

Type Inference for PolyA *

Henning Makhholm

<http://www.macs.hw.ac.uk/~makholm/>
Heriot-Watt University

J. B. Wells

<http://www.macs.hw.ac.uk/~jbw/>
Heriot-Watt University

Heriot-Watt University
School of Mathematical and Computing Sciences
Technical Report HW-MACS-TR-0013
January 2004

Abstract

We present an automatic type inference algorithm for PolyA, a type system for Mobile Ambients presented in earlier work by us together with Torben Amtoft. We present not only a basic inference algorithm, but also several optimizations to it aimed at reducing the size of the inferred types. The final algorithm has been implemented and verified to work on small examples. We discuss some small problems that still exist and methods for solving them.

1 Introduction

This report is intended to be part of a DART project deliverable and reports on work done as of 2003-12-31. We expect to do some further work and to make available early in 2004 an extended and revised version of this document.

PolyA is a novel type system paradigm for Mobile Ambients (Cardelli and Gordon, 1998), presented by the authors and Torben Amtoft in (Amtoft et al., 2003).

Amtoft et al. (2003) defined the general system of PolyA types and described a restricted class of PolyA such that that each term provably has a maximally precise (or *principal*) type within that class. Although we proved the existence of principal types for this restriction of PolyA, the proof is non-constructive and does not give an algorithm for finding principal types (or any types).

*Partially supported by EC FP5 grant IST-2001-33477, EPSRC grant GR/L 41545/01, NSF grants 9806745 (EIA), 9988529 (CCR), and 0113193 (ITR), and Sun Microsystems equipment grant EDUD-7826-990410-US.

In this report we pursue the more practical goal of automatically producing sufficiently precise types for showing interesting properties of process terms. We describe the principles behind a prototype type inference algorithm that we have constructed, and show examples of its output.

Our motivation for doing this is twofold: First, our implementation serves as a proof of concept: It shows that it is possible and realistic to use PolyA (or a system like it) for automatic reasoning about mobility calculi. Second, we plan to use the implementation itself as a vehicle for further research about the practical properties of PolyA. Having an automatic generator of types for arbitrary terms will be essential for investigating how to present types in an easily understood way, trade-offs between precise types and small/readable ones, modularity of type inference, et cetera.

At the current state of the research, we do not yet try to show that the types we construct are principal (for some particular restricted version of PolyA) and the algorithms we describe are not compositional. We plan further work to construct modular versions of the analysis.

The rest of the report is structured as follows: Section 2 introduces the basic PolyA system. Section 3 describes the basic assumptions of our type inference. Section 4 develops a basic inference algorithm in a limited case. Section 5 describes a number of optimisations with respect to the basic algorithm. Section 6 collects our complete current, optimized algorithm in one place. Section 7 reports current implementation status and briefly discusses further work.

2 PolyA in brief

We assume familiarity with the ambient calculus in general, but in order to make the report reasonably self-contained begin by reproducing the description of PolyA from Amtoft et al. (2003).

(This entire section consists of extracts from Amtoft et al. (2003), slightly edited to remove concepts not relevant to our inference algorithms).

2.1 Overview

The basic concept in PolyA is that of a *shape predicate*. The actual definition of this is somewhat involved, partly due to the need of handling communication, so let us introduce the concept gently with a toy example where the only capability is “in”:

Toy shape predicates: $\sigma ::= 0 \mid (\sigma \mid \sigma) \mid a[\sigma] \mid \text{in } a$

The meaning of a shape predicate is a set of terms, given by the following matching relation:

$$\frac{\vdash P : \sigma}{\vdash a[P] : (\dots \mid a[\sigma] \mid \dots)} \qquad \frac{\vdash P : \sigma \quad \vdash Q : \sigma}{\vdash P \mid Q : \sigma}$$

$$\frac{}{\vdash \text{in } a.0 : (\dots \mid \text{in } a \mid \dots)} \qquad \frac{}{\vdash 0 : \sigma}$$

(These rules are actually true in our full theory, but only the ones on the right are primitive). With these rules we can derive the judgement $\vdash P_0 : \sigma_0$, where

$$P_0 = a[\text{in } b.0 \mid \text{in } c.0] \mid b[d[\text{in } a.0]] \mid c[e[\text{in } a.0]]$$

$$\sigma_0 = \mathbf{a}[\mathbf{in\ b\ |\ in\ c}] \mid \mathbf{b}[\mathbf{d}[\mathbf{in\ a}]] \mid \mathbf{c}[\mathbf{e}[\mathbf{in\ a}]]$$

But we can also derive, say, $\vdash \mathbf{a}[\mathbf{in\ b.0}] \mid \mathbf{a}[\mathbf{in\ c.0}] : \sigma_0$ — the matching rules do not care that the \mathbf{b} and \mathbf{c} on the top level are missing, nor that the $\mathbf{a}[\mathbf{in\ b\ |\ in\ c}]$ part of the shape predicate is used twice.

PolyA *types* are shape predicates such that the set of terms matching a type is closed under reduction. The shape predicate σ_0 above is not a type, because $P_0 \leftrightarrow P_1 = \mathbf{b}[\mathbf{a}[\mathbf{in\ c.0}] \mid \mathbf{d}[\mathbf{in\ a.0}]] \mid \mathbf{c}[\dots]$ yet $\not\vdash P_1 : \sigma_0$. One type that P_0 does have is

$$\begin{aligned} \sigma_1 = & \mathbf{a}[\mathbf{in\ b\ |\ in\ c}] \mid \mathbf{b}[\mathbf{a}[\mathbf{in\ b\ |\ in\ c\ |\ d}[\mathbf{in\ a}]] \mid \mathbf{d}[\mathbf{in\ a}]] \\ & \mid \mathbf{c}[\mathbf{a}[\mathbf{in\ b\ |\ in\ c\ |\ e}[\mathbf{in\ a}]] \mid \mathbf{e}[\mathbf{in\ a}]] \end{aligned}$$

The $\mathbf{a}[\dots]$ predicate inside \mathbf{b} still allows the $\mathbf{in\ b}$. This must be so because shape predicates do not care about the number of identical items (unlike what is the case in Teller et al. (2002)), so one of the terms matched by σ_1 is $\mathbf{a}[\mathbf{in\ b.0\ |\ in\ b.0}] \mid \mathbf{b}[0]$, which reduces to $\mathbf{b}[\mathbf{a}[\mathbf{in\ b}]]$.

A more subtle point about σ_1 is that it *disallows* having an \mathbf{e} inside an \mathbf{a} inside a \mathbf{b} or a \mathbf{d} inside an \mathbf{a} inside a \mathbf{c} . This example therefore illustrates the most basic kind of polymorphism possible: The same initial \mathbf{a} ambient can evolve differently in different possible futures, and the type system can prove that those different futures do not interfere with each other.

This is the way polymorphism works in PolyA: An ambient can start out with a very small type such as $\mathbf{a}[\mathbf{in\ b\ |\ in\ c}]$, and then when it chooses to move into \mathbf{b} rather than \mathbf{c} , it may change its type to a supertype, $\mathbf{a}[\mathbf{in\ b\ |\ in\ c\ |\ \dots}]$. Seen from the point of the moving ambient, its type has evolved — though of course the new type has been present from the beginning somewhere in the overall type of the entire computation.

PolyA lets any supertype (i.e., a type that is matched by a larger set of terms) be used as a “polymorphic variant” if it appears in the right place of the overall typing. The overall typing contains all of the polymorphic variants that will ever be needed for each ambient in the particular context it is being typed in.

Other features of PolyA are as follows. There are singleton types of ambient names and explicit dependencies on communication, as illustrated by this judgement:

$$\vdash (\langle \mathbf{a} \rangle.0 \mid (\mathbf{x}).\mathbf{x}[0]) : (\langle \mathbf{a} \rangle.0 \mid (\mathbf{x}).\mathbf{x}[0] \mid \mathbf{a}[0])$$

Types can be infinitely deep trees, e.g.:

$$\vdash !\mathbf{a}[\mathbf{!in\ a.0}] : \mathbf{letrec\ X} = (\mathbf{a}[\mathbf{X}] \mid \mathbf{in\ a.0}) \mathbf{in\ a}[\mathbf{X}]$$

We only consider types that can be given a finite term representation.

PolyA can optionally track the sequencing of actions, a possibility pioneered by Amtoft et al. (2001, 2002). For example, $\mathbf{a}[\mathbf{in\ b.in\ c.0}] \mid \mathbf{b}[\mathbf{c}[0]] \mid \mathbf{c}[\mathbf{open\ a.0}]$ has a PolyA type proving that \mathbf{a} will never be opened.

2.2 The ambient calculus

The type inference presently handles a calculus without name restriction. We expect the addition of support for a name restriction operator (along the lines presented in the original paper’s appendix) will be simple.

Fig. 1 defines the syntax and semantics of our base calculus. Whenever it has been defined that some (meta)variable letter, say “ x ”, ranges over a given set of objects, the notation \boxed{x} shall mean that set of objects.

The syntactic category of *prefixes* is not in traditional ambient calculus formulations. We treat ambient boundaries and other computational actions uniformly where this makes our theory simpler. Our calculus treats ambient boundaries as capabilities; “**amb** a ” is the capability that creates an ambient named a when executed. In our formulation, an ambient with contents P is written “**amb** $a.P$ ”. The traditional notation “ $a[P]$ ” is syntactic sugar for **amb** $a.P$; we use this whenever convenient. When there is no risk of confusion, we will also write $a[\]$ as syntactic sugar for the capability **amb** a .

The capability **amb** a can in principle be passed in a message. We allow such communication more because it is syntactically convenient than because we expect processes to actually do it.

The parentheses around parallel composition $P \mid Q$ can be omitted when unambiguous. Prefixes bind tighter than parallel composition.

The special capability “ \bullet ” is not supposed to be found in the initial term. It signifies a substitution result that would otherwise be syntactically invalid. For example, the term $\langle \text{in } c \rangle \mid (\text{b}).\text{in } \text{a}.\text{open } \text{b}.0$ reduces to $\text{in } \text{a}.\bullet.0$ instead of the (hypothetical) “ $\text{in } \text{a}.\text{open } (\text{in } c).0$ ”. Traditional ambient calculus accounts usually leave such a communication result undefined, implicitly understanding that the system would crash either at the communication time or when the ill-formed capability executes after the $\text{in } \text{a}$ capability has fired. Our approach supports both views, according to how one interprets \bullet . In either case, it is technically convenient to reify the failure as a special term.

