ISSN 1361 - 6161

THE UNIVERSITY of MANCHESTER

Computer Science
University of Manchester

A Fibration Semantics for Pi-Calculus Modules via Abstract MONSTR Rule Systems

R. Banach

Department of Computer Science
University of Manchester
Technical Report Series
UMCS-96-7-3

A Fibration Semantics for Pi-Calculus Modules via Abstract MONSTR Rule Systems*

R. Banach

Department of Computer Science
University of Manchester
Oxford Road, Manchester, U.K.
banach@cs.man.ac.uk

31 July 1996

* Refereed by: Franck van Breugel.

A Fibration Semantics for Pi-Calculus Modules via Abstract MONSTR Rule Systems

R. Banach
Computer Science Dept., Manchester University, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk

Abstract: A translation of the π-calculus into MONSTR rulesets is described. The syntax-directed aspects of the translation enable a categorical semantics of modules to be constructed. The relationship between name-sensitive and name-insensitive categories is explored, and by moving to abstract rulesets to avoid difficulties with soundness arising from potential symbol clashes, a categorical semantics of modules which is both a fibration and an opfibration ensues. The fibration/opfibration relationship between syntax and rulesets is extended to a semantics in terms of systems (to which it is related by further fibration/opfibrations), and then finally to a semantics in terms of traces, (the latter relationship being only an opfibration over the former).

1 INTRODUCTION

MONSTR is a generalised term graph rewriting language that was used as the intermediate language for the Flagship machine. See [Banach et al. (1988), Banach and Watson (1989), Banach (1993), Banach (1996), Banach (1995), Watson and Watson (1987), Watson et al. (1987), Watson et al. (1989)]. Since the expressiveness of the language was closely matched to the capabilities of a distributed store parallel machine, it is not surprising that MONSTR has proved itself well suited to describing features of many programming paradigms of current interest, intended for highly parallel implementation. For instance: concurrent logic programming [Banach and Papadopoulos (1993)], linear concurrent programming [Banach and Papadopoulos (1995a)], concurrent constraint programming [Banach and Papadopoulos (1995b)], and concurrent object oriented programming [Banach and Papadopoulos (1995c)]. In all cases MONSTR's primitive notions proved ideally suited to the situation at hand.

The π-calculus [Milner et al. (1992), Milner (1993a)] generalises CCS [Milner (1989)], allowing process networks to evolve dynamically. Such process algebra languages are another candidate for highly parallel situations, embodying the basic ideas of the chemical abstract machine (CHAM) paradigm [Berry and Boudol (1990)]. In [Banach et al. (1995)], there is a detailed and rigorous study of a translation of the π-calculus into MONSTR. This makes clear the very different nature of the fundamental operations in process algebra languages and in more conventionally based languages.

[Banach et al. (1995)] concentrates on matching up π-calculus terms and concrete MONSTR counterparts. In this paper we consider a more abstract aspect of the translation. We show how the semantics of π-calculus modules may be understood categorically via the translation. Three things help here. First, that each π-calculus constructor corresponds to its own piece of translation, and these just compose together. Second, that communication is handled by independent rules for sender and receiver. Thus when arbitrary components are composed, they can communicate right away. Third, that scope extrusion is trivial in the translation. When a scope is extruded in a π-calculus expression, there is no change in the representing MONSTR execution graph. Thus when composition yields an expression with inadequate scope for some restriction, nothing need be done. These things are clear from results in [Banach et al. (1995)].

The structure of the rest of this paper is as follows. Section 2 gives a brief description of MONSTR, while section 3 sets out the version of π-calculus that we will use. Section 4 describes the key ideas of the translation, which is reproduced tersely in the Appendix. Section 5 considers modules, giving a name-sensitive and name-insensitive categorical semantics. Abstract rulesets are introduced which give a fibration and opfibration semantics to π-calculus modules. This fibration and opfibration is then lifted via another fibration and opfibration to give a semantics of ground systems and modules, and finally this latter is related via an opfibration to a semantics in terms of traces, at both the π-calculus and MONSTR levels. Section 6 concludes.

2 MONSTR

MONSTR arose as a result of the attempt to reconcile the desire for an intermediate language with rewriting-based semantics, with the reality of a parallel machine where the primitive atomic actions were in principle of much smaller granularity than atomic rewrites of arbitrary size. The result was a term graph rewriting language MONSTR, for which the implementation problem did not make excessive demands on the architecture's semantics. We give here a necessarily brief description of MONSTR rewriting. See [Banach et al. (1995)], or other references, for a mathematically more precise account.

The operational semantics of MONSTR deals with the transformation of term graphs. These are graphs in which the nodes are labelled with fixed arity symbols; each node having a number of out-arcs determined by its symbol, while itself being the target of an arbitrary number of in-arcs. The nodes and arcs of MONSTR graphs are further decorated with certain markings which relate to reduction strategy. If a node is marked with $*$, then it is active and can serve as the root of a redex. If it is marked with $\#^n$, then it is suspended waiting for n notifications (see below), and then and only then n of its out-arcs are marked with the notification mark ^. (This correspondence between $\#^n$ and n ^'s is called balancedness, a vital invariant of MONSTR graphs.) The only other possibilities are that nodes and arcs are unmarked (i.e. idle).

Computation proceeds by arbitrarily selecting an active node t in the execution graph and attempting to find a rule that matches at t. The easiest case to describe is when there is no such rule, whereupon notification takes place: The active marking is removed from t and a "notification" is sent up along each ^-marked in-arc of t. When this notification arrives at its (necessarily) $\#^n$-marked source node p, the ^ mark is removed from the arc, and the n in p's $\#^n$ marking is decremented, preserving balancedness. ($\#^0$ is replaced by $*$, so suspended nodes eventually wake when all their subcomputations have notified.) For our purposes, we can assume that all the above takes place within a single atomic action.

The other case, when a rule matches at t, needs a little more background for its description. The symbols that label nodes are statically divided into three classes: functions, constructors, and stateholders. Functions label roots of LHSs of rules but are not allowed to occur at subroot positions; they may be redirected (see below). Constructors and stateholders can only occur at subroot positions of LHSs of rules; and while stateholders may be redirected, constructors may not. Compared to the use of term graph rewriting as an implementation vehicle for functional languages, where only functions and constructors are needed, the presence of stateholders within MONSTR considerably increases the flexibility of the language for conveniently modelling imperative notions such as storage cells, synchronisation objects, and the logical variable. They will play a vital role in our translation of the π-calculus below by representing channels and encoding protocol states.

Here is a MONSTR rule in concrete syntax:

```
F[Cons[a b] x:Var] => #G[a ^*b] , x := *SUCCEED
```

The LHS and RHS are separated by **=>**. On the left is the pattern with root labelled by function **F**, whose arguments are a constructor labelled **Cons** with two unspecified arguments, and a stateholder **x** labelled by **Var**. The notation makes it plain that **F**-labelled nodes have two out-arcs, and that if this rule is to match at t, then the subgraph immediately descending from t must be of the specified form.

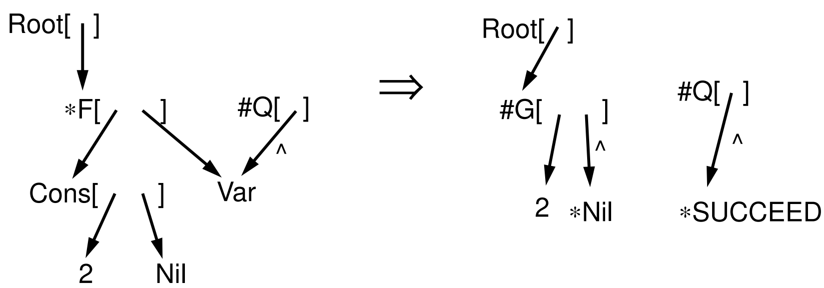
Suppose the rule indeed matches at active t (note that as a matter of convention, we will distinguish syntactic objects, such as the rule above and its constituents, from semantic objects such as the execution graph and its constituents (eg. the root t), by writing the former in Courier font and the latter in Helvetica font). The RHS of the rule specifies new pieces to be glued into the execution graph, namely firstly, a node labelled by G, created suspended and with the same arguments as the LHS Cons node, the second of which is activated (i.e. has its marking changed to $*$ if previously idle), and such that G is waiting for this second argument to notify; and secondly, a node labelled by SUCCEED, created active. This part of the rewrite is called the contractum building phase. The **=>** further indicates that the LHS root F is to be redirected to G, and **x := *SUCCEED** indicates that the node matched to **x** is to be redirected to the new SUCCEED node. Redirection itself consists of swinging all in-arcs of the LHS node of the redirection (eg. F) so that they point to the RHS node of the redirection (eg. G). All redirections are performed simultaneously, this being the redirection phase of the rewrite. Also, t is deactivated. All the individual subactions mentioned in this paragraph again take place within a single atomic action of the rewriting model.

By convention, MONSTR systems are assumed to start from a fixed execution graph containing a single active node labelled by Initial. Obviously there must be a rule for **Initial** if the system is to make progress.

According to the above, no node in a MONSTR execution graph is ever destroyed. The rewriting model does not prescribe what nodes are ever to be removed from an execution graph because the rather arbitrary way in which arcs of the graph may cause it to be connected up, means that we cannot statically determine which nodes are useless. Destruction of nodes is therefore done by garbage collection. Apart from specially designated system roots, active nodes are live, as are nodes accessible from a live node along an idle arc, as are also nodes which can access a live node along a notification arc. All others are garbage. This definition turns out to be sound.

Below is a small example of a rewrite according to our rule above. It is clear that there is a redex for the rule in the graph on the left — the active F-labelled node is the root of the redex, and the nodes labelled 2 and Nil match to **a** and **b** respectively. At the start of the rewrite, the F-labelled node is made idle. During contractum building, the new nodes #G[…] and $*$SUCCEED are introduced into the graph, the former with children

2 and Nil, and the later childless. Neither of these new nodes has a parent at this stage. During the redirection phase, the arc from Root is swung over to point to G instead of to F effecting the redirection **`F => G`**, and the arc from Q is swung over to point to SUCCEED instead of to Var, effecting the redirection **`x := *SUCCEED`**. Also the Nil node must be activated. At this point the F, Cons, and Var nodes are still present in the execution graph but are garbage. When the garbage is collected, these nodes and their out-arcs disappear, leaving the graph on the right. Subsequently, the two active nodes, a constructor and a stateholder respectively, will be able to notify.



