Published in U. Montanari and V. Sassone, editors, *CONCUR'96: Concurrency Theory*, volume 1119 of *LNCS*, pages 389–405. Springer-Verlag, 1996.

Constraints as Processes

Björn Victor * Joachim Parrow †

Abstract

We present a compositional encoding of the γ-calculus into the π-calculus. The former, used in the Oz semantics, is a recent small language with equational constraints over logical variables; the latter is a basic calculus of interacting processes. We establish a close correspondence between the reductions in the γ-calculus and its encoding, using weak barbed bisimulation congruence.

1 Introduction

In this paper we shall relate two different strands in the semantics of programming languages. One is the concurrent constraints paradigm where facts about data values combine to resolve goals; the other is the communicating processes paradigm where parallel agents interact through data transmission. We shall use two small semantic calculi which embody these paradigms in a rigorous and tractable format: the γ-calculus for concurrent constraints and the π-calculus for communicating processes. These calculi have been deemed successful as semantic bases for programming languages in their respective areas. Our main result, that the γ-calculus has a natural encoding in the π-calculus, implies that many of the techniques and tools developed for process calculi can be transferred to the concurrent constraints community.

*Dept. of Computer Systems, Uppsala University, Box 325, S-751 05 Uppsala, Sweden. Work partially supported by ESPRIT BRA project No. 8130: LOMAPS.

†Dept. of Teleinformatics, Royal Institute of Technology, Electrum 204, S-164 40 Kista, Sweden. Work partially supported by The Human Capital and Mobility Project EXPRESS.

The relevance of the γ-calculus lies in its relationship to Oz [Smo95b, Smo94, Smo95a]. Oz is a real programming language; its performance is comparable to commercial implementations of Lisp or Prolog, it has an extensive programming environment, and it supports important concepts like higher-order functions, object-oriented programming, and concurrent constraints. Its computational model, the Oz calculus [Smo95a], has been inspired by process calculi but analysis methods such as equivalence checking and model checking have not been applied to it. This is partly because the calculus contains elements such as logical variables with no direct counterparts in other calculi. The γ-calculus [Smo94] is a simplified version of the Oz calculus; it retains the concept of logical variable and a primitive form of constraint solving, *elimination*. An example is in the reduction

$$E \wedge x = a \quad \longrightarrow \quad E\{x/a\}.$$

Here $E \wedge x = a$ is a γ-term, namely a conjunction of a subterm E with the "fact" that x is a; the reduction results in $E\{x/a\}$, which is E where the logical variable x is "eliminated" and replaced by a.

In a process calculus the terms denote processes, called *agents*, interacting by sending data over named ports. The distinctive feature of the π-calculus [MPW92, Mil91] is that the data transmitted are again ports. There is no difference in kind between ports and data, and they are collectively called *names*. The calculus has a considerable expressive power, as witnessed by its use in, e.g., the semantics of the programming language PICT [PRT93] and of object-oriented programming [Wal95]; it can naturally encode higher-order communications [San92] and the λ-calculus [Mil92]. There is a variety of proof methods based on bisimulations [MPW92, San96, San92, San95, PS96], rewrite systems [PS95] and model checking [Dam93, AD96], and automated tools for these are emerging [VM94, FMQ95].

An example π-calculus reduction is the following *interaction*:

$$b(x)\,.\,P \mid \overline{b}\langle a\rangle\,.\,Q \quad \longrightarrow \quad P\{a/x\} \mid Q.$$

Here $b(x)\,.\,P$ is an "input" which receives something for x along b and continues as the subagent P, and $\overline{b}\langle a\rangle\,.\,Q$ is an "output" which transmits a along b and continues as the subagent Q. When combined with the parallel operator "$|$" they may interact and reduce to $P\{a/x\} \mid Q$, where bound occurrences of x are replaced everywhere in P by a.

At first it may seem a trivial task to encode elimination by interaction, since in both cases the result is a substitution of atomic entities. The difficulty lies in the extent that other reductions are permissible. In the example of elimination above, the subterm E may have other reductions and these are inherited by the whole term $E \wedge x = a$. In this way an elimination can be deferred in favour of reductions where the (not yet eliminated) logical variable x can play a role. In contrast, the π-calculus input $b(x) \, . \, P$ must await its input along b before any reduction within P can occur. No reductions involving names bound by input are possible. Another difference between the calculi is in the conditional construct: in the π-calculus an outermost conditional can always be decided (because the only conditions are equalities between names), while in the γ-calculus the conditions include "undecided" equalities between not yet eliminated logical variables.

We will demonstrate an encoding of the γ-calculus into the π-calculus which is compositional, i.e., the encoding of a term is derived from the encodings of its subterms. The main problem is the treatment of logical variables. We solve it by introducing special *handler* agents, one for each variable, which administer eliminations and resolve conditions. To argue that the encoding is correct relative to the predefined reduction relations we use the concept of *barbed bisimulation congruence*, originally introduced for the π-calculus [MS92a, San92] but equally applicable to the γ-calculus. In essence, two agents (or two terms) are bisimilar if their reductions can mimic indefinitely, and they are congruent if they are bisimilar in all contexts. Our correctness criterion is that two terms are congruent if and only if their encodings are congruent. In short, this means that the encoding is neither too sloppy in that it disregards significant aspects of terms, nor too picky in that it takes insignificant aspects into consideration.

Related work: Smolka [Smo94] inspired our research by relating the γ-calculus to a variant of the π-calculus with variables, equations and an elimination rule. We show that these additions can be dropped. Niehren and Müller [NM95] relate the γ-calculus to the ρ-calculus with equational constraints ($\rho(x = y)$), and observe that the ρ-calculus without constraints ($\rho(\emptyset)$) is a subcalculus both of $\rho(x = y)$ and of the π-calculus. We strengthen their result by formally showing the correspondence between the γ-calculus and π-calculus directly.

Ross [Ros93] gives π-calculus specifications of simplified logic variables and unification, and Li [Li94] gives a π-calculus specification of Prolog, including logic variables and unification, with support for backtracking. Neither of these provide proofs of formal correctness.

The programming languages PICT [PRT93] and Facile [TLP+93] have semantics formulated in π-calculus and CHOCS [Tho90], respectively. They support concurrent, functional and (in the PICT case) object-oriented programming, but not constraints.

Outline: The rest of the paper is structured as follows. In Sections 2 and 3 we give brief introductions to the γ-calculus and the π-calculus. In Section 4 we present our encoding of the γ-calculus in the π-calculus, and in Section 5 we prove that it is correct relative to weak barbed bisimulation. Finally in Section 6 we conclude with some areas for future work.