The symbol \bullet does not have any reduction rules associated with it. As far as our theory is concerned it just sits there. Likewise, there are no reduction rules for “placeholder” capabilities of the form “ a ”. In applications, one may or may not want to treat it as an error if such a capability shows up in a place where it wants to be executed. Our type system makes it possible to approximate conservatively *whether* one of these capabilities may occur, but it is up to the type system user to decide which conclusion to draw if they do.

Convention 2.1. A term P is **well-formed** iff its free names are distinct from the names bound by any “ (\vec{a}) ” within the term and it does not contain any nested bindings of the same name. We consider only well-formed terms.

This guarantees that the reduction rules of the ambient calculus will never perform a substitution where there is a risk of name capture. Reductions preserve well-formedness, because the syntax of the calculus prevents substitution from injecting a (\vec{a}) within the body of another (\vec{a}) . (This is in contrast to the λ -calculus, where substitutions routinely insert λ -abstractions into other abstractions).

Because of this, we do not need to recognise α -equivalence for $(\vec{a}).P$. This is a significant technical simplification, because for many purposes we can treat (\vec{a}) as any other action, without needing special machinery for α -equivalence of the bound names. Convention 2.1 does not limit expressiveness. Any program (term) in a more conventional ambient calculus formulation that does recognise α -equivalence has a well-formed α -variant which can be used in our type system.¹

¹If one later discovers that one really wanted to use another variant, for example because

Syntax:

Names: $a, b ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \dots$
 Opcodes: $O ::= \text{in} \mid \text{out} \mid \text{open} \mid \text{amb}$
 Capabilities: $C ::= a \mid Oa \mid \bullet$
 Messages: $M, N ::= C \mid M.N \mid \varepsilon$
 Prefixes: $p ::= M \mid \langle \vec{M} \rangle \mid (\vec{a})$
 Processes: $P, Q, R, S ::= p.P \mid !P \mid (P \mid Q) \mid 0$

See main text for further syntactic restrictions (scoping).

Process equivalence:

$$\begin{array}{c} \overline{P \mid Q \equiv Q \mid P} \quad \overline{P \mid (Q \mid R) \equiv (P \mid Q) \mid R} \quad \overline{0 \mid P \equiv P} \\ \overline{!P \equiv P \mid !P} \quad \overline{!0 \equiv 0} \quad \overline{(M.N).P \equiv M.(N.P)} \quad \overline{\varepsilon.P \equiv P} \quad \overline{P \equiv P} \\ \frac{P \equiv Q}{p.P \equiv p.Q} \quad \frac{P \equiv Q}{P \mid R \equiv Q \mid R} \quad \frac{P \equiv Q}{!P \equiv !Q} \quad \frac{Q \equiv P}{P \equiv Q} \quad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R} \end{array}$$

Substitution:

A **term substitution** \mathcal{S} is a (total) function from names to messages such that $\mathcal{S}(a) \neq a$ for only finitely many a 's. We often notate it $\mathcal{S} = [a_1 \mapsto M_1, \dots, a_k \mapsto M_k]$, understanding implicitly that $\mathcal{S}(a) = a$ when a is not one of the a_i 's. Shorter notations are $[a_i \mapsto M_i]_{1 \leq i \leq k}$ or $[a \mapsto M_a]_{a \in A}$.

For messages: $\mathcal{S}(M.N) = (\mathcal{S}M).(\mathcal{S}N)$ $\mathcal{S}\varepsilon = \varepsilon$
 $\mathcal{S}a = \mathcal{S}(a)$ $\mathcal{S}\bullet = \bullet$
 $\mathcal{S}(Oa) = \begin{cases} O\mathcal{S}(a) & \text{if } \mathcal{S}(a) \text{ is a name} \\ \bullet & \text{otherwise} \end{cases}$

For other prefixes: $\mathcal{S}\langle M_1, \dots, M_k \rangle = \langle \mathcal{S}M_1, \dots, \mathcal{S}M_k \rangle$
 $\mathcal{S}(a_1, \dots, a_k) = \begin{cases} (a_1, \dots, a_k) & \text{if } \mathcal{S}(a_i) = a_i \text{ for all } i \\ \bullet & \text{otherwise} \end{cases}$

For terms: $\mathcal{S}(p.P) = (\mathcal{S}p).(\mathcal{S}P)$ $\mathcal{S}(!P) = !(\mathcal{S}P)$
 $\mathcal{S}(P \mid Q) = (\mathcal{S}P) \mid (\mathcal{S}Q)$ $\mathcal{S}0 = 0$

Reduction rules:

$$\begin{array}{c} \overline{a[\text{in } b.P \mid Q] \mid b[R] \hookrightarrow b[a[P \mid Q] \mid R]} \\ \overline{b[a[\text{out } b.P \mid Q] \mid R] \hookrightarrow a[P \mid Q] \mid b[R]} \quad \overline{a[P] \mid \text{open } a.Q \hookrightarrow P \mid Q} \\ \overline{\langle M_1, \dots, M_n \rangle.P \mid (a_1, \dots, a_n).Q \hookrightarrow P \mid [a_i \mapsto M_i]_{1 \leq i \leq n} Q} \\ \frac{P \hookrightarrow Q}{a[P] \hookrightarrow a[Q]} \quad \frac{P \hookrightarrow Q}{P \mid R \hookrightarrow Q \mid R} \quad \frac{P \equiv Q \quad Q \hookrightarrow R \quad R \equiv S}{P \hookrightarrow S} \end{array}$$

Figure 1: Syntax and semantics of the ambient calculus

Fig. 1 contains no provisions for avoiding name capture in $\mathcal{S}(\vec{a})$ — this is handled by Convention 2.1. The \bullet possibility for $\mathcal{S}(\vec{a})$ is never supposed to be used; substitutions leading to it will not arise by our rules.

Lemma 2.2. *Substitution is compatible with term equivalence: $P \equiv Q$ implies $\mathcal{S}P \equiv \mathcal{S}Q$.*

2.3 Shape predicates

The following pseudo-grammar defines the (abstract) syntax of our type system:

$$\begin{array}{ll}
\text{Message types: } & \mu ::= \{C_1, C_2, \dots, C_k\}^* \quad (k \geq 0) \\
& \quad \quad \quad | \langle C_1.C_2.\dots.C_k \rangle \quad (k \geq 0) \\
& \quad \quad \quad | \{a\} \\
\text{Prefix types: } & \pi ::= C \mid (\vec{a}) \mid \langle \vec{\mu} \rangle \\
\text{Shape predicates: } & \sigma ::= (\pi_1.\sigma_1 \mid \dots \mid \pi_k.\sigma_k) \quad (k \geq 0)
\end{array}$$

Message types μ have an additional syntactic restriction: No message type may mention any capability C more than once. This restricts the expressiveness, but is needed to ensure that only finitely many different message types can be built from a given finite set of available names. The empty shape predicate ($k = 0$) may also be written 0 .

Definition 2.3 (matching of shape predicates). *The following rules define the relations $\vdash M : \mu$, $\vdash p : \pi$, and $\vdash P : \sigma$:*

$$\begin{array}{c}
\frac{M \notin \boxed{a} \quad M.0 \equiv C'_1.\dots.C'_n.0 \quad \{C'_1, \dots, C'_n\} \subseteq \{C_1, \dots, C_k\}}{\vdash M : \{C_1, \dots, C_k\}^*} \text{KleeneStar} \\
\\
\frac{M \notin \boxed{a} \quad M.0 \equiv C_1.\dots.C_k.0}{\vdash M : \langle C_1.\dots.C_k \rangle} \text{Sequenced} \qquad \frac{}{\vdash a : \{a\}} \text{Name} \\
\\
\frac{}{\vdash C : C} \text{Cap} \qquad \frac{}{\vdash (\vec{a}) : (\vec{a})} \text{Recv} \qquad \frac{\vdash M_1 : \mu_1 \quad \dots \quad \vdash M_k : \mu_k}{\vdash \langle M_1, \dots, M_k \rangle : \langle \mu_1, \dots, \mu_k \rangle} \text{Send} \\
\\
\frac{\vdash p : \pi \quad \vdash P : \sigma}{\vdash p.P : (\dots \mid \pi.\sigma \mid \dots)} \text{Pfx} \qquad \frac{\vdash M.(N.P) : \sigma}{\vdash (M.N).P : \sigma} \text{Seq} \qquad \frac{\vdash P : \sigma}{\vdash \varepsilon.P : \sigma} \text{Nop} \\
\\
\frac{\vdash P : \sigma \quad \vdash Q : \sigma}{\vdash P \mid Q : \sigma} \text{Par} \qquad \frac{}{\vdash 0 : \sigma} \text{Null} \qquad \frac{\vdash P : \sigma}{\vdash !P : \sigma} \text{Bang}
\end{array}$$

The side conditions $M \notin \boxed{a}$ and $M.0 \equiv C_1.\dots.C_k.0$ on rules KleeneStar and Sequenced amount to specifying that these two forms of message types are matched modulo associativity of “.” and neutrality of “ ε ” — with the exception that messages that are raw names (i.e., a as opposed to $a.\varepsilon$ or in a) are handled specially. They are matched only by the message type $\{a\}$.

Our language of message types is rather restricted. It might seem more sensible to let a message type be a regular expression over capabilities, and one could easily construct a *sound* type system that way. But our methods for proving *completeness* properties (including principal types) would not work.

the free names are only learned incrementally during compositional analysis, it is a simple matter to correct it retroactively by swapping all instances of one name with a fresh new one, in terms and types alike.

Our message types have been carefully constructed to allow the proofs to go through.

No prefix type matches a prefix of the form M where M is not a naked capability. Such prefixes are handled by rules **Seq** and **Nop**.

