific to the π-calculus and does not handle locations or ambient-style mobility. On the other hand, it is considerably more flexible than Poly✶ within its domain and can be instantiated to do such things as deadlock and race detection which are beyond the capabilities of Poly✶.

Yoshida [31] used graph types much like our shape predicates to reason about the order of messages exchanged on each channel in the π-calculus. Since this type system reasoned about *time* rather than *location*, it is not directly comparable to Poly✶, despite the rather similar type structure.

References

[1] M. Abadi, A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inform. & Comput.*, 148(1), 1999.

[2] T. Amtoft, H. Makholm, J. B. Wells. PolyA: True type polymorphism for Mobile Ambients. In J.-J. Levy, E. W. Mayr, J. C. Mitchell, eds., *Exploring New Frontiers of Theoret. Informatics: IFIP 18th World Comput. Congress – TC1 3rd Int'l Conf. Theoret. Comput. Sci. (TCS '04)*. Kluwer Academic Publishers, 2004.

[3] T. Amtoft, H. Makholm, J. B. Wells. PolyA: True type polymorphism for Mobile Ambients. Technical Report HW-MACS-TR-0015, Heriot-Watt Univ., School of Math. & Comput. Sci., 2004.

[4] T. Amtoft, F. Turbak. Faithful translations between polyvariant flows and polymorphic types. In *Programming Languages & Systems, 9th European Symp. Programming*, vol. 1782 of *LNCS*. Springer-Verlag, 2000.

[5] G. Boudol. The π-calculus in direct style. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997.
[6] C. Braghin, A. Cortesi, S. Filippone, R. Focardi, F. L. Luccio, C. Piazza. BANANA: a tool for Boundary Ambients Nesting ANAlysis. In *Tools and Algorithms for the Construction and Analysis of Sytstems, TACAS'03*, vol. 2619 of *LNCS*. Springer-Verlag, 2003.
[7] M. Bugliesi, G. Castagna, S. Crafa. Boxed ambients. In *4th International Conference on Theoretical Aspects of Computer Science (TACS'01)*, vol. 2215 of *LNCS*. Springer-Verlag, 2001.
[8] M. Bugliesi, S. Crafa, M. Merro, V. Sassone. Communication interference in mobile boxed ambients. In *FST & TCS 2002*, 2002.
[9] M. Bugliesi, S. Crafa, A. Prelic, V. Sassone. Secrecy in untrusted networks. In *Proc. 30th Int'l Coll. Automata, Languages, and Programming*, vol. 2719 of *LNCS*. Springer-Verlag, 2003.
[10] L. Cardelli, G. Ghelli, A. D. Gordon. Mobility types for mobile ambients. In J. Wiedermann, P. van Emde Boas, M. Nielsen, eds., *ICALP'99*, vol. 1644 of *LNCS*. Springer-Verlag, 1999. Extended version appears as Microsoft Research Technical Report MSR-TR-99-32, 1999.
[11] L. Cardelli, A. D. Gordon. Mobile ambients. In M. Nivat, ed., *FoSSaCS'98*, vol. 1378 of *LNCS*. Springer-Verlag, 1998.
[12] L. Cardelli, P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), 1985.
[13] G. Castagna, G. Ghelli, F. Z. Nardelli. Typing mobility in the Seal calculus. In K. G. Larsen, M. Nielsen, eds., *CONCUR*, vol. 2154 of *LNCS*. Springer-Verlag, 2001.
[14] S. Chaki, S. K. Rajamani, J. Rehof. Types as models: Model checking message-passing programs. In *Conf. Rec. POPL '02: 29th ACM Symp. Princ. of Prog. Langs.*, 2002.
[15] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, I. Salvo. M3: Mobility types for mobile processes in mobile ambients. In *CATS 2003*, vol. 78 of *ENTCS*, 2003.
[16] A. D. Gordon, A. S. A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoret. Comput. Sci.*, 300(1–3), 2003.
[17] R. R. Hansen, J. G. Jensen, F. Nielson, H. R. Nielson. Abstract interpretation of mobile ambients. In *Proc. 6th Int'l Static Analysis Symp.*, vol. 1694 of *LNCS*. Springer-Verlag, 1999.
[18] A. Igarashi, N. Kobayashi. A generic type system for the pi-calculus. In *Conf. Rec. POPL '01: 28th ACM Symp. Princ. of Prog. Langs.*, 2001.
[19] F. Levi, C. Bodei. A control flow analysis for safe and boxed ambients. In *Programming Languages & Systems, 13th European Symp. Programming*, vol. 2986 of *LNCS*. Springer-Verlag, 2004.
[20] F. Levi, D. Sangiorgi. Controlling interference in ambients. In *POPL'00, Boston, Massachusetts*. ACM Press, 2000.
[21] S. Maffeis. Sequence types for the π-calculus. In *Intersection Types and Related Systems*, 2004. To appear in ENTCS.
[22] R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Inform. & Comput.*, 100(1), 1992.
[23] H. R. Nielson, F. Nielson, H. Pilegaard. Spatial analysis of BioAmbients. In R. Giacobazzi, ed., *Static Analysis: 11th Int'l Symp.*, vol. 3148 of *LNCS*, Verona, Italy, 2004. Springer-Verlag.
[24] J. Palsberg, C. Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Funct. Programming*, 11(3), 2001.
[25] S. M. Pericas-Geertsen. *XML-Fluent Mobile Ambients*. PhD thesis, Boston University, 2001.
[26] I. Phillips, M. G. Vigliotti. On reduction semantics for the push and pull ambient calculus. In R. A. Baeza-Yates, U. Montanari, N. Santoro, eds., *Theoretical Computer Science: 2nd IFIP Int'l Conf.*, vol. 223 of *IFIP Conference Proceedings*. Kluwer, 2002.
[27] G. Sander. Graph layout through the VCG tool. In R. Tamassia, I. G. Tollis, eds., *Graph Drawing: DIMACS International Workshop, GD '94*, vol. 894 of *LNCS*. Springer-Verlag, 1994.
[28] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995. Report no ECS-LFCS-96-345.
[29] J. Vitek, G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, vol. 1686 of *LNCS*. Springer-Verlag, 1999.
[30] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, vol. 2380 of *LNCS*. Springer-Verlag, 2002.
[31] N. Yoshida. Graph types for monadic mobile processes. In *Foundations of Software Technology and Theoret. Comput. Sci., 16th Conf.*, vol. 1180 of *LNCS*. Springer-Verlag, 1996.