2 The γ-calculus

The γ-calculus is a simplified version of the Oz calculus, with equational constraints and where search is omitted. In this section we give a brief presentation of its syntax and semantics. For a fuller treatment we refer the reader to [Smo94].

There are three entities in the γ-calculus: *names* (ranged over by $a, b, c, \ldots$), *variables* (ranged over by $x, y, z, \ldots$), and *terms* (ranged over by $E, F, \ldots$). Names and variables are collectively referred to as *references* (ranged over by $u, v, w, \ldots$).

The syntax of the γ-calculus is given by the following BNF equation (the terminology in the equation comes from [Smo94]):

$$
\begin{array}{rcll}
E & ::= & \top & \textit{null} \\
 & | & E_1 \wedge E_2 & \textit{composition} \\
 & | & \exists u E & \textit{declaration} \\
 & | & u = v & \textit{equation} \\
 & | & u\colon \tilde{x}/E & \textit{abstraction} \\
 & | & u\tilde{v} & \textit{application} \\
 & | & \texttt{if } u = v \texttt{ then } E_1 \texttt{ else } E_2 & \textit{conditional} \\
 & | & u\colon v & \textit{cell}
\end{array}
$$

We have generalized the syntax from [Smo94] by allowing variables as locations of abstractions and cells. This admits terms like $\exists a, b, c(a\colon x/\exists y(x\colon y) \wedge ab \wedge bca)$. (The semantics remains the same – only abstractions and cells with

1. $E \equiv_\gamma F$ if E is α-convertible to F
2. Abelian monoid laws for $\wedge$ and $\top$:
   $E \wedge F \equiv_\gamma F \wedge E, E \wedge (F \wedge G) \equiv_\gamma (E \wedge F) \wedge G, E \wedge \top \equiv_\gamma E$
3. $\exists u \top \equiv_\gamma \top$, $\exists u \exists v E \equiv_\gamma \exists v \exists u E$, $\exists u E \wedge F \equiv_\gamma \exists u(E \wedge F)$ if $u \notin \mathrm{fr}(F)$
4. $u = v \equiv_\gamma v = u$

Figure 1: Structural congruence for γ.

names as locations can take part in a reduction using the $\textsc{Appl}_\gamma$ or $\textsc{Exch}_\gamma$ rules which follow.)

Briefly, $\top$ represents an inactive term; each term in a composition can reduce in parallel; a declaration $\exists u E$ introduces a new reference u with scope E; an equation $u = v$ imposes the equality of u and v on all terms; an abstraction $a\colon \tilde{x}/E$ can communicate with an application $a\tilde{v}$, producing the new term $E\{\tilde{v}/\tilde{x}\}$. A cell $a\colon u$ can be seen as holding the value u, and can interact with an application avw, updating the cell's value to w and simultaneously extracting the old value rendering the equation $u = v$. A conditional $\texttt{if}\ u = v\ \texttt{then}\ E_1\ \texttt{else}\ E_2$ tests whether u and v are equal.

The semantics of the γ-calculus is defined by a structural congruence (Figure 1) and a reduction relation (Figure 2). A reference u is bound in $\exists u E$ and in $v\colon u/E$ (but *not* in $v\colon u$). We use $\mathrm{br}(E)$ to refer to the bound references of P, and $\mathrm{fr}(E)$ for the free references. A substitution $\{\tilde{u}/\tilde{v}\}$ simultaneously replaces each v_i with u_i.

An example of a reduction sequence shows that even though variables cannot be used for interaction in the $\textsc{Appl}_\gamma$ or $\textsc{Exch}_\gamma$ rules, they can take part in reductions:

$$\exists x(x = y \wedge \texttt{if}\ x = y\ \texttt{then}\ E\ \texttt{else}\ E') \overset{\textsc{Elim}_\gamma}{\to_\gamma} \texttt{if}\ y = y\ \texttt{then}\ E\ \texttt{else}\ E' \overset{\textsc{Then}_\gamma}{\to_\gamma} E$$

$$\textsc{Struct}_\gamma : \frac{E \equiv_\gamma E' \qquad E' \rightarrow_\gamma F' \qquad F' \equiv_\gamma F}{E \rightarrow_\gamma F}$$

$$\textsc{Comp}_\gamma : \frac{E \rightarrow_\gamma E'}{E \wedge F \rightarrow_\gamma E' \wedge F} \qquad \textsc{Decl}_\gamma : \frac{E \rightarrow_\gamma E'}{\exists u E \rightarrow_\gamma \exists u E'}$$

$$\textsc{Appl}_\gamma : \; a\tilde{u} \wedge a\colon \tilde{x}/E \rightarrow_\gamma E\{\tilde{u}/\tilde{x}\} \wedge a\colon \tilde{x}/E \;\; \text{, if } \tilde{u} \text{ free for } \tilde{x} \text{ in } E \text{ and } |\tilde{u}| = |\tilde{x}|$$

$$\textsc{Then}_\gamma : \; \texttt{if } u = u \texttt{ then } E \texttt{ else } F \rightarrow_\gamma E$$

$$\textsc{Else}_\gamma : \; \texttt{if } a = b \texttt{ then } E \texttt{ else } F \rightarrow_\gamma F \;\; \text{, if } a \neq b$$

$$\textsc{Exch}_\gamma : \; a\colon u \wedge avw \rightarrow_\gamma a\colon w \wedge u = v$$

$$\textsc{Elim}_\gamma : \; \exists x(x = u \wedge E) \rightarrow_\gamma E\{u/x\} \;\; \text{, if } x \neq u \text{ and } u \text{ free for } x \text{ in } E$$

Figure 2: Reduction relation for γ.

3 The π-calculus

In this section we give a brief presentation of the syntax and semantics of the π-calculus. For fuller treatment we refer the reader to, e.g., [Mil91, San92].