Theorem 2.4. *If $P \equiv Q$ then $\vdash P : \sigma \Leftrightarrow \vdash Q : \sigma$ for all σ .*

Definition 2.5. *The **meaning** of a shape predicate (message type, prefix type) is the set of terms (messages, prefixes) that match it:*

$$\llbracket \mu \rrbracket = \{ M \mid \vdash M : \mu \} \quad \llbracket \pi \rrbracket = \{ p \mid \vdash p : \pi \} \quad \llbracket \sigma \rrbracket = \{ P \mid \vdash P : \sigma \}$$

Definition 2.6. *Define **containment** of shape predicates (message types, prefix types) as ordinary subset containment of their meanings:*

$$\mu \leq \mu' \iff \llbracket \mu \rrbracket \subseteq \llbracket \mu' \rrbracket \quad \pi \leq \pi' \iff \llbracket \pi \rrbracket \subseteq \llbracket \pi' \rrbracket \quad \sigma \leq \sigma' \iff \llbracket \sigma \rrbracket \subseteq \llbracket \sigma' \rrbracket$$

Each of the three containment relations is a preorder (transitive and reflexive). They are not antisymmetric, however. For message types and prefix types, antisymmetry could be achieved by identifying message types such as $\{\}^*$ and $\langle \rangle$ that have identical meanings; such pairs can be recognised by simple syntactic conditions. But for shape predicates, there are equivalent pairs such as $\pi_1.\pi_2.0$ and $\pi_1.\pi_2.0 \mid \pi_1.0$ which cannot be identified by any simple syntactic conditions.

2.4 Recursive shape predicates

Our strategy in analysing a term is to look for a shape predicate describing all of its possible computational futures. Because many terms can create arbitrarily deep nestings of ambients (e.g., $!a[!in\ a.0]$), the finite trees we have used for shape predicates so far are not up to the task². We need “infinite” shape predicates. We should, however, restrict ourselves to infinite shape predicates with finite *representations* — in other words, regular trees.

In principle, we could use any of the well-known schemes for expressing regular trees, such as a fix-point operator at the shape-predicate level or (for a more efficient representation) a “letrec” construction with mutual recursion. However, we find it easier, formally as well as intuitively, to view regular trees as *graphs*. Therefore, we replace the abstract syntax for shape predicates with:

$$\begin{array}{ll} \text{Node identifiers: } X, Y, Z ::= \mathbf{x1} \mid \mathbf{x2} \mid \mathbf{x3} \mid \dots & \\ \text{Edges: } e ::= X \xrightarrow{\alpha} Y & \\ \text{Shape graphs: } G \in \mathcal{P}_{\text{fin}}(\mathbb{E}) & \\ \text{Shape predicates: } \sigma ::= \langle X \mid G \rangle & \end{array}$$

A shape predicate is now a shape graph together with a pointer to a distinguished “root” node. The version of the **Pfx** rule that works with this notation is

$$\frac{\vdash p : \pi \quad X \xrightarrow{\alpha} Y \in G \quad \vdash P : \langle Y \mid G \rangle}{\vdash p.P : \langle X \mid G \rangle} \text{Pfx}$$

²This happens even for “harmless” terms such as $b[in\ a.0] \mid a[open\ b.0]$, because shape predicates cannot distinguish them from $!b[!in\ a.0] \mid !a[open\ b.0]$. Thus, nearly every nontrivial use of **open** will need recursive σ 's. As already observed by Cardelli et al. (1999), **open** often complicates analysis significantly.

Thm. 2.4 is still true with this formulation, because it was proven by induction on term the term equivalence rather than shape-predicate structure.

This graph-based formulation is the basis for our formal development. In general, defining some property for shape graphs implicitly defines it for shape predicates: The shape predicate $\langle X | G \rangle$ has the property iff G has.

Proposition 2.7. *The relations of Defn. 2.3 are effectively (and efficiently) decidable when shape predicates are given as graphs.*

2.5 Effective characterisations of containment

Definition 2.6 defined a containment order on shape predicates (and message/prefix types) in an intuitively appealing way, but it did not provide a decision procedure. This subsection develops a more effective characterisation.

Definition 2.8. *Let R be a relation between shape predicates. R is a **shape simulation** iff $\langle X | G \rangle R \langle X' | G' \rangle$ and $X \xrightarrow{\pi} Y \in G$ imply that there is $\pi' \geq \pi$ and Y' such that $X' \xrightarrow{\pi'} Y' \in G'$ and $\langle Y | G \rangle R \langle Y' | G' \rangle$.*

Theorem 2.9. *Shape containment \leq is the largest shape simulation; it is the union of all shape simulations.*

Thus, to prove that $\sigma \leq \sigma'$ it is sufficient to find a shape simulation R such that $\sigma R \sigma'$. And if there is such an R , there is also a small one:

Lemma 2.10. *For any shape graph G , define its **support**:*

$$\overline{G} = \{ \langle X | G \rangle \mid X \xrightarrow{\pi} Y \in G \} \cup \{ \langle Y | G \rangle \mid X \xrightarrow{\pi} Y \in G \}$$

Let G and G' be arbitrary shape graphs and let R be any shape simulation. Then $R \cap (\overline{G} \times \overline{G}')$ is also a shape simulation.

2.6 Type substitutions

Definition 2.11. *A **type substitution** \mathcal{T} is a (total) function from names to message types such that $\mathcal{T}(a) \neq \{a\}$ for only finitely many a 's. Like term substitutions, type substitutions are denoted by $[a_1 \mapsto \mu_1, \dots, a_k \mapsto \mu_k]$ or $[a \mapsto \mu_a]_{a \in A}$.*

Fig. 2 defines type substitution on capabilities, message types, shape graphs and shape predicates. The operation on shape predicates is the important one. Its main property is that if $\vdash P : \sigma$ and \mathcal{S} and \mathcal{T} are term and type substitutions such that $\vdash \mathcal{S}(a) : \mathcal{T}(a)$ for all a , then $\vdash \mathcal{S}P : \mathcal{T}\sigma$.

2.7 Closed shape predicates

Definition 2.12. *The shape predicate σ is **semantically closed** iff its meaning is closed under reduction, i.e., if $\vdash P : \sigma$ and $P \hookrightarrow Q$ imply $\vdash Q : \sigma$.*

This definition is intuitively appealing, but it is not immediately clear how to decide it. However, we have local rules that imply (and are under certain conditions equivalent to) semantic closure:

Definition 2.13. *The shape graph G is **locally closed** at X_0 iff*

Type substitution for capabilities: $\mathcal{T}C$ is a message type, not a capability.

$$\mathcal{T}a = \mathcal{T}(a) \quad \mathcal{T}(Oa) = \begin{cases} \langle Ob \rangle & \text{if } \mathcal{T}(a) = \{b\} \\ \langle \bullet \rangle & \text{otherwise} \end{cases} \quad \mathcal{T}\bullet = \langle \bullet \rangle$$

Substitution for message types: $\mathcal{T}\mu$ is a message type given by:

To compute $\mathcal{T}\{C_1, \dots, C_k\}^*$, let $\mu_i = \mathcal{T}C_i$ for $1 \leq i \leq k$. The result is $\{C'_1, \dots, C'_n\}^*$, where the C'_j 's are all capabilities that occur in any of the μ_i 's, with duplicates removed (and in some canonical order).

To compute $\mathcal{T}\langle C_1 \dots C_k \rangle$, let $\mu_i = \mathcal{T}C_i$ for $1 \leq i \leq k$. If any μ_i has the form $\{a\}$, then replace it by $\langle a \rangle$. If any μ_i has the form $\{\dots\}^*$, or if there is any C that appears in more than one μ_i , then the result is $\mathcal{T}\{C_1, \dots, C_k\}^*$. Otherwise, each μ_i has the form $\langle \dots \rangle$. Concatenate all of these capability lists (in numerical order of the μ_i 's) and return \langle the concatenated list \rangle .

Finally, $\mathcal{T}\{a\}$ is simply $\mathcal{T}(a)$.

Substitution for shape graphs: $\mathcal{T}G$ is a shape graph. To construct $\mathcal{T}G$, first construct an intermediate graph G_ε which can contain special “null edges” written $X \xrightarrow{\varepsilon} Y$. G_ε contains contributions from each edge $Y_1 \xrightarrow{\pi} Y_2 \in G$:

1. When $\pi = a$ and $\mathcal{T}(a) = \{C_1, \dots, C_k\}^*$, choose a fresh node Z , and add to G_ε the following edges:

$$Y_1 \xrightarrow{\varepsilon} Z \xrightarrow{C_1} Z \xrightarrow{C_2} \dots \xrightarrow{C_k} Z \xrightarrow{\varepsilon} Y_2$$

2. When $\pi = a$ and $\mathcal{T}(a) = \langle C_1 \dots C_k \rangle$, choose fresh nodes Z_0 through Z_k , and add to G_ε the edges

$$Y_1 \xrightarrow{\varepsilon} Z_0 \xrightarrow{C_1} Z_1 \xrightarrow{C_2} \dots \xrightarrow{C_k} Z_k \xrightarrow{\varepsilon} Y_2$$

3. When $\pi = a$ and $\mathcal{T}(a) = \{b\}$, add to G_ε the edge $Y_1 \xrightarrow{b} Y_2$.
4. When $\pi = Oa$, $\mathcal{T}(Oa)$ will always have the form $\langle C' \rangle$. Add to G_ε the edge $Y_1 \xrightarrow{C'} Y_2$.
5. When $\pi = (a_1, \dots, a_k)$, check that $\mathcal{T}a_i = \{a_i\}$ for all i , and then add the edge $Y_1 \xrightarrow{\pi} Y_2$ to G_ε . Otherwise, add $Y_1 \xrightarrow{\bullet} Y_2$.
6. When $\pi = \langle \mu_1, \dots, \mu_k \rangle$, add to G_ε the edge $Y_1 \xrightarrow{\langle \mathcal{T}\mu_1, \dots, \mathcal{T}\mu_k \rangle} Y_2$.

Now set $\mathcal{T}G = \{X_k \xrightarrow{\pi} Y \mid (X_k \xrightarrow{\varepsilon} X_{k-1} \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} X_0 \xrightarrow{\pi} Y) \in G_\varepsilon, k \geq 0\}$.