In order that executions have various desirable properties, MONSTR imposes a further collection of restrictions. All functions must have a default rule (i.e. one that does no pattern matching at all in the LHS), so that every active function node can rewrite. (Correspondingly, all active constructors and stateholders only ever notify.) Rule selection is controlled using the nondeterministic rule separator **|** and the sequential rule separator **`;`**. Pattern matching by functions is only allowed to be one level deep, and each function has a statically fixed set of positions at which non-default rules must pattern match; at most (a statically fixed) one of these is permitted to match a stateholder. A further restriction is imposed that any notification arc must point to a node which is either non-idle or a stateholder (arcs are said to be state saturated). Finally, if an argument needed for matching by a rewrite is found to be non-idle, the rewrite suspends until the subcomputation notifies in the normal way, although this is a feature of MONSTR semantics that we will not need below.

The above constitutes a brief sketch of MONSTR and its abstract execution model. For a definitive account see [Banach (1996)], [Banach (1995)]. Lack of space prevents us from saying more here.

3 THE PI-CALCULUS

The π-calculus first appeared in [Milner et al. (1992)] and since that time has been seen in a number of minor variants. We will use a version of the monadic calculus, [Milner (1993a)], since it proves to be convenient. The syntax is given by

$$P \;=\; \pi_1.Q_1 + \pi_2.Q_2 + \ldots + \pi_n.Q_n \;\mid\; Q_1 \,|\, Q_2 \,|\, \ldots \,|\, Q_n \;\mid\; \nu x Q \;\mid\; !Q \;\mid\; \mathbf{0}$$

Speaking briefly, the first case is guarded summation where each π_i is of the form $x(y)$ or $\overline{x}z$. The input $x(y).Q$, means that some channel name q say, is to be read over the channel x, and then q is bound to all free occurences of y in Q. The output $\overline{x}z.Q$ means that the channel name z is to be written to the channel x. A process which is a sum can evolve into exactly one of the alternatives. The second case is parallel composition.

The third case is restriction; in $\nu x Q$, the channel name x is bound, and refers to a channel that is private to $\nu x Q$. The fourth case is replication, in which $!Q$ is equivalent to the parallel composition of arbitrarily many copies of Q. Finally $\mathbf{0}$ does nothing.

The dynamics of the π-calculus is given by the two rewrite rules

$$!P \rightarrow_{\mathrm{R}} P \mid !P \quad \text{and} \quad (\ldots + x(y).P + \ldots) \mid (\ldots + \bar{x}z.Q + \ldots) \rightarrow_{\mathrm{C}} P\{z/y\} \mid Q$$

The first is an active version of replication (which is usually handled by demanding that $\rightarrow_{\mathrm{R}}$ is a congruence), and the second is internal communication, with the input process substituting all instances of the input channel by the received name.

With $\rightarrow_Y$ standing for either of $\rightarrow_{\mathrm{R}}$ and $\rightarrow_{\mathrm{C}}$, the CHAM aspect of the π-calculus is expressed by allowing dynamic behaviour only at top level, as sanctioned by

$$\frac{P \rightarrow_Y Q}{P \mid R \rightarrow_Y Q \mid R} \quad \text{and} \quad \frac{P \rightarrow_Y Q}{\nu x P \rightarrow_Y \nu x Q}$$

Communications that do not take place either entirely inside, or entirely outside the scope of a νx binder do not make sense as they export a bound name outside the scope of its binder. Therefore scope extrusion is introduced by the rule

$$(\nu z Q_1 \mid Q_2 \mid \ldots \mid Q_n) \Rrightarrow \nu z'(Q_1\{z'/z\} \mid Q_2 \mid \ldots \mid Q_n)$$

where the prime indicates an alpha conversion of the bound channel to avoid capture of free names. This operation is required for the dynamics of the π-calculus, and is regarded as a congruence in [Milner (1993a)], but there is no counterpart of it in the translation of [Banach et al. (1995)], since channel identity is handled directly by graph connectivity, rather than by awkward syntactic means. When the rule has been applied enough times to ensure all potential communications do not cause escape of scopes, a π-calculus expression is in *standard form*. This will prove useful below.

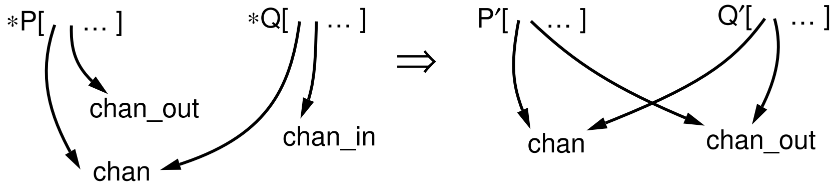
The definition in [Milner (1993a)] is supplemented by demanding that summation and parallel composition are monoidal operators with $\mathbf{0}$ as unit. For this paper, it is variadic *prefixed* summation that is primitive, which is thus not associative. We will also not demand that $\mathbf{0}$ is a unit for simplicity. Whether parallel composition is commutative and associative, and whether prefixed summation is commutative, can be regarded as matters of choice. Either they are, in which case little more needs to be said, or they are not, in which case the rule schemas above need to be supplemented with a suite of rather obvious structural variants. Either viewpoint can be handled within the translation.

Below is a small example of a π-calculus system and one possible evolution. Readers may check that the same system may also evolve to $v(t).\mathbf{0} \mid \mathbf{0} \mid x(y).\mathbf{0}$ or to $x(u).u(t).\mathbf{0} \mid \bar{v}s.\mathbf{0} \mid \mathbf{0}$ or to $x(u).u(t).\mathbf{0} \mid \mathbf{0} \mid \mathbf{0}$.

$$x(u).u(t).\mathbf{0} \mid (\bar{x}v.\bar{v}s.\mathbf{0} + \bar{x}v.\mathbf{0}) \mid x(y).\mathbf{0} \rightarrow_{\mathrm{C}} v(t).\mathbf{0} \mid \bar{v}s.\mathbf{0} \mid x(y).\mathbf{0} \rightarrow_{\mathrm{C}} \mathbf{0} \mid \mathbf{0} \mid x(y).\mathbf{0}$$

4 A REVIEW OF THE TRANSLATION

The idea of the translation is that π-calculus processes which are potentially able to communicate by virtue of not being ancestrally guarded, are represented by active function nodes in a MONSTR execution graph. Channels are represented by stateholder nodes, and all processes with an interest in a given channel share the stateholder representing that channel. Suppose then a pair of active processes P and Q share a channel chan, (we assume that the symbols P and Q encode the potential behaviours of the two processes), and P wishes to send a channel chan_out along chan, and Q wishes in turn to receive a channel along chan to bind to its channel chan_in. We represent this action as the term graph transformation below.



This achieves the substitution of chan_in by chan_out via the redirection chan_in := chan_out. P′ and Q′ represent the subsequent potential behaviours of P and Q.

We cannot turn this directly into a MONSTR rewrite rule, since the left hand side above is birooted. MONSTR forbids this as it increases the complexity of pattern matching, but there is no obstacle posed by graph rewriting semantics. (In fact [Montanari and Pistore (1995)] give a translation of the π-calculus into a less pragmatically constrained graph rewriting formalism, where internal communication is done by a rule much like the above transformation. Also a multirooted extension of MONSTR has been exploited in [Banach and Papadopoulos (1995b)].)

To overcome the multiroot problem, the translation utilises a protocol to enable P and Q to independently initiate a putative communication, in the hope that a suitable partner process offers to cooperate. Offers by processes to communicate must be rescindable otherwise deadlock could occur, and all the actions that constitute the runs of the protocol for the representation of a π-calculus system must be equivalent to some serial shedule of atomic communication events in the original system.

Here is an outline of our preferred protocol. The states of a channel are represented by the three stateholder symbols Empty, Busy_Unlocked, and Busy_Locked. Empty means no reader has recorded an offer on this channel. Busy_Unlocked means that a reader has registered an offer on this channel. At this stage, either the offer may be rescinded by the original reader and the state of the channel reverts to Empty, or a writer completes a rendezvous with the reader, installing its data, and changing the state to Busy_Locked. In the latter case it becomes the reader's responsibility to extract the data and bind it to the input channel, and to reset the channel state to Empty. For simplicity, we allow the protocol to busy wait.

The behaviour of a process $P = (\ldots + chan(chan\_in).Q + \ldots)$ is represented by a collection of rules, one for each summand. The rule for the summand displayed, records a decision, made nondeterministically, for P to attempt to communicate via *chan*, and when successful to evolve to Q. The node P rewrites to P_Q and spawns a helper process Help_r[chan chan_in] to manage the protocol. If the attempt is unsuccessful, P_Q backtracks to P once more. For input, the input channel is instantiated just when the decision to attempt to input is taken.

The creation of a collection of processes executing in parallel $P = Q_1 \mid Q_2 \mid \ldots \mid Q_n$ is represented by a rule which spawns the requisite MONSTR function nodes $\mathsf{Q}_1 \ldots \mathsf{Q}_n$.

Similarly for replication. The **0** process is represented by a rule which rewrites to a non-garbagable Root node, so that even the trivial process expression is represented by something. Finally restriction $P = \nu x Q$ just instantiates the bound variable.

The translation therefore consists of three parts. The first two are the constant communication protocol rules, and the translation of the process expression structure. The latter is as follows. The function symbols required are generated by decorating the parse tree of the expression, allowing syntactically identical subexpressions to be similarly labelled, but otherwise avoiding clashes, eg. $\{\nu x Q_Q\}_P$ where the subscripts are the labels. For such a function symbol P, we have a mapping $Args_P : \mathrm{Free}(P) \to \boldsymbol{\mathtt{args}}_P$ which records a choice of bijection between the free names of the process subexpression, and argument positions of the function symbol. The translation of a construct then consists of rule(s) for **`P`**, (the concrete syntax counterpart of P), such that **`P`** rewrites to **`Q`**; and a relationship between the $Args_P$ and $Args_Q$ maps, reflecting the structure of the rule, that ensures channel nodes are passed from **`P`** to **`Q`** in a proper manner, (see the Appendix). The last part of the translation is the rule for **`Initial`** which instantiates the top level symbol and top level free channel nodes.

