A Sort Inference Algorithm for the Polyadic π-Calculus

Simon J. Gay*
Department of Computing,
Imperial College of Science,
Technology and Medicine,
180 Queen's Gate,
London, UK
SW7 2BZ
(sjg3@doc.ic.ac.uk)

Abstract

In Milner's polyadic π-calculus there is a notion of *sorts* which is analogous to the notion of types in functional programming. As a well-typed program applies functions to arguments in a consistent way, a well-sorted process uses communication channels in a consistent way. An open problem is whether there is an algorithm to infer sorts in the π-calculus in the same way that types can be inferred in functional programming. Here we solve the problem by presenting an algorithm which infers the most general sorting for a process in the first-order calculus, and proving its correctness. The algorithm is similar in style to those used for Hindley-Milner type inference in functional languages.

1 Introduction

The benefits of programming with a typed language are widely accepted. The type of a function or procedure is the simplest possible form of specification, but nevertheless a very useful one. The type system aids modular program construction by providing a means of expressing procedure interface specifications, and compile-time type checking, which is a standard feature of both imperative and functional languages, is able to detect many programming errors. In the realm of functional programming with polymorphism, type checking is extended by the provision of type inference: the compiler is able to (attempt to) infer the most general polymorphic type of an expression. This relieves the programmer of the task of supplying type annotations for all variables and functions, and helps to ensure that function definitions reflect any genericity present in the algorithms which they are encoding. From a theoretical point of view, the analogy between types in functional programming and propositions in intuitionistic logic (the Curry-Howard isomorphism, also known as the propositions-as-types paradigm) forms the basis of the elegant connections between functional programs, intuitionistic proofs and cartesian closed categories. On the practical side, type checking is recognised as one of the most successful applications to date of formal methods in computer science.

The success of type systems in sequential programming makes it natural to ask whether there is a similar notion of types in some given formalism for describing concurrent processes, and whether the same kind of type checking and type inference can be used to good effect. In this paper we study this question in the case of one such formalism, namely Milner's π-calculus [MPW89, Mil91]. Milner answers the first part of the question by defining the notions of *sort* and *sorting*: a sorting is analogous to an environment of typed terms in functional programming type inference, and the sort of an agent (process) is analogous to the type of a function. The second part of the question is an open problem. We solve this problem by presenting an algorithm which constructs the most general (in a sense to be made precise; this is analogous to polymorphism) sorting in which a given agent can be assigned a sort, and also the sort itself. The algorithm is in the style of those in widespread use for Hindley-Milner type inference [Mil78, Hin69] in functional language systems.

In order to make the paper as self-contained as possible, we begin with an introduction to (or review of) the π-calculus in Section 2 and describe Milner's notion of sorting. Then in Section 3 we give the (slightly differ-

*Research supported by an SERC studentship.

ACM-20th PoPL-1/93-S.C., USA
© 1993 ACM 0-89791-561-5/93/0001/0429...$1.50

ent) definition of sorting to be used in the rest of the paper, and develop some theory of these sortings. In Section 4 the sort inference algorithm is presented as a collection of rules allowing agents to be constructed along with a suitable sorting, in the same way that type inference algorithms for functional programming languages are usually presented. Some examples of the algorithm in action also appear in this section. Section 5 contains the proof that the algorithm constructs the correct sorting. Finally in Section 6 the algorithm needed to combine two existing sortings into a sorting for a compound agent is given in a more concrete form, which should make it clear that the sort inference algorithm is amenable to implementation.

2 The π-Calculus

The π-calculus is a notation for communicating processes. As the name suggests, it has a flavour of the λ-calculus in the way terms are constructed. It is similar to CCS [Mil89] in that processes can be built up by adding prefixes, or communication points, to existing processes, but generalises CCS by allowing the communications resulting from prefixes to send and receive arbitrary names. For example, adding the prefix $x(y)$ to a process P forms the process $x(y).P$ which can receive a name (referred to as y within P) along a communication channel called x, and then become P. The notation for prefixes which output names is similar: $\overline{x}u.Q$ outputs u along x and becomes Q. The names which can be transmitted along channels in this way are of the same nature as the names of the channels themselves, so that $x(y).y(z).P$ is a valid construction. This ability to use names to refer to both channels and data is one of the key features of the π-calculus, and is what gives it much of its power. It allows the effect of transmitting processes along channels to be obtained: instead of sending a process, it is sufficient to send a name which can be used to access the process. Processes can be combined in parallel, which introduces the possibility of communication just as in CCS: $x(y).P|\overline{x}u.Q$ reduces to $P\{u/y\}|Q$. There is also a $+$ operator which again behaves in the manner familiar from CCS: $x.P + y.Q$ can communicate with an $\overline{x}$ or a $\overline{y}$ prefix and become P or Q accordingly.