There are two entities in the π-calculus: *names* (ranged over by lowercase letters) and *agents* (ranged over by $P, Q, \ldots$). The syntax of the π-calculus is given by the following BNF equation:

$$\begin{array}{rcll}
P & ::= & \mathbf{0} & \textit{null} \\
 & | & P_1 \mid P_2 & \textit{composition} \\
 & | & P_1 + P_2 & \textit{summation} \\
 & | & (\boldsymbol{\nu} x)P & \textit{restriction} \\
 & | & a(\tilde{b}) \,.\, P & \textit{input} \\
 & | & \overline{a}\langle\tilde{b}\rangle \,.\, P & \textit{output} \\
 & | & [a = b](P_1 \,,\, P_2) & \textit{conditional} \\
 & | & !\, P & \textit{replication} \\
 & | & A(a_1, \ldots, a_k) & \textit{identifier}
\end{array}$$

where A ranges over some set of identifiers with associated nonnegative arities k and agent definitions $A(b_1, \ldots, b_k) \stackrel{def}{=} P$.

Briefly, $\mathbf{0}$ represents an inactive agent; each agent in a composition can execute in parallel; a summation $P_1 + P_2$ offers a choice between the agents P_1 and P_2; a restriction $(\boldsymbol{\nu} b)P$ introduces a new name b with scope P; an input prefix $a(\tilde{b}) . P_1$ can communicate with an output $\overline{a}\langle\tilde{c}\rangle . P_2$, reducing to the agent $P_1\{\tilde{c}/\tilde{b}\} \mid P_2$. A conditional $[a = b](P_1\,,\, P_2)$ reads as *"if $a = b$ then P_1 else P_2"* and tests whether a and b are equal; a replication $!\,P$ corresponds to an unbounded number of copies of P in parallel. For convenience we use both replication and recursive definitions although one could be expressed in terms of the other (see [Mil91, section 3]), as long as we use a finite number of identifiers. Note that we include the binary conditional [PS95] whereas in most presentations of the π-calculus there is only the unary form $[x = y]P$. The binary form facilitates our encoding since the γ-calculus has an if-then-else conditional.

We only use guarded summation, i.e., summands must be prefix forms. This admits a definition of the semantics of the π-calculus by the structural congruence in Figure 3 and the reduction relation in Figure 4. Note that the $\textsc{Match}_\pi$ and $\textsc{Mismatch}_\pi$ rules have no premises. This format enables us to deduce $\textsc{Comm}_\pi$ reductions of agents with conditionals in parallel compositions.

A name a is bound in $(\boldsymbol{\nu} a)P$ and in $b(a) . P$. We use $\mathrm{bn}(P)$ to refer to the bound names of P, and $\mathrm{fn}(P)$ for the free names of P. A substitution $\{\tilde{a}/\tilde{b}\}$ simultaneously replaces each b_i with a_i.

Some notational conventions: $\overline{a}\langle b, \boldsymbol{\nu} c, d\rangle . P$ is shorthand for $(\boldsymbol{\nu} c)\overline{a}\langle b, c, d\rangle . P$; we use an underscore _ to signify a name not used anywhere else; and we often omit trailing $\mathbf{0}$ in inputs and outputs (e.g., $\overline{a}\langle b\rangle$ stands for $\overline{a}\langle b\rangle . \mathbf{0}$).

4 Encoding γ in π

4.1 Handlers

The main difficulty when encoding the γ-calculus is the treatment of variables. As a result of the $\textsc{Elim}_\gamma$ rule a variable can be substituted by another reference. The only mechanism for substitution in the π-calculus is in the interaction between input and output prefixes. Unfortunately we cannot use

1. $P \equiv_\pi Q$ if P is α-convertible to Q
2. Abelian monoid laws for $+$ and $\mathbf{0}$:
   $P + Q \equiv_\pi Q + P, P + (Q + R) \equiv_\pi (P + Q) + R, P + \mathbf{0} \equiv_\pi P$
3. Abelian monoid laws for $\mid$ and $\mathbf{0}$:
   $P \mid Q \equiv_\pi Q \mid P, P \mid (Q \mid R) \equiv_\pi (P \mid Q) \mid R, P \mid \mathbf{0} \equiv_\pi P$
4. $(\boldsymbol{\nu} x)\mathbf{0} \equiv_\pi \mathbf{0}, (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv_\pi (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P, (\boldsymbol{\nu} x)P \mid Q \equiv_\pi (\boldsymbol{\nu} x)(P \mid Q)$ if $x \notin \mathrm{fn}(Q)$
5. $!\,P \equiv_\pi P \mid\, !\,P$
6. $A(\tilde{x}) \equiv_\pi P\{\tilde{x}/\tilde{y}\}$ if $A(\tilde{y}) \stackrel{def}{=} P$ and $|\tilde{x}| = |\tilde{y}|$

Figure 3: Structural congruence for π.

that mechanism directly to emulate the $\textsc{Elim}_\gamma$ rule. A π-calculus name which can be substituted in an interaction must be bound in an input prefix, and this prefix must reduce in the $\textsc{Comm}_\pi$ rule before the rest of the agent can reduce. In contrast, in the γ-calculus a term can reduce before its variables have been eliminated.

Our solution to this difficulty is to introduce one π-calculus *handler* agent for each reference not bound by abstraction. A handler interacts with its environment (the encoding of the term where the reference occurs) to perform the substitutions corresponding to the $\textsc{Elim}_\gamma$ rule, and to report its current *value*. Originally the value of a handler is the corresponding reference, but the value can change as a result of *updates* emanating from γ-calculus equations. The handlers thus collectively contain a global state corresponding to the eliminations in a reduct.

We distinguish between three kinds of handlers. $N(\underline{a})$ is a handler for a name a (the underscore will be explained below). It will not update to another reference and always gives its value as a. A variable x which has been updated to a reference u is handled by $R(\underline{x}, \underline{u})$. When queried for its value or asked to update it will relay the request to the handler for u. A

$$\textsc{Struct}_\pi : \frac{P \equiv_\pi P' \qquad P' \to_\pi Q' \qquad Q' \equiv_\pi Q}{P \to_\pi Q}$$

$$\textsc{Match}_\pi : [x = x](P\,,\,Q) \to_\pi P \quad \textsc{Mismatch}_\pi : [x = y](P\,,\,Q) \to_\pi Q \text{ if } x \neq y$$

$$\textsc{Par}_\pi : \frac{P \to_\pi P'}{P \mid Q \to_\pi P' \mid Q} \qquad \textsc{Res}_\pi : \frac{P \to_\pi P'}{(\boldsymbol{\nu} x)P \to_\pi (\boldsymbol{\nu} x)P'}$$

$$\textsc{Comm}_\pi : (\cdots + x(\tilde{y})\,.\,P) \mid (\overline{x}\langle\tilde{z}\rangle\,.\,Q + \cdots) \to_\pi P\{\tilde{z}/\tilde{y}\} \mid Q \text{ if } |\tilde{y}| = |\tilde{z}|$$

Figure 4: Reduction relation for π.

variable x not yet eliminated is handled by $V(\underline{x})$. It gives x as value, and when asked to update to u it changes state to $R(\underline{x}, \underline{u})$. In this way we think of N as standing for "name", V for "variable" and R for "relay" in our encoding.