Let $\to_{\mathrm{C}(E)}$, $\to_{\mathrm{R}(E)}$, $\to_{\mathrm{A}(E)}$ be the rewrite relations for the communication protocol, replication rewrites, and the other structural rules respectively, for a π-calculus expression E. In [Banach et al. (1995)], the following correctness theorems are proved.

Theorem 4.1 Let $E \equiv E_0$ be a π-calculus expression, and $\mathrm{Tr}(E)$ its translation. Then for any trace of communications and replications from E enhanced by reduction to a standard form at each step, eg.

$$E \equiv E_0 \Rrightarrow^* F_0 \to_{\mathrm{C}} E_1 \Rrightarrow^* F_1 \to_{\mathrm{R}} E_2 \Rrightarrow^* F_2 \to \dots$$

there is a rewriting sequence as below such that for each i, G_i represents both E_i and F_i.

$$\begin{tikzcd}
G_0 \arrow[r, "{\mathrm{C}(E)^+}"'] & H_0 \arrow[r, "{\mathrm{A}(E)^*}"'] & G_1 \arrow[r, "{\mathrm{R}(E)}"'] & H_1 \arrow[r, "{\mathrm{A}(E)^*}"'] & G_2 \arrow[r] & \dots \\
\ast\text{Initial} \arrow[u, "{\mathrm{A}(E_0)}"', "*"] & & \ast\text{Initial} \arrow[u, "{\mathrm{A}(E_1)}"', "*"] & & \ast\text{Initial} \arrow[u, "{\mathrm{A}(E_2)}"', "*"] & \\
\ast\text{Initial} \arrow[uu, bend left, "{\mathrm{A}(F_0)}", "*"'] & & \ast\text{Initial} \arrow[uu, bend left, "{\mathrm{A}(F_1)}", "*"'] & & \ast\text{Initial} \arrow[uu, bend left, "{\mathrm{A}(F_2)}", "*"'] &
\end{tikzcd}$$

(N.B. A MONSTR execution graph *represents* a π-calculus expression, iff there are bijections between top level parallel subexpressions (TLPSEs) and active function nodes, and between free channel names and channel nodes, such that TLPSEs' free channel names map to arcs between function and channel nodes in the expected way.)

Theorem 4.2 Let E be a π-calculus expression, and $\mathrm{Tr}(E)$ its translation. Let $\mathcal{H} = [H_0, H_1, \dots]$ be a weakly fair execution of $\mathrm{Tr}(E)$. Then there is a trace $\mathcal{T}$ of communications and replications from E, enhanced by reductions to standard form, such that for a standard execution $\mathcal{G} = [G_0, G_1, \dots]$ of $\mathrm{Tr}(E)$ corresponding via theorem 4.1 to $\mathcal{T}$, for some prefix $[G_0, G_1, \dots, G_N]$ of $\mathcal{G}$, with $0 \le N \le \infty$, if $i \le N$

(1) For every node ${}^{(i)}x \in G_i \in \mathcal{G}$, ${}^{(i)}x \; \Xi \; {}^{(j)}y$ for some ${}^{(j)}y \in H_j \in \mathcal{H}$, and if ${}^{(i)}x$ is live then ${}^{(j)}y$ is live.

(2) For every arc $({}^{(i)}p_k, {}^{(i)}c) \in G_i \in \mathcal{G}$, $({}^{(i)}p_k, {}^{(i)}c) \; \Xi \; ({}^{(j)}q_k, {}^{(j)}d)$ for some arc $({}^{(j)}q_k, {}^{(j)}d) \in H_j \in \mathcal{H}$.

(3) If $H_j \to H_{j+1} \in \mathcal{H}$ is a rewrite of a redex rooted at ${}^{(j)}y \in H_j$ and governed by either a rule for a replication symbol, or a communication commit rule, there is a corresponding rewrite $G_i \to G_{i+1} \in \mathcal{G}$ of a redex rooted at a node ${}^{(i)}x \in G_i$, which is governed by the same rule, and ${}^{(i)}x \,\Xi\, {}^{(j)}y$. Further, if $H_j \to H_{j+1}$ and $H_{j'} \to H_{j'+1}$ are two distinct such rewrites, their corresponding $G_i \to G_{i+1}$ and $G_{i'} \to G_{i'+1}$ are also distinct, and all such rewrites occur in the same order in $\mathcal{H}$ and $\mathcal{G}$.

(N.B. The relation Ξ between nodes of two executions of a ruleset, is such that: (i) the two initial nodes are Ξ-related, (ii) the redirections and injections of rewrites and notification steps preserve Ξ-relatedness, (iii) the respective copies of contractum nodes of rewrites governed by the same rule at redexes with Ξ-related roots are Ξ-related. See [Banach et al. (1995)] for more details.)

5 MODULES

In this section we show how to understand π-calculus modules and their translation counterparts in a categorical setting. We start with the π-calculus. A π-calculus module is just a π-calculus expression with some number (possibly zero) of holes, the latter being place holders for where other π-calculus modules or expressions can be substituted. Modules can be generated from the π-calculus syntax by adjoining the case

$$P = H$$

where H is a constant standing for a hole. We write a π-calculus module with n holes as $C(H_1 \ldots H_n)$ where we decorate the hole constant with subscripts (or whatever else seems convenient), to disambiguate different hole occurences in the meta language. (Of course different occurences may be formally disambiguated by using the canonical addressing structure in terms.) The surviving free names $\mathrm{SFn}(C)$ of a module are given in terms of the surviving free names imported into the holes, $\mathrm{SFn}(H_1) \ldots \mathrm{SFn}(H_n)$, by adapting the usual recursive definition of free names of a π-calculus expression (we just show dyadic + and |), viz.

$$\begin{aligned}
&\mathrm{SFn}(x(y).Q_1 + \bar{z}w.Q_2) = \{x, z, w\} \cup (\mathrm{SFn}(Q_1) - \{y\}) \cup \mathrm{SFn}(Q_2)\\
&\mathrm{SFn}(Q_1 \mid Q_2) = \mathrm{SFn}(Q_1) \cup \mathrm{SFn}(Q_2)\\
&\mathrm{SFn}(\nu x Q) = \mathrm{SFn}(Q) - \{x\}\\
&\mathrm{SFn}(!Q) = \mathrm{SFn}(Q)\\
&\mathrm{SFn}(\mathbf{0}) = \varnothing
\end{aligned}$$

Thus for a π-calculus module C with n holes, $\mathrm{SFn}(C) : \mathcal{P}\mathbf{N}^n \to \mathcal{P}\mathbf{N}$, where $\mathbf{N}$ is the alphabet of π-calculus names, and $\mathcal{P}$ is the powerset constructor. We can organise all of this into a double category ${}^{+|}\mathcal{B}$, (for double categories see [Ehresmann (1963)], [Palmquist (1970)], [Bastiani and Ehresmann (1974)]; a tutorial treatment also appears in [Gadducci and Montanari (1995)]).

The objects of ${}^{+|}\mathcal{B}$ are finite sequences of finite sets of π-calculus names, i.e. members of $[\mathcal{P}\mathbf{N}]$, where [-] is the sequence constructor. (We will use underlined letters to name sequences, and square brackets to explicitly list the contents of a sequence.) The vertical arrows of ${}^{+|}\mathcal{B}$ are ordered pointwise inclusions, i.e. assuming the lengths of $\underline{X}$ and $\underline{X}'$ are n, there is a vertical arrow $\underline{X} \subseteq \underline{X}'$ exactly whenever for all $1 \le k \le n$, $X_k \subseteq X'_k$. Composition of vertical arrows is pointwise, and vertical identities are identity ordered pointwise inclusions.

The horizontal arrows of $^{+|}\mathcal{B}$ are finite sequences of π-calculus modules together with certain maps that connect sets of π-calculus names to the modules' inputs. Let $\underline{X} = [X_1 \dots X_m]$ and $\underline{Y} = [Y_1 \dots Y_n]$ be two objects of $^{+|}\mathcal{B}$. A horizontal arrow

$$([C_1 \dots C_n], \theta) : \underline{X} \to \underline{Y}$$

is a pair where: $C_1 \dots C_n$ are π-calculus modules and the length of $\underline{Y}$ is n, and θ is a surjection from the set of occurrences of holes of $[C_1 \dots C_n]$ onto (the set of occurrences of elements of) $\underline{X}$. Such a pair must satisfy for all $1 \le k \le n$,

$$\mathrm{SFn}(C_k(\theta(\not{H}_k))) = Y_k$$

where $\not{H}_k$ is the set of occurrences of holes of C_k, and $\theta(\not{H}_k)$ yields the sets of names to be used by the SFn function for the holes of C_k.

Horizontal composition of arrows $([C_1 \dots C_m], \theta) : [X_1 \dots X_l] \to [Y_1 \dots Y_m]$ and $([D_1 \dots D_n], \phi) : [Y_1 \dots Y_m] \to [Z_1 \dots Z_n]$ is given by

$$([D_1\{\phi(\not{H}_1) / \not{H}_1\}, \dots , D_n\{\phi(\not{H}_n) / \not{H}_n\}], \theta) : [X_1 \dots X_l] \to [Z_1 \dots Z_n]$$

where $D_k\{\phi(\not{H}_k) / \not{H}_k\}$ is the substitution for $\not{H}_k$ within D_k, of those π-calculus modules in $[C_1 \dots C_m]$ identified by ϕ. Horizontal identities are $([\Lambda \dots \Lambda], \eta) : \underline{X} \to \underline{X}$ where Λ is the empty π-calculus module with identity SFn function, and η is the obvious bijection.