The above examples live in the monadic π-calculus; in moving to the polyadic π-calculus, which we deal with in this paper, two further generalisations are made. Any number of names can appear in the argument of a prefix, for example $x(uvw)$, and also the addition of a prefix to a process is broken down into two steps. To form what we have previously written as $x(y).P$, the first step is to form the *abstraction* $(\lambda y)P$ from the process P, and the second step is to *locate* this abstraction at x by forming $x.(\lambda y)P$; we can then abbreviate this expression by $x(y).P$ to recover the previous notation. Similarly for outputs: $\overline{x}u.Q$ is an abbreviation for $\overline{x}.[u]Q$, where the *concretion* $[u]Q$ has been *co-located* at the *co-name* $\overline{x}$. Abstraction and concretion behave slightly differently with respect to the binding of names: x is bound in $(\lambda x)P$ but free in $[x]P$. The other way of binding names is with the restriction operator ν. In $(\nu x)P$, uses of x within P are distinct from any external uses; the combination $(\nu x)[x]P$ gives the effect of a concretion which binds its arguments.

The last construction to mention is the replication operator. The process $!P$ is equivalent to $P|!P$; this allows processes to be constructed which behave as if defined by recursive equations.

The preceding discussion describes the standard (first-order) π-calculus. We will extend the calculus by the addition of a new atomic process ERROR, which will be used to represent the kind of run-time error which sortings are intended to avoid.

The constructions described above are formalised by the following grammar for agents, in which M, N are *normal processes*, P, Q are *processes*, F is an *abstraction*, C a *concretion* and A an *agent*. Also x is an arbitrary name, and α is an arbitrary name or co-name.

$$\begin{array}{lcl} N & ::= & \alpha.A \mid 0 \mid M + N \mid \text{ERROR} \\ P & ::= & N \mid P|Q \mid !P \mid (\nu x)P \\ F & ::= & P \mid (\lambda x)F \mid (\nu x)F \\ C & ::= & P \mid [x]C \mid (\nu x)C \\ A & ::= & F \mid C \end{array}$$

In order to describe the benefits of introducing sortings, we need to discuss the operational semantics of the π-calculus. First of all, the reduction relation between processes can be defined most simply if we work not with raw process terms, but modulo a certain *structural congruence* relation. Writing $\mathcal{P}$ for the collection of processes and $\mathcal{NP}$ for the collection of normal processes, we define $\equiv$ to be the smallest congruence relation over $\mathcal{P}$ which satisfies the properties listed below. The free and bound names of a process, $\text{fn}(P)$ and $\text{bn}(P)$, are defined in the natural way. This is the standard definition of $\equiv$ for the π-calculus, extended to cover ERROR.

1. Processes are identified if they differ only by a change of bound names (*ie* α-conversion).
2. $(\mathcal{NP}/\equiv, +, 0)$ is a commutative monoid.
3. $(\mathcal{P}/\equiv, |, 0)$ is a commutative monoid.
4. $!P \equiv P|!P$.
5. If $x \notin \text{fn}(P)$ then $(\nu x)(P|Q) \equiv P|(\nu x)Q$.

6. If $x \neq y$ then $(\nu y)(\lambda x)F \equiv (\lambda x)(\nu y)F$.
7. If $x \neq y$ then $(\nu y)[x]C \equiv [x](\nu y)C$.
8. $(\nu x)(\nu y)A \equiv (\nu y)(\nu x)A$, $(\nu x)(\nu x)A \equiv (\nu x)A$.
9. $N + \text{ERROR} \equiv \text{ERROR}$.
10. $P|\text{ERROR} \equiv \text{ERROR}$.
11. $!\text{ERROR} \equiv \text{ERROR}$.
12. $(\lambda x)\text{ERROR} \equiv \text{ERROR}$, $[x]\text{ERROR} \equiv \text{ERROR}$, $(\nu x)\text{ERROR} \equiv \text{ERROR}$.

Now we define the reduction relation $\rightarrow$ over processes to be the smallest relation satisfying the following rules.

$$\frac{}{(\cdots + x.F)|(\cdots + \overline{x}.C) \rightarrow F \bullet C}\ \text{Comm}$$

$$\frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}\ \text{Par} \qquad \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}\ \text{Res}$$

$$\frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}\ \text{Struct}$$

The term $F \bullet C$ is the *pseudo-application* of the abstraction F to the concretion C. To define it, note that structural congruence can be used to write abstractions and concretions in standard forms $F \equiv (\lambda \vec{x})P$, $C \equiv (\nu \vec{y})[\vec{z}]Q$. If F and C are in this form with $\vec{x} \cap \vec{y} = \emptyset$, we define $F \bullet C$ to be $(\nu \vec{y})(P\{\vec{z}/\vec{x}\}|Q)$ if $|\vec{z}| = |\vec{x}|$, and ERROR otherwise. Thus the ERROR process is used to represent the result of an attempted communication in which the sending and receiving processes disagree about the number of names to be transmitted. The other possibility would be not to have a reduction in the case of such a disagreement, so that the result would be a deadlock. But by introducing ERROR we can identify precisely the kind of bad behaviour which can be eliminated by using well-sorted processes, although even well-sorted processes can deadlock for other reasons.

The aim of introducing sortings is to ensure that names are used consistently. If two processes $x.F$ and $\overline{x}.C$ have been constructed within the discipline of some ambient sorting then forming the composition $x.F|\overline{x}.C$ is guaranteed to make sense (ie will not reduce to ERROR), because F and C will have the same number of names abstracted or concreted out. Furthermore, both processes agree about how each name transmitted can be used, and so ERROR will not appear in any subsequent reductions. This will be made precise later in the paper.