Since the reference u in $R(\underline{x}, \underline{u})$ may itself be represented by a handler of type R (if u is eliminated after x) the handlers will form a forest. At the root of each tree in the forest sits a handler of type V or N; interior nodes are type R handlers who will relay any queries to its parent. An update of a V root will graft its tree to another tree. Each tree forms an equivalence class of references under the substitutions from $\textsc{Elim}_\gamma$ rules. We call such a tree an *equivalence tree*. See Figure 5 for an example of how an update affects the equivalence trees.

The γ-calculus is not symmetric in its treatment of variables and names — e.g., the $\textsc{Else}_\gamma$ rule only applies when a and b are names — so a handler additionally must report whether its reference is a name. For the same reason it is convenient to let handlers deal with the queries of equality between references that emanate from the conditional terms in the γ-calculus. Such queries can only be resolved in case the references are identical references or distinct names; an equality between distinct variables must be deferred to a later stage.

To let the handlers perform their tasks we introduce, for each γ-calculus

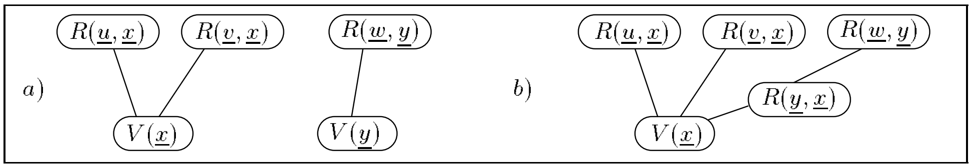

Figure 5: Equivalence trees before (a) and after (b) w updates to v. The handler for w relays the update to its root y, which updates to the value of v, namely x.

reference u, the five π-calculus names u, $u \triangleright \mathsf{name}$, $u \triangleright \mathsf{value}$, $u \triangleright \mathsf{update}$, $u \triangleright \mathsf{eq}$, called the *handler interface* names, which we collectively refer to using the notation $\underline{u}$. The symbol $\triangleright$ is part of the names. We use a notation $x\colon (a, b, c)$ to specify that names a, b, c are transmitted along x in the following explanation.

— u itself is used for the interaction between encodings of applications and abstractions or cells.

— $u \triangleright \mathsf{name}\colon (a)$ ascertains that u has the γ-calculus name a as root in its equivalence tree.

— $u \triangleright \mathsf{value}\colon (c, v)$ queries u for its value, by passing a π-calculus name c where the value is to be received, and the identity v of the source of the query. The value of a reference is the root of its equivalence tree. The source of the query, v, is used to avoid circular queries.

— $u \triangleright \mathsf{update}\colon (\underline{v})$ tells u to update to the reference v. Care must be taken to avoid circular updates.

— $u \triangleright \mathsf{eq}\colon (\underline{v}, y, n)$ asks u to decide whether it is in the same equivalence tree as v, and if it is, to signal on y, or if they are both names and not equal, to signal on n.

Finally, note that variables bound by abstraction have no handlers. This is because the $\textsc{Appl}_\gamma$ rule can be emulated with the $\textsc{Comm}_\pi$ rule.

4.2 The Encoding

The definitions of handlers can now be presented in Figure 6 on the next page.

A handler $N(\underline{a})$ never changes state to another type of handler; it can always handle requests and therefore uses replication in its definition. If it

$$
\begin{array}{rcl}
\underline{u} & \equiv & u,\, u{\triangleright}\mathsf{name},\, u{\triangleright}\mathsf{value},\, u{\triangleright}\mathsf{update},\, u{\triangleright}\mathsf{eq} \\[2ex]
N(\underline{a}) & \equiv & \quad !\, \overline{a{\triangleright}\mathsf{name}}\langle \underline{a} \rangle \\
 & & \mid\, !\, a{\triangleright}\mathsf{value}(c, \_)\,.\, \overline{c}\langle \underline{a} \rangle \\
 & & \mid\, !\, a{\triangleright}\mathsf{update}(\underline{u})\,.\, \overline{u{\triangleright}\mathsf{update}}\langle \underline{a} \rangle \\
 & & \mid\, !\, a{\triangleright}\mathsf{eq}(\underline{u}, y, n)\,.\, u{\triangleright}\mathsf{name}(\underline{b})\,.\, [a = b](\overline{y}\,,\, \overline{n}) \\[2ex]
V(\underline{x}) & \stackrel{def}{=} & \quad x{\triangleright}\mathsf{value}(c, \_)\,.\, \overline{c}\langle \underline{x} \rangle\,.\, V(\underline{x}) \\
 & & +x{\triangleright}\mathsf{update}(\underline{u})\,. \\
 & & \qquad [x = u]\Big(V(\underline{x})\,,\, \overline{u{\triangleright}\mathsf{value}}\langle \boldsymbol{\nu} c, x \rangle\,.\, c(\underline{u})\,. \\
 & & \qquad\qquad\qquad\qquad [x = u](V(\underline{x})\,,\, R(\underline{x}, \underline{u}))\Big) \\
 & & +x{\triangleright}\mathsf{eq}(\underline{u}, y, n)\,. \\
 & & \qquad \Big(\quad V(\underline{x}) \\
 & & \qquad\ \ \mid\, \overline{u{\triangleright}\mathsf{value}}\langle \boldsymbol{\nu} c, x \rangle\,.\, c(\underline{u})\,.\, [x = u](\overline{y}\,,\, \overline{x{\triangleright}\mathsf{eq}}\langle \underline{u}, y, n \rangle)\Big) \\[2ex]
R(\underline{x}, \underline{u}) & \equiv & \quad !\, u{\triangleright}\mathsf{name}(\underline{a})\,.\, \overline{x{\triangleright}\mathsf{name}}\langle \underline{a} \rangle \\
 & & \mid\, !\, x{\triangleright}\mathsf{value}(c, v)\,.\, [u = v]\Big(\overline{c}\langle \underline{u} \rangle\,,\, \overline{u{\triangleright}\mathsf{value}}\langle c, v \rangle\Big) \\
 & & \mid\, !\, x{\triangleright}\mathsf{update}(\underline{v})\,.\, \overline{u{\triangleright}\mathsf{update}}\langle \underline{v} \rangle \\
 & & \mid\, !\, x{\triangleright}\mathsf{eq}(\underline{v}, y, n)\,.\, \overline{u{\triangleright}\mathsf{eq}}\langle \underline{v}, y, n \rangle
\end{array}
$$

Figure 6: Definitions of handlers.

is asked to update to some other reference u through $a{\triangleright}\mathsf{update}(\underline{u})$ it simply reflects the request and asks u to update a instead. The case where a name is asked to update to another name then results in a divergence, but this is not a problem, since an equation $a = b$ has no computational significance in the γ-calculus. In an equality check $a{\triangleright}\mathsf{eq}$, the other reference is established to be a γ-calculus name, and this name and a itself are compared using a π-calculus conditional, and y or n is signalled appropriately.