A bit more formally, a double category consists of double arrows; and in our case these are given by arbitrary pairs of coinitial horizontal and vertical arrows, each such pair, coinitial on $\underline{X}$ say, defining a unique commuting square as shown below. (The map θ' appearing in the square should be obvious to the reader.)

$$\begin{tikzcd}
\underline{X} \arrow[r, "{([C_1 \dots C_m], \theta)}"] \arrow[d, "\subseteq"'] \arrow[dr, phantom, "\mathbf{a}"] & \underline{Y} \arrow[d, "\subseteq"] \\
\underline{X}' \arrow[r, "{([C_1 \dots C_m], \theta')}"'] & \underline{Y}'
\end{tikzcd}$$

Such squares may be composed both vertically ($*_\mathrm{v}$) and horizontally ($*_\mathrm{h}$), and the crucial interchange law of square composition, that $(\mathbf{a}*_\mathrm{v}\mathbf{a}')*_\mathrm{h}(\mathbf{b}*_\mathrm{v}\mathbf{b}') = (\mathbf{a}*_\mathrm{h}\mathbf{b})*_\mathrm{v}(\mathbf{a}'*_\mathrm{h}\mathbf{b}')$, is easily checked to hold. A predictable collection of properties of identities and compositions of double arrows is required (see the references). What we called horizontal and vertical arrows above, are easily obtained by forgetting one or other component of a double arrow, or more formally, by requiring that the component be an identity.

The set of double arrows with a given horizontal arrow $([C_1 \dots C_m], \theta') : \underline{X}' \to \underline{Y}'$ as codomain, has a least element, namely the double arrow with $([C_1 \dots C_m], \theta_\varnothing) : \underline{\varnothing}_l \to \underline{Y}_\varnothing$ where $\underline{\varnothing}_l = [\varnothing \dots \varnothing]$ a sequence of l empty sets, as domain horizontal arrow. We write $\underline{Y}_\varnothing$ to denote that $\underline{Y}_\varnothing$ is the sequence of smallest sets obtainable from $[C_1 \dots C_m]$ and the appropriate smallest $\theta_\varnothing$.

There is a category $^{+|}\mathcal{B}_\varnothing$ whose objects are the $\underline{\varnothing}_l$, and whose arrows are pairs consisting of smallest horizontal arrows $\underline{\varnothing}_l \to \underline{Y}_\varnothing$ as above, and inclusions $\underline{\varnothing}_m \subseteq \underline{Y}_\varnothing$,

$$\langle([C_1 \dots C_m], \theta_\varnothing) : \underline{\varnothing}_l \to \underline{Y}_\varnothing, \underline{\varnothing}_m \subseteq \underline{Y}_\varnothing\rangle : \underline{\varnothing}_l \to \underline{\varnothing}_m$$

Identities are given by pairs of identity horizontal and vertical arrows, and the composition of $\alpha \equiv \langle([C_1 \ldots C_m], \theta_\varnothing) : \underline{\varnothing}_l \to \underline{Y}_\varnothing, \underline{\varnothing}_m \subseteq \underline{Y}_\varnothing\rangle : \underline{\varnothing}_l \to \underline{\varnothing}_m$ with $\beta \equiv \langle([D_1 \ldots D_n], \phi_\varnothing) : \underline{\varnothing}_m \to \underline{Z}_\varnothing, \underline{\varnothing}_n \subseteq \underline{Z}_\varnothing\rangle : \underline{\varnothing}_m \to \underline{\varnothing}_n$ is

$$\begin{aligned}&\langle([D_1\{\phi'(\underline{H}_1) / \underline{H}_1\}, \ldots , D_n\{\phi'(\underline{H}_n) / \underline{H}_n\}], \theta_\varnothing) : \underline{\varnothing}_l \to \underline{Z'}_\varnothing,\\ &\qquad \underline{\varnothing}_n \subseteq \underline{Z}_\varnothing \subseteq \underline{Z'}_\varnothing\rangle : \underline{\varnothing}_l \to \underline{\varnothing}_n\end{aligned}$$

where $\underline{Z}_\varnothing \subseteq \underline{Z'}_\varnothing$ and $([D_1 \ldots D_n], \phi') : \underline{Y}_\varnothing \to \underline{Z'}_\varnothing$ are determined by the double arrow arising from $\underline{\varnothing}_m \subseteq \underline{Y}_\varnothing$ and $([D_1 \ldots D_n], \phi_\varnothing) : \underline{\varnothing}_m \to \underline{Z}_\varnothing$. The composition is illustrated below. (Note that the inclusion component in an arrow of ${}^{+|}\mathcal{B}_\varnothing$ could be regarded as redundant, but we will keep it anyway.)

$$\begin{tikzcd}
 & & & \underline{Z'}_\varnothing & \\
 & \underline{Y}_\varnothing \arrow[urr, "(\underline{D},\, \phi')"] & & \underline{Z}_\varnothing \arrow[u, "\subseteq"'] & \\
\underline{\varnothing}_l \arrow[ur, "(\underline{C},\, \theta_\varnothing)"] & & \underline{\varnothing}_m \arrow[ul, "\subseteq"] \arrow[ur, "(\underline{D},\, \phi_\varnothing)"'] & & \underline{\varnothing}_n \arrow[ul, "\subseteq"']
\end{tikzcd}$$

Consider the projection ${}^{+|}P_\varnothing : {}^{+|}\mathcal{B} \to {}^{+|}\mathcal{B}_\varnothing$ that takes objects $\underline{Y}$ and vertical arrows $\underline{Y} \subseteq \underline{Y}'$ to the appropriate $\underline{\varnothing}_m$; and horizontal arrows $(\underline{D}, \phi) : \underline{Y} \to \underline{Z}$ and double arrows to the appropriate $\langle(\underline{D}, \phi_\varnothing) : \underline{\varnothing}_m \to \underline{Z}_\varnothing, \underline{\varnothing}_n \subseteq \underline{Z}_\varnothing\rangle : \underline{\varnothing}_m \to \underline{\varnothing}_n$. ${}^{+|}P_\varnothing$ has as good properties as anyone has a right to expect from a relationship between a double category and a normal category. There is a partial converse ${}^{+|}F_\varnothing : {}^{+|}\mathcal{B}_\varnothing \to \mathcal{C}at$ which takes objects $\underline{\varnothing}_m$ to the discrete category whose objects are inclusions $\underline{Y} \subseteq \underline{Y}'$ such that $\underline{\varnothing}_m \subseteq \underline{Y}$, and where arrows $\langle(\underline{D}, \phi_\varnothing) : \underline{\varnothing}_m \to \underline{Z}_\varnothing, \underline{\varnothing}_n \subseteq \underline{Z}_\varnothing\rangle : \underline{\varnothing}_m \to \underline{\varnothing}_n$ take $\underline{Y} \subseteq \underline{Y}'$ (over $\underline{\varnothing}_m$) to $\underline{Z} \subseteq \underline{Z}'$ (over $\underline{\varnothing}_n$), giving essentially the double arrow generated by $\underline{Y} \subseteq \underline{Y}'$ and the appropriate $(\underline{D}, \phi)$. This characterises horizontal composition in ${}^{+|}\mathcal{B}$ via an opfibration. Note that this opfibration approach cannot be extended to all of the machinery of ${}^{+|}\mathcal{B}$, since that would necessitate that double arrows were composable diagonally, which is not true in general.

${}^{+|}\mathcal{B}$ gives a description roughly at the level of a system linker, while ${}^{+|}\mathcal{B}_\varnothing$ is more abstract, in fact being quite close to a Lawvere theory, though we will not pursue this here.

The presuperscripts $+|$ on ${}^{+|}\mathcal{B}$ and ${}^{+|}\mathcal{B}_\varnothing$ indicate that we are not assuming commutativity of (prefixed) + nor commutataivity/associativity of $|$ for modules. We can relax these restrictions individually or together. Thus ${}^{|}\mathcal{B}$ is obtained by allowing commutativity of prefixed +, identifying eg. arrows $(\pi_1.H_1 + \pi_2.H_2, \theta)$ and $(\pi_2.H_2 + \pi_1.H_1, \theta^{\sim})$ where θ and $\theta^{\sim}$ are such as to ensure each H_i picks up the same input names in both arrows. Similarly ${}^{|}\mathcal{B}_\varnothing$. Likewise $\mathcal{B}$ and $\mathcal{B}_\varnothing$ where we allow all three structural laws for + and $|$. In each case the relationship between the two categories is as above, and there are obvious functor pairs $(\Theta^+, \Theta^+_\varnothing) : ({}^{+|}\mathcal{B}, {}^{+|}\mathcal{B}_\varnothing) \to ({}^{|}\mathcal{B}, {}^{|}\mathcal{B}_\varnothing)$, and $(\Theta^|, \Theta^|_\varnothing) : ({}^{|}\mathcal{B}, {}^{|}\mathcal{B}_\varnothing) \to (\mathcal{B}, \mathcal{B}_\varnothing)$.

Note however that incorporating (π-calculus) alpha convertibility cannot be allowed. The scope of some bound name in some module may contain a hole, and then the behaviour of a combination of modules will not be preserved after alpha conversion since a different name will be captured by the binder.

Now we turn to the translation in order to fit this into a similar categorical framework. We will regard the space of MONSTR function symbols as split into three parts namely {**`Initial`**}, the constant five function symbols of the communication protocol, and a set $\mathbf{F}^\pi$ of user-selectable function symbols used for the translation body. At the most concrete level, we recognise that node identifiers in concrete syntax rules are just tags for relationships between argument positions of occurrences of the parent symbols, so we always consider rules up to alpha conversion of nodeids. For the following, the translation of a π-calculus module E will be $\mathrm{Tr}^{-}(E)$, the set of body rules generated together with the interface functions $Args_P : \mathrm{Free}(P) \to \boldsymbol{args}_P$, needed for the input symbols, and generated for the output symbols. Note that for what follows to work properly, we need to understand that the translation of a π-calculus module can yield all possible orderings of channel names to arguments for function symbols (i.e. all possible $Args_P$ functions), not just the ones displayed (for definiteness' sake) in the Appendix.