Milner's definition of sorting is as follows. We assume a collection $\mathcal{S}$ of *subject sorts* and for each $S \in \mathcal{S}$ an infinite collection of names with subject sort S, written $x : S$. An *object sort* is a sequence of subject sorts; that is, an element of $\mathcal{S}^*$. A *sorting* over $\mathcal{S}$ is a non-empty partial function $ob : \mathcal{S} \rightharpoonup \mathcal{S}^*$ which describes, for any name $x : S$, the sort of name-vector it can carry when used to locate or co-locate an abstraction or concretion. Given a sorting ob, an agent A *respects* ob, or is *well-sorted* for ob, if it can be assigned an object sort s by the use of certain rules which ensure that the use of names in A is compatible with ob. So the condition is that $A : s$ can be inferred for some object sort s from the rules

$$\frac{x : S \quad F : ob(S)}{x.F : ()} \qquad \frac{x : S \quad C : ob(S)}{\bar{x}.C : ()}$$

$$0 : () \qquad \frac{M : () \quad N : ()}{M + N : ()}$$

$$\frac{P : () \quad Q : ()}{P|Q : ()} \qquad \frac{P : ()}{!P : ()} \qquad \frac{A : s}{(\nu x)A : s}$$

$$\frac{x : S \quad F : s}{(\lambda x)F : (S)^\frown s} \qquad \frac{x : S \quad C : s}{[x]C : (S)^\frown s}$$

and then s is the (possibly empty) sequence of sorts of the names over which A is abstracted or concreted. Each rule is to be read as a deduction of the conclusion below the line from the hypotheses above the line, in the manner of sequent calculus.

In the rest of the paper we will describe how these rules form the basis of a sort inference algorithm.

3 Sortings

From now on we assume that there is a fixed collection $\mathcal{N}$ of names, which includes all the names used in all the agents being considered. In the following theory, everything could be explicitly parameterised over $\mathcal{N}$, but this seems excessively general. We also assume that all bound names are distinct from each other and from all free names; this can be done as α-conversion is part of the definition of structural congruence. For sortings, we use

Definition 1 A *sorting* Σ is a pair (R, ob) consisting of an equivalence relation R on $\mathcal{N}$ and a partial function $ob : \mathcal{N}/R \rightharpoonup (\mathcal{N}/R)^*$. An *object sort* in the sorting Σ is an element of $(\mathcal{N}/R)^*$.

The idea behind this definition is that the equivalence classes of R represent the assignment of subject sorts to names, so that names x and y have the same subject sort if and only if xRy. Thus we have abandoned

the names of the subject sorts, and retained only the essential information about which names have the same subject sort. The definition of what it means for an agent to respect a sorting is essentially the same as in the previous section, except that we use judgements of the form $\Sigma \vdash \text{Abs } A : s$ to show the sorting being used, the sort assigned to the agent A, and also the class of agent into which A falls.

Definition 2 An agent A *respects* a sorting $\Sigma = (R, ob)$, or is *well-sorted* for Σ, if for some object sort s the judgement $\Sigma \vdash \text{Agent}A : s$ can be derived from the rules in Figure 1.

To show the benefit of working with well-sorted processes, we state (but do not prove)

Proposition 1 If $\Sigma \vdash \text{Proc } P : ()$ and $P \rightarrow Q$ then $\Sigma \vdash \text{Proc } Q : ()$.

from which, noting that the agent ERROR does not respect any sorting, avoidance of run-time error follows.

Corollary 2 If $\Sigma \vdash \text{Proc } P : ()$ then $P \not\rightarrow^* \text{ERROR}$.

Before considering the question of how to infer a sorting for a given agent, we will develop an algebraic theory of sortings a little; this can be conveniently expressed in the language of partial orders. The idea is that sortings form a poset in which moving up the order relation represents instantiating a sorting by combining certain subject sorts and extending the domain of definition of ob.

Definition 3 Let **Sort** be the set of sortings $\Sigma = (R, ob)$, with a partial order defined by $\Sigma \sqsubseteq \Sigma'$ if and only if $R \subseteq R'$ and whenever $ob([x]_R) = ([x_1]_R \ldots [x_n]_R)$, $ob'([x]_{R'}) = ([x_1]_{R'} \ldots [x_n]_{R'})$.

It is easy to check that this does define a partial order. Note that R induces an equivalence relation on each equivalence class of R', so the classes of R' are themselves partitioned into R-classes; these are also the classes into which R partitions $\mathcal{N}$. Viewing this the other way round, the picture is of R' being constructed by amalgamating some of the R-classes of $\mathcal{N}$, as long as this results in ob' being well-defined when it is induced by ob in the obvious way.

Note that **Sort** has a least element, the sorting $(=, \emptyset)$, which we will denote by $\perp$ as usual.

