A Calculus for Interaction Nets

Maribel Fernández[1] and Ian Mackie[2]

[1] LIENS (CNRS UMR 8548), École Normale Supérieure
45 Rue d'Ulm, 75005 Paris, France
maribel@dmi.ens.fr
[2] CNRS-LIX (UMR 7650), École Polytechnique
91128 Palaiseau Cedex, France
mackie@lix.polytechnique.fr

Abstract. Interaction nets are graphical rewriting systems which can be used as either a high-level programming paradigm or a low-level implementation language. However, an operational semantics together with notions of strategy and normal form which are essential to reason about implementations, are not easy to formalize in this graphical framework. The purpose of this paper is to study a textual calculus for interaction nets, with a formal operational semantics, which provides a foundation for implementation. In addition, we are able to specify in this calculus various strategies, and a type system which formalizes the notion of partition used to define semi-simple nets. The resulting system can be seen as a kernel for a programming language, analogous to the λ-calculus.

1 Introduction

Interaction nets, introduced by Lafont [12], offer a graphical paradigm of computation based on net rewriting. They have proven themselves successful for application in computer science, most notably with the coding of the λ-calculus, where optimal reduction (specifically Lamping's algorithm [13]) has been achieved [9].

Although the graphical representation of interaction nets is very intuitive, graphical interfaces (editors) have not been forthcoming; a textual language is therefore necessary. Such a language would require ways of representing the interaction nets and rules, together with a reduction system to express how a rule should be applied. Lafont suggested in [12] a rather beautiful textual notation for interaction rules, but a general study of it has not emerged.

In this paper we define a calculus of interaction nets based on this notation. We provide the coding of interaction nets and rules, and a reduction system which can be seen as a decomposed system of interaction. Various notions of normal form and strategies for reduction can be formalized in this calculus, which provides the starting point for a more general treatment of abstract machines for implementing interaction nets. To enforce a discipline of programming, a type assignment system with user-defined types is introduced, which incorporates the notion of partition used by Danos and Regnier to generalize the multiplicative connectives of linear logic [5], and by Lafont to define semi-simple nets [12].

G. Nadathur (Ed.): PPDP'99, LNCS 1702, pp. 170–187, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Apart from the fact that it simplifies the actual writing of programs, a formal, textual, operational account of interaction nets has many advantages. Static properties of nets, such as types, can be defined in a more concise and formal way (compare the definition of the type system given in Sect. 3 and the definitions given in e.g. [6,12]). By giving a formal account of the rewriting process, the calculus provides the basis of an implementation of interaction nets. Interaction nets are strongly confluent, but like in all rewriting systems, there exist different notions of strategies and normal forms (for instance, irreducible nets, or weak normal forms associated to lazy reduction strategies). These are hard to formalize in a graphical framework, but we will see that they can be precisely defined in the calculus. Such strategies have applications for encodings of the λ-calculus, where interaction nets have had the greatest impact, and where a notion of a strategy is required to avoid non-termination of disconnected nets (see [16]).

Reduction algorithms and strategies also play a crucial rôle in the study of the operational semantics of interaction nets, in particular, operational equivalences of nets. Applications of this include [7] where the definition of a strategy was essential to show the correspondence between bisimilarity and contextual equivalence. In [7] an informal textual notation was used throughout the paper to help writing nets, but the formal definitions of strategy of evaluation and operational equivalence had to be given in the graphical framework. The calculus defined in this paper provides a formal and uniform notation for writing nets and defining their properties.

Related Work. Banach [3] showed that interaction nets are closely related to connection graphs of Bawden [4] and gave a formal account of these formalisms via hypergraph rewriting, using a categorical approach. Honda and Yoshida [10,18,19] studied various graphical and textual process calculi that generalize interaction nets; their emphasis is in the study of concurrent computations. The Interaction Systems of Laneve [14] are a class of combinatory reduction systems closely related to interaction nets (the intuitionistic nets). Strategies have been well studied in this framework, in particular for optimal reduction. Related to this work is also the encoding of interaction nets as combinatory reduction systems given in [8]. The notations that we introduce in the present work are inspired by formalisms for cyclic rewriting [2], and proof expressions for linear logic [1].

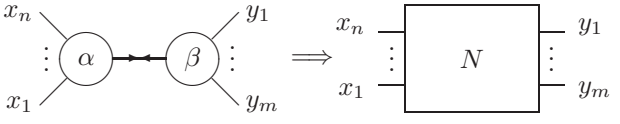
Overview. The rest of this paper is structured as follows: In the next section we recall some basic preliminaries on interaction nets and present several textual languages for interaction nets. Section 3 gives a thorough study of the calculus and presents the type system. Section 4 shows how strategies and reduction algorithms can be easily expressed in this framework. Finally, we conclude the paper in Section 5.

2 Background

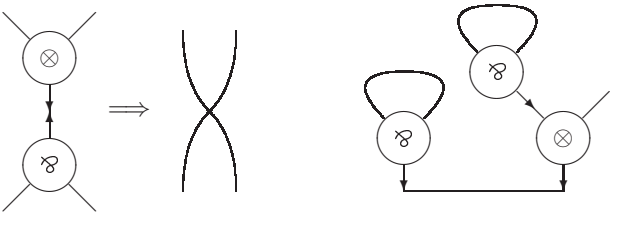
An interaction net system is specified by a set Σ of symbols, and a set $\mathcal{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occur-

rence of a symbol $\alpha \in \Sigma$ will be called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a distinguished one called the *principal port* depicted by an arrow, and n *auxiliary ports* corresponding to the arity of the symbol. We will say that the agent has $n+1$ *free* ports.

A *net* N built on Σ is a graph (not necessarily connected) with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge at every port (edges may connect two ports of the same agent). A net may also have edges with free extremes, called *wires*, and their extremes are called ports by analogy. The *interface* of a net is the set of free ports it has. There are two special instances of a net: a wiring (no agents), and the empty net. A pair of agents $(\alpha, \beta) \in \Sigma^2$ connected together on their principal ports is called an *active pair*; the interaction net analogue of a redex. An *interaction rule* $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ replaces an occurrence of the active pair (α, β) by the net N. Rules have to satisfy two very strong conditions: the interface must be preserved, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.



As a running example, we use the system of interaction for proof nets of multiplicative linear logic, which consists of two agents, $\parr$ and $\otimes$ of arity 2, and one interaction rule. The following diagram indicates the interaction rule and an example net.