A Proof sketch for the subject reduction result

Definition A.1. Write $G \vdash \mathcal{V} : \mathcal{U}$ iff all of the following hold:

1. $\mathrm{Dom}\ \mathcal{V} = \mathrm{Dom}\ \mathcal{U}$
2. $\mathcal{V}(\mathring{x}) = \mathcal{U}(\mathring{x})$ for all relevant $\mathring{x}$.
3. $\vdash \mathcal{V}(\mathring{m}) : \mathcal{U}(\mathring{m})$ for all relevant $\mathring{m}$.
4. $\vdash \mathcal{V}(\mathring{p}) : \langle G \,|\, \mathcal{U}(\mathring{p})\rangle$ for all relevant $\mathring{p}$. □

Proposition A.2. *Let the shape predicate $\pi = \langle G \,|\, X_0\rangle$ be given.*

a. Assume that $\underline{P}$ satisfies the restrictions that apply to left-hand-side templates. Let $\mathcal{V}$ be given, and assume that $\vdash \mathcal{V}^{\mathbf{P}}\,\underline{P} : \pi$. Then there is a $\mathcal{U}$ such that $G \vdash \mathcal{V} : \mathcal{U}$ and $\mathcal{U} \models_{\mathsf{L}} \underline{P} : \pi$.

b. If G is flow closed and $\mathcal{U} \models_{\mathsf{R}} \underline{P} : \pi$, then for every $\mathcal{V}$ such that $G \vdash \mathcal{V} : \mathcal{U}$, it holds that $\vdash \mathcal{V}^{\mathbf{P}}\,\underline{P} : \pi$. □

Proof of Thm. 3.12. By an easy induction on the derivation of $\mathcal{R} \vdash P \hookrightarrow Q$, using Proposition A.2 for the cases that depend on $\mathcal{R}$. □

B Proof sketch for principality of inferred typings

Definition B.1. A **simulation** from $\pi = \langle G \,|\, X_0\rangle$ to $\pi' = \langle G' \,|\, X_0'\rangle$ is a finite map σ from node names to node names such that

1. $\sigma(X_0) = X_0'$
2. For all $(X \xrightarrow{\varphi} Y) \in G$, there is $(\sigma(X) \xrightarrow{\varphi'} \sigma(Y)) \in G'$ such that $[\![\varphi]\!] \subseteq [\![\varphi']\!]$.
3. For all $(X \dashrightarrow[\mathcal{T}] Y) \in G$ there is $(\sigma(X) \xrightarrow{\mathcal{T}'} \sigma(Y)) \in G'$ such that $\mathrm{Dom}\ \mathcal{T} = \mathrm{Dom}\ \mathcal{T}'$ and $[\![\mathcal{T}(x)]\!] \subseteq [\![\mathcal{T}'(x)]\!]$ for all relevant x. □

It is clear that the existence of a simulation from π to π' implies $[\![\pi]\!] \subseteq [\![\pi']\!]$.

Definition B.2. The shape predicate π is **pre-principal** for P if $\vdash P : \pi$ and for each restricted *type* π' such that $\vdash P : \pi'$, there is a simulation from π to π'. □

If π is both pre-principal for P and itself a restricted type, then it is evidently a **principal typing** [30] for P.

Lemma B.3. *Each step of the type inference for ν-free terms preserve pre-principality of the shape predicate being closed.* □

Proof. It is clear that neither the addition of edges and nodes nor unification of nodes can cause $\vdash P : \pi$ to stop holding. Assume therefore that π' is a restricted type for P; we must show that the existence of a simulation to π' is preserved.

If the type inference step consists of adding edges that are necessary for π to be closed, one easily sees that π' must already contain the corresponding edges, because it is assumed to be a type. Therefore we can readily extend the simulation with mappings from the new nodes in π to the corresponding points in π'.

If the step consists of unifying two nodes because the width or depth restriction demands it, the edges that violate the restriction in π must have counterparts in π'. Because π' is assumed to satisfy the restrictions (and because a simulation maps $\approx$-related form types to $\approx$-related form types), the simulation must already map the nodes in π that are being unified to a single node in π'. Thus the simulation can be preserved across the unification, as required. □

Lemma B.4. *The initial shape predicate, corresponding directly to* P*'s syntax tree, is pre-principal.* □

This is immediate. The two lemmas lead us to the principality result.

Notice that the proof does not depend on the type inference always unifying nodes *immediately* when it is necessary to uphold the width and depth restrictions. Our implementation exploits this by using a weakened version of the depth restrictions which is cheaper to compute and still guarantees termination. The full depth restriction is only checked when the shape predicate is otherwise fully closed.