Substitution for shape predicates: $\mathcal{T}\sigma$ is a shape predicate given by:

$$\mathcal{T}\langle X \mid G \rangle = \langle X \mid \mathcal{T}G \rangle$$

Figure 2: Definition of type substitution

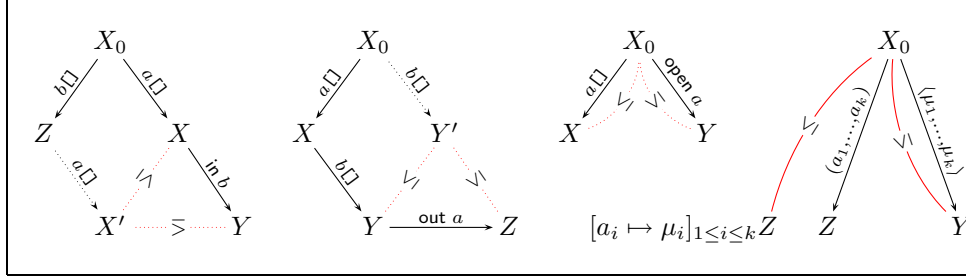


Figure 3: Definition 2.13 in diagram form

1. $\{(X_0 \xrightarrow{a\Box} X), (X \xrightarrow{\text{in } b} Y), (X_0 \xrightarrow{b\Box} Z)\} \subseteq G$
 $\Rightarrow \exists X' : Z \xrightarrow{a\Box} X' \in G \wedge \langle X | G \rangle \leq \langle X' | G \rangle \wedge \langle Y | G \rangle \leq \langle X' | G \rangle,$
2. $\{(X_0 \xrightarrow{a\Box} X), (X \xrightarrow{b\Box} Y), (Y \xrightarrow{\text{out } a} Z)\} \subseteq G$
 $\Rightarrow \exists Y' : X_0 \xrightarrow{b\Box} Y' \in G \wedge \langle Y | G \rangle \leq \langle Y' | G \rangle \wedge \langle Z | G \rangle \leq \langle Y' | G \rangle,$
3. $\{(X_0 \xrightarrow{a\Box} X), (X_0 \xrightarrow{\text{open } a} Y)\} \subseteq G$
 $\Rightarrow \langle X | G \rangle \leq \langle X_0 | G \rangle \wedge \langle Y | G \rangle \leq \langle X_0 | G \rangle,$ and
4. $\{(X_0 \xrightarrow{(\mu_1, \dots, \mu_k)} Y), (X_0 \xrightarrow{(a_1, \dots, a_k)} Z)\} \subseteq G$
 $\Rightarrow \langle Y | G \rangle \leq \langle X_0 | G \rangle \wedge [a_i \mapsto \mu_i]_{1 \leq i \leq k} \langle Z | G \rangle \leq \langle X_0 | G \rangle.$

Fig. 3 shows diagrams corresponding to each of the cases in this definition.

Definition 2.14. Let $\sigma = \langle X | G \rangle$ be a shape predicate. The **active nodes** in σ , written $\text{active}(\sigma)$, is the least set of node names such that

$$\text{active}(\sigma) = \{X\} \cup \{Z \mid \exists Y \in \text{active}(\sigma) : Y \xrightarrow{a\Box} Z \in G\}.$$

Definition 2.15. The shape predicate $\langle X | G \rangle$ is **syntactically closed** iff G is locally closed at every $X \in \text{active}(\langle X | G \rangle)$.

Theorem 2.16. Let σ be a shape predicate. If σ is syntactically closed, then it is semantically closed.

Definition 2.17. A **type** τ is a syntactically closed shape predicate. Given a type τ , the term P has type τ iff $\vdash P : \tau$.

This notion of types has the basic properties expected of any type system: It enjoys subject reduction (Thm. 2.16), it can be effectively decided whether a given term has a given type (Prop. 2.7), and types can be distinguished from non-types (Amtoft et al. (2003) describe how to decide the containment relation, so it is easy to check syntactic closure). Various properties of a term's computational behaviour can be approximated by looking at its types:

- If P has the type $\sigma = \langle X | G \rangle$ and G contains no edge $Y \xrightarrow{\bullet} Z$ with $Y \in \text{active}(\sigma)$, then executing P will never *execute* a malformed substitution result such as $[a \mapsto M.N](\text{in } a)$.
- If P has the type $\langle X | G \rangle$ and G contains no edge $Y \xrightarrow{\bullet} Z$, then executing P will never create a malformed substitution result.

- If P has the type $\langle X | G \rangle$ and G does not contain a sequence $X \xrightarrow{a[]}$ $Y \xrightarrow{b[]}$ Z , then executing P will never result in an ambient named a directly containing an ambient named b .

3 Basics of type inference

3.1 Algorithm overview

The type inference algorithm works in two steps:

1. Construct the minimal shape predicate σ_0 that describes the term which is to be typed. This is easy, as the set of possible paths in the term is finite and thus trivial to express as an automaton.
2. *Close* the shape predicate; that is, add nodes and edges as necessary to make it syntactically closed. Most of the rest of this report concerns how to write a terminating algorithm for closing shape predicates.

It is natural to expect the closing algorithm to compute the *least* closed shape predicate (in some well-defined set) that contains its input. Our closure algorithm does not quite do that, but does have a related property, which will be explained at the end of Section 5.2.

Each instance of the rules for syntactic closure requires that the containment relation holds between certain nodes in the type graph. It is natural to support this in the closure algorithm by keeping track of a set of containment pairs that have been found to be necessary for the original graph to be closed.

Thus, the closure algorithm does not work on raw shape graphs, but on the *combination* of a shape graph G and a relation R between its nodes. The goal of the algorithm is to extend G and R simultaneously such that G becomes syntactically closed and R becomes a shape simulation that certifies all the containment pairs necessary to show that G is in fact syntactically closed.

We will view G and R as being parts of the same *inference graph*, which contains the edges of G (which are labelled with prefix types) plus one edge for each pair in R (which are all labelled with the symbol \leq).

We have found it useful to think of the edges from G as *black edges* and the edges from R as *red edges*. (More colours will be introduced later). By convention a red edge is oriented *from* the subtype *to* the supertype; then the direction of the red edge is the same as the direction of data flow it symbolises.

3.2 Modular analysis

As described here, the algorithm is not compositional. One may imagine compositional variants of the algorithm, with steps like

- To type the term $P | Q$, first type P and Q separately to get σ_P and σ_Q . Then construct the graph



(where X is a fresh node), and close it.

- To type the term $a[P]$, first type P to get σ_P ; then construct the graph

$$\begin{array}{ccc}
 X & \xrightarrow{a\Box} & Y \\
 & & \downarrow \text{V} \\
 & & \sigma_P
 \end{array} \tag{2}$$

(where X and Y are fresh), and close it.

The red edge in (2) will ensure that the closure operation will leave σ_P unchanged (though this is not obvious yet). The closure operation here is not necessarily trivial; if σ_P describes $\mathbf{b}[\text{out } \mathbf{a}]$, there will be new possible reductions in (2).

We have not yet tried to implement this variant. It will probably not be desirable to implement it as strictly as sketched here (with a full closure operation taking place for each node in the term's abstract syntax tree), but it might nevertheless be useful to insert closure steps at strategic places in the derivation of the initial shape predicate from the term to be analysed.

However, it is not obvious that the final closure operation for (1) will be faster than closing a raw minimal shape graph for the entire term, if P and Q contain processes that interact strongly. Further analysis is needed here before anything conclusive can be said.

3.3 Deterministic shape graphs

The system of syntactically closed but otherwise unrestricted shape predicates is very strong and expressive, but is also too flexible for our current state of the art in inference. Therefore we need to impose some restrictions on types. (Note that these are different from the ones in (Amtoft et al., 2003)).

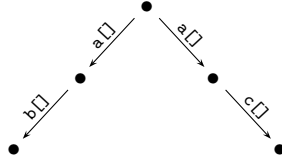
The most serious restriction is that we require all shape predicates to be *deterministic*. This is necessary for our algorithms to terminate.

Definition 3.1. *The shape graph G is **deterministic** if, for every node X and every prefix type π , there is at most one Y such that G contains $X \xrightarrow{\pi} Y$.*

A deterministic shape predicate can be thought of as a (prefix-closed) deterministic finite automaton that describes the possible paths from the root of a term, when the term is viewed as a tree, and replication and parallel composition are ignored in the paths.

Thus, a set of terms \mathbf{P} can be realized by a deterministic shape predicate only if $P \in \mathbf{P}$ can be decided just by looking at the set of paths from the root of P . For example, a deterministic σ that describes $P_1 = \mathbf{a}[\mathbf{b}[0]] \mid \mathbf{a}[\mathbf{c}[0]]$ must also describe $P_2 = \mathbf{a}[\mathbf{b}[0] \mid \mathbf{c}[0]]$, because the path sets of these two terms are identical.

This means that determinism is a real restriction of the expressive power of types; in contrast the non-deterministic shape predicate



does describe P_1 but not P_2 .

3.4 Simpler containment

For the purposes of most of the type inference, we will read $\pi' \geq \pi$ in Definition 2.8 as if it was $\pi' = \pi$. Together with determinism, this means that the Y' in Definition 2.8 is uniquely given by X and π , which simplifies reasoning about shape simulations.

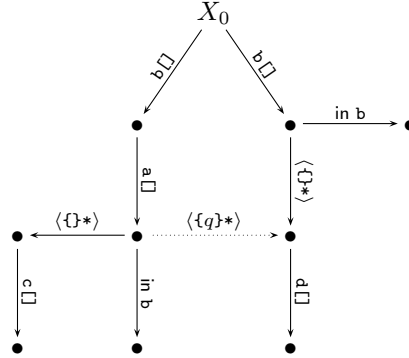
The union of all such restricted shape simulation is not quite the containment relation defined semantically in Definition 2.6. For example,

$$\langle X \mid X \xrightarrow{\langle \{a\}^* \rangle} Y \rangle \leq \langle X \mid X \xrightarrow{\langle \{a,b\}^* \rangle} Y \rangle$$

is true under the original containment relation, but is not in the union of all restricted shape simulations. However, the rest of the development of PolyA holds equally well if we interpret \leq to mean “the union of all (restricted) shape simulations” instead of the relation defined by Definition 2.6.³

Because the modified definition strengthens the requirements for being a shape simulation somewhat and thus makes it a little bit harder for a shape predicate to be a type. However, the difference is small in practise, because the only nontrivial inequalities between prefix types are those between send actions.

The role of the restricted definition is to forbid using spuriously enlarged prefix types to circumvent the determinism requirement for send actions. For example, should



be accepted as a closure of the graph without the dotted edge? Under the semantic definition of containment, it is indeed syntactically closed (and deterministic), but it depends on inventing the message q out of thin air. Our restriction of the containment relation forbids this trick, which would be hard to automate.