Three textual languages for interaction nets have been proposed in the literature. Inspired by Combinatory Reduction Systems [11], a net can be "flattened" by replacing ports of agents by names, and representing edges by two occurrences of a name. The expression $\otimes(a,b,c), \parr(a,d,d), \parr(b,e,e)$ represents the example net above, where the first argument of each agent is the principal port. The interaction rule is written: $\otimes(a,b,c), \parr(a,d,e) \Longrightarrow I(c,e), I(b,d)$ where $I(a,b)$ is a wire with extremes a, b. This language has been used as a notation for interaction nets in [7,8], and is quite straightforward to relate to the graphical notation.

A second textual notation [15] eliminates the variables. The ports of each agent are indexed $\alpha.i$, with the principal port given by $\alpha.0$. A wiring relation is used to express the connectivity of the ports of agents in a net: $\alpha.i \equiv \beta.j$ indicates that there is a link between port i of agent α and port j of agent β. For example,

$(\parr_i, \otimes_j) \longrightarrow (\emptyset, \{\parr_i.1 \equiv \otimes_j.1, \parr_i.2 \equiv \otimes_j.2\})$ represents the interaction rule for proof nets. The left-hand side denotes an active pair and the right-hand side is a set of agents (empty in this example) together with a wiring.

Lafont [12] proposed a textual notation for interaction rules, where the example rule is written as $\otimes(x, y) \bowtie \parr(x, y)$. This notation is much lighter syntactically, but more esoteric. It seems the most useful as a formal language for interaction nets, although some practice is needed to become acquainted with it. This notation inspired the calculus that we develop in the rest of the paper.

3 The Calculus

We begin by introducing a number of syntactic categories.

Agents: Let Σ be a set of symbols, ranged over by $\alpha, \beta, \ldots$, each with a given *arity* $\mathsf{ar} : \Sigma \to \mathbb{N}$. An occurrence of a symbol will be called an *agent*. The arity of a symbol corresponds precisely to the number of auxiliary ports.

Names: Let N be a set of names, ranged over by x, y, z, etc. N and Σ are assumed disjoint.

Terms: A term is built on Σ and N by the grammar: $t ::= x \mid \alpha(t_1, \ldots, t_n)$, where $x \in N$, $\alpha \in \Sigma$, $\mathsf{ar}(\alpha) = n$ and $t_1, \ldots, t_n$ are terms, with the restriction that each name can appear at most twice. If $n = 0$, then we omit the parentheses. If a name occurs twice in a term, we say that it is *bound*, otherwise it is *free*. Since free names occur exactly once, we say that terms are *linear*. We write $\vec{t}$ for a list of terms $t_1, \ldots, t_n$. A term of the form $\alpha(\vec{t})$ can be seen as a tree with the principal port of α at the root, and where the terms $t_1, \ldots, t_n$ are the subtrees connected to the auxiliary ports of α.

Equations: If t and u are terms, then the (unordered) pair $t = u$ is an *equation*. Δ, $\Theta, \ldots$ will be used to range over multisets of equations. Examples of equations include: $x = \alpha(\vec{t})$, $x = y$, $\alpha(\vec{t}) = \beta(\vec{u})$.

Rules: Rules are pairs of terms written as $\alpha(\vec{t}) \bowtie \beta(\vec{u})$, where $(\alpha, \beta) \in \Sigma^2$ is the active pair of the rule. All names occur exactly twice in a rule, and there is at most one rule for each pair of agents.

Definition 1 (Names in terms). *The set $\mathcal{N}(t)$ of names of a term t is defined in the following way, which extends to multisets of equations and rules in the obvious way.*

$$\begin{aligned} \mathcal{N}(x) &= \{x\} \\ \mathcal{N}(\alpha(t_1, \ldots, t_n)) &= \mathcal{N}(t_1) \cup \cdots \cup \mathcal{N}(t_n) \end{aligned}$$

Given a term, we can replace its free names by new names, provided the linearity restriction is preserved.

Definition 2 (Renaming). *The notation $t[y/x]$ denotes a renaming that replaces the free occurrence of x in t by a new name y. Remark that since the name x occurs exactly once in the term, this operation can be implemented directly as an assignment, as is standard in the linear case. This notion extends to equations, and multisets of equations in the obvious way.*

More generally, we consider substitutions that replace free names in a term by other terms, always assuming that the linearity restriction is preserved.

Definition 3 (Substitution). *The notation $t[u/x]$ denotes a substitution that replaces the free occurrence of x by the term u in t. We only consider substitutions that preserve the linearity of the terms. Note that renaming is a particular case of substitution.*

Lemma 1. *Assume that $x \notin \mathcal{N}(v)$. If $y \in \mathcal{N}(u)$ then $t[u/x][v/y] = t[u[v/y]/x]$, otherwise $t[u/x][v/y] = t[v/y][u/x]$.*

Definition 4 (Instance of a rule). *If r is a rule $\alpha(t_1, \ldots, t_n) \bowtie \beta(u_1, \ldots, u_m)$, then $\hat{r} = \alpha(\widehat{t_1}, \ldots, \widehat{t_n}) \bowtie \beta(\widehat{u_1}, \ldots, \widehat{u_m})$ denotes a new generic* instance *of r, that is, a copy of r where we introduce a new set of names.*

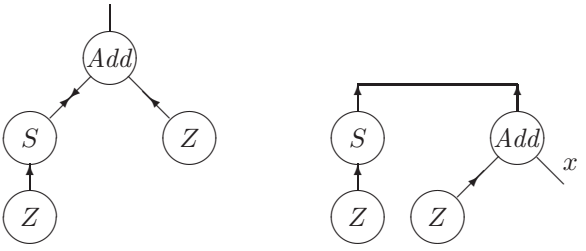
We now have all the machinery that we need to define interaction nets.

Definition 5 (Configurations). *A* configuration *is a pair: $c = (\mathcal{R}, \langle \mathbf{t} \mid \Delta \rangle)$, where $\mathcal{R}$ is a set of rules, $\mathbf{t}$ a multiset $\{t_1, \ldots, t_n\}$ of terms, and Δ a multiset of equations. Each variable occurs at most twice in c. If a name occurs once in c then it is* free*, otherwise it is* bound*. For simplicity we sometimes omit $\mathcal{R}$ when there is no ambiguity. We use c, c' to range over configurations. We call $\mathbf{t}$ the* head *and Δ the* body *of a configuration.*

Intuitively, $\langle \mathbf{t} \mid \Delta \rangle$ represents a net that we evaluate using $\mathcal{R}$; Δ gives the set of active pairs and the renamings of the net. The roots of the terms in the head of the configuration and the free names correspond to ports in the interface of the net. We work modulo α-equivalence for bound names as usual, but also for free names. Configurations that differ only on the names of the free variables are equivalent, since they represent the same net.