When inferring sortings, we will need to be able to combine sortings which have been inferred for two subprocesses in order to find a sorting for a compound process. The new sorting should be an instance of both previous sortings, and in fact it is clear that we would like it to be the most general such sorting. So we will be interested in least upper bounds in **Sort**. A concrete algorithm to construct $\Sigma_1 \vee \Sigma_2$ will be described later; in order to make it easier to prove the correctness of the algorithm, we will now give a characterisation of the least upper bound of two sortings in terms of a least fixed point construction. Let $\Sigma_1 = (R_1, ob_1)$ and $\Sigma_2 = (R_2, ob_2)$ be sortings for which $\Sigma_1 \vee \Sigma_2$ exists. Consider the function F on the poset $(\text{EqRel}(\mathcal{N}), \subseteq)$, defined in Figure 2, where the function STC is the symmetric transitive closure and $\text{EqRel}(X)$ is the set of equivalence relations on a set X. This definition relies on the existence of $\Sigma_1 \vee \Sigma_2$ to ensure that when $ob_1[x]_{R_1}$ and $ob_2[y]_{R_2}$ are defined with $(x, y) \in R$, their values are sequences of the same length. Since F is continuous and the poset $(\text{EqRel}(\mathcal{N}), \subseteq)$ has a least element (equality), we can define a sorting $\Sigma = (R, ob)$ by taking R to be the least fixed point of F, and ob to be induced by its values on the equivalence classes of R_1 and R_2 (which are bound to be compatible). We now have

Lemma 3 $\Sigma = \Sigma_1 \vee \Sigma_2$.

Proof We have $\Sigma_i \sqsubseteq \Sigma$ since $R_i \subseteq R$ and because of the way in which ob is induced by ob_i. If $\Sigma' = (R', ob')$ is such that $\Sigma_1, \Sigma_2 \sqsubseteq \Sigma'$, then R' is a fixed point of F. Hence $R \subseteq R'$. Also $\Sigma_1, \Sigma_2 \sqsubseteq \Sigma'$ means that ob' is defined on at least those classes for which its definition can be induced from the ob_i, and its values there are those which would be induced. So $\Sigma \sqsubseteq \Sigma'$. □

4 The Sort Inference Algorithm

We can now describe the algorithm which constructs the most general sorting for a given agent, if it exists, assuming the existence of an algorithm which computes $\vee$ or reports that it is undefined. Following the usual practice in presenting type inference algorithms in functional programming, we give a collection of inference rules for sorting judgements, similar to those in the definition of well-sortedness, but which build up a sorting as well as constructing an agent. If the judgement $\Sigma_A \vdash \text{Agent}A : s$ can be deduced from the rules given below, then the agent A is well-sorted for Σ_A and Σ_A is the most general sorting for A. If at any point a computation of $\vee$ is required which is not defined, then A does not respect any sorting. The rules are shown in Figure 3. An important feature of these rules is that there is exactly one for each construction in the grammar which generates agents. Thus given an agent A, the construction of Σ_A is deterministic.

Before proving that this algorithm constructs the desired sorting, we will illustrate it with a couple of exam-

$$\frac{}{\Sigma \vdash \text{NormProc } 0 : ()}\ \text{Zero}$$

$$\frac{\Sigma \vdash \text{NormProc } M : () \quad \Sigma \vdash \text{NormProc } N : ()}{\Sigma \vdash \text{NormProc } M + N : ()}\ \text{Plus}$$

$$\frac{\Sigma \vdash \text{Abs } F : ob[\alpha]_R}{\Sigma \vdash \text{NormProc } \alpha.F : ()}\ \text{Loc} \qquad \frac{\Sigma \vdash \text{Conc } C : ob[\alpha]_R}{\Sigma \vdash \text{NormProc } \alpha.C : ()}\ \text{CoLoc}$$

$$\frac{\Sigma \vdash \text{NormProc } N : ()}{\Sigma \vdash \text{Proc } N : ()}\ \text{Prom}_1$$

$$\frac{\Sigma \vdash \text{Proc } P : () \quad \Sigma \vdash \text{Proc } Q : ()}{\Sigma \vdash \text{Proc } P|Q : ()}\ \text{Comp}$$

$$\frac{\Sigma \vdash \text{Proc } P : ()}{\Sigma \vdash \text{Proc } !P : ()}\ \text{Rep} \qquad \frac{\Sigma \vdash \text{Proc } P : ()}{\Sigma \vdash \text{Proc } (\nu x)P : ()}\ \text{Res}_1$$

$$\frac{\Sigma \vdash \text{Proc } P : ()}{\Sigma \vdash \text{Abs } P : ()}\ \text{Prom}_2 \qquad \frac{\Sigma \vdash \text{Proc } P : ()}{\Sigma \vdash \text{Conc } P : ()}\ \text{Prom}_3$$

$$\frac{\Sigma \vdash \text{Abs } F : s}{\Sigma \vdash \text{Abs } (\lambda x)F : ([x]_R)\hat{}s}\ \text{Abs} \qquad \frac{\Sigma \vdash \text{Conc } C : s}{\Sigma \vdash \text{Conc } [x]F : ([x]_R)\hat{}s}\ \text{Conc}$$

$$\frac{\Sigma \vdash \text{Abs } F : s}{\Sigma \vdash \text{Abs } (\nu x)F : s}\ \text{Res}_2 \qquad \frac{\Sigma \vdash \text{Conc } C : s}{\Sigma \vdash \text{Conc } (\nu x)C : s}\ \text{Res}_3$$

$$\frac{\Sigma \vdash \text{Abs } F : ()}{\Sigma \vdash \text{Agent } F : ()}\ \text{Prom}_4 \qquad \frac{\Sigma \vdash \text{Conc } C : ()}{\Sigma \vdash \text{Agent } C : ()}\ \text{Prom}_5$$

Figure 1: Rules for sort derivation.