Let ${}^{+|}\mathcal{B}_{\mathrm{H}}$ be the horizontal category of ${}^{+|}\mathcal{B}$. Above ${}^{+|}\mathcal{B}_{\mathrm{H}}$ we build a category ${}^{+|}\mathcal{H}_{\mathrm{CH}}$ and a projection ${}^{+|}\Gamma_{\mathrm{CH}} : {}^{+|}\mathcal{H}_{\mathrm{CH}} \to {}^{+|}\mathcal{B}_{\mathrm{H}}$. The objects of ${}^{+|}\mathcal{H}_{\mathrm{CH}}$ are pairs $(\underline{Y} \to \mathsf{F}^{\pi}, [\underline{Y}_1 \ldots \underline{Y}_n])$, the first being a mapping from elements of $\underline{Y}$ to distinct function symbols in F^{π}, and the second being a choice of ordering for each of the elements of $\underline{Y}$. Such a pair is equivalent to a sequence of $Args_P : \mathrm{Free}(P) \to \boldsymbol{args}_P$ functions used in the translation, one for each of the distinct function symbols corresponding to elements of $\underline{Y}$, but does not mention arbitrary nodeids. The projection ${}^{+|}\Gamma_{\mathrm{CH}}$ sends each $(\underline{Y} \to \mathsf{F}^{\pi}, [\underline{Y}_1 \ldots \underline{Y}_n])$ to $\underline{Y}$.

The arrows of ${}^{+|}\mathcal{H}_{\mathrm{CH}}$ between objects $(\underline{X} \to \mathsf{F}^{\pi}, [\underline{X}_1 \ldots \underline{X}_m])$ and $(\underline{Y} \to \mathsf{F}^{\pi}, [\underline{Y}_1 \ldots \underline{Y}_n])$ arise from sequences of translations of π-calculus modules whose totality of inputs are specified by the input sequence of interface functions $Args_{\mathsf{X}_i} : \mathrm{Free}(\mathsf{X}_i) \to \boldsymbol{args}_{\mathsf{X}_i}$, and for each element of which, the generated output interface function is the appropriate function $Args_{\mathsf{Y}_j} : \mathrm{Free}(\mathsf{Y}_j) \to \boldsymbol{args}_{\mathsf{Y}_j}$. Such an arrow of ${}^{+|}\mathcal{H}_{\mathrm{CH}}$ projects down under ${}^{+|}\Gamma_{\mathrm{CH}}$ to an arrow $([C_1 \ldots C_n], \theta) : \underline{X} \to \underline{Y}$ of ${}^{+|}\mathcal{B}_{\mathrm{H}}$ where C_j is the module whose top level symbol in the translation is Y_j and where θ sends all hole occurences tagged X_i in $[C_1 \ldots C_n]$ to the element $\mathrm{Free}(\mathsf{X}_i)$ of $\underline{X}$. Arrow composition is just the sequence of unions of rulesets which have common interfaces, with suppression of the $Args$ functions pertaining to the matched interfaces.

All of this is fine, and the semantics is complete (in the sense that any behaviour of π-calculus modules can be reproduced by the translation), but is not sound because of the possibility of name clashes between symbols used in modules that are composed. If the same symbol is used (with the same arity) to encode different subexpressions in the two different modules (despite being used consistently within each module separately), then the composed translation is capable of behaviours not corresponding to behaviours of the composed π-calculus modules. To overcome this we move to *abstract rulesets* to capture the properties of translations but avoiding clashes.

We can define a law of composition of rulesets, which is union of rulesets as before, *but only if there is no clash* (except for matching at interfaces). Obviously this cannot be a composition in a category, but now we take equivalence classes of such rulesets (under clash free retagging of the originating π-calculus modules, and under permutations of the channel name to argument position correspondences of non-interface function symbols). We now get a fresh category ${}^{+|}\mathcal{H}_{\mathrm{H}}$ whose objects are $[\underline{Y}_1 \ldots \underline{Y}_n]$, and whose arrows are sequences of the equivalence classes mentioned. The composition of two equivalence classes with a compatible interface, is that equivalence class containing the union of any clash free pair of elements, one from each. To objects $[\underline{Y}_1 \ldots \underline{Y}_n]$ of ${}^{+|}\mathcal{H}_{\mathrm{H}}$ we adjoin formal identities $[\underline{\Lambda} \ldots \underline{\Lambda}]$. These correspond to empty π-calculus expressions as before. (N.B. ${}^{+|}\mathcal{H}_{\mathrm{CH}}$ has similar identities.)

There is now a projection ${}^{+|}\Gamma_{\mathrm{H}} : {}^{+|}\mathcal{H}_{\mathrm{H}} \to {}^{+|}\mathcal{B}_{\mathrm{H}}$. It is not hard to see that ${}^{+|}\Gamma_{\mathrm{H}}$ is both a fibration and an opfibration. The opfibration viewpoint is appropriate when the "generative" aspect is being stressed, building systems bottom up by taking small components and combining them using various enclosing modules to get a big final result. The fibration viewpoint is appropriate when the "substitutive" aspect is being stressed, building systems top down by taking a high level structuring module and filling the holes with increasingly low level implementations to get the final system.

The above dealt with the horizontal aspect of ${}^{+|}\mathcal{B}$. For the vertical aspect there is ${}^{+|}\Gamma_{\mathrm{CV}} : {}^{+|}\mathcal{H}_{\mathrm{CV}} \to {}^{+|}\mathcal{B}_{\mathrm{V}}$, where ${}^{+|}\mathcal{B}_{\mathrm{V}}$ is the vertical category of ${}^{+|}\mathcal{B}$. The objects of ${}^{+|}\mathcal{H}_{\mathrm{CV}}$ are as for ${}^{+|}\mathcal{H}_{\mathrm{CH}}$, and the arrows between objects ($\underline{Y} \to \mathbf{F}^{\pi}$, $[\underline{Y}_1 \ldots \underline{Y}_n]$) and ($\underline{Y}' \to \mathbf{F}^{\pi}$, $[\underline{Y}'_1 \ldots \underline{Y}'_n]$) arise from pointwise inclusions of subsequences, i.e. when for all $1 \le k \le m$, $\underline{Y}_k$ is a subsequence of $\underline{Y}'_k$. The action of ${}^{+|}\Gamma_{\mathrm{CV}}$ is now clear.

The horizontal and vertical combine into a projection of double categories ${}^{+|}\Gamma_{\mathrm{CD}} : {}^{+|}\mathcal{H}_{\mathrm{CD}} \to {}^{+|}\mathcal{B}$. Unlike in the case of ${}^{+|}\mathcal{B}$, a coinitial pair is no longer sufficient to determine a double arrow of ${}^{+|}\mathcal{H}_{\mathrm{CD}}$ uniquely, because of the independent choices of tagging and permutations (for function symbols corresponding to interior nodes of the parse tree of a module), in the horizontal arrow codomain and domain of a double arrow.

The vertical dimension for abstract rulesets is simple enough. It is ${}^{+|}\Gamma_{\mathrm{V}} : {}^{+|}\mathcal{H}_{\mathrm{V}} \to {}^{+|}\mathcal{B}_{\mathrm{V}}$, where the objects are $[\underline{Y}_1 \ldots \underline{Y}_n]$ as before, and the arrows are just the pointwise inclusions of subsequences. Here, it is even easier to see that ${}^{+|}\Gamma_{\mathrm{V}}$ is both a fibration and an opfibration.

Predictably, the two dimensions at the abstract level combine to give a double category and associated projection ${}^{+|}\Gamma_{\mathrm{D}} : {}^{+|}\mathcal{H}_{\mathrm{D}} \to {}^{+|}\mathcal{B}$. This time, a coinitial pair *is* sufficient to determine a double arrow of ${}^{+|}\mathcal{H}_{\mathrm{D}}$ uniquely, because the various independent choices have been abstracted away. ${}^{+|}\Gamma_{\mathrm{D}}$ is both a fibration and an opfibration of double categories, by which we mean it is a fibration and an opfibration, in both the horizontal and vertical dimensions independently, and also in the diagonal direction. (This is in contrast to the situation with ${}^{+|}P_{\varnothing}$, where there was not enough information in ${}^{+|}\mathcal{B}_{\varnothing}$ to distinguish the cases where two diagonally adjacent double arrows were diagonally composable, from the cases where they were not.) Also the relationship of the projection ${}^{+|}\Gamma_{\mathrm{D}}$ to the Lawvere-like category ${}^{+|}\mathcal{B}_{\varnothing}$, is now clear via the projection ${}^{+|}P_{\varnothing}$.

As before, the $+|$ superscripts indicate fussiness about structural aspects of $+$ and $|$. All that we have done readily generalises to "without +" and "without + and |" versions via the evident $({\Theta_{\mathrm{D}}}^{+}, \Theta^{+}, \Theta^{+}_{\varnothing}) : ({}^{+|}\mathcal{H}_{\mathrm{D}}, {}^{+|}\mathcal{B}, {}^{+|}\mathcal{B}_{\varnothing}) \to ({}^{|}\mathcal{H}_{\mathrm{D}}, {}^{|}\mathcal{B}, {}^{|}\mathcal{B}_{\varnothing})$, and $({\Theta_{\mathrm{D}}}^{|}, \Theta^{|}, \Theta^{|}_{\varnothing}) : ({}^{|}\mathcal{H}_{\mathrm{D}}, {}^{|}\mathcal{B}, {}^{|}\mathcal{B}_{\varnothing}) \to (\mathcal{H}_{\mathrm{D}}, \mathcal{B}, \mathcal{B}_{\varnothing})$. These variants actually correspond to issues of concrete versus abstract syntax for MONSTR rulesets. We comment on these now.

Non-commutativity of + maps to textual ordering of rule groups corresponding to prefixed alternatives, a byproduct of concrete syntax. Similarly, non-commutativity of | maps to textual ordering of function nodes on the RHS of the rule for parallel composition. In the abstract syntax of MONSTR, the specification of a rule has a *pattern*, a graph-like object where the nodes form a *set*. Similarly rules are parcelled up into sets rather than sequences. Non-associativity of | translates to extra structural rules that decompose a multilevel parallel composition in stages, rather than all at once. This is a different issue, and the intended equivalence is at the level of simulation.