3.5 Monomorphic recursion

Another restriction that the type inference makes use of is the following principle:

If a shape graph contains a chain of edges

$$X_0 \xrightarrow{\pi_1} X_1 \xrightarrow{\pi_2} \dots \xrightarrow{\pi_n} X_n$$

and $\pi_1 = \pi_n$, then it is reasonable to require $X_1 = X_n$ as well.

³This is not true of the *completeness* and *principality* results of Amtoft et al. (2003), which we do not use in this report.

We refer to this requirement as the **monomorphic recursion restriction**. We do not think that it is as severe as the determinism requirement in practise; this is based on the assumption that situations where it applies will most often be artifacts of the type analysis (e.g., because PolyA does not keep track of how many identical processes that may exist side by side) rather than true behavioural features of terms. We do not have practical experience to support this assumption, but it feels like a reasonable working hypothesis.

Our inference algorithm does not enforce the monomorphic-recursion principle everywhere in the graph; only a partial variant that is sufficient to guarantee termination. More details about this will be given below, in Sections 4.3 and 5.2.

4 The closure algorithm

4.1 Closure steps

The basic element of the closure algorithm is the *closure step*. A step consists of locating one active node (Definition 2.14) in the type graph where one of the conditions in Definition 2.13 does not hold, and adding items (edges and possibly new nodes) to the inference graph that will make it hold.

We will initially consider a restricted form of syntactic closure where the only rule that matters is 2.13(1): the one for the *in* capability. We will add steps corresponding to the other rules later (Section 6.1), when we have shown how to integrate the *in* rule in a closure algorithm.

A first approximation of the closure step for *in* is:

- The step $\text{StepIn}(X_0, X, Y, Z, a, b)$ **applies** iff all of the following hold:
 1. X_0 is active.
 2. There is an edge $X_0 \xrightarrow{a\sqcap} X$,
 3. There is an edge $X \xrightarrow{\text{in } b} Y$.
 4. There is an edge $X_0 \xrightarrow{b\sqcap} Z$.
 5. Either there is no X' such that $Z \xrightarrow{a\sqcap} X'$ exists, or there is one (in which case it is unique), but the red edges $X \leq X'$ and $Y \leq X'$ do not both exist.
- When the step applies, we can **fire** it by
 - (a) If there is no X' such that $Z \xrightarrow{a\sqcap} X'$ exists, then create a fresh node X' and add the edge $Z \xrightarrow{a\sqcap} X'$.
 - (b) Add red edges $X \leq X'$ and $Y \leq X'$.

It will be an invariant that no edges will ever be *removed* from the inference graph, so after the step has fired, it cannot apply again because of condition (5).

Part (b) of firing the rule does not guarantee that X' will actually contain X , but only records the fact that we'll want to *make* it hold, as a red edge. We use another kind of step to guarantee this:

- The step $\text{Propagate}(X_0, Y, Z, \pi)$ **applies** iff all of the following hold:
 1. There is an edge $X_0 \xrightarrow{\pi} Y$.
 2. There is a red edge $X_0 \leq Z$.
 3. Either there is no Y' such that $Z \xrightarrow{\pi} Y'$ exists, or there is one (in which case it is unique), but there is no red edge $Y \leq Y'$.
- When the step applies, we can **fire** it by
 - a. If there is no Y' such that $Z \xrightarrow{\pi} Y'$ exists, then create a fresh node Y' and add the edge $Z \xrightarrow{\pi} Y'$.
 - b. Add a red edge from Y to Y' .

The Propagate rule corresponds to the definition of a shape simulation. Here it is in diagram form, similar to Figure 3:

$$\begin{array}{ccc}
 Z & \xrightarrow{\pi} & Y' \\
 \downarrow \text{red} & & \downarrow \text{red} \\
 X_0 & \xrightarrow{\pi} & Y
 \end{array}$$

4.2 Scheduling of steps

Superficially, the closure algorithm is simple: Keep looking for a closure step that applies, and fire it. If at some point we find that there is no node that applies, we'll know that the black edges of the inference graph constitute a syntactically closed shape predicate (if we assume for the moment that each π in the graph has one of the shapes in a or $a[]$). Then the closure algorithm terminates.

Of course, we do not yet have any guarantee that the algorithm *will* terminate; since the steps may add nodes to the graph, it is possible that there will always be new steps that apply, such that the algorithm continues to build an ever larger inference graph. We will handle that problem in a moment.

First, however, a brief discussion of how to locate steps that apply. In principle, one could just enumerate all tuples of the form $\text{StepIn}(X_0, X, Y, Z, a, b)$ and $\text{Propagate}(X_0, Y, Z, \pi)$, and test the applicability of each in sequence – at any point in time there are finitely many nodes to try, and the set of names that may ever need to be used can be determined before we ever start closing the shape predicate.

Such a strategy would of course be stupidly slow. In our prototype implementation, we instead do the following:

1. Construct a pending-set consisting of all nodes *from* which a red edge originates in the graph to be closed.
2. Visit all active nodes X_0 in some order. They are easily located by, say, a depth-first traversal. For each of them, do
 - Enumerate each pair (a, a') in $\{a \mid \exists X : X_0 \xrightarrow{a} X\}^2$, and check for each whether $\text{StepIn}(X_0, X, Y, Z, a, b)$ applies for some X, Y, Z (which will be uniquely determined by X_0, a and b). If it does, then fire it.

- (Similar steps for the other three rules of Definition 2.13 will be added here).

Each time an edge (either black or red) is added to the graph, add the node *from* which it goes to the pending-set.

3. Now, the X_0 of every **Propagate** that applies will be in the pending set. As long as the pending-set is not empty, select and remove a X_0 and look for applicable **Propagate** rules there. Continue adding nodes to the pending-set whenever edges from them are created.
4. When (if) the pending set becomes empty, check if any step at all was fired in phase (2) or (3) of the algorithm. If not, then we're done and the graph is closed. Otherwise, go back to phase (2).

This algorithm is reasonably smart as far as scheduling **Propagate** goes. There is room for improvement in the scheduling of **StepIn** steps. Intuitively, except for the first round of the closure, it should only be necessary to check for applicable steps “near” the areas of the inference graph that have changed since the last check. But it is not trivial to formulate what “near” means here, because it may be any of the edges in the diagram for a step that is added last and therefore makes the step apply.

It certainly appears to be *possible* to store enough information locally in the graph to be able to locate the possible new step instances when a (black) edge is added to the graph, but it would be rather complex; each new reduction rule in the underlying calculus might need its own set of information tacked onto the graph. Though something will eventually have to be done in this direction, we feel that optimisation here would be premature at the current stage of our research.

4.3 Heritages: guaranteeing termination

The algorithm as presented thus far will not necessarily terminate. For example, if we try to close the graph $\{X \xrightarrow{\pi} Y, X \leq Y\}$, repeated applications of **Propagate** will construct a growing chain of parallel black and red arrows, such that there is always a fresh instance of **Propagate** that applies.



Our approach to ensuring termination is based on preventing the number of nodes from growing without bound. If only we achieve this, then the number of possible closure steps will be bounded too; since each step can be fired only once, the process will eventually reach a state when there are no more steps to fire.

Observe that both of the step types **StepIn** and **Propagate** have the property that a new node is only created together with a *black* edge from an existing node. The existing node was either present before we started closing the type, or was itself created as the target of a black edge in a closure step, so we can trace back through a series of black edges to a node that existed in the input to the closure operation. Let us call this series the new node’s **heritage**. Now apply the monomorphic recursion principle to the heritage: If the π with which we

are to decorate the new edge is already found in the heritage, then the principle says that it is “reasonable” to reuse the target of the existing π arrow instead instead of creating a new node.

For example if we want to add a new π_2 edge from node Y whose heritage is

$$X_0 \xrightarrow{\pi_1} X_1 \xrightarrow{\pi_2} X_2 \xrightarrow{\pi_3} X_3 \xrightarrow{\pi_4} Y \quad (3)$$

then instead of creating a fresh Y' we set $Y' = X_2$ and add a back edge $Y \xrightarrow{\pi_2} X_2$ in the graph:

$$X_0 \xrightarrow{\pi_1} X_1 \xrightarrow{\pi_2} X_2 \xrightarrow{\pi_3} X_3 \xrightarrow{\pi_4} Y$$

Because no new names can be introduced during closure, the number d of different prefix types is limited too. This bounds the out-degree of the graph (thanks to determinism) as well as the maximal length of heritages. Thus the total number of nodes in the graph is bounded by $\frac{d^{d+1}-1}{d-1}$ for each node in the original graph.⁴

Notice that the heritage-based bounding of graph size is not a full implementation of the monomorphic recursion restriction. For example, assume that the graph that (3) comes from already contained an edge $X_2 \xrightarrow{\pi_4} Z$. Then, full enforcement of monomorphic recursion would require that Z was identified with Y , because the new edge introduces the chain

$$X_3 \xrightarrow{\pi_4} Y \xrightarrow{\pi_2} X_2 \xrightarrow{\pi_4} Z$$

which matches the monomorphism requirement. However, we don’t enforce this, and even after the addition of $Y \xrightarrow{\pi_2} X_2$, the heritage of X_2 is considered to consist of only X_0 and X_1 .

In the implementation, the heritage for each node is cached as a partial map from prefix types to node names. For example, the heritage of Y in (3) can be represented as the map

$$\{\pi_1 \mapsto X_1, \pi_2 \mapsto X_2, \pi_3 \mapsto X_3, \pi_4 \mapsto Y\}$$

It is not necessary to represent the linear structure of the heritage.

5 Improvements of the basic algorithm

So far we have described a *correct* and *terminating* design for an inference procedure, except that we still need to add closure steps corresponding to the out, open and communication reduction rules. Before defining the missing rules, we will describe some optimisations of the algorithm.

5.1 Reflexivity of containment

The first optimisation is simple: Whenever a node X is created, we immediately create the red edge $X \leq X$. This does not entail any changes to the graph that wouldn’t happen otherwise (before shape-graph containment is reflexive), but

⁴This is of course an extremely conservative estimate and should not be taken to be indicative of the likely size of the closed graph. We doubt that it is possible to construct original graphs that in fact grow by that much before they are closed.

means that we don't have to spend time dealing with them later. These self-edges need not be explicitly represented in the implementation, because they exist uniformly for all nodes.