The definition of a handler $V(\underline{x})$ is more intricate. When handling $x{\triangleright}\mathsf{update}(\underline{u})$ to change into $R(\underline{x}, \underline{u})$, we must make sure the equivalence tree does not become circular; if it did, the handlers would end up relaying requests to each other. This could happen in two cases: (1) if $V(\underline{x})$ is asked to update to itself or to a reference in the same equivalence tree, so we must test for this situation by reading the value of the target u and check if it is x using a π-calculus conditional. Circularity can also occur (2) if two or more variables try to update to each other simultaneously. This can be avoided by not allowing any other variable to update to x while $V(\underline{x})$ is handling an $x \triangleright \mathsf{update}$ request. This is accomplished by *not* handling $x \triangleright \mathsf{value}$ requests during update sections, and vice versa, since another variable must read the value of the reference it is updating to. But this means that in the case that x is told to update to itself, $V(\underline{x})$ now cannot respond to its own $x \triangleright \mathsf{value}$ query. We therefore add a conditional which tests the "un-evaluated" target for equality with x. The case that x is told to update to itself through a number of relay handlers must also be taken care of: the last relay handler in the chain cannot get a reply to its $x \triangleright \mathsf{value}$ query. This situation is dealt with in the $x \triangleright \mathsf{value}$ case of the R handlers, where the R handler will respond on behalf of the source of the request, which in this case will be $V(\underline{x})$.

The treatment of $x \triangleright \mathsf{eq}$ requests in a $V(\underline{x})$ handler is straightforward: the value of the other reference is read, and if it is x itself, we generate a positive response. Otherwise the request is reiterated, deferring the response until a decision can be reached, i.e., until either both have updated to names, or both are in the same equivalence tree.

The $R(\underline{x}, \underline{u})$ handler is almost transparent. On receipt of $u \triangleright \mathsf{name}(\underline{a})$, the relay can repond to $x \triangleright \mathsf{name}$ requests. The $x \triangleright \mathsf{update}$ and $x \triangleright \mathsf{eq}$ requests are simply relayed to the handler for u. The treatment of $x \triangleright \mathsf{value}$ was mentioned above: if the source of the request is the handler of u, it may be in a state where it cannot respond to its own query, so the R handler responds by proxy. If the source was some other reference, the request is relayed.

Given the handling of references, the encoding of γ-calculus terms themselves can be presented in Figure 7 on the following page. Null of the γ-calculus ($\top$) corresponds to null of the π-calculus ($\mathbf{0}$), and $\wedge$ (composition) to $\mid$. Declaration $\exists$ corresponds to restriction $\boldsymbol{\nu}$, with the addition of a handler for the reference being declared.

The encoding of a γ-calculus cell ensures that its location is a γ-calculus name, and proceeds according to the π-calculus definition *Cell*. An encoded γ-calculus application must also ensure its location to be a γ-calculus name, and then outputs the vector of references together with their handler interface names. A γ-calculus abstraction is encoded as a replication which first

$$
\begin{array}{rcl}
\llbracket \top \rrbracket & = & \mathbf{0} \\
\llbracket E \wedge F \rrbracket & = & \llbracket E \rrbracket \mid \llbracket F \rrbracket \\
\llbracket \exists a E \rrbracket & = & (\boldsymbol{\nu} \underline{a})(N(\underline{a}) \mid \llbracket E \rrbracket) \\
\llbracket \exists x E \rrbracket & = & (\boldsymbol{\nu} \underline{x})(V(\underline{x}) \mid \llbracket E \rrbracket) \\
\llbracket u \colon \tilde{x}/E \rrbracket & = & !\, u \triangleright \mathsf{name}(\underline{a}) \,.\, a(\underline{\tilde{x}}) \,.\, \llbracket E \rrbracket \\
\llbracket u \colon v \rrbracket & = & u \triangleright \mathsf{name}(\underline{a}) \,.\, \mathit{Cell}(\underline{a}, \underline{v}) \\
\llbracket u \tilde{v} \rrbracket & = & u \triangleright \mathsf{name}(\underline{a}) \,.\, \overline{a} \langle \underline{\tilde{v}} \rangle \\
\llbracket u = v \rrbracket & = & \overline{u \triangleright \mathsf{update}} \langle \underline{v} \rangle \\
\multicolumn{3}{l}{\llbracket \texttt{if } u = v \texttt{ then } E \texttt{ else } F \rrbracket} \\
 & = & (\boldsymbol{\nu} y, n)(\overline{u \triangleright \mathsf{eq}} \langle \underline{v}, y, n \rangle \mid y \,.\, \llbracket E \rrbracket \mid n \,.\, \llbracket F \rrbracket) \\
 & & \\
\mathit{Cell}(\underline{a}, \underline{u}) & \stackrel{\mathit{def}}{=} & a(\underline{v}, \underline{w}) \,.\, \Big(\llbracket u = v \wedge a \colon w \rrbracket \Big)
\end{array}
$$

Figure 7: Encoding of the γ-calculus.

makes sure its location is a γ-calculus name, inputs the references along with their handler interface names, and proceeds as the encoding of the body of the abstraction. The replication is necessary since, in the $\text{A}\textsc{ppl}_\gamma$ rule, the abstraction recurs.