Example 1 (Configurations). The empty net is represented by $\langle \mid \rangle$, and the configuration $\langle \mid \parr(a, a) = \otimes(b, b) \rangle$ represents a net without an interface, containing an active pair. A configuration $\langle \mathbf{t} \mid \rangle$ represents a net without active pairs and without cycles of principal ports.

There is an obvious (although not unique) translation between the graphical representation of interaction nets, and the configurations that we are using. Briefly, to translate a net into a configuration, we first orient the net as a collection of trees, with all principal ports facing in the same direction. Each pair of trees connected at their principal ports is translated as an equation, and any tree whose root is free or any free port of the net goes in the head of the configuration. We give a simple example to explain this translation. The usual encoding of the addition of natural numbers uses the agents $\Sigma = \{Z, S, Add\}$, $\mathsf{ar}(Z) = 0$, $\mathsf{ar}(S) = 1$, $\mathsf{ar}(Add) = 2$. The diagrams below illustrate the net representing the addition $1 + 0$ in the "usual" orientation, and also with all the principal ports facing up.



We then obtain the configuration $\langle x \mid S(Z) = Add(x, Z)\rangle$ where the only port in the interface is x, which we put in the head of the configuration.

The reverse translation simply requires that we draw the trees for the terms, connect the common variables together, and connect the trees corresponding to the members of an equation together on their principal ports.

Remark 1. The multiset $\mathbf{t}$ is the *interface of the configuration*, and this can be all, or just part of, the interface of the net represented by $\langle \mathbf{t} \mid \Delta\rangle$. For example, the net consisting of the agents *Succ* and *True* can be represented by $\langle True \mid y = Succ(x)\rangle$, $\langle True, Succ(x) \mid \rangle$, or $\langle \mid x = True, y = Succ(z)\rangle$. In other words, we can select a part of the interface of the net to be displayed as "observable" interface; the same net can be represented by several different configurations. See [7] for an example of use of observable interface.

Definition 6 (Computation Rules). *The operational behaviour of the system is given by a set of four computation rules.*

Indirection: *If* $x \in \mathcal{N}(u)$*, then* $x = t, u = v \longrightarrow u[t/x] = v$.
Interaction: *If* $\alpha(t'_1, \ldots, t'_n) \bowtie \beta(u'_1, \ldots, u'_m) \in \mathcal{R}$*, then*

$$\alpha(t_1, \ldots, t_n) = \beta(u_1, \ldots, u_m) \longrightarrow \\ t_1 = \widehat{t'_1}, \ldots, t_n = \widehat{t'_n}, u_1 = \widehat{u'_1}, \ldots, u_m = \widehat{u'_m}$$

Context: *If* $\Delta \longrightarrow \Delta'$*, then* $\langle \mathbf{t} \mid \Gamma, \Delta, \Gamma'\rangle \longrightarrow \langle \mathbf{t} \mid \Gamma, \Delta', \Gamma'\rangle$.
Collect: *If* $x \in \mathcal{N}(\mathbf{t})$*, then* $\langle \mathbf{t} \mid x = u, \Delta\rangle \longrightarrow \langle \mathbf{t}[u/x] \mid \Delta\rangle$.

The calculus puts in evidence the real cost of implementing an interaction step, which involves generating an instance, i.e. a new copy, of the right-hand side of the rule, plus renamings (rewirings). Of course this also has to be done when working in the graphical framework, even though it is often seen as an atomic step. The calculus therefore shows explicitly how an interaction step is performed.

Example 2 (Natural Numbers). We show two different encodings of natural numbers. The first is the standard one, the second is a more efficient version which offers a constant time addition operation.

1. Let $\Sigma = \{Z, S, Add\}$ with $\mathsf{ar}(Z) = 0$, $\mathsf{ar}(S) = 1$, $\mathsf{ar}(Add) = 2$, and $\mathcal{R}$:

$$\begin{array}{ll} Add(S(x), y) & \bowtie S(Add(x, y)) \\ Add(x, x) & \bowtie Z \end{array}$$

   As shown previously, the net for 1+0 is given by the configuration $(\mathcal{R}, \langle a \mid Add(a, Z) = S(Z)\rangle)$. One possible sequence of rewrites for this net is the following:

$$\begin{array}{l} \langle a \mid Add(a, Z) = S(Z)\rangle \\ \quad \longrightarrow \langle a \mid a = S(x'), y' = Z, Z = Add(x', y')\rangle \\ \quad \longrightarrow^* \langle S(x') \mid Z = Add(x', Z)\rangle \\ \quad \longrightarrow \langle S(x') \mid x'' = x', x'' = Z\rangle \\ \quad \longrightarrow^* \langle S(Z) \mid \rangle \end{array}$$

2. Let $\Sigma = \{S, N, N^*\}$, $\mathsf{ar}(S) = 1$, $\mathsf{ar}(N) = \mathsf{ar}(N^*) = 2$. Numbers are represented as a list of S agents, where N is the constructor holding a pointer to the head and tail of the list. 0 is defined by $N(x, x)$, and n by $N(S^n(x), x)$. The operation of addition can then be encoded by the configuration

$$\langle N(b, c), N^*(a, b), N^*(c, a) \mid \rangle$$

   which simply appends two numbers. There is only one rule that we need: $N(a, b) \bowtie N^*(b, a)$, which is clearly a constant time operation. To show how this works, we give an example of the addition of 1+1:

$$\begin{array}{l} \langle N(b, c) \mid N(S(x), x) = N^*(a, b), N(S(y), y) = N^*(c, a)\rangle \\ \quad \longrightarrow^* \langle N(b, c) \mid b = S(a), a = S(c)\rangle \\ \quad \longrightarrow^* \langle N(S(S(c)), c) \mid \rangle \end{array}$$

Example 3 (Proof Nets for Multiplicative Linear Logic). The usual encoding of proof nets in interaction nets uses $\Sigma = \{\otimes, \parr\}$, $\mathsf{ar}(\otimes) = \mathsf{ar}(\parr) = 2$, and the interaction rule: $\otimes(x, y) \bowtie \parr(x, y)$. We show a configuration representing the net used as a running example in the previous section, and a reduction sequence:

$$\begin{array}{l} \langle c \mid \parr(a, a) = \otimes(\parr(b, b), c)\rangle \\ \quad \longrightarrow \langle c \mid x_1 = a, y_1 = a, x_1 = \parr(b, b), y_1 = c\rangle \\ \quad \longrightarrow^* \langle c \mid c = \parr(b, b)\rangle \\ \quad \longrightarrow \langle \parr(b, b) \mid \rangle \end{array}$$

The next example shows how we can capture semi-simple nets [12], where vicious circles of principal ports cannot be created during computation. The notion of partition was used in [12] with the purpose of defining this class of nets. For each agent $\alpha \in \Sigma$ the (possibly empty) set of auxiliary ports is divided into one or several classes, each of them called *a partition*. A *partition mapping* establishes, for each agent in Σ, the way its auxiliary ports are grouped into partitions. We recall the graphical definition of semi-simple nets and at the same time show the translation to configurations.