5.2 Unifying containment cycles

Whenever the graph contains a (non-trivial) cycle of red edges, the definition of shape simulation means that the (black) subgraphs reachable from each of the nodes in the cycle must all be isomorphic when the closure operation is finished. Thus we can save work by unifying the nodes in a cycle quickly after it arises.

We do this by inserting a phase (2b) between phases (2) and (3) in the algorithm from Section 4.2:

- 2b. Traverse the subgraph consisting of red edges reachable from any red edge that has been added since the last time phase (2b) ran, and identify strongly connected components in it.

After the traversal, unify the nodes in each component with more than one member. In order to facilitate this, access to the nodes during the closure operation goes through a union-find structure. Whenever two or more nodes in the component both have an outgoing black edge with the same π on it, and add a cycle of red edges between their targets (which will later cause them to be unified), and remove all but one of the edges.

Repeat the phase until no nontrivial components are found.

The termination condition in phase (4) of the algorithm now becomes: “No steps were fired in phase (2) or (3) and no nontrivial components were found in phase (2b)”.

Unification and heritage When two or more nodes are unified, the question arises of what the unified node's heritage should be. For the purposes of ensuring termination, it would be sufficient to take the heritage of an arbitrary one of the unified nodes. In an attempt to be more symmetric and predictable (but we do not yet have enough experience with the algorithm to tell whether it is a good idea), our implementation instead *combines* the heritages of all the nodes to be unified.

A combined heritage is now represented as a map from prefix types to *sets* of nodes. When one wants to add a π -edge from some node Y , and Y 's heritage is defined for π , the nodes in the heritage set for π must be unified (or rather: scheduled to be unified by adding a cycle of red edges among them) and one of them is (arbitrarily) selected as the target of the new π -edge.

The natural consequence of this combining of heritages would be to propagate the unified node's combined heritage to those nodes that had one of its predecessors in its previous heritage. Our implementation does not currently do that, primarily because it would complicate the coding to allow a node's heritage to be extended after it has been created.

This omission, however, means that the final result of the closure operation may in principle depend on the (arbitrary) order in which the nodes are processed in phase (2) of the algorithm, because that may influence the exact time when a red cycle is detected and thus which heritages the nodes will have at later stages in the closure. We have not yet discovered an actual use case

where this seems to be an issue, but they can probably be constructed specifically to trigger the effect. We plan, in further work, to investigate the effects of propagating widened heritages more aggressively.

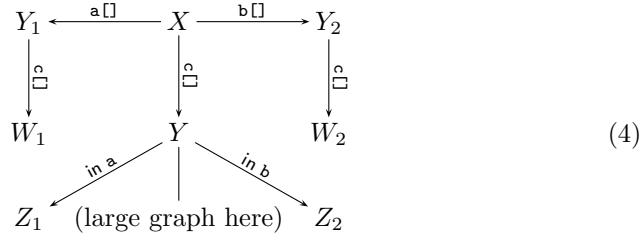
An aside: larger heritages A possible method for avoiding the uncertainty just described would be to enforce the monomorphic-recursion restriction globally. This could be done by defining a node’s heritage as simply all of the black edges that can reach it directly or indirectly through other edges. Whenever we find an edge $X \xrightarrow{\pi} Y$ such that there is an edge $\xrightarrow{\pi} Z$ in X ’s heritage with $Y \neq Z$, we would then unify X and Y .

We have not implemented this, because it is clear that it would introduce no less sharing than the rule from Section 4.3, and would therefore produce less precise types. Furthermore, the principle is not directly compatible with the concept of “blue edges” we will introduce in a moment.

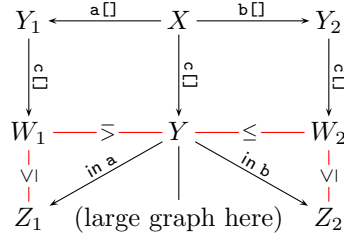
The thought experiment is interesting nevertheless, because a closure algorithm based on such an aggressive heritage concept would clearly yield a final type that was minimal among the deterministic, fully monomorphic-recursive syntactically closed shape predicates that match the original term. In other words, the system of those types has computable principal types. Furthermore, our actual algorithm will always compute a type that is at least as precise as the principal type in that system.

5.3 Avoiding graph duplication

Consider closing the graph



rooted at X . First, rule **StepIn** fires twice and adds some red edges:

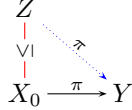


Next, the **Propagate** phase will cause the two red edges starting at Y to ripple through the entire large subgraph alluded to at the bottom of the figure, and create two fresh copies of it rooted at W_1 and W_2 . That is an essential part of the way **PolyA** can be said to be polymorphic, because other elements at, say,

Y_1 may interact with the fresh copy at W_1 without propagating the changes to the fresh copy at W_2 .

However, if there is nothing at Y_1 or W_1 that can interact with the large graph, it is wasteful to make an entire copy. Ideally, we'd like to copy graphs *lazily*, that is, only when there is a possibility of interaction between the copy and its new context.

Consider therefore the following variant of the **Propagate** step that does not create fresh copies:



The rule *applies* in the same conditions as the old one, but when it *fires* and a π -edge from Z does not already exist, it creates an edge directly to Y instead of to a new (or heritage-reused) node with a red edge from Y .

That solves the task of not immediately duplicating entire subgraphs, but we still need a way to start duplicating the subgraph when we find that it interacts with something in its new context. For this, we need to put a special mark on the diagonal edge $Z \xrightarrow{\pi} Y$ added by the modified **Propagate**; we call it a *blue edge*. The special property of blue edges is that they can be *replaced* by black edges to new nodes when and if we find it necessary. In black-and-white illustrations, we show blue edges with a dashed arrow.

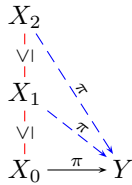
Blue edges are treated just like black ones when deciding whether a closure step *applies*. However, once we have begun *firing* a rule (except **Propagate** which is special), everything we add to the graph must be reached from the rule's X_0 by *black* edges, and we must lift blue edges to new black ones as necessary.

Part (a) of the firing process for **StepIn** in Section 4.1 must be replaced by

- a1. If the edge $X_0 \xrightarrow{b\Box} Z$ is *blue*, then erase it, and create a new (or heritage-reused) Z' with a new edge $X_0 \xrightarrow{b\Box} Z'$. Then let Z denote Z' in the rest of the firing.
- a2. If there is no X' such that $Z \xrightarrow{a\Box} X'$ exists *and is not blue*, then create a new (or heritage-reused) X' and add the edge $Z \xrightarrow{a\Box} X'$.

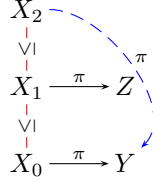
Maintenance of blue edges After the modified rule has fired, there is still some cleaning-up to do. It is common to all situations where a blue edge has been replaced with a black one, so we handle it generically outside the specific context of the **StepIn** step.

Assume that part of the graph looks like

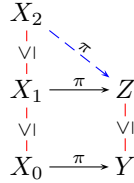


and that the blue edge from X_1 is replaced with a black one, say, to Z . The

graph now looks like



What is necessary to make this self-consistent? First, the blue edge from X_2 should be redirected to end at Z instead of Y . Second, the original Propagate says that there must be a red edge from Y to Z :



We replace phase (3) of the algorithm in Section 4.2 by a new phase that keeps these invariants up to date:

3. Do the following for each π : Traverse all nodes in the graph in topological order according to the red edges, such that $Y \leq X$ means that Y will be visited before X . (This is possible because there are no red cycles left after phase 2b). For each node X_0 , do the following:
 - a. Compute a set of **candidate** black edges. An edge $Z \xrightarrow{\pi} W$ is a candidate iff there is a series of red edges $Z \leq Y_1 \leq \dots \leq Y_n \leq X_0$ with $n \geq 0$ (but $Z \neq X$) such that no Y_i 's is the origin of a π -marked black edge.
 - b. The candidate $Z \xrightarrow{\pi} W$ *dominates* another candidate $Z' \xrightarrow{\pi} W'$ iff there is a series of red edges $Z \leq \dots \leq Z'$, no matter whether the intermediate nodes are origins of π -edges.
Remove all candidates that are dominated by other candidates.
 - c. For each π such that there is more than one surviving candidate edge marked π , make sure that X_0 is the origin of a black π -edge; if necessary by adding a fresh node or reusing a heritage node.
 - d. For each black edge $X_0 \xrightarrow{\pi} X'$ from X_0 , including ones created in (c), save the red edge $W \leq X'$ for each candidate $Z \xrightarrow{\pi} W$. Remove those candidates after processing them.
 - e. Each remaining candidate $Z \xrightarrow{\pi} W$ is the unique candidate marked π . Add (or redirect) a blue edge $X_0 \xrightarrow{\pi} W$.

After all nodes have been processed, add the red edges saved in (d) to the graph. (This cannot happen during the traversal itself, because it might invalidate the topological order). Repeat the entire phase (3) if any of those red edges did not exist already.

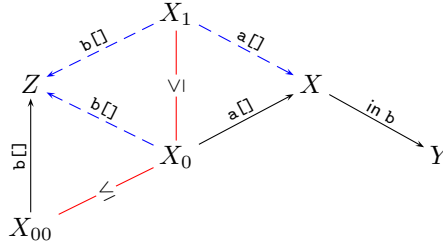
Identifying candidates in (a) may seem to require a graph representation that can identify $\{Y \mid Y \leq X\}$ given X . However, it is easy to pre-compute the

candidate sets for yet unprocessed nodes as part of the visiting of their predecessors.

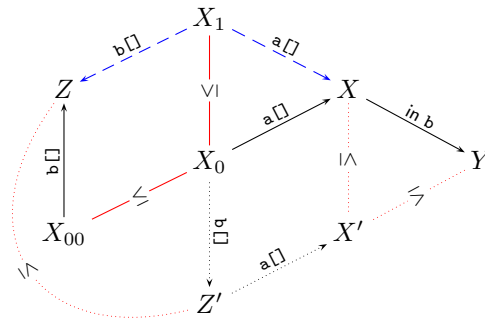
One thing to note about this algorithm is that it does not care at all about the blue edges *before* the phase runs. Our implementation does use information from the old blue edges to try to compute a smaller set of X_0 's to traverse than the entire graph – but the implementation is slightly flawed here; its heuristics may cause it to miss dominators in step (b). This does not affect the correctness or precision of the types computed, but may in certain cases cause spurious copying of subgraphs.⁵

Closure step scheduling and blue edges The maintenance phase for blue edges described above runs only once the entire graph has been searched for applicable closure steps and cycles have been unified. We feel this is a reasonable design decision because it is somewhat expensive; it would be wasteful to rerun it each time a black or red edge were added to the graph.