4.3 Variants of the encoding

The use of summation (the + operator) in the definition of type V reference handlers can be avoided by using a standard π-calculus technique: we replace the summation of inputs on "selector" names $a_1, \ldots, a_n$, $a_1(\tilde{x}_1) . P_1 + \cdots + a_n(\tilde{x}_n) . P_n$, with *one* input on a "contact" name b: $b(a_1, \ldots, a_n) . (a_1(\tilde{x}_1) . P_1 \mid \cdots \mid a_n(\tilde{x}_n) . P_n)$, where $\forall i : \{a_j : j \neq i\} \cap \mathrm{fn}(P_i) = \emptyset$, and replace the "selector" outputs $\overline{a_i}\langle \tilde{c}_i \rangle . P_i$ with $\overline{b}\langle \boldsymbol{\nu} a_1, \ldots, \boldsymbol{\nu} a_n \rangle . \overline{a_i}\langle \tilde{c}_i \rangle . P_i$, with the same restriction on $\mathrm{fn}(P_i)$. This also makes handler interface names more local.

Our encoding introduces divergence in several places. As already mentioned the encoding of a γ-calculus equation $a = b$ causes the handlers for the names a and b to ask each other to update indefinitely. This can easily be avoided by not reflecting update requests to other names. The reiteration of the $\overline{x \triangleright \mathsf{eq}}\langle \underline{u}, y, n\rangle$ request in the definition of $V(\underline{x})$ can also result in a divergence. This can be avoided by "freezing" the request by putting it under a prefix (local to $V(\underline{x})$), which is activated if x is later updated, possibly deciding the condition. Finally, indefinitely iterated checks that a reference is a name, i.e., interactions along $u \triangleright \mathsf{name}$, occur in relay handlers and in abstractions. Again divergence can be avoided if these are modified to only perform the check once. This is enough since whenever the check succeeds it will always continue to succeed — a name can never cease to be a name.

Divergence does not affect the formal correctness proofs to follow, since termination is not among the observables in our notion of correctness. In other words, the fact that a γ-term terminates but its encoding diverges is not considered relevant. This can be construed as a weakness from the point of view of the γ-calculus and Oz programming. By adopting the variants above we can eliminate divergence at the expense of a more complicated encoding. Although we have not carried out all details of the proofs, we strongly conjecture that the correctness results of Section 5 are still valid in these variants.

5 Correctness

In this section we show that our encoding is correct, in the sense that if two γ-calculus terms are behaviourally equivalent, we know that the encodings of these terms into the π-calculus are also equivalent, and vice versa. We also show that γ-calculus reductions correspond to π-calculus reductions in our encoding.

5.1 Free references

As in [Smo94], we assume that there are no free γ-calculus variables at top level of a term. However, free γ-calculus names may still appear. To complete the encoding of terms $[\![\cdot]\!]$, we must add an N handler for each free name. Also there may be more free π-calculus names than there are free γ-calculus names. E.g., for the term $E \equiv a{:}\, x/ax$, $\mathrm{fr}(E) = \{a\}$, while $\mathrm{fn}([\![E]\!]) = \{\underline{a}\} = \{a, a \triangleright \mathsf{name}, a \triangleright \mathsf{value}, a \triangleright \mathsf{update}, a \triangleright \mathsf{eq}\}$. To correct this and make sure the handler interface names cannot be used improperly by some external agent, we add a restriction on them.

We define a closing operation $\mathcal{V}$ to add the name handlers and the restriction:

Definition 1 *If* $\mathrm{fr}(E) = \{a_1, \ldots, a_n\}, n \geq 0$,

$$\mathcal{V}(E) \equiv (\boldsymbol{\nu}\tilde{a})(N(\underline{a_1}) \mid \cdots \mid N(\underline{a_n}) \mid [\![E]\!])$$

where $\tilde{a} \equiv \bigcup_{i=1}^{n} \{a_i \triangleright \mathsf{name}, a_i \triangleright \mathsf{value}, a_i \triangleright \mathsf{update}, a_i \triangleright \mathsf{eq}\}$

An example:

$$\begin{array}{rcl} E & \equiv & a\colon x/ax \\ [\![E]\!] & \equiv & !\, a\triangleright\mathsf{name}(\underline{a})\,.\,a(\underline{x})\,.\,a\triangleright\mathsf{name}(\underline{a})\,.\,\overline{a}\langle\underline{x}\rangle \\ \mathcal{V}(E) & \equiv & (\boldsymbol{\nu} a\triangleright\mathsf{name}, a\triangleright\mathsf{value}, a\triangleright\mathsf{update}, a\triangleright\mathsf{eq})\big(N(\underline{a}) \mid [\![E]\!]\big) \end{array}$$

5.2 Behavioural equivalences

We must define what we mean by two terms or agents being "equivalent". Using the results of Milner and Sangiorgi [MS92a, San92], we propose a version of *barbed bisimulation*, based on the reduction relations of Sections 2 and 3, adding predicates for observability.

Definition 2 (Observability in γ-calculus)

$a\tilde{v} \downarrow_\gamma^a$, $a\colon \tilde{x}/E \downarrow_\gamma^a$, $a\colon u \downarrow_\gamma^a$
$(\exists u E) \downarrow_\gamma^a$ *if* $E \downarrow_\gamma^a$ *and* $a \neq u$
$(E \wedge F) \downarrow_\gamma^a$ *if* $E \downarrow_\gamma^a$ *or* $F \downarrow_\gamma^a$

Definition 3 (Observability in π-calculus)

$\overline{a}\tilde{b} \downarrow_\pi^a$, $a(\tilde{b}).P \downarrow_\pi^a$
$(\boldsymbol{\nu} b)P \downarrow_\pi^a$ *if* $P \downarrow_\pi^a$ *and* $a \neq b$
$(P \mid Q) \downarrow_\pi^a$ *if* $P \downarrow_\pi^a$ *or* $Q \downarrow_\pi^a$
$(P + Q) \downarrow_\pi^a$ *if* $P \downarrow_\pi^a$ *or* $Q \downarrow_\pi^a$

The relation $\Rightarrow_\gamma$ is the reflexive and transitive closure of $\rightarrow_\gamma$, and similarly for $\Rightarrow_\pi$ and $\rightarrow_\pi$. We write $E \Downarrow_\gamma^a$ if $E \Rightarrow_\gamma \downarrow_\gamma^a$, and $P \Downarrow_\pi^a$ if $P \Rightarrow_\pi \downarrow_\pi^a$.

We can now define observational equivalences for γ-calculus terms and for π-calculus agents. We start by giving a parameterized definition of weak barbed bisimulation [MS92a, San92]. Let $\Box$ be γ or π.