Example 4 (Semi-simple Nets). Semi-simple nets are inductively defined as:

1. The empty net is semi-simple, and is represented by the configuration $\langle \mid \rangle$.
2. A wire is a semi-simple net, and is represented by a configuration of the form $\langle y, y \mid \rangle$.
3. If N_1, N_2 are semi-simple nets, their juxtaposition (called Mix) gives a new semi-simple net which is represented by the configuration $\langle \mathbf{s}, \mathbf{t} \mid \Delta, \Theta \rangle$, where $\langle \mathbf{t} \mid \Delta \rangle$ and $\langle \mathbf{s} \mid \Theta \rangle$ are the configurations representing N_1 and N_2 respectively (without loss of generality we assume that these configurations do not share variables, since we work modulo α-conversion).
4. If N_1, N_2 are semi-simple nets, a Cut between ports i and j builds a semi-simple net represented by the configuration $\langle \mathbf{s} - s_i, \mathbf{t} - t_j \mid \Delta, \Theta, s_i = t_j \rangle$, where $N_1 = \langle \mathbf{s} \mid \Delta \rangle$ and $N_2 = \langle \mathbf{t} \mid \Theta \rangle$, $\mathbf{s} - s_i$ is the multiset $\mathbf{s}$ without the element s_i, and $\mathbf{t} - t_j$ is the multiset $\mathbf{t}$ without the element t_j.
5. If $N_1, \ldots, N_n$ are semi-simple nets, the Graft of an agent α to the nets $N_1, \ldots, N_n$ according to its partitions builds a semi-simple net represented by $\langle \alpha(\vec{s_1}, \ldots, \vec{s_n}), \mathbf{t}_1, \ldots, \mathbf{t}_n \mid \Delta_1, \ldots, \Delta_n \rangle$, where $N_i = \langle \mathbf{s}_i, \mathbf{t}_i \mid \Delta_i \rangle$, $1 \leq i \leq n$, and we assume without loss of generality that these configurations do not share variables.

Example 5 (Non-termination). Consider the net $\langle x, y \mid \alpha(x) = \beta(\alpha(y)) \rangle$ and the rule $\alpha(a) \bowtie \beta(\beta(\alpha(a)))$. The the following non-terminating reduction sequence is possible: $\langle x, y \mid \alpha(x) = \beta(\alpha(y)) \rangle \longrightarrow \langle x, y \mid x = a, \beta(\alpha(a)) = \alpha(y) \rangle \longrightarrow \langle a, y \mid \beta(\alpha(a)) = \alpha(y) \rangle \longrightarrow \cdots$.

There is an obvious question to ask about this language with respect to the graphical formalism: can we write all interaction rules? Under some assumptions, the answer is yes. There are in fact two restrictions. The first one is that there is no way of writing a rule with an active pair in the right-hand side. This is not a problem since interaction nets can be assumed to satisfy the optimization condition [12], which requires no active pairs in right-hand sides. The second problem is the representation of interaction rules for active pairs without interface. In the calculus, an active pair without interface can only rewrite to the empty net. In other words, it will be erased, and so can be ignored. This coincides with the operational semantics defined in [7].

3.1 A Type Discipline

We now define a typed version of the calculus using type variables $\varphi_1, \varphi_2, \ldots$, and type constructors with fixed arities (such as **int**, **bool**, ... of arity 0; $\mathsf{list}, \ldots$ of arity 1; $\times$, $\otimes$, $\parr$, ...of arity 2; ...). The terms built on this signature ($\tau = \mathsf{list}(\mathbf{int}), \mathsf{list}(\mathsf{list}(\mathbf{bool})), \otimes(\varphi_1, \varphi_2), \ldots$) are the types of the system, and they are used-defined in the same way as the set Σ of agents is. Note that we may have type constructors with the same names as agents, as in the case of $\otimes$ which is traditionally used as a type constructor and as an agent in proof nets.

Types may be decorated with *signs*: τ^+ will be called an *output type*, and τ^- an *input type*. The *dual* of an atomic type σ^+ (resp. σ^-) is σ^- (resp. σ^+), and in general we will denote by $\sigma^\perp$ the dual of the type σ, which is defined by a set of type equations of the form $C(\sigma_1,\ldots,\sigma_n)^\perp = C^\perp(\sigma_1^\perp,\ldots,\sigma_n^\perp)$ such that $(\sigma^\perp)^\perp = \sigma$. For example

$$\begin{array}{l}(\mathsf{list}^-(\sigma))^\perp = \mathsf{list}^+(\sigma^\perp)\\ (\mathsf{list}^+(\sigma))^\perp = \mathsf{list}^-(\sigma^\perp)\\ (\otimes(\sigma,\tau))^\perp = \parr(\sigma^\perp,\tau^\perp)\\ (\parr(\sigma,\tau))^\perp = \otimes(\sigma^\perp,\tau^\perp)\end{array}$$

Moreover, we might have other equations defining equalities between types, such as: $(\sigma\times\tau)\times\rho = \sigma\times(\tau\times\rho)$.

Types will be assigned to the terms of the calculus by using the following inference rules, which are a form of one-sided sequents.