Finally, we return to (π-calculus) alpha conversion and scope extrusion. It has been emphasised that alpha conversion cannot be considered for syntactic reasons. Still, this does not affect the aptness of our semantics of modules (even of modules without holes). This is because alpha conversion is only *needed* in dynamics situations, when a transition threatens to cause unintended name capture. When an expression is merely regarded as a *specification* of possible behaviours, it is not needed. The same applies with even more force to scope extrusion. Even if statically, an expression needs an extrusion to enable a communication to take place, this is a dynamic requirement, and its translation needs no alteration. The dynamic behaviour of the generated rulesets copes with the situation automatically. This is because the representing graphs of an expression before and after extrusion are isomorphic ([Banach et al. (1995)], lemma 7.5). This illustrates how the translation separates concerns in this area. The specificational aspects of a π-calculus expression or module map to the ruleset generated by the translation. The dynamic aspects of its behaviours map to the properties of the representing graphs generated when the ruleset is augmented with the protocol rules and a suitable initial rule, and set in motion. We now discuss the categorical aspects of this within the framework we have built. From now on, we ignore the potential for fussiness about + and/or | ; everything below can be presuperscripted with +| or | and appropriate projections exist.

We start by concentrating on dynamics at the pure π-calculus level, ignoring the translation for the moment. We build another double category $\mathcal{S}$. The objects of $\mathcal{S}\,\underline{E}$ say, are sequences of *ground* π-calculus expressions, that is to say, sequences of π-calculus expressions without holes, (such expressions are intended to be interpreted now as systems, i.e. as specifications of dynamic behaviour). The horizontal arrows of $\mathcal{S}$ are given by similar data to that for $\mathcal{B}$, i.e. $([C_1 \ldots C_n], \theta) : \underline{E} \to \underline{F}$ is a horizontal arrow iff substituting the various $\underline{E}$ into C_i as prescribed by θ yields F_i. The vertical arrows of $\mathcal{S}$ are given by unrestriction, viz. $\underline{\overline{\nu}} : \underline{E} \to \underline{E}'$ is a vertical arrow iff the ground expressions in $\underline{E}'$ are obtained by stripping off some ν's at top level in the concrete syntax of the ground expressions in $\underline{E}$. Obviously a horizontal and a vertical arrows combine to form a double arrow, as for $\mathcal{B}$, and the identities of $\mathcal{S}$ are clear. There is an obvious projection $Q : \mathcal{S} \to \mathcal{B}$. This takes objects of $\mathcal{S}\,\underline{E}$, to their sequences of sets of free names $\underline{X}$ which are objects of $\mathcal{B}$; horizontal arrows $([C_1 \ldots C_n], \theta) : \underline{E} \to \underline{F}$ are projected down to horizontal arrows $([C_1 \ldots C_n], \theta) : \underline{X} \to \underline{Y}$ where $Q(\underline{E}) = \underline{X}$, etc.; vertical arrows $\underline{\overline{\nu}} : \underline{E} \to \underline{E}'$ project to vertical arrows $\underline{X} \subseteq \underline{X}'$; and double arrows project by extrapolation from the preceding. It is not hard to see that like Γ_{D}, Q is a fibration and opfibration in horizontal, vertical and diagonal directions.

Now we build a further double category $\mathcal{T}^{\delta}$ and projection $R : \mathcal{T}^{\delta} \to \mathcal{S}$. The objects of $\mathcal{T}^{\delta}$, which we write as $\underline{A}$ say, are sequences of sets of detailed traces of the sequence of objects $\underline{E}$ below them; that is to say, A_i is a set of detailed traces of E_i. We had better say what a detailed trace of E_i is. It is a sequence $[s_0, p_1, s_1, p_2, \ldots]$ where the s_i are states of the labelled transition system of E_i (according to the standard rules of inference of π-calculus), and each p_i is a proof that there is a transition from s_{i-1} to s_i. Since we are dealing with dynamics, detailed traces are defined up to alpha conversion of bound names, and evidently $E_i = s_0$ up to such conversion, for each detailed trace in A_i.

Vertical arrows are generated by unrestriction and inclusion (modulo alpha conversion). If $[s_0, p_1, s_1, p_2, \ldots]$ is a detailed trace of E_i, and $\overline{\nu}_i : E_i \to E_i'$ is a component of a vertical arrow of $\mathcal{S}$, then $[\overline{\nu}_i(s_0), \overline{\nu}_i(p_1), \overline{\nu}_i(s_1), \overline{\nu}_i(p_2), \ldots]$ is a detailed trace of E_i', where $\overline{\nu}_i(s_0)$ is given by stripping off the appropriate top level ν's from s_0 (with the specific free names $x' \ldots z'$ of $\overline{\nu}_i(s_0)$ that are generated from the relevant alpha convertible bound names $\nu x \ldots \nu z$ of s_0 being infered from the structural correspondence between E_i and E_i'), and the stripping and free name identification is propagated to the remainder of the detailed trace. The vertical arrows of $\mathcal{T}^\delta$ are now given by set continuity and sequence construction from the preceding.

Horizontal arrows are given by data $([C_1 \ldots C_n], \theta) : \underline{A} \to \underline{B}$, whose action is as follows. Suppose we have the arrow $([C_1 \ldots C_n], \theta) : \underline{A} \to \underline{B}$ which projects by R to $([C_1 \ldots C_n], \theta) : \underline{E} \to \underline{F}$. Then roughly speaking, $\underline{B}$ will be the sequence of sets of all detailed traces of $\underline{F}$ such that for each trace b_i in B_i, there are traces $a_1 \ldots a_m$ in $A_1 \ldots A_m$ of any of $E_1 \ldots E_m$ which occur substituted for hole occurrences in F_i by θ, such that whenever a step of b_i involves one or more of the subcomponents $A_1 \ldots A_m$, the contribution(s) from the relevant A_j(s) agree suitably with appropriate steps of the relevant a_j(s).

More precisely, consider the rules for + and | in the *original* inference system of [Milner et al. (1992)] for the π-calculus[1].

$$\frac{P \xrightarrow{\alpha} Q}{P + R \xrightarrow{\alpha} Q + R} \qquad \frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R} \qquad \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \varnothing$$

We say that the term R is *not involved* in any transition for which the proof includes such a step. Otherwise, for every other term in every rule occurrence of a proof, the term *is involved* in the transition. With the notation used above let

$$\begin{aligned}
&a_1 = [s_{01}, p_{11}, s_{11}, p_{21}, \ldots],\\
&\qquad a_1{}' = [s_{11}, p_{21}, \ldots], \ldots,\\
&a_m = [s_{0m}, p_{1m}, s_{1m}, p_{2m}, \ldots],\\
&\qquad a_m{}' = [s_{1m}, p_{2m}, \ldots], \ldots,\\
&b_i = [t_{0i}, q_{1i}, t_{1i}, q_{2i}, \ldots],\\
&\qquad b_i{}' = [t_{1i}, q_{2i}, \ldots].
\end{aligned}$$

Define the set valued function $\Psi_{C,\theta,\underline{E}}$ by

$b_i \in \Psi_{C_i,\theta,\underline{E}}(a_1 \ldots a_m)$ iff

EITHER $C_i(E_1 \ldots E_m) \xrightarrow{\alpha} C_i'(E_1 \ldots E_m)$
and none of the $E_1 \ldots E_m$ are involved in the transition
and $b_i' \in \Psi_{C_i',\theta,\underline{E}}(a_1 \ldots a_m)$

OR $C_i(E_1 \ldots E_j \ldots E_m) \xrightarrow{\alpha} C_i'(E_1 \ldots E_j' \ldots E_m)$
and exactly one of the $E_1 \ldots E_m$, namely E_j (suitably substituted), is involved in the transition, and (a suitably substituted copy of) p_{1j} is a subproof of q_{1i}
and $b_i' \in \Psi_{C_i',\theta,\underline{E}[E_j'/E_j]}(a_1 \ldots a_j' \ldots a_m)$

OR $C_i(E_1 \ldots E_j \ldots E_k \ldots E_m) \xrightarrow{\alpha} C_i'(E_1 \ldots E_j' \ldots E_k' \ldots E_m)$
and exactly two of the $E_1 \ldots E_m$, namely E_j and E_k (suitably substituted), are involved in the transition,
and (suitably substituted copies of) p_{1j} and p_{1k} are

1. This allows us to speak about external communication easily. The system of section 3 makes that aspect of π-calculus more awkward to treat.

subproofs of q_{1i}
and $b_i' \in \Psi_{C_i',\theta,\underline{E}[E_j'/E_j][E_k'/E_k]}(a_1 \ldots a_j' \ldots a_k' \ldots a_m)$.

In the above, $\underline{E}[E_j'/E_j]$ is just short for $E_1 \ldots E_j' \ldots E_m$, and there are three cases because a transition can be wholly contained in the context part, or can involve at most two subterms among the $E_1 \ldots E_m$. The "suitably substituted" caveat for subterms and proof subobjects covers all cases where an external communication of say E_j contributes to an internal communication of $C_i(E_1 \ldots E_m)$ and/or a free name of E_j is captured by a binder in C_i.

The original inference system of [Milner et al. (1992)] for the π-calculus does not feature replication as per our BNF in section 3. We can address this most simply by adjoining the rule $!P \xrightarrow{!} P \mid !P$ to our system. Then the first case for $\Psi_{C,\theta,\underline{E}}$ has to be modified with the observation that a ! transition in the context duplicates a subterm, and in particular may duplicate an E_j subterm. In such a case each residual of E_j in the resulting term must be regarded as independent, and each must pick up a separate copy of the relevant detailed trace a_j. Thus the subscript list $\underline{E}$ and parameter list $a_1 \ldots a_m$ of Ψ grow when this happens. Alternative rules for !, which infer a transition for $!P$ on the premiss that P has such a transition, can be dealt with similarly, except that the two residuals of an E_j (and hence the corresponding detailed traces) would not be the same; one being the original E_j and trace, the other being what remained after the transition.

With all this, the action of the horizontal arrow $([C_1 \ldots C_n], \theta) : \underline{A} \to \underline{B}$ is given by set continuity over the A_i, and sequence construction. Also it is clear that a coinitial vertical and horizontal arrow pair combine to form a unique double cell.

One can check straightforwardly that $R : \mathcal{T}^\delta \to \mathcal{S}$ is an opfibration in horizontal, vertical and diagonal directions, but this time is not a fibration. The latter can be seen in eg. the horizontal direction, by noticing that name capture by a context C can conceal whether a particular set of traces is the maximal set specified by Ψ of some other action of a context D; i.e. we may have $\Psi_C(B) = \Psi_{C(D)}(A)$ without $B = \Psi_D(A)$, spoiling the unique factorisation required.