This has implications for the scheduling of closure steps, however. Imagine this situation:



It is easy to see that the closure steps $\text{StepIn}(X_0, X, Y, Z, a, b)$ and $\text{StepIn}(X_1, X, Y, Z, a, b)$ both apply. If we fire the former (but do not yet update the blue edges), the graph becomes:



The old blue $X_0 \xrightarrow{a[]} Z$ has been replaced with a new black $X_0 \xrightarrow{a[]} Z'$ (because X_{00} does not need to match $a[b[\dots]]$). However, the blue edge from X_1 has not yet been updated to point to Z' , so $\text{StepIn}(X_1, X, Y, Z, a, b)$ *still* applies. If we fire it now, it will create new Z'' and X'' , but that is not really necessary.

The same thing happens if we fire the two steps in the other order; in that case the extraneous nodes cannot even be prevented by updating the blue edges after each step.

⁵This was discovered so shortly before this report was to be delivered that we have not yet thought of a way to fix the problem.

To avoid these effects, we need to refine the scheduling of closure steps in phase (2) of the closure algorithm. For this, we change the algorithm in two ways:

- The active nodes must be visited in topological order according to the red edges, just as for the modified phase (3) from above. This makes sure that in examples like the above one, the step for X_0 is met before the step for X_1 .
- Whenever a step with a certain X_0 is fired, prevent any nodes that is reachable by red edges from this X_0 from being fired until the blue edges have been updated.

Furthermore, in order to make the “topological order” meaningful, it is necessary to insert a cycle-unification phase before phase (2) just as there was already one before phase (3).

6 The final algorithm

6.1 All closure steps

We are now ready to define all of the closure steps we need for PolyA, rather than just the one for the in reduction. For completeness, we also give the final version of `StepIn`, with the amendments we have described since its first introduction:

- The step `StepIn(X_0, X, Y, Z, a, b)` **applies** iff all of the following hold:
 1. X_0 is active.
 2. There is a (blue or black) edge $X_0 \xrightarrow{a\Box} X$,
 3. There is a (blue or black) edge $X \xrightarrow{\text{in } b} Y$.
 4. There is a (blue or black) edge $X_0 \xrightarrow{b\Box} Z$.
 5. Either there is no X' such that $Z \xrightarrow{a\Box} X'$ exists, or there is one, but the red edges $X \leq X'$ and $Y \leq X'$ do not both exist.

When the step applies, we can **fire** it by

- a1. If the edge $X_0 \xrightarrow{b\Box} Z$ is blue, then erase it, and create a new (or heritage-reused) Z' with a new edge $X_0 \xrightarrow{b\Box} Z'$. Then let Z denote Z' in the rest of the firing.
 - a2. If there is no X' such that $Z \xrightarrow{a\Box} X'$ exists and is not blue, then create a new (or heritage-reused) X' and add the edge $Z \xrightarrow{a\Box} X'$.
 - b. Add red edges $X \leq X'$ and $Y \leq X'$.
- The step `StepOut(X_0, X, Y, Z, a, b)` **applies** iff all of the following hold:
 1. X_0 is active.
 2. There is a (blue or black) edge $X_0 \xrightarrow{a\Box} X$.
 3. There is a (blue or black) edge $X \xrightarrow{b\Box} Y$.
 4. There is a (blue or black) edge $Y \xrightarrow{\text{out } a} Z$.

5. Either there is no Y' such that $X_0 \xrightarrow{b\sqcup} Y'$ exists, or there is one, but the red edges $Y \leq Y'$ and $Z \leq Y'$ do not both exist.

When the step applies, we can **fire** it by

- a. If there is no Y' such that $X_0 \xrightarrow{b\sqcup} Y'$ exists and is not blue, then create a new (or heritage-reused) Y' and add the edge $X_0 \xrightarrow{b\sqcup} Y'$.
 - b. Add red edges $Y \leq Y'$ and $Z \leq Y'$.
- The step **StepOpen**(X_0, X, Y, a) **applies** iff all of the following hold:
 1. X_0 is active.
 2. There is a (blue or black) edge $X_0 \xrightarrow{a\sqcup} X$.
 3. There is a (blue or black) edge $X_0 \xrightarrow{\text{open } a} Y$.
 4. The red edges $X \leq X_0$ and $Y \leq X_0$ do not both exist.

When the step applies, we can **fire** it by

- a. Add red edges $X \leq X_0$ and $Y \leq X_0$.
- The step **StepComm**($X_0, Y, Z, a_1, \dots, a_k, \mu_1, \dots, \mu_k$) **applies** iff all of the following hold:
 1. X_0 is active.
 2. There is a (blue or black) edge $X_0 \xrightarrow{\langle \mu_1, \dots, \mu_k \rangle} Y$.
 3. There is a (blue or black) edge $X_0 \xrightarrow{\langle a_1, \dots, a_k \rangle} Z$.
 4. The node $[a_i \mapsto \mu_i]_{1 \leq i \leq k} Z$ has not been generated yet, or it has but the red edges $[a_i \mapsto \mu_i]_{1 \leq i \leq k} Z \leq X_0$ and $Y \leq X$ do not both exist.

When the step applies, we can **fire** it by

- a. Generate $[a_i \mapsto \mu_i]_{1 \leq i \leq k} Z$, if necessary.
- b. Add red edges $[a_i \mapsto \mu_i]_{1 \leq i \leq k} Z \leq X_0$ and $Y \leq X$.

Doing substitutions The description of **StepComm** talks about generating the node $\mathcal{T}Z$ for some type substitution Z . We have not yet described what this means. Mostly it is just the construction given in Figure 2 (on page 9), with a few changes.

To generate $\mathcal{T}Z$, first let G' be the subgraph of the inference graph that is reachable from Z by blue and black edges. Construct $\mathcal{T}G'$ within the inference graph, as defined in Figure 2, with the following modifications:

- Instead of reusing node names from G' (which are already in use in the inference graph), construct one fresh node (with empty heritage) for each node in G' .
- The inference graph cannot contain “null edges” $X \xrightarrow{\varepsilon} Y$, but it can be represented as the red edge $Y \leq X$.
- Omit the final step that sets $\mathcal{T}G' = \{X_k \xrightarrow{\pi} Y \mid (X_k \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} X_0 \xrightarrow{\pi} Y) \in G_\varepsilon\}$. The edges it adds for $k \neq 0$ will arise automatically as blue edges due to our representation of null edges.

The correctness of this procedure is not quite obvious – what if more edges are added to G' after we have generated the substituted graph? This will not happen if the following invariants hold:

- A node that is reachable from the target of a $\xrightarrow{(\dots)}$ is *not* reachable from the root except through paths that includes at least one $\xrightarrow{(\dots)}$. In particular such nodes are never active.
- A node that is reachable from the target of a $\xrightarrow{(\dots)}$ is *not* the target of a red edge.
- If $X \xrightarrow{(a_1, \dots, a_k)} Y$ and $X' \xrightarrow{(a_1, \dots, a_k)} Y'$ (with the same sequence of a_i 's), then $Y = Y'$.

Unfortunately the latter two of these invariants are only preserved if the original term contains no nested (\dots) abstractions. Our type inference may therefore not be correct on terms that do contain nested abstractions. We believe that this is not a fundamental restriction of our approach, and are currently investigating several possible schemes for solving the problem.

The property that nodes in the substituted graph are created without heritage does not threaten the termination of the closure process. Because there is a finite number of names in the original graph, there is also a finite number of possible different type substitutions, and the same type substitution is not applied twice to the same node.

6.2 Summary of the closure algorithm

1. Construct a pending-set consisting of all nodes *from* which a red edge originates in the graph to be closed.
- 1b. Traverse the subgraph consisting of red edges reachable from any red edge that has been added since the last time phase (1b) or (2b) ran, and identify strongly connected components in it.

After the traversal, unify the nodes in each component with more than one member. In order to facilitate this, access to the nodes during the closure operation goes through a union-find structure. Whenever two or more nodes in the component both have an outgoing black edge with the same π on it, and add a cycle of red edges between their targets (which will later cause them to be unified), and remove all but one of the edges.

Repeat this phase until no nontrivial components are found.
2. Compute a topological order of all of the active nodes. Visit them in this order. For each active node X_0 , see if it is the root of any closure steps that apply. If so, fire the steps and exclude any node reachable from X_0 via red edges from being visited in this instance of (2).
- 2a. Unify cycles once again, as in phase (1a).
3. Do the following for each π : Traverse all nodes in the graph in topological order according to the red edges, such that $Y \leq X$ means that Y will be visited before X . (This is possible because there are no red cycles left after phase 2b). For each node X_0 , do the following:

- a. Compute a set of **candidate** black edges. An edge $Z \xrightarrow{\pi} W$ is a candidate iff there is a series of red edges $Z \leq Y_1 \leq \dots \leq Y_n \leq X_0$ with $n \geq 0$ (but $Z \neq X$) such that no Y_i 's is the origin of a π -marked black edge.
- b. The candidate $Z \xrightarrow{\pi} W$ *dominates* another candidate $Z' \xrightarrow{\pi} W'$ iff there is a series of red edges $Z \leq \dots \leq Z'$, no matter whether the intermediate nodes are origins of π -edges.
Remove all candidates that are dominated by other candidates.
- c. For each π such that there is more than one surviving candidate edge marked π , make sure that X_0 is the origin of a black π -edge; if necessary by adding a fresh node or reusing a heritage node.
- d. For each black edge $X_0 \xrightarrow{\pi} X'$ from X_0 , including ones created in (c), save the red edge $W \leq X'$ for each candidate $Z \xrightarrow{\pi} W$. Remove those candidates after processing them.
- e. Each remaining candidate $Z \xrightarrow{\pi} W$ is the unique candidate marked π . Add (or redirect) a blue edge $X_0 \xrightarrow{\pi} W$.

After all nodes have been processed, add the red edges saved in (d) to the graph. (This cannot happen during the traversal itself, because it might invalidate the topological order). Repeat the entire phase (3) if any of those red edges did not exist already.