Identity Group. The Axiom allows variables of dual types to be introduced to the system, and the Cut rule says that an equation is typeable (denoted $t = u:\diamond$) if both sides are typeable with dual types.

$$\frac{}{x:\sigma, x:\sigma^\perp}\,(\mathsf{Ax}) \qquad \frac{\Gamma, t:\sigma \quad \Delta, u:\sigma^\perp}{\Gamma,\Delta, t = u:\diamond}\,(\mathsf{Cut})$$

Structural Group. The Exchange rule allows permutations on the order of the sequent, and the Mix rule allows the combination of two sequents:

$$\frac{\Gamma, t:\sigma, u:\tau, \Delta}{\Gamma, u:\tau, t:\sigma, \Delta}\,(\mathsf{Exchange}) \qquad \frac{\Gamma \quad \Delta}{\Gamma,\Delta}\,(\mathsf{Mix})$$

User-defined Group. For each agent $\alpha\in\Sigma$, there is a rule that specifies the way types are assigned to terms rooted by α. The general format is:

$$\frac{\Gamma_1, t_1:\sigma_1,\ldots,t_{i_1}:\sigma_{i_1} \quad \ldots \quad \Gamma_k, t_k:\sigma_k,\ldots,t_{i_k}:\sigma_{i_k}}{\Gamma_1,\ldots,\Gamma_k,\alpha(t_1,\ldots,t_n):\sigma}\,(\mathsf{Graft})$$

The Graft rule for α specifies its partitions, that is, the way the subterms $t_1,\ldots,t_n$ are distributed in the premises. For example, a set of Graft rules for the system defined in Example 2, Part 1 (arithmetic), together with a polymorphic erasing agent ϵ defined by rules of the form $\epsilon \bowtie \alpha(\epsilon,\ldots,\epsilon)$, can be defined as follows:

$$\frac{\Gamma, t:\mathbf{int}^+}{\Gamma, S(t):\mathbf{int}^+}\,(S) \quad \frac{\Gamma}{\Gamma, Z:\mathbf{int}^+}\,(Z) \quad \frac{\Gamma, t_1:\mathbf{int}^- \quad \Delta, t_2:\mathbf{int}^+}{\Gamma,\Delta, Add(t_1,t_2):\mathbf{int}^-}\,(Add) \quad \frac{\Gamma}{\Gamma,\epsilon:\sigma}\,(\epsilon)$$

Definition 7 (Typeable Configurations). *Let $\{x_1,\ldots,x_m\}$ be the set of free names of t, then t is a term of type σ if there is a derivation ending in*

$x_1 : \tau_1, \ldots, x_m : \tau_m, t : \sigma$. *Equations are typed in a similar way: if* $t = u$ *is an equation with free names* $\{x_1, \ldots, x_m\}$, *then it is typeable if there is a derivation for* $x_1 : \tau_1, \ldots, x_m : \tau_m, t = u : \diamond$. *This notion can be extended to multisets of equations in a straightforward way. A configuration* $\langle \mathbf{t} \mid s_1 = u_1, \ldots, s_m = u_m \rangle$ *with free names* $x_1, \ldots, x_p$ *is typeable by* $\sigma_1, \ldots, \sigma_n$, *if there is a derivation for* $x_1 : \rho_1, \ldots, x_p : \rho_p, t_1 : \sigma_1, \ldots, t_n : \sigma_n, s_1 = u_1 : \diamond, \ldots, s_m = u_m : \diamond$.

Example 6. The following is a type derivation for the net $\langle a \mid Add(a, Z) = S(Z) \rangle$ in Example 2, Part 1, using the set of Graft rules defined above.

$$\dfrac{\dfrac{\dfrac{}{a : \mathbf{int}^+, a : \mathbf{int}^-}\,(\mathsf{Ax}) \qquad \dfrac{}{Z : \mathbf{int}^+}\,(Z)}{a : \mathbf{int}^+, Add(a, Z) : \mathbf{int}^-}\,(Add) \qquad \dfrac{\dfrac{}{Z : \mathbf{int}^+}\,(Z)}{S(Z) : \mathbf{int}^+}\,(S)}{a : \mathbf{int}^+, Add(a, Z) = S(Z) : \diamond}\,(\mathsf{Cut})$$

The typing of rules is more delicate, since we have to ensure that both sides are consistent with respect to types so that the application of the Interaction Rule preserves types (Subject Reduction). We do this in two steps: first we find a type derivation for the active pair (using arbitrary names for the free variables), and then use this derivation as a template to build a derivation for the right-hand side.

Definition 8 (Typeable Rules). *Let* Σ *be a given set of agents, with a corresponding set of Graft rules. We say that a rule* $\alpha(t_1, \ldots, t_n) \bowtie \beta(s_1, \ldots, s_m)$ *is typeable if*

1. *there is a derivation* D *with conclusion* $\alpha(x_1, \ldots, x_n) = \beta(y_1, \ldots, y_m) : \diamond$ *and leaves containing assumptions for* $x_1, \ldots, x_n, y_1, \ldots, y_m$, *built by application of the Cut rule and the Graft rules for* α, β,
2. *there is a type derivation with the same assumptions leading to the conclusion* $x_1 = t_1 : \diamond, \ldots, x_n = t_n : \diamond, y_1 = s_1 : \diamond, \ldots, y_m = s_m : \diamond$,
3. *and whenever an equation* $\alpha(t'_1, \ldots, t'_n) = \beta(s'_1, \ldots, s'_m)$ *is typeable, its type derivation is obtained by using the Cut rule and instances of the Graft rules for* α, β *applied in* D.

Example 7. The interaction rules for addition given in Example 2, Part 1, are typeable. We show the type derivation for $Add(S(x), y) \bowtie S(Add(x, y))$. First we build the most general derivation for the active pair (to ensure that condition 3 in Definition 8 holds):

$$\dfrac{\dfrac{x_1 : \mathbf{int}^- \quad x_2 : \mathbf{int}^+}{Add(x_1, x_2) : \mathbf{int}^-}\,(Add) \qquad \dfrac{x_3 : \mathbf{int}^+}{S(x_3) : \mathbf{int}^+}\,(S)}{Add(x_1, x_2) = S(x_3) : \diamond}\,(\mathsf{Cut})$$

Then we use this template to build a second derivation:

$$\dfrac{x_1\colon \mathbf{int}^- \qquad \dfrac{x_2\colon \mathbf{int}^+ \qquad \dfrac{x_3\colon \mathbf{int}^+ \qquad \dfrac{\dfrac{\dfrac{}{x\colon \mathbf{int}^-, x\colon \mathbf{int}^+}\,(\mathsf{Ax})}{x\colon \mathbf{int}^-, S(x)\colon \mathbf{int}^+}\,(S) \qquad \dfrac{}{y\colon \mathbf{int}^-, y\colon \mathbf{int}^+}\,(\mathsf{Ax})}{y\colon \mathbf{int}^-, S(x)\colon \mathbf{int}^+, Add(x,y)\colon \mathbf{int}^-}\,(Add)}{y\colon \mathbf{int}^-, S(x)\colon \mathbf{int}^+, x_3 = Add(x,y)\colon \diamond}\,(\mathsf{Cut})}{S(x)\colon \mathbf{int}^+, x_2 = y\colon \diamond, x_3 = Add(x,y)\colon \diamond}\,(\mathsf{Cut})}{x_1 = S(x)\colon \diamond, x_2 = y\colon \diamond, x_3 = Add(x,y)\colon \diamond}\,(\mathsf{Cut})$$