$$F(R) = \text{STC}(R \cup R_1 \cup R_2 \cup \{(x_i, y_i) : \exists x, y.(x, y) \in R \& ob_1[x]_{R_1} = ([x_i]_{R_1})_{i=1}^n \& ob_2[y]_{R_2} = ([y_j]_{R_2})_{j=1}^n\})$$

Figure 2: The definition of $F : (\text{EqRel}(\mathcal{N}), \subseteq) \to (\text{EqRel}(\mathcal{N}), \subseteq)$.

$$\frac{}{\bot \vdash \text{NormProc } 0 : ()}\ \text{Zero}$$

$$\frac{\Sigma \vdash \text{NormProc } M : () \quad \Sigma' \vdash \text{NormProc } N : ()}{\Sigma \vee \Sigma' \vdash \text{NormProc } M + N : ()}\ \text{Plus}$$

$$\frac{(R, ob) \vdash \text{Abs } F : ob[\alpha]_R}{(R, ob) \vdash \text{NormProc } \alpha.F : ()}\ \text{Loc}_1 \qquad \text{if } [\alpha]_R \in \text{dom}(ob)$$

$$\frac{(R, ob) \vdash \text{Abs } F : s}{(R, ob \cup \{[\alpha]_R \mapsto s\}) \vdash \text{NormProc } \alpha.F : ()}\ \text{Loc}_2 \qquad \text{if } [\alpha]_R \notin \text{dom}(ob)$$

$$\frac{(R, ob) \vdash \text{Conc } C : ob[\alpha]_R}{(R, ob) \vdash \text{NormProc } \bar{\alpha}.C : ()}\ \text{CoLoc}_1 \qquad \text{if } [\alpha]_R \in \text{dom}(ob)$$

$$\frac{(R, ob) \vdash \text{Conc } C : s}{(R, ob \cup \{[\alpha]_R \mapsto s\}) \vdash \text{NormProc } \bar{\alpha}.C : ()}\ \text{CoLoc}_2 \qquad \text{if } [\alpha]_R \notin \text{dom}(ob)$$

$$\frac{\Sigma \vdash \text{NormProc } N : ()}{\Sigma \vdash \text{Proc } N : ()}\ \text{Prom}_1$$

$$\frac{\Sigma \vdash \text{Proc } P : () \quad \Sigma' \vdash \text{Proc } Q : ()}{\Sigma \vee \Sigma' \vdash \text{Proc } P|Q : ()}\ \text{Comp}$$

$$\frac{\Sigma \vdash \text{Proc } P : ()}{\Sigma \vdash \text{Proc } !P : ()}\ \text{Rep} \qquad \frac{\Sigma \vdash \text{Proc } P : ()}{\Sigma \vdash \text{Proc } (\nu x)P : ()}\ \text{Res}_1$$

$$\frac{\Sigma \vdash \text{Proc } P : ()}{\Sigma \vdash \text{Abs } P : ()}\ \text{Prom}_2 \qquad \frac{\Sigma \vdash \text{Proc } P : ()}{\Sigma \vdash \text{Conc } P : ()}\ \text{Prom}_3$$

$$\frac{\Sigma \vdash \text{Abs } F : s}{\Sigma \vdash \text{Abs } (\lambda x)F : ([x]_R)\hat{}s}\ \text{Abs} \qquad \frac{\Sigma \vdash \text{Conc } C : s}{\Sigma \vdash \text{Conc } [x]F : ([x]_R)\hat{}s}\ \text{Conc}$$

$$\frac{\Sigma \vdash \text{Abs } F : s}{\Sigma \vdash \text{Abs } (\nu x)F : s}\ \text{Res}_2 \qquad \frac{\Sigma \vdash \text{Conc } C : s}{\Sigma \vdash \text{Conc } (\nu x)C : s}\ \text{Res}_3$$

$$\frac{\Sigma \vdash \text{Abs } F : ()}{\Sigma \vdash \text{Agent } F : ()}\ \text{Prom}_4 \qquad \frac{\Sigma \vdash \text{Conc } C : ()}{\Sigma \vdash \text{Agent } C : ()}\ \text{Prom}_5$$

Figure 3: Rules for sort inference.

ples, one of success and one of failure. Our first example comes from Milner's translation of the lazy λ-calculus into the π-calculus. This translation, written as $[\![-]\!]$, is given by

$$\begin{aligned} [\![x]\!] &= (\lambda u)\overline{x}.[u]0 \\ [\![\lambda x.M]\!] &= (\lambda u)u.(\lambda x)[\![M]\!] \\ [\![MN]\!] &= \\ & (\lambda u)(\nu v)([\![M]\!]v|(\nu x)(\overline{v}.[xu]0|!x.[\![N]\!])) \ . \end{aligned}$$