Definition 4 *A symmetric relation* $\mathcal{R}$ *on* $\Box$ *expressions is a* weak $\Box$-barbed bisimulation *if for all* $(X, Y) \in \mathcal{R}$, *1. if $X \rightarrow_{\Box} X'$, then $Y \Rightarrow_{\Box} Y'$ such that $X'\mathcal{R}Y'$*

2. if $X \downarrow^{a}_{\Box}$, then $Y \Downarrow^{a}_{\Box}$.

Two expressions X and Y are weak $\Box$-barbed bisimilar*, written $X \dot{\approx}_{\Box} Y$, if $X\mathcal{R}Y$ for some weak $\Box$-barbed bisimulation $\mathcal{R}$.*

These definitions of bisimilarity are too weak; e.g, the agents $a\,.\,b\,.\,\mathbf{0}$ and $a\,.\,\mathbf{0}$ are considered weak π-barbed bisimilar. We remedy this by closing under contexts:

Definition 5 *Two terms E and F are* weak γ-barbed congruent*, written $E \approx_{\gamma} F$, if for all γ-calculus contexts $C[\cdot]$, it holds that $C[E] \dot{\approx}_{\gamma} C[F]$.*

In the π-calculus case, we choose to close not under all π-calculus contexts, but only under γ contexts, i.e., contexts occurring in encodings of γ-calculus terms. Formally, these can be defined by introducing term variables; the γ contexts are then encodings of γ terms with one occurrence of a term variable.

Definition 6 *Two agents P and Q are* weak π-barbed congruent*, written $P \approx_{\pi} Q$, if for all γ contexts $C[\cdot]$, it holds that $C[P] \dot{\approx}_{\pi} C[Q]$.*

If arbitrary π-calculus contexts were used in the previous definition, the relation would coincide with weak early bisimulation [San92]. Our congruence is coarser since only contexts corresponding to the γ-calculus are considered. For example, the terms $a\colon x/(x = x)$ and $a\colon x/\top$ are equivalent in the sense that their encodings have the same reductions and observations in arbitrary γ-calculus contexts. But there is a π-calculus context which distinguishes them, for example $(\overline{a}\langle \underline{u} \rangle \mid \bullet)$. Inserting the encoding of $a\colon x/(x = x)$ we will get observations along the interface names in $\underline{u}$, in contrast to when inserting $a\colon x/\top$. A similar situation is present in the encoding of the λ-calculus into the π-calculus [Mil92], and indeed it would be unreasonable to expect any encoding to be robust under alien contexts.

We now have definitions of the congruences we will use in the full abstraction theorem below. However, to make the proofs easier, we use the *up-to* $\approx$ technique of [MS92b, San92]. Again we give a parameterized definition. Let $\Box$ be γ or π.

Definition 7 *A symmetric relation $\mathcal{R}$ on $\Box$ expressions is a* weak $\Box$-barbed bisimulation up-to $\approx_\Box$ *if for all $(X,Y) \in \mathcal{R}$,*

1. *if $X \Rightarrow_\Box X'$, then $Y \Rightarrow_\Box Y'$ such that $X' \approx_\Box \mathcal{R} \approx_\Box Y'$*
2. *if $X \Downarrow^a_\Box$, then $Y \Downarrow^a_\Box$.*

Again, we close these bisimilarities under contexts.

Definition 8 *Two terms E and F are* weak γ-barbed congruent up-to $\approx_\gamma$, *written $E \approx^+_\gamma F$, if for all γ-calculus contexts $C[\cdot]$, it holds that $(C[E], C[F]) \in \mathcal{R}$ for some $\mathcal{R}$ which is a weak γ-barbed bisimulation up-to $\approx_\gamma$.*

Definition 9 *Two agents P and Q are* weak π-barbed congruent up-to $\approx_\pi$, *written $P \approx^+_\pi Q$, if for all γ contexts $C[\cdot]$, it holds that $(C[P], C[Q]) \in \mathcal{R}$ for some $\mathcal{R}$ which is a weak π-barbed bisimulation up-to $\approx_\pi$.*

Proposition 1

1. *If $E \approx^+_\gamma F$, then $E \approx_\gamma F$.*
2. *If $P \approx^+_\pi Q$, then $P \approx_\pi Q$.*

Proof: Analogous to Lemma 4.5 in [Mil89]. $\Box$

5.3 Full abstraction

We now have enough machinery to proceed to show full abstraction of the encoding, i.e., that two γ-calculus terms are congruent iff their encodings in the π-calculus are congruent. We omit the proofs because of the limited space available; they are often quite detailed.

Theorem 2 *$E \approx_\gamma F$ iff $\mathcal{V}(E) \approx_\pi \mathcal{V}(F)$.*

In the rest of this section we sketch the proof.

We use straight-forward Lemmas concerning operational correspondence between γ-calculus terms E and their encodings $\mathcal{V}(E)$. The first Lemma says that if a γ-calculus term E can reduce to E', its encoding $\mathcal{V}(E)$ can reduce to an agent congruent to the encoding of E'.

Lemma 3 *$E \rightarrow_\gamma E'$ implies $\mathcal{V}(E) \Rightarrow_\pi \approx_\pi \mathcal{V}(E')$.*
PROOF: By induction over reduction rules. $\Box$

An immediate corollary is the following:

Corollary 4 *$E \Rightarrow_\gamma E'$ implies $\mathcal{V}(E) \Rightarrow_\pi \approx_\pi \mathcal{V}(E')$.*

The next Lemma goes in the other direction: it states that however $\mathcal{V}(E)$ reduces in the π-calculus, it reduces to an agent which is congruent to the encoding of a γ-calculus term to which E can reduce in the γ-calculus.

Lemma 5 *$\mathcal{V}(E) \Rightarrow_\pi P'$ implies $\exists E' : E \Rightarrow_\gamma E'$ and $P' \approx_\pi \mathcal{V}(E')$.*
PROOF: By induction on the structure of terms E. $\Box$

We move on to the observations: if we can observe a on E, we can also observe a on the encoding of E.

Lemma 6 *$E \downarrow^a_\gamma$ implies $\mathcal{V}(E) \Downarrow^a_\pi$.*
PROOF: By induction on the structure of terms E. $\Box$

A corollary of this (and Corollary 4) is the following:

Corollary 7 *$E \Downarrow^a_\gamma$ implies $\mathcal{V}(E) \Downarrow^a_\pi$.*

The other direction is also true: if we can observe a on the encoding of a term E, we can observe it on E itself.