Consider now the encoding of proof nets (Example 3), with the interaction rule $\otimes(a,b) \bowtie \parr(a,b)$. We will show that a cut-elimination step in linear logic proof nets is typeable. Here is the most general derivation for the active pair:

$$\dfrac{\dfrac{x_1\colon \sigma_1 \quad x_2\colon \sigma_2}{\otimes(x_1,x_2)\colon \otimes(\sigma_1,\sigma_2)}\,(\otimes) \qquad \dfrac{x_3\colon \sigma_1^\perp, x_4\colon \sigma_2^\perp}{\parr(x_3,x_4)\colon \parr(\sigma_1^\perp,\sigma_2^\perp)}\,(\parr)}{\otimes(x_1,x_2) = \parr(x_3,x_4)\colon \diamond}\,(\mathsf{Cut})$$

We build now the second derivation using this template:

$$\dfrac{x_1\colon \sigma_1 \qquad \dfrac{x_2\colon \sigma_2 \qquad \dfrac{\dfrac{}{a\colon \sigma_1^\perp, a\colon \sigma_1}\,(\mathsf{Ax}) \qquad \dfrac{\dfrac{}{b\colon \sigma_2^\perp, b\colon \sigma_2}\,(\mathsf{Ax}) \qquad x_3\colon \sigma_1^\perp, x_4\colon \sigma_2^\perp}{b\colon \sigma_2^\perp, x_3\colon \sigma_1^\perp, x_4 = b\colon \diamond}\,(\mathsf{Cut})}{a\colon \sigma_1^\perp, b\colon \sigma_2^\perp, x_3 = a\colon \diamond, x_4 = b\colon \diamond}\,(\mathsf{Cut})}{a\colon \sigma_1^\perp, x_2 = b\colon \diamond, x_3 = a\colon \diamond, x_4 = b\colon \diamond}\,(\mathsf{Cut})}{x_1 = a\colon \diamond, x_2 = b\colon \diamond, x_3 = a\colon \diamond, x_4 = b\colon \diamond}\,(\mathsf{Cut})$$

The last condition in Definition 8 is crucial for Subject Reduction, as the following example shows.

Example 8 (Untypeable Rule). Let $\Sigma = \{\alpha, S\}$ where $\mathsf{ar}(\alpha) = \mathsf{ar}(S) = 1$, together with the interaction rule $\alpha(x) \bowtie S(x)$. The agents are typed with the Graft rules:

$$\dfrac{\Gamma, t\colon \mathbf{int}^+}{\Gamma, S(t)\colon \mathbf{int}^+}\,(S) \qquad \dfrac{\Gamma, t\colon \sigma}{\Gamma, \alpha(t)\colon \mathbf{int}^-}\,(\alpha)$$

We first build a type derivation for the active pair:

$$\dfrac{\dfrac{x_1\colon \mathbf{int}^-}{\alpha(x_1)\colon \mathbf{int}^-}\,(\alpha) \qquad \dfrac{y_1\colon \mathbf{int}^+}{S(y_1)\colon \mathbf{int}^+}\,(S)}{\alpha(x_1) = S(y_1)\colon \diamond}\,(\mathsf{Cut})$$

With this template we can build a type derivation for the right-hand side:

$$\dfrac{x_1 : \mathbf{int}^- \qquad \dfrac{\dfrac{}{x : \mathbf{int}^+, x : \mathbf{int}^-}\,(\mathsf{Ax}) \qquad y_1 : \mathbf{int}^+}{x : \mathbf{int}^+, y_1 = x : \diamond}\,(\mathsf{Cut})}{x_1 = x : \diamond, y_1 = x : \diamond}\,(\mathsf{Cut})$$

But the equation $\alpha(x') = S(y')$ is also typeable by:

$$\dfrac{\dfrac{x' : \mathbf{bool}^-}{\alpha(x') : \mathbf{int}^-}\,(\alpha) \qquad \dfrac{y' : \mathbf{int}^+}{S(y') : \mathbf{int}^+}\,(S)}{\alpha(x') = S(y') : \diamond}\,(\mathsf{Cut})$$

which is not an instance of the first derivation. This means that condition 3 in Definition 8 is not satisfied, therefore the rule is not typeable. If we accept this rule, we obtain a system that is not sound with respect to types: the net $\langle \mid \alpha(\mathit{True}) = S(Z) \rangle$ where True has type $\mathbf{bool}$ and Z $\mathbf{int}$, is typeable, but reduces to $\langle \mid \mathit{True} = Z \rangle$ which is not typeable.

Theorem 1 (Subject Reduction). *The computation rules Indirection, Interaction, Context and Collect, preserve typeability and types: For any set $\mathcal{R}$ of typeable rules and configuration c on Σ, if $c \to^* c'$ then c' can be assigned the same types as c.*

Proof. The cases of Indirection with Context and Collect are straightforward, we show the case of Interaction. Let

$$c = \langle \mathbf{t} \mid \Gamma, \alpha(\vec{u}) = \beta(\vec{v}), \Gamma' \rangle \to c' = \langle \mathbf{t} \mid \Gamma, \overrightarrow{u = \widehat{u'}}, \overrightarrow{v = \widehat{v'}}, \Gamma' \rangle$$

using the interaction rule $\alpha(\vec{u'}) \bowtie \beta(\vec{v'})$. Assume that c is typeable (more precisely, there is a derivation for the sequent $\Delta, \alpha(\vec{u}) = \beta(\vec{v}) : \diamond$ corresponding to c), and that the rule $\alpha(\vec{u'}) \bowtie \beta(\vec{v'})$ is typeable. We show that the sequent $\Delta, \overrightarrow{u = \widehat{u'}} : \diamond, \overrightarrow{v = \widehat{v'}} : \diamond$ corresponding to c' is derivable. By Definition 8 (part 2), there is a proof tree for $\overrightarrow{x = \widehat{u'}} : \diamond, \overrightarrow{y = \widehat{v'}} : \diamond$ using as template the proof tree of $\alpha(\vec{x}) = \beta(\vec{y}) : \diamond$. By Definition 8 (part 3), the (α) and (β) typing rules used in the proof of c are instances (say with a substitution S) of the ones used in the proof of $\alpha(\vec{x}) = \beta(\vec{y}) : \diamond$. Hence, we can build the derivation tree for the sequent associated to c' by using the instance (by substitution S) of the proof tree of $\overrightarrow{x = \widehat{u'}} : \diamond, \overrightarrow{y = \widehat{v'}} : \diamond$, replacing x_i, y_i by u_i, v_i, and replacing the leaves containing assumptions for x_i, y_i by the corresponding proofs of u_i, v_i (subtrees of the proof of c). □

We remark that the notion of partition is built-in in our type system: the partitions of an agent correspond to the hypotheses in its Graft rule. This means that our type system can be used to check semi-simplicity of nets.