This translation respects the sorting which in Milner's notation is written as

$$\{\text{VAR} \mapsto (\text{ARGS}), \text{ARGS} \mapsto (\text{VAR}, \text{ARGS})\}$$

and which, in our notation, means that there are two equivalence classes of names, called VAR and ARGS. We will not dwell on the details of this translation, but just have a look at the π-calculus term arising from a simple case - the identity function. According to the above definitions, $[\![\lambda x.x]\!] = (\lambda v)v.(\lambda x)(\lambda u)\overline{x}.[u]0$. The π-calculus term contains the names x, u, v; we can describe a sorting by indicating the partitions of x, u, v and the action of ob, so for example the sorting $\perp$ will be written

$$\{x\} \mapsto \perp, \{u\} \mapsto \perp, \{v\} \mapsto \perp$$

where by $\{x\} \mapsto \perp$ we mean that ob is undefined on $\{x\}$. We will omit the undefined instances of ob where this does not cause confusion. The derivation of the sorting

$$\{x\} \mapsto (\{u\}), \{u\} \mapsto \perp, \{v\} \mapsto (\{x\}, \{u\})$$

and the assignment of the sort $(\{v\})$ to the agent $(\lambda v)v.(\lambda x)(\lambda u)\overline{x}.[u]0$ is shown in Figure 4 (uses of the promotion rules Prom_n are omitted). This is the most general sorting respected by this agent; the sorting given by Milner, when expressed in our notation, is

$$\{x\} \mapsto (\{u, v\}), \{u, v\} \mapsto (\{x\}, \{u, v\})$$

(so VAR $= \{x\}$, ARGS $= \{u, v\}$). We can see that Milner's sorting is an instantiation of the one found by our algorithm. This is not to say that his sorting is not the most general one appropriate for the translation of the lazy λ-calculus, but merely demonstrates the fact that a particular process arising from the translation may not constrain the use of its names sufficiently to define that sorting fully.

For an example of the algorithm detecting a badly-sorted process, consider $a.(\lambda x)\overline{x}.0$ and $\overline{a}.[u]u.(\lambda z)0$. The first of these receives a name x on a, then sends a signal on x. The second sends a name u on a, then receives a name z on u. Thus the two processes do not use the channel a in compatible ways, because they make different use of the name transmitted on a. Figure 5 shows the attempt to infer a sorting respected by $(a.(\lambda x)\overline{x}.0)|(\overline{a}.[u]u.(\lambda z)0)$. The problem arises when it is necessary to form the join of the sortings

$$\{x\} \mapsto (), \{u\} \mapsto \perp, \{z\} \mapsto \perp, \{a\} \mapsto (\{x\})$$

and

$$\{x\} \mapsto \perp, \{u\} \mapsto (\{z\}), \{z\} \mapsto \perp, \{a\} \mapsto (\{u\}).$$

Comparing $ob\{a\}$ in the two sortings shows that the combined sorting must have an equivalence class containing $\{x, u\}$, but looking at $ob\{x\}$ in the first sorting and $ob\{u\}$ in the second shows that the combined sorting must have $ob\{x, u\} = ()$ and $ob\{x, u\} = ([z])$, which is contradictory. Hence the two sortings have no join, and the process respects no sorting.

5 Correctness of the Algorithm

We now have to demonstrate that Σ_A is the most general sorting respected by A, in a suitable sense. First we need to know that $\mathbf{Sort}_A$, the set of sortings respected by A, is closed under instantiation.

Lemma 4 If $\Sigma \in \mathbf{Sort}_A$ and $\Sigma \sqsubseteq \Sigma'$ then $\Sigma' \in \mathbf{Sort}_A$.

Proof As usual, let $\Sigma = (R, ob)$ and $\Sigma' = (R', ob')$. Given an object sort $s = ([x_1]_R \ldots [x_n]_R)$, define $s' = ([x_1]_{R'} \ldots [x_n]_{R'})$. From a proof of $\Sigma \vdash A : s$ we construct a proof of $\Sigma' \vdash A : s'$, by induction on the length of the proof. Consider the possible cases for the last step of the proof.

- The base case: if it is

  $$\frac{}{\Sigma \vdash \text{NormProc } 0 : ()} \text{ Zero}$$

  then since $()' = ()$,

  $$\frac{}{\Sigma' \vdash \text{NormProc } 0 : ()} \text{ Zero}$$

  is the desired proof.

- If it is

  $$\frac{\Sigma \vdash \text{Abs } F : ob([\alpha]_R)}{\Sigma \vdash \text{NormProc } \alpha.F : ()} \text{ Loc}$$

  then by induction there is a proof of

  $$\Sigma' \vdash \text{Abs } F : \{ob([\alpha]_R)\}';$$

  but $\{ob([\alpha]_R)\}' = ob'([\alpha]_{R'})$, so this is a proof of

  $$\Sigma' \vdash \text{Abs } F : ob'([\alpha]_{R'})$$

  and hence the Loc rule can be used to prove

  $$\Sigma' \vdash \text{NormProc } \alpha.F : ()'.$$

  The case for the rule CoLoc is similar.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{x\},\{u\},\{v\} \mapsto \bot \vdash \text{NormProc } 0 : ()}\text{Zero}}{\{x\},\{u\},\{v\} \mapsto \bot \vdash \text{Conc } [u]0 : (\{u\})}\text{Conc}}{\{x\} \mapsto (\{u\}) \vdash \text{NormProc } \overline{x}.[u]0 : ()}\text{CoLoc}_1}{\{x\} \mapsto (\{u\}) \vdash \text{Abs } (\lambda u)\overline{x}.[u]0 : (\{u\})}\text{Abs}}{\{x\} \mapsto (\{u\}) \vdash \text{Abs } (\lambda x)(\lambda u)\overline{x}.[u]0 : (\{x\},\{u\})}\text{Abs}}{\{x\} \mapsto (\{u\}),\{v\} \mapsto (\{x\},\{u\}) \vdash \text{NormProc } v.(\lambda x)(\lambda u)\overline{x}.[u]0 : ()}\text{Loc}_1}{\{x\} \mapsto (\{u\}),\{v\} \mapsto (\{x\},\{u\}) \vdash \text{Abs } (\lambda v)v.(\lambda x)(\lambda u)\overline{x}.[u]0 : (\{v\})}\text{Abs}$$