Lemma 8 *$\mathcal{V}(E) \Downarrow^a_\pi$ implies $E \Downarrow^a_\gamma$.*
PROOF: By induction on the structure of terms E. $\Box$

Proposition 9 *$E \approx^+_\gamma F$ iff $\mathcal{V}(E) \approx^+_\pi \mathcal{V}(F)$.*
PROOF: Using Corollaries 4 and 7 and Lemmas 5 and 8, induction, and the fact that the name handlers and restrictions introduced in $\mathcal{V}$ distribute over γ-calculus contexts. $\Box$

Now, Theorem 2 is a consequence of Propositions 9 and 1.

6 Conclusion

We have demonstrated a compositional encoding of the γ-calculus into the π-calculus and established its correctness by means of barbed bisimulation congruence. This opens several avenues of further work.

The constraints in the γ-calculus are just equalities between variables and names. In this way γ is equipotent with a special case of the ρ-calculus[NM95]. A natural direction for further work is to investigate other versions of the ρ-calculi and the Oz-calculus. These calculi use constraints over arbitrary data types and although data type encodings in the π-calculus are well understood it is less clear how to deal with constraints over them.

Further the γ-calculus lacks disjunction and failure. An encoding of disjunction could be envisaged with the π-calculus summation operator (which is akin to committed choice in that non selected branches are irrevocably lost) plus a backtracking mechanism. An encoding of failures is straightforward by giving unsatisfiable equations such as $a = b$ a computational content beyond that in our present scheme (where equations merely trigger updates of a global state).

Calculi with concurrent constraints typically come with "computational models" which express how terms represent computational behaviour. In a sense our encoding is such a model, since it dictates how the global state is represented and how it can be updated. The model for the γ-calculus in [Smo94] uses a central store (a "blackboard") for noncritical resources such as abstractions, but effectuates state updates through global substitutions of variables. Our encoding is therefore closer to an implementation, and it might be interesting to compare it with various implementation strategies.

Finally, the equivalence introduced in this paper has not been investigated. As explained in Section 5, our π-calculus congruence is coarser than weak early bisimulation congruence [San92]. Nevertheless the barbed congruence in the γ-calculus is a potential area of further study and presumably many techniques will carry over from the π-calculus.

References

[AD96] R. Amadio and M. Dam. Toward a modal theory of types for the π-calculus. 1996. To appear in the proceedings of FTRTFT'96, Lecture Notes in Computer Science.

[Dam93] M. Dam. Model checking mobile processes. In E. Best, editor, *CONCUR'93, 4^{th} Intl. Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 22–36. Springer-Verlag, 1993. Full version in Research Report R94:01, Swedish Institute of Computer Science, Kista, Sweden.

[FMQ95] G. Ferrari, G. Modoni and P. Quaglia. Towards a semantic-based verification environment for the pi-calculus. In *Proceedings of 5th Italian Conference on Theoretical Computer Science: ICTCS-95*. World Scientific, 1995.

[Li94] B. Li. A π-calculus specification of Prolog. In D. Sannella, editor, *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 379–393, Edinburgh, 1994. Springer-Verlag.

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

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, Oct. 1991. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.

[Mil92] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MPW92] R. Milner, J. Parrow and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, Sept. 1992.

[MS92a] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proc. of 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[MS92b] R. Milner and D. Sangiorgi. The problem of "weak bisimulation up-to". In R. Cleaveland, editor, *CONCUR'92, 3^{rd} Intl. Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992. Revised version available as ftp://ftp.dcs.ed.ac.uk/pub/sad/x.ps.Z, entitled *Techniques of "weak bisimulation up-to"*.

[NM95] J. Niehren and M. Müller. Constraints for free in concurrent computation. In K. Kanchanasut and J.-J. Lévy, editors, *Asian Computing Science Conference on Algorithms, Concurrency and Knowledge (ACSC '95)*, volume 1023 of *Lecture Notes in Computer Science*, Pathumthani, Thailand, 11–13 Dec. 1995. Springer-Verlag.

[PRT93] B. C. Pierce, D. Rémy and D. N. Turner. A typed higher-order programming language based on the pi-calculus. In *Workshop on Type Theory and its Application to Computer Systems, Kyoto University*, July 1993.

[PS95] J. Parrow and D. Sangiorgi. Algebraic theories for name-passing calculi. *Journal of Information and Computation*, 120(2):174–197, 1995.

[PS96] M. Pistore and D. Sangiorgi. A partition refinement algorithm for the π-calculus. 1996. To appear in the proceedings of CAV'96, Lecture Notes in Computer Science.

[Ros93] B. Ross. A π-calculus semantics of logical variables and unification. In S. Purushothaman and A. Zwarico, editors, *Proceedings of First North American Process Algebra Workshop, Stony Brook, 1992*, *Workshops in Computing*, pages 216–230. Springer-Verlag, 1993.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, UK, 1992.

[San95] D. Sangiorgi. On the proof method for bisimulation (extended abstract). In J. Wiederman and P. Hájek, editors, *Mathematical Foundations of Computer Science 1995 (MFCS'95)*, volume 969 of *Lecture Notes in Computer Science*, pages 479–488. Springer-Verlag, 1995. Full version available electronically as `ftp://ftp.dcs.ed.ac.uk/pub/sad/bis-proof.ps.z`.

[San96] D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. Earlier version published as Report ECS-LFCS-93-270, University of Edinburgh. An extended abstract appeared in LNCS 715 (Proc. CONCUR'93).

[Smo94] G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *Proc. 1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72. Springer-Verlag, Sept. 1994. Available as Research Report RR-94-16 from DFKI Kaiserslautern.

[Smo95a] G. Smolka. The definition of Kernel Oz. In A. Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. Springer-Verlag, 1995.

[Smo95b] G. Smolka. An Oz primer. DFKI Oz documentation series, DFKI, Saarbrücken, Germany, 1995. Available as ftp://ps-ftp.dfki.uni-sb.de/pub/oz/documentation/Primer.ps.Z.

[Tho90] B. Thomsen. *Calculi for Higher Order Communicating Systems.* PhD thesis, Imperial College, University of London, Sept. 1990.

[TLP+93] B. Thomsen, L. Leth, S. Prasad, T.-S. Kuo, A. Kramer, F. Knabe and A. Giacalone. Facile Antigua release – programming guide. Technical Report ECRC-93-20, ECRC, München, Germany, 1993.

[VM94] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In D. Dill, editor, *Computer Aided Verification (Proc. of CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[Wal95] D. Walker. Objects in the π-calculus. *Journal of Information and Computation*, 116(2):253–271, 1995.