Proposition 1. *Typeable configurations are semi-simple (without vicious circles of principal ports).*

Proof. Induction on the type derivation. Note that the names given to the type rules of our system coincide with the names given to the operations that build semi-simple nets (cf. Example 4). □

Related systems. Danos and Regnier [5] generalized the multiplicative connectives of linear logic, showing the general format of the introduction rule (for a connective and its dual), and the cut-elimination step. Each connective corresponds to an agent and a type constructor in our system, and their rules coincide with our Graft rules. The cut-elimination steps are defined through interaction rules in our system (and are not necessarily rewirings, as in the case of multiplicatives).

Lafont [12] introduced a basic type discipline for the graphical framework of interaction nets, using a set of constant types ($\tau \in \{\mathsf{atom}, \mathsf{nat}, \mathsf{list}, \ldots\}$). For each agent, ports are classified between *input* and *output*: input ports have types of the form τ^-, whereas output ports have types τ^+. A net is *well-typed* in Lafont's system if each agent is used with the correct type and input ports are connected to output ports of the same type. Our system is a generalization of Lafont's system (since we introduce polymorphism and a more general notion of duality) integrating the notion of partition. It is easy to see that the typed nets of Lafont are represented by configurations which are typeable in our calculus (the proof is by a straightforward induction on the structure of the configuration).

In [6] another type system for interaction nets is discussed, using type variables and intersection types. The intersection free part of this system is also a subsystem of ours, but we consider also polymorphic type constructors and build-in the notion of partition.

3.2 Properties of the Calculus

This section is devoted to showing various properties of the rewriting system defined by the rules Indirection, Interaction, Context, and Collect. These results are known for the graphical formalism of interaction nets, but here we show that the calculus, which is a decomposed system of interaction, also preserves these properties.

Proposition 2 (Confluence). $\longrightarrow$ *is strongly confluent.*

Proof. All the critical pairs are joinable in one step, using Lemma 1. □

We write $c \Downarrow c'$ iff $c \to c' \not\to$. As an immediate consequence of the previous property we obtain:

Proposition 3 (Determinacy). $c \Downarrow c'$ *and* $c \Downarrow c''$ *implies* $c' = c''$.

It is a rather interesting phenomenon that interaction nets representing infinite loops (infinite computations) can terminate. This is analogous to "black hole" detection, which is well-known in graph reduction for functional languages. Two examples of this are the following configurations: $\langle y \mid x = \delta(x, y) \rangle$, where δ is the duplicator agent, and $\langle \mid x = x \rangle$, which both can be thought of as representing the cyclic term $\alpha = \alpha$ in cyclic term rewriting [2]. Both of these configurations are irreducible. The latter is a net without an interface, and the first is the same thing with an interface. Hence our interaction net configurations allow us to distinguish these two cases.

Proposition 4. *The rules Indirection and Collect are terminating.*

Proof. Both rules decrease the number of equations in a configuration. □

Non-termination arises because of the Interaction rule (cf. Example 5). There are several criteria for termination of nets, which were defined for the graphical framework, see for example [3] and [8]. These can be recast in the textual language in a much cleaner, concise way.

4 Normal Forms and Strategies

Although we have stressed the fact that systems of interaction are strongly confluent, there are clearly many ways of obtaining the normal form (if one exists), and moreover there is scope for alternative kinds of normal form, for instance those induced by weak reduction.

In this section we study several different notions of normal form and strategies for evaluation of interaction nets. The calculus provides a simple way of expressing these concepts which are quite hard to formalize in the graphical framework. In addition, we define some extra rules that can optimize the computations.

There are essentially two standard notions of normal form for rewriting systems: full normal form and weak normal form (also known as root stable, weak head normal form, etc). In the λ-calculus one also has notions such as head normal form, which allows reduction under the top constructor.

These notions can be recast in our calculus, providing in this way a theory for the implementation of interaction nets. One of the most fruitful applications of this would be the implementation of nets containing disconnected nonterminating computations, which are crucial for the coding of the λ-calculus, and functional programming languages for instance.

We begin with the weakest form of reduction that we will introduce.

Definition 9 (Interface Normal Form). *A configuration* $(\mathcal{R}, \langle \mathbf{t} \mid \Delta \rangle)$ *is in* interface normal form *(INF) if each* t_i *in* $\mathbf{t}$ *is of one of the following forms:*

- $\alpha(\vec{s})$. *E.g.* $\langle S(x) \mid x = Z, \Delta \rangle$.
- x *where* $x \in \mathcal{N}(t_j)$, $i \neq j$. *This is called an* open path. *E.g.* $\langle x, x \mid \Delta \rangle$
- x *where* x *occurs in a* cycle *in* Δ. *E.g.* $\langle x \mid y = \alpha(\beta(y), x), \Delta \rangle$.

Clearly any net with no interface $\langle \mid \Delta \rangle$ *is in interface normal form.*

Intuitively, an interaction net is in interface normal form when there are agents with principal ports on all of the observable interface, or, if there are ports in the interface that are not principal, then they will never become principal by reduction (since they are in an open path or a cycle). This idea can also be adapted to the typed framework, where we may require that only terms with positive types appear in the head of the configuration in interface normal form. This corresponds exactly to the Canonical Forms defined in [7], where negative ports are not observable. Additionally, this notion of normal form can be generalized to deal with a user defined set of values in the interface (Value Normal Form), which can allow some reduction under the top constructor.

The second notion of normal form that we introduce is the strongest one in that all reductions possible have been done, and corresponds to the usual notion of normal form for interaction nets.

Definition 10 (Full Normal Form). *A configuration* $(\mathcal{R}, \langle \mathbf{t} \mid \Delta \rangle)$ *is in* full normal form *if either* Δ *is empty, or all equations in* Δ *are of the form* $x = t$ *where* $x \in t$ *or* x *is free in* $\langle \mathbf{t} \mid \Delta \rangle$.

Having defined the notions of normal form, we now give the corresponding algorithms to compute them. We suggest different ways of evaluating a net to normal form, each of which could be useful for various applications.