Figure 4: An example of sort inference.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{x\},\{a\},\{u\},\{z\} \mapsto \bot \vdash 0 : ()}\text{Zero}}{\{x\} \mapsto () \vdash \overline{x}.0 : ()}\text{CoLoc}_2}{\{x\} \mapsto () \vdash (\lambda x)\overline{x}.0 : (\{x\})}\text{Abs}}{\{x\} \mapsto (),\{a\} \mapsto (\{x\}) \vdash a.(\lambda x)\overline{x}.0 : ()}\text{Loc}_2 \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{x\},\{a\},\{u\},\{z\} \mapsto \bot \vdash 0 : ()}\text{Zero}}{\{x\},\{a\},\{u\},\{z\} \mapsto \bot \vdash (\lambda z)0 : (\{z\})}\text{Abs}}{\{u\} \mapsto \{z\} \vdash u.(\lambda z)0 : ()}\text{Loc}_2}{\{u\} \mapsto \{z\} \vdash [u]u.(\lambda z)0 : (\{u\})}\text{Conc} \qquad \{u\} \mapsto \{z\},\{a\} \mapsto (\{u\}) \vdash \overline{a}.[u]u.(\lambda z)0 : ()}\text{CoLoc}_2}{}\text{Comp}$$

The sortings in the premises have no join.

Figure 5: Detecting a badly-sorted process.

- If it is

$$\dfrac{\Sigma \vdash \text{Abs } F : s}{\Sigma \vdash \text{Abs } (\lambda x)F : ([x]_R)\hat{}s}\text{Abs}$$

then by induction there is a proof of

$$\Sigma' \vdash \text{Abs } F : s'$$

which can be extended to a proof of

$$\Sigma' \vdash \text{Abs } (\lambda x)F : ([x]_{R'})\hat{}s'$$

by the Abs rule, and this is the required proof since $([x]_R)\hat{}s' = ([x]_{R'})\hat{}s'$. The case for the rule Conc is similar.

- All the other cases are trivial since they do not change the sorting of the agent involved.

□

Thus instantiation preserves respectability. The most general sorting respected by A should be the least instantiated sorting in $\mathbf{Sort}_A$, which means the least element of the poset. Before proving this, we should check that A does respect the sorting Σ_A, if Σ_A exists.

Lemma 5 All the sorting judgements appearing in the algorithm which constructs Σ_A are valid.

Proof We check that the rules in the algorithm preserve validity of judgements. There are three essentially different cases.

- The rule Zero is sound because

$$\Sigma \vdash \text{NormProc } 0 : ()$$

for any sorting Σ.

- For the rule Plus, observe that since $\Sigma, \Sigma' \sqsubseteq \Sigma \vee \Sigma'$, M and N respect $\Sigma \vee \Sigma'$, so we have

$$\Sigma \vee \Sigma' \vdash \text{NormProc } M : ()$$

and

$$\Sigma \vee \Sigma' \vdash \text{NormProc } N : ()$$

from which we can deduce

$$\Sigma \vee \Sigma' \vdash \text{NormProc } M + N : ().$$

- The rule Loc_2 is sound because $(R, ob) \sqsubseteq (R, ob')$ and by the same argument as for the previous case.

□

Corollary 6 If Σ_A is defined then $\Sigma_A \in \mathbf{Sort}_A$.

Finally, we can prove the desired property of Σ_A.

Proposition 7 For any agent A, if Σ_A is defined then it is the least element of $\mathbf{Sort}_A$.

Proof We show by induction that if A respects Σ and Σ' appears during the construction of Σ_A, $\Sigma' \sqsubseteq \Sigma$; hence if $\Sigma \in \mathbf{Sort}_A$ and Σ_A is defined, $\Sigma_A \sqsubseteq \Sigma$. There are essentially three cases to consider.

- A respects the sorting $\bot$ which starts the construction off.
- For the rule Plus we have by induction that $\Sigma_1, \Sigma_2 \sqsubseteq \Sigma$, and so $\Sigma_1 \vee \Sigma_2 \sqsubseteq \Sigma$ since it is the least upper bound.
- For the rule Loc_2 we have by induction that $(R_1, ob_1) \sqsubseteq \Sigma$. In the proof that A respects Σ, the rule Loc_2 must be used with sorting Σ, which means that $ob : [\alpha]_R \mapsto s$. If

  $$(ob_1 \cup \{[\alpha]_{R_1} \mapsto s\})([x]_{R_1}) = ([x_1]_{R_1} \ldots [x_n]_{R_1})$$