As previously, we can transport the whole situation to the MONSTR level without difficulty. We build a double category $\mathcal{M}$ whose objects are sequences of (abstract) *system* rulesets, i.e. sequences of (equivalence classes of) collections of rules which include a rule for Initial, and such that the contractum of none of them creates a function node whose symbol does not have rules within the set. The objects also include the $[\underline{Y}_1 \ldots \underline{Y}_n]$ sequences of free name sequences that specify the argument orderings for the top level function symbols created by rewriting the Initial rules. The vertical arrows are pointwise inclusions of $[\underline{Y}_1 \ldots \underline{Y}_n]$ sequences corresponding to unrestrictions at the abstract ruleset level[1]. The horizontal arrows are sequences of abstract rulesets as for $\mathcal{H}_\mathrm{D}$, the law of composition of an arrow and a system being abstract ruleset union combined with the obvious replacement of the rule for Initial. As usual a coinitial vertical and hor-

1. An unrestriction at the concrete ruleset level discards the rules and symbols for the relevant sequence of top level restrictions $\nu x \ldots \nu z$, and replaces the Initial rule with one which creates the contractum function created by the rule corresponding to the last of them, the relevant channel nodes now being created free by the new Initial rule (see the Appendix). An unrestriction at the abstract ruleset level just does this for each concrete ruleset in the equivalence class.

izontal pair form a unique double cell, and it is not hard to see that there are functors $\Delta : \mathcal{M} \to \mathcal{H}_\mathrm{D}$ and $\Gamma_\mathrm{S} : \mathcal{M} \to \mathcal{S}$, both of which are fibrations and opfibrations in horizontal, vertical and diagonal directions.

The last construction we will make in this paper is the MONSTR analogue of $\mathcal{T}^\delta$ which we will call $\mathcal{N}^\delta$. The objects of $\mathcal{N}^\delta$ are sequences of sets of (abstract) detailed MONSTR traces, so let us say what a concrete detailed MONSTR trace is first. A concrete detailed MONSTR trace is a sequence $\mathcal{G} = [G_0, \pi_1, G_1, \pi_2, G_2, \ldots]$ whose elements G_i are execution graphs of the ruleset in question, and whose elements π_i consist of the paraphernalia of the execution step $G_{i-1} \to G_i$; i.e. firstly the injection and redirection functions between nodes of G_{i-1} and G_i, i_{G_{i-1},G_i}, r_{G_{i-1},G_i}; and secondly, for notifications and suspensions, the active node of G_{i-1} in question, while for rewrites, the redex matching and contractum node identification functions[1]. Since we want to consider external communication at the MONSTR level, we have to augment these steps by *external* π_i steps corresponding to state transitions of interface channel nodes attributable to the actions of fictitious external agents cooperating in an external communication of the system. For simplicity theorems 4.1 and 4.2 did not consider such situations, but there is no technical obstacle to doing so in principle, so we will take such possibilities for granted below. An abstract detailed MONSTR trace equivalences all of these by graph isomorphism (done pointwise over the trace) and clash free renaming of translation body function symbols (done consistently through the trace).

The vertical arrows of $\mathcal{N}^\delta$ correspond naturally to the dynamics of unrestriction. The first few rewrites of a detailed trace which correspond syntactically to the first few ν's and bound channel instantiations, are omitted in favour of a rewrite by a different Initial rule which just directly instantiates these channels as free channels, and creates the function node that spawns the rest of the trace. The horizontal arrows, given by the data in the horizontal arrows of $\mathcal{H}_\mathrm{D}$, relate an abstract trace $\mathcal{G}$ to all the abstract traces $\mathcal{H}$ of the larger ruleset (with modified Initial rule), such that the steps corresponding to the included subsystem match those of $\mathcal{G}$. It is possible to identify these quite precisely at the concrete level because of the very detailed information maintained within the π_i elements of traces. Whence we proceed by set continuity and sequence construction as previously. At the abstract level, coinitial vertical and horizontal pairs form unique double cells. Finally there are projections $\Gamma_\mathrm{T} : \mathcal{N}^\delta \to \mathcal{T}^\delta$ and $\Upsilon : \mathcal{N}^\delta \to \mathcal{M}$. The former is a fibration and opfibration in horizontal, vertical and diagonal directions, while the latter is just an opfibration in horizontal, vertical and diagonal directions. Υ takes a trace of a system $\mathcal{R}$ to the system $\mathcal{R}$ itself, with the concommitant action on objects, arrows and double cells. Γ_T takes any MONSTR detailed trace to its serialisation and corresponding π-calculus detailed trace via theorem 4.2, (strictly speaking via the mild extension of theorem 4.2 to non weakly fair executions, discussed in [Banach et al. (1995)]). For Γ_T, proof objects p_i correspond to the redexes mentioned in a sequence of π_i elements of a serial simulation of the transition which p_i witnesses. The action on objects, arrows and double cells is now also clear. We summarise the constructions of this section in the commutative diagram below.

1. All this information enables us to say exactly how each node of G_i arises from nodes of G_{i-1} and the rule data. See [Banach (1996)].

$$
\begin{tikzcd}
\mathcal{N}^{\delta} \arrow[r, "\Upsilon"] \arrow[d, "\Gamma_{\mathrm{T}}"'] & \mathcal{M} \arrow[r, "\Delta"] \arrow[d, "\Gamma_{\mathrm{S}}"] & \mathcal{H}_{\mathrm{D}} \arrow[d, "\Gamma_{\mathrm{D}}"] & \\
\mathcal{T}^{\delta} \arrow[r, "R"'] & \mathcal{S} \arrow[r, "Q"'] & \mathcal{B} \arrow[r, "P_{\varnothing}"'] & \mathcal{B}_{\varnothing}
\end{tikzcd}
$$

6 CONCLUSIONS

In the preceding sections, we have reviewed a translation of the π-calculus into MONSTR rulesets, and have examined the categorical properties of the semantics of modules from that perspective. In particular we have seen the relationship between name-sensitive and name-insensitive categories ($\mathcal{B}$ vs. $\mathcal{B}_{\varnothing}$), and have seen how abstract rulesets can avoid the unpleasant consequences of name clashes. We have seen how this semantics of modules and their free names, the functor Γ_{D}, has a very natural extension to a semantics of modules and ground systems, the functor Γ_{S}, and thence an equally natural extension to a semantics of modules and traces of ground systems, the functor Γ_{T}. The functor pairs (R, Υ) and (Q, Δ), which relate these, tie up semantics via MONSTR systems to semantics versed in terms of the transition system of π-calculus quite tightly. Nevertheless this richness of structure creates almost more new avenues to explore than it closes paths of enquiry: Similar techniques can be expected to work for other calculi. The structures here should be related to both the work of [Gadducci and Montanari (1995)] and to the interaction categories of eg. [Abramsky (1994)]. Other operations on traces, and more generally other dynamic semantic techniques such as bisimulations, ought to fit into such a framework. These and other issues will be explored elsewhere.

References

[Abramsky (1994)] Abramsky S., Interaction Categories and Communicating Sequential Processes. *in*: A Classical Mind: Essays in honour of C. A. R. Hoare, Roscoe (ed.), 1-16, Prentice Hall, (1994).

[Banach (1993)] Banach R., MONSTR: Term Graph Rewriting for Parallel Machines. *in*: Term Graph Rewriting: Theory and Practice, Sleep, Plasmeijer, van Eekelen (eds.), 243-252, John Wiley, (1993).

[Banach (1995)] Banach R., MONSTR II — Suspending MONSTR Semantics and Independence. *Submitted to* J.UCS, (1995).

[Banach (1996)] Banach R., MONSTR I — Fundamental Issues and the Design of MONSTR. J.UCS, **2**, (1996), 164-216. http://www.iicm.tu-graz.ac.at/jucs

[Banach and Papadopoulos (1993)] Banach R., Papadopoulos G., Parallel Term Graph Rewriting and Concurrent Logic Programs. *in*: Proc. WPDP-93, Bulgarian Acad. of Sci., Boyanov (ed.), (1993), 303-322. (North Holland, to appear.)

[Banach and Papadopoulos (1995a)] Banach R., Papadopoulos G., Linear Behaviour of Term Graph Rewriting Programs. *in*: Proc. A.C.M. SAC-95, (1995), 157-163.

[Banach and Papadopoulos (1995b)] Banach R., Papadopoulos G., A Highly Parallel Model for Object-Oriented Concurrent Constraint Programming. Proc. IEEE ICA^{3}PP-95, **1**, (1995), 61-70.

[Banach and Papadopoulos (1995c)] Banach R., Papadopoulos G., Term Graph Rewriting as a Specification and Implementation Framework for Concurrent Object-Oriented Programming Languages. Proc. IEEE MPPM-95, (1995), to appear.

[Banach and Watson (1989)] Banach R., Watson P., Dealing with State in Flagship: the MONSTR Computational Model. *in*: Proc. CONPAR-88, Jesshope, Reinhartz (eds.), 595-604, B.C.S. Workshop Series, Cambridge University Press, (1989).

[Banach et al. (1988)] Banach R., Sargeant J., Watson I., Watson P., Woods V., The Flagship Project. *in*: Proc. UK-IT-88, (Alvey Technical Conference), 242-245, Information Engineering Directorate, Department of Trade and Industry, IEE Publications, (1988).

[Banach et al. (1995)] Banach R., Balazs J., Papadopoulos G., A Translation of the Pi-Calculus into MONSTR. J.UCS **1**, (6), (1995), 335-394. http://www.iicm.tu-graz.ac.at/jucs

[Bastiani and Ehresmann (1974)] Bastiani A., Ehresmann C., Multiple Functors I: Limits Relative to Double Categories. Cahiers de Topologie et Géométrie Differentielle, **15**, (1974), 545-621.

[Berry and Boudol (1990)] Berry G., Boudol G., The Chemical Abstract Machine. *in*: 17th Annual Symposium on Principles of Programming Languages, A.C.M., (1990). *Also in*: Theoretical Computer Science **96**, (1992), 217-248.