Full Reduction. To obtain the full normal form of a net, we apply the computation rules until we obtain an irreducible configuration. Since interaction nets are strongly confluent, there is clearly no need to impose a strategy, since all reductions will eventually be done. However, there are additional factors, involving the size of the net and non-termination, that suggest that different strategies can be imposed:

Priority Reduction. As many of the examples that we have already given indicate, there are interaction rules which reduce the size of the net, keep the size constant, and increase the size of the net. More formally, we define the size of a term:

$$\begin{aligned} |x| &= 0 \\ |\alpha(t_1, \ldots, t_n)| &= 1 + |t_1| + \cdots + |t_n| \end{aligned}$$

The size of an equation is given by $|t = u| = |t| + |u|$. For an interaction rule $r = \alpha(\vec{t}) \bowtie \beta(\vec{u})$, let $s = |\vec{t}| + |\vec{u}|$. Then r is said to be expansive if $s > 2$, reducing if $s < 2$, otherwise it is stable. We can then define an ordering on the rules, and give priority to the ones which are reducing.

Connected reduction. Using this strategy, an application of Interaction to $t = u$ in $\langle \mathbf{t} \mid t = u, \Delta \rangle$ would only be allowed if $\mathcal{N}(\mathbf{t}) \cap \mathcal{N}(t = u) \neq \emptyset$. Thus subnets that are not connected to the observable interface will not be evaluated. As a direct application of this strategy we can define an *accelerated garbage collection.*

Interaction nets are very good at capturing the explicit dynamics of garbage collection. However, nets have to be in normal form before they can be erased. Therefore, non-terminating nets can never be erased. With this strategy, we can ignore isolated nets since they will never contribute to the interface of the net. It is therefore useful to add an additional rule, called **Cleanup**, which explicitly removes these components, without reducing to normal form first. If there are no free names in the equation $t = u$, then

$$\langle \mathbf{t} \mid t = u, \Delta \rangle \longrightarrow \langle \mathbf{t} \mid \Delta \rangle$$

Weak Reduction. Computing interface normal forms suggests that we do the minimum work required to bring principal ports to the interface; computing value normal form requires that each agent in the observable interface is a value. Both of these can be described by placing conditions on the rewrite system. Here we just focus on interface normal form. The following conditional version of the computation rules is enough to obtain this. Given a configuration of the form: $\langle t_1, \ldots, x, \ldots, t_n \mid t = u, \Delta \rangle$, where $x \in \mathcal{N}(t = u)$, then any of the computation rules can be applied to $t = u$. This process is repeated until the Collect rule has brought agents to the interface, or the configuration is irreducible. We can formalize this as a set of evaluation rules, for instance one such way is:

Axiom:

$$\frac{c \in INF}{c \Downarrow c}$$

Collect:

$$\frac{\langle t_1, \ldots, t, \ldots, t_n \mid \Delta \rangle \Downarrow c}{\langle t_1, \ldots, x, \ldots, t_n \mid x = t, \Delta \rangle \Downarrow c}$$

Indirection: if $x \in \mathcal{N}(u)$ and $y \in \mathcal{N}(t, u = v)$

$$\frac{\langle t_1, \ldots, y, \ldots, t_n \mid u[t/x] = v, \Delta \rangle \Downarrow c}{\langle t_1, \ldots, y, \ldots, t_n \mid x = t, u = v, \Delta \rangle \Downarrow c}$$

Interaction: if $x \in \mathcal{N}(\alpha(\vec{t}) = \beta(\vec{u})), \alpha(\vec{t'}) = \beta(\vec{u'}) \in \mathcal{R}$

$$\frac{\langle s_1, \ldots, x, \ldots, s_n \mid \overrightarrow{t = \widehat{t'}}, \overrightarrow{u = \widehat{u'}}, \Delta \rangle \Downarrow c}{\langle s_1, \ldots, x, \ldots, s_n \mid \alpha(\vec{t}) = \beta(\vec{u}), \Delta \rangle \Downarrow c}$$

Note the simplicity of this definition, compared with the definitions given in the graphical framework.

5 Conclusions

We have given a calculus for interaction nets, together with a type system, and its operational theory. This calculus provides a solid foundation for an implementation of interaction nets. In particular, we can express strategies, and define

notions of weak reduction which are essential for actual implementations of interaction nets for realistic computations.

The language has also been extended to attach a value to an agent, and allow interaction rules to use this value in the right-hand side of the rule. Such a system is analogous to the extension of the λ-calculus with δ rules, as done for instance with the language **PCF**.

There are various directions for further study. The operational account has lead to the development of an Abstract Machine for interaction nets [17]. This could be used as a basis for formalizing an implementation and could also serve as the basis for the development of a parallel abstract machine.

References

1. S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993. 171
2. Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139(2):154–233, 1997. 171, 183
3. R. Banach. The algebraic theory of interaction nets. Technical Report UMCS-95-7-2, University of Manchester, 1995. 171, 183
4. A. Bawden. Connection graphs. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 258–265, 1986. 171
5. V. Danos and L. Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28:181–203, 1989. 170, 182
6. M. Fernández. Type assignment and termination of interaction nets. *Mathematical Structures in Computer Science*, 8(6):593–636, 1998. 171, 182
7. M. Fernández and I. Mackie. Coinductive techniques for operational equivalence of interaction nets. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 321–332. IEEE Computer Society Press, 1998. 171, 171, 172, 175, 177, 184
8. M. Fernández and I. Mackie. Interaction nets and term rewriting systems. *Theoretical Computer Science*, 190(1):3–39, 1998. 171, 172, 183
9. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, 1992. 170
10. K. Honda. Types for dyadic interaction. In *CONCUR 93*, Lecture Notes in Computer Science. Springer-Verlag, 1993. 171
11. J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993. 172
12. Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990. 170, 170, 170, 171, 173, 176, 176, 177, 182
13. J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 16–30. ACM Press, 1990. 170
14. C. Laneve. *Optimality and Concurrency in Interaction Systems*. PhD thesis, Dipartmento di Informatica, Università degli Studi di Pisa, 1993. 171

15. I. Mackie. Static analysis of interaction nets for distributed implementations. In P. van Hentenryck, editor, *Proceedings of the 4th International Static Analysis Symposium (SAS'97)*, number 1302 in Lecture Notes in Computer Science, pages 217–231. Springer-Verlag, 1997. 172
16. I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998. 171
17. J. S. Pinto. An abstract machine for interaction nets, 1999. École Polytechnique. 186
18. N. Yoshida. Graph notation for concurrent combinators. In *Proceedings of TPPP'94*, number 907 in Lecture Notes in Computer Science, pages 364–397. Springer-Verlag, 1995. 171
19. N. Yoshida. Minimality and separation results on asynchronous mobile processes: Representability theorems by concurrent combinators. In *Proceedings of CONCUR'98*, number 1466 in Lecture Notes in Computer Science, pages 131–146. Springer-Verlag, 1998. 171