  then either

  $$ob_1([x]_{R_1}) = ([x_1]_{R_1} \ldots [x_n]_{R_1})$$

  or $[x]_{R_1} = [\alpha]_{R_1}$. In the first case,

  $$ob_1([x]_R) = ([x_1]_R \ldots [x_n]_R)$$

  since $(R_1, ob_1) \sqsubseteq \Sigma$, and in the second case

  $$s = ([x_1]_{R_1} \ldots [x_n]_{R_1}) = ob([\alpha]_R).$$

  Hence

  $$(R_1, ob_1 \cup \{[\alpha]_{R_1} \mapsto s\}) \sqsubseteq \Sigma.$$

□

6 Computing Least Upper Bounds of Sortings

We now need to describe the algorithm which given $\Sigma_1 = (R_1, ob_1)$ and $\Sigma_2 = (R_2, ob_2)$, calculates $\Sigma = \Sigma_1 \vee \Sigma_2 = (R, ob)$ or reports that it does not exist. The algorithm is shown in Figure 6. It uses the functions TC and STC which compute the transitive closure and symmetric transitive closure of their arguments, respectively.

We now prove the correctness of this algorithm in an informal way. Of course, as we are now at the stage of considering real computation, we assume that $\mathcal{N}$ is finite; the actual set $\mathcal{N}$ of names used in a given agent can be determined syntactically.

Lemma 8 Algorithm 1 always terminates.

Algorithm 1 To compute the least upper bound of two sortings, if possible.

procedure lub(R_1, ob_1, R_2, ob_2,R,ob)
$R := \mathrm{TC}(R_1 \cup R_2)$
repeat
  $l := \emptyset$
  foreach $(x, y) \in \mathcal{N}^2$ **do**
    if $(x, y) \in R$
      and $ob_1[x]_{R_1} = ([x_i]_{R_1})_{i=1}^m$
      and $ob_2[y]_{R_2} = ([y_j]_{R_2})_{j=1}^n$ **then**
        if $m \neq n$ **then**
          fail
        else
          foreach $i = 1 \ldots n$ **do**
            if $(x_i, y_i) \notin R$ **then**
              $l := l \cup \{(x_i, y_i)\}$
            endif
          endfor
        endif
    endif
  endfor
  foreach $(x, y) \in l$ **do**
    $R := \mathrm{STC}(R \cup \{(x, y)\})$
  endfor
until $l = \emptyset$
foreach $x \in \mathcal{N}$ **do**
  if $ob_1[x]_{R_1} = ([x_i]_{R_1})_{i=1}^n$
    and $ob_2[x]_{R_2} = ([y_j]_{R_2})_{j=1}^n$ **then**
      $ob([x]_R) := ([x_i]_R)_{i=1}^n$
  endif
endfor
end procedure

Figure 6: Computing least upper bounds.

Proof The set l is used during each pass through $\mathcal{N}^2$ to accumulate the extra pairs which need to be added to R. Each such pair which is found represents a pair of equivalence classes of R which need to be joined together. Since there are only finitely many equivalence classes in $\mathrm{TC}(R_1 \cup R_2)$ to start with, it must be the case that after some finite number of passes through $\mathcal{N}^2$, no more pairs will be found on the next pass. When this happens, $l = \emptyset$ at the end of a pass, and this is the case in which the algorithm terminates. □

Proposition 9 When Algorithm 1 terminates,

$$(R, ob) = \Sigma_1 \vee \Sigma_2.$$

Proof The algorithm is directly computing the least fixed point of the function F by iteration. The only difference is that the algorithm makes explicit the fact that the existing relations R_1 and R_1 only need to be incorporated at the first iteration. □

A few words about the complexity of the sort inference algorithm are in order. When computing the most general sorting respected by a given agent A, the function $\vee$ is calculated $s + c$ times where s and c are the numbers of occurrences of $+$ and $|$ in A. As for the complexity of computing $\vee$, if $|\mathcal{N}| = n$ then computing $\mathrm{TC}(R_1 \cup R_2)$ takes time $O(n^3)$, Combine is called $O(n)$ times, joining two equivalence classes takes time $O(n)$ and finding the length of an object sort should take time $O(n)$ with the result that computing $\vee$ is $O(n^3)$. The problem with this analysis is that the length of an object sort cannot be bounded by n because of the fact that introducing a name into a concretion does not bind it and so the sort of a concretion can be arbitrarily long. If the maximum size of a concretion is m then the time taken to compute $\vee$ is $O(n^3 + n^2 m)$. We can bound $s + c$, n and m by the size p of the textual representation of the process; this gives a bound of $O(p^4)$ on the time taken to calculate the most general sorting.

7 Conclusions

The idea of sorts in the π-calculus extends the benefits of typing from a sequential to a concurrent setting, providing a useful tool in program design. The algorithm presented in this paper demonstrates that sort checking and sort inference could also be incorporated into an implementation of the π-calculus offering the same support to the programmer as type checking and type inference do in functional language implementations.

8 Acknowledgements

It should be noted that work in the general area of sortings, including sort inference, has been carried out independently by David N. Turner (personal communication); his work is as yet unpublished and, I believe, uses a rather different approach.

I would like to thank Samson Abramsky, Roy Crole, Radha Jagadeesan, Hiu Fai Chau and Sebastian Hunt for their helpful comments and suggestions during the preparation of this paper; also the anonymous referees who made some valuable comments.

References

[Hin69] J.R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil91] R. Milner. The polyadic π-calculus: A tutorial. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1991.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1989.