[Ehresmann (1963)] Ehresmann C., Catégories Doubles et Catégories Structurées. C. R. Acad. Sci. Paris, **256**, (1963), 1198-1201.

[Gadducci and Montanari (1995)] Gadducci F., Montanari U., Enriched Categories as Models of Computation. Proc. 3rd Italian Conference on Theoretical Computer Science, Ravello, (1995).

[Milner (1989)] Milner R., Communication and Concurrency. Prentice-Hall, (1989).

[Milner (1993a)] Milner R., The Polyadic Pi-Calculus: A Tutorial. *in*: Logic and Algebra of Specification, Bauer, Brauer, Schwichtenberg (eds.), 203-246, Springer, (1993).

[Milner et al. (1991)] Milner R., Parrow J., Walker D., Modal Logics for Mobile Processes. *in*: Proc. CONCUR-91, Baeten, Groote (eds.), LNCS **527** 45-60, Springer, (1991).

[Milner et al. (1992)] Milner R., Parrow J., Walker D., A Calculus of Mobile Processes – I / II. Inf. and Comp. **100**, (1992), 1-40, 41-77.

[Montanari and Pistore (1995)] Montanari U., Pistore M., Concurrent Semantics for the π-Calculus. *in*: Proc. MFPS-95, ENTCS **1**, (1995). http://www.elsevier.nl/mcs/tcs/pc/volume01.htm

[Palmquist (1970)] Palmquist P., The Double Category of Adjoint Squares. *in*: Reports of the Midwest Category Seminar V, Gray, MacLane (eds.), LNM **195**, 123-153, Springer, (1970).

[Watson and Watson (1987)] Watson P., Watson I., Evaluating Functional Programs on the Flagship Machine. *in*: Proc. FLCA-87, Kahn (ed.), LNCS **274** 80-97, Springer, (1987).

[Watson et al. (1987)] Watson I., Woods V., Watson P., Banach R., Greenberg M., Sargeant J., Flagship: A Parallel Architecture for Declarative Programming. *in*: Proc. 15th Annual International Symposium on Computer Architecture, Hawaii, ACM, (1987).

[Watson et al. (1989)] Watson I., Sargeant J., Watson P., Woods V., The Flagship Parallel Machine. *in*: Proc. CONPAR-88, Jesshope, Reinhartz (eds.), 125-133, BCS Workshop Series, Cambridge University Press, (1989).

Appendix

The Translation Body

$\{\mathbf{0}\}_P$: Free(P) = $\varnothing$; $Args_P = \varnothing$; Rules(P) $\equiv$ { $\texttt{P => *Root}$ }

$\{\nu x Q\}_P$: Free(P) = Free(Q) − $\{x\}$;

If $x \notin$ Free(Q) then Set $Args_P = Args_Q$ and $\boldsymbol{args}_P = \boldsymbol{args}_Q$;

Rules(P) $\equiv$ { $\texttt{P[}\boldsymbol{args}_P\texttt{] => *Q[}\boldsymbol{args}_Q\texttt{]}$ }

else ($x \in$ Free(Q)) then assume for simplicity that $\texttt{x}$ occurs last in $\boldsymbol{args}_Q$;

Set $Args_P = Args_Q \big|_{\mathrm{Free}(P)}$ and $\boldsymbol{args}_P = all\_except\_the\_last\_of(\boldsymbol{args}_Q)$;

Rules(P) $\equiv$ { $\texttt{P[}\boldsymbol{args}_P\texttt{] => *Q[}\boldsymbol{args}_P\texttt{ x:Empty]}$ }

$\{!Q\}_P$: Free(P) = Free(Q) ; Set $Args_P = Args_Q$ and $\boldsymbol{args}_P = \boldsymbol{args}_Q$;

Rules(P) $\equiv$ { $\texttt{P[}\boldsymbol{args}_P\texttt{] => *Q[}\boldsymbol{args}_Q\texttt{],*P[}\boldsymbol{args}_P\texttt{]}$ }

$\{Q_1 \mid Q_2 \mid \dots \mid Q_n\}_P$: Free(P) = $\bigcup_{i \in [1\dots n]}$ Free(Q_i) ;

Let $blend_P$ be a function that merges a set of sequences into a single sequence without repetitions (so providing an implementation of set union). Set

$\boldsymbol{args}_P = blend_P(\boldsymbol{args}_{Q_1}, \boldsymbol{args}_{Q_2}, \dots, \boldsymbol{args}_{Q_n})$;

$\forall i \bullet \forall d \in$ Free(Q_i) $\bullet\ Args_P(d) = Args_{Q_i}(d)$;

Rules(P) $\equiv$ { $\texttt{P[}\boldsymbol{args}_P\texttt{] =>}$

$\texttt{*Q}_1\texttt{[}\boldsymbol{args}_{Q_1}\texttt{],*Q}_2\texttt{[}\boldsymbol{args}_{Q_2}\texttt{],...,*Q}_n\texttt{[}\boldsymbol{args}_{Q_n}\texttt{]}$ }

$\{\Sigma_{i \in [1\dots n]}\, x_i(u_i).Q_i^R + \Sigma_{j \in [1\dots m]}\, \bar{z}_j w_j.Q_j^W\}_P$:

Free(P) = $\{x_1,\dots, x_n, z_1,\dots, z_m, w_1,\dots, w_m\}$

$\cup \bigcup_{i \in [1\dots n]} (\mathrm{Free}(Q_i^R) - \{u_i\}) \cup \bigcup_{j \in [1\dots m]} \mathrm{Free}(Q_j^W)$;

Assume a $blend_P$ function as above. Set

$\boldsymbol{args}_P = blend_P\,([x_1,\dots, x_n, z_1,\dots, z_m, w_1,\dots, w_m],$

$\boldsymbol{args}_{Q_1^R}, \dots, \boldsymbol{args}_{Q_n^R}, \boldsymbol{args}_{Q_1^W}, \dots, \boldsymbol{args}_{Q_m^W})$;

and

$\forall i \bullet \forall d \in \mathrm{Free}(Q_i^R) - \{u_i\} \bullet Args_P(d) = Args_{Q_i^R}(d)$;

$\forall j \bullet \forall d \in \mathrm{Free}(Q_j^W) \bullet Args_P(d) = Args_{Q_j^W}(d)$;

Rules(P) is given by firstly $\bigcup_{i \in [1\dots n]}$ of

$\texttt{P[}\boldsymbol{args}_P\texttt{] => \#P\_Q}_i^R\texttt{ [\^{}**Help\_r[x}_i\texttt{ u}_i\texttt{] }\boldsymbol{args}_P\texttt{ u}_i\texttt{:Empty]}$

$\texttt{P\_Q}_i^R\texttt{ [Yes }\boldsymbol{args}_P\texttt{ u}_i\texttt{] => *Q}_i^R\texttt{ [}\boldsymbol{args}_{Q_i^R}\texttt{]}$

(note that if $u_i \in \mathrm{Free}(Q_i^R)$ then $\texttt{u}_i$ occurs in $\boldsymbol{args}_{Q_i^R}$)

$\texttt{P\_Q}_i^R\texttt{ [No }\boldsymbol{args}_P\texttt{ u}_i\texttt{] => *P[}\boldsymbol{args}_P\texttt{]}$

$\texttt{P\_Q}_i^R\texttt{ [a }\boldsymbol{args}_P\texttt{ u}_i\texttt{] => *P\_Q}_i^R\texttt{ [a }\boldsymbol{args}_P\texttt{ u}_i\texttt{]}$

and secondly $\bigcup_{j \in [1\dots m]}$ of

$\texttt{P[}\boldsymbol{args}_P\texttt{] => \#P\_Q}_j^W\texttt{ [\^{}**Help\_w[z}_j\texttt{ w}_j\texttt{] }\boldsymbol{args}_P\texttt{]}$

$\texttt{P\_Q}_j^W\texttt{ [Yes }\boldsymbol{args}_P\texttt{] => *Q}_j^W\texttt{ [}\boldsymbol{args}_{Q_j^W}\texttt{]}$

$\texttt{P\_Q}_j^W\texttt{ [No }\boldsymbol{args}_P\texttt{] => *P[}\boldsymbol{args}_P\texttt{]}$

$\texttt{P\_Q}_j^W\texttt{ [a }\boldsymbol{args}_P\texttt{] => *P\_Q}_j^W\texttt{ [a }\boldsymbol{args}_P\texttt{]}$

The Communication Protocol

Rules for read helper: **`Help_r`** initiates an offer, no offers in progress; default rule.

```
Help_r[chan:Empty chan_in] =>
  *Help_r_test_chan[chan':Busy_Unlocked chan_in], chan:=chan'
Help_r[chan chan_in] => *No
```

Rules for write helper: **`Help_w`** sees a channel containing an offer, starts a rendezvous; default rule.

```
Help_w[chan:Busy_Unlocked chan_out] => *Yes,
  chan := chan':Busy_Locked[chan_out]
```

```
Help_w[chan chan_out] => *No
```

Rules for **Help_r_test_chan**: **Help_r_test_chan** waits a bit longer; revokes its own offer; detects a rendezvous; default rule.

```
Help_r_test_chan[chan:Busy_Unlocked chan_in] =>
  *Help_r_test_chan[chan chan_in]
Help_r_test_chan[chan:Busy_Unlocked chan_in] => *No,
  chan := chan':Empty
Help_r_test_chan[chan:Busy_Locked[data] chan_in] =>
  *Help_r_assign_data[chan data chan_in]
Help_r_test_chan[chan chan_in] =>
  *Help_r_test_chan[chan chan_in]
```

Rules for **Help_r_assign_data**. **Help_r_assign_data** assigns data to input; default rule.

```
Help_r_assign_data[chan data chan_in:Empty] =>
  *Help_r_unlock[chan], chan_in:=data
Help_r_assign_data[chan data chan_in] =>
  *Help_r_assign_data[chan data chan_in]
```

Rules for **Help_r_unlock**: **Help_r_unlock** unlocks chan and resets protocol; default rule.

```
Help_r_unlock[chan:Busy_Locked[data]] => *Yes,
  chan:=chan':Empty
Help_r_unlock[chan] => *Help_r_unlock[chan]
```