4. Check if any step at all was fired in phase (2), or if any blue or red edges were changed in phase (3), or if any nontrivial components were found in phase (1a) or (2a). If not, then we're done and the graph is closed. Otherwise, go back to phase (1a).

7 Implementation; further work

We have produced a prototype implementation of the type inference process described above. As of the date of this report, it is available for download at:

<http://www.macs.hw.ac.uk/DART/software/PolyA/>

The prototype consists of source code written in Moscow ML (version 2.0), which is available at:

<http://www.dina.dk/~sestoft/mosml.html>

At output it produces graph descriptions suitable for viewing with VCG (Sander, 1994), which is available at:

<http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html>

An example inference graph produced by the implementation is shown as Figure 4. Here the input is the simple in-only example from page 3.

As soon as the input term and its behaviour gets much more complicated, the output graph becomes hard to read; see for example Figure 5, which shows the inferred type for one of the simpler messenger examples from Amtoft et al. (2003). Although some of the visual complexity in this output is arguably due

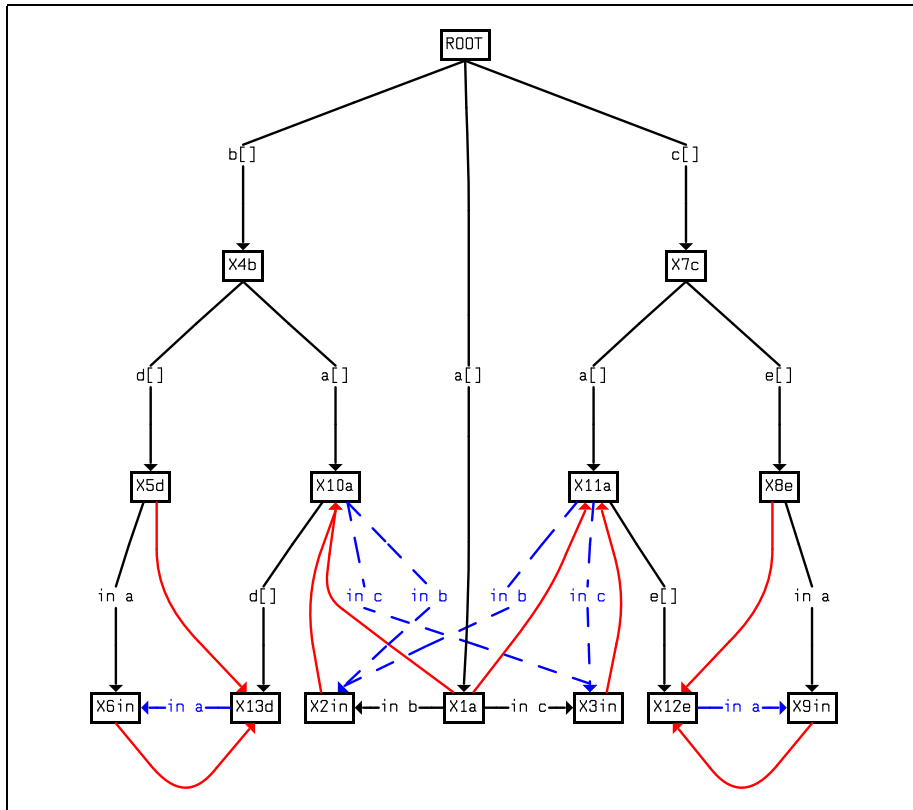


Figure 4: Example output from the prototype implementation. The input term is $a[in\ b.0 \mid in\ c.0] \mid b[d[in\ a.0]] \mid c[e[in\ a.0]]$.

to shortcomings in the graph layout tool (consider, for example, the strange shape of the $X15m \xrightarrow{\text{in } c} X11r$ arrow), it is clear that the graph also has a high inherent complexity.

In practical use, we would not expect humans to directly read these graphs. Instead, we expect desired properties to be formulated as queries that the type graph might or might not satisfy. However, our current difficulty in producing readable graphs slows us down in understanding the strengths and weaknesses of PolyA itself empirically, which we need to do for further development of the PolyA approach. We are therefore currently investigating ways to improve the presentation of inference graphs

Another direction for further work is of course to improve the efficiency of the closure algorithm; it should be clear from the description that improved scheduling and other algorithmic improvement could bring about dramatic runtime improvements. However, at its present state, the algorithm is fast enough for our purposes – it took less than 0.1 second to infer the graph in Figure 5, so it would not be useful to improve this further until we have presentation techniques that can make sense of outputs that take longer than this to produce.

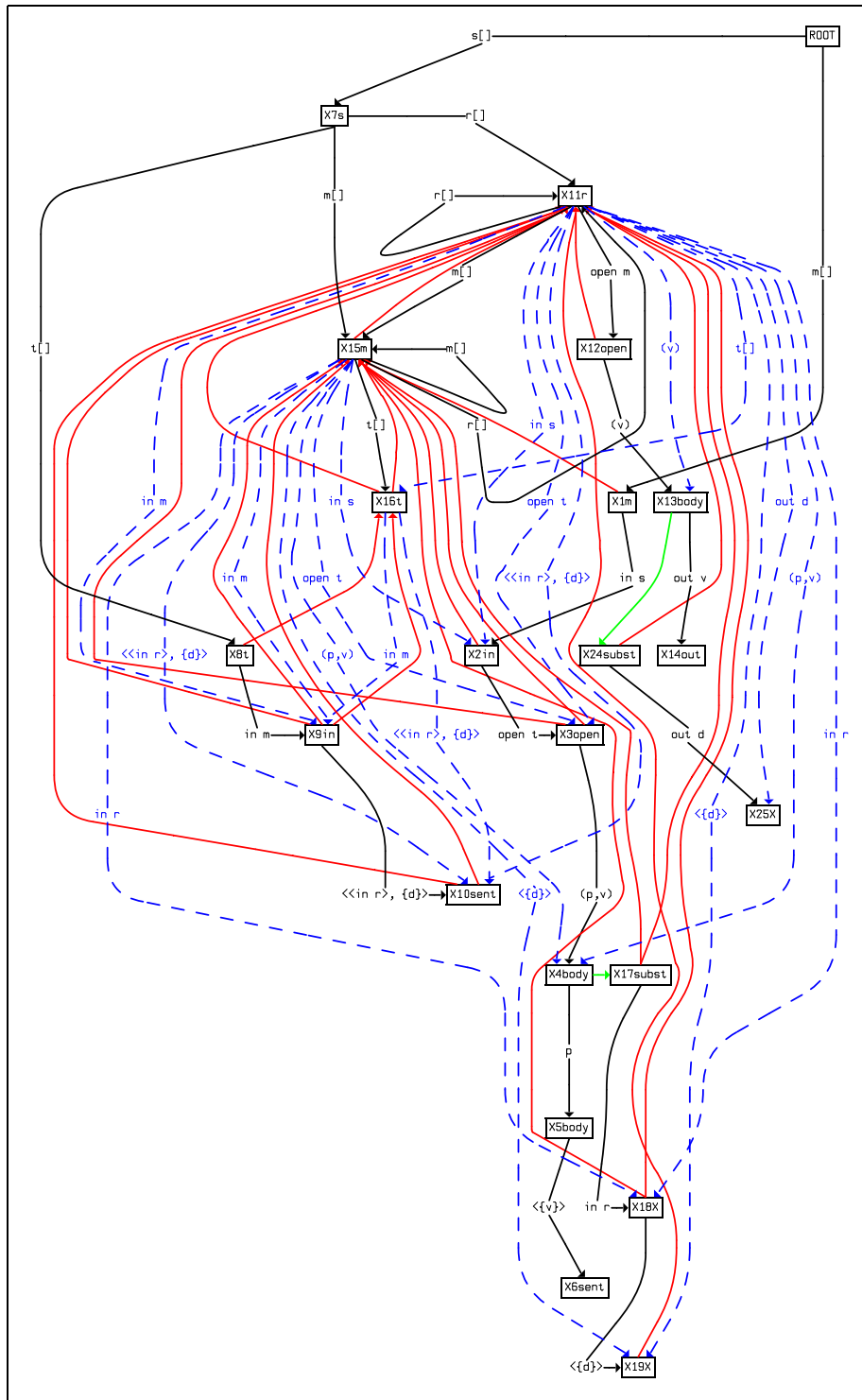


Figure 5: The prototype implementation's output on a simple messenger example from (Amtoft et al., 2003)

References

- Amtoft, T., Kfoury, A. J., and Pericas-Geertsen, S. M. (2000). What are polymorphically-typed ambients? Technical Report BUCS-TR-2000-021, Comp. Sci. Dept., Boston Univ.
- Amtoft, T., Kfoury, A. J., and Pericas-Geertsen, S. M. (2001). What are polymorphically-typed ambients? In Sands, D., editor, *ESOP 2001, Genova*, volume 2028 of *LNCS*, pages 206–220. Springer-Verlag. An extended version appears as Amtoft et al. (2000).
- Amtoft, T., Kfoury, A. J., and Pericas-Geertsen, S. M. (2002). Orderly communication in the ambient calculus. *Computer Languages*. To appear.
- Amtoft, T., Makhholm, H., and Wells, J. B. (2003). PolyA: True type polymorphism for mobile ambients. Unpublished draft. Supersedes Amtoft and Wells (2002).
- Amtoft, T. and Wells, J. B. (2002). Mobile processes with dependent communication types and singleton types for names and capabilities. Technical Report 2002-3, Kansas State University, Department of Computing and Information Sciences.
- Cardelli, L., Ghelli, G., and Gordon, A. D. (1999). Mobility types for mobile ambients. In Wiedermann, J., van Emde Boas, P., and Nielsen, M., editors, *ICALP'99*, volume 1644 of *LNCS*, pages 230–239. Springer-Verlag. Extended version appears as Microsoft Research Technical Report MSR-TR-99-32, 1999.
- Cardelli, L. and Gordon, A. D. (1998). Mobile ambients. In Nivat, M., editor, *FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag.
- Sander, G. (1994). Graph layout through the VCG tool. In Tamassia, R. and Tollis, I. G., editors, *Graph Drawing: DIMACS International Workshop, GD'94*, volume 894 of *LNCS*, pages 194–205. Springer-Verlag.
- Teller, D., Zimmer, P., and Hirschhoff, D. (2002). Using ambients to control resources. In *CONCUR'02*, volume 2421 of *LNCS*, pages 288–303. Springer-Verlag.