# Interaction Nets and Term Rewriting Systems (Extended Abstract)

Maribel Fernández and Ian Mackie

DMI - LIENS (CNRS URA 1327)
École Normale Supérieure
45 Rue d'Ulm, 75005 Paris, France
maribel@dmi.ens.fr

LIX (CNRS URA 1439)
École Polytechnique
91128 Palaiseau Cedex, France
mackie@lix.polytechnique.fr

**Abstract.** Term rewriting systems provide a framework in which it is possible to specify and program in a one-dimensional syntax. Interaction nets, on the other hand, provide a graphical syntax for the same purpose. In this paper, our aim is to bridge the gap between the above formalisms by showing how to understand interaction nets in a term rewriting framework. This allows us to transfer results from one paradigm to the other; in particular with respect to modularity properties.

## 1 Introduction

Term rewriting systems have been well studied as a general formalism for specifying and reasoning about computation. One of their main assets is that they provide a "universal programming language", or more precisely a specification language, in a traditional one-dimensional syntax which can be used to express different paradigms (functional, logical, parallel, etc.). However, it is fair to say that they have not been too successful in the study of the dynamics of computation since they are free from strategies — there is no intentional behaviour implied by the rules. For example, overlapping patterns leads to nondeterminism.

Recently, a new paradigm in rewriting has been proposed by Lafont under the name *interaction nets* [13]. This framework is equally general and simple like term rewriting systems, but is based on rewriting *networks* rather than terms; hence a graphical syntax. One of the main assets of interaction nets is that they give a more refined view of computation — this is exemplified by the successful use in the study of the dynamics of computation (sharing in the $\lambda$-calculus for example). Interaction nets could be regarded as being closer to an implementation than term rewriting since the rewrite rules are non-ambiguous and confluent, and the reduction process is local and asynchronous; hence they have the potential to be trivially implemented on parallel hardware.

The two formalisms in rewriting outlined above have been developed separately, and very little has been said about the relation between them. The aim of this work is two-fold. First, and primary, the hope is to bridge the gap between the two formalisms. This would then allow us to:

– reason about interaction nets in a traditional syntax. Term rewriting is now a very rich field, with well established theories and results such as type-

systems, modularity and termination proof techniques. An encoding of interaction nets into term rewriting systems should allow all this knowledge to be harnessed.
- reason about term rewriting systems in a graphical syntax, thus allowing the use of properties and graphical intuitions of interaction nets to deduce properties of term rewriting systems. There is also the possibility of applying some of the recent developments in semantics for interaction nets to term rewriting systems.

The second hope is that by translating between interaction nets and term rewriting systems, we can characterise new classes of each formalism with good properties (obtained as images of the translations). Some classes will not have an image in the translation function, which will give rise to the definition of new notions of term rewriting systems and interaction nets. For example, the parallel-or function cannot be written in the interaction net formalism, but can be defined as a rewrite system.

In this paper we begin this work by showing how to understand interaction nets in the term rewriting system world (see [14] for a different approach). We feel this is the most important starting point since term rewriting systems are more established and therefore could, hopefully, be of more interest. For the reverse direction of this translation we require a generalisation of interaction nets so that we can encode interesting term rewriting systems; this is reported in [5]. In the full paper we address the rest of the above issues.

The expression "term rewriting system" is normally used for rewrite systems on first-order terms. As we will see in the following sections, a natural translation of the graphical syntax of interaction nets to terms involves the use of bound variables, which take us away from the world of standard term rewriting systems. In the rest of the paper we will use the expression term rewriting system in a broad sense, including first-order systems and its extensions such as combinatory reduction systems, shared-rewriting systems, and infinitary rewrite systems.

After presenting two different styles of translations from interaction nets to term rewriting systems, we will show that useful properties like confluence and termination are preserved under the translations. In particular, this allows us to apply the techniques developed for term rewriting systems in proofs of termination of interaction net systems. Moreover, we will show that many of the modularity results for termination of term rewriting can be reformulated in a simple way for systems of interaction nets.

The rest of this paper is organised as follows. In the next section we review certain classes of term rewriting system and interaction net that we will use in this paper. In Sections 3 and 4 we provide translations of interaction nets into term rewriting systems. In Section 5 we study some applications of the translations, in particular modularity of termination. Finally, we conclude our ideas and suggest some further directions of study in Section 6.

*Acknowledgements* The work of Ian Mackie was funded by an HCM Fellowship.

## 2 Basic Concepts

### 2.1 Term Rewriting Systems

We recall briefly the definition of first-order term rewriting systems and some extensions, such as shared rewriting and combinatory reduction systems. For more details we refer the reader to the surveys [3, 9, 10].

**First-order Term Rewriting Systems** A *signature* $\mathcal{F}$ is a finite set of *function symbols* together with their (fixed) arity. $\mathcal{X}$ denotes a denumerable set of *variables*, and $T(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* built up from $\mathcal{F}$ and $\mathcal{X}$. Terms are identified with finite labelled trees as usual. *Positions* are strings of positive integers. The *subterm* of $t$ at position $p$ is denoted by $t|_p$ and the result of replacing $t|_p$ with $u$ at position $p$ in $t$ is denoted by $t[u]_p$. This notation also indicates that $u$ is a subterm of $t$. $\mathcal{V}ar(t)$ denotes the set of variables appearing in $t$. A term $t$ is *linear* if variables in $\mathcal{V}ar(t)$ occur at most once in $t$. Substitutions are written as in $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ where $t_i$ is assumed different from $x_i$. We use Greek letters for substitutions and postfix notation for their application.

A *term rewriting system* is a set of rewrite rules $R = \{l_i \rightarrow r_i\}_i$, where $l_i \notin \mathcal{X}$ and $\mathcal{V}ar(r_i) \subseteq \mathcal{V}ar(l_i)$. A term $t$ rewrites to a term $u$ at position $p$ with the rule $l \rightarrow r$ and the substitution $\sigma$, written $t \xrightarrow[l \rightarrow r]{p} u$, or simply $t \rightarrow_R u$, if $t|_p = l\sigma$ and $u = t[r\sigma]_p$. We denote by $\rightarrow_R^+$ (resp. $\rightarrow_R^*$) the transitive (resp. transitive and reflexive) closure of the rewrite relation $\rightarrow_R$. The subindex $R$ will be omitted when it is clear from the context.

A term rewriting system $R$ is *confluent* if $t \rightarrow^* u$ and $t \rightarrow^* v$ implies $u \rightarrow^* s$ and $v \rightarrow^* s$ for some $s$; *terminating* if all reduction sequences are finite; *linear* if all the terms in the rewrite rules are linear; *non-duplicating* (or *conservative*) if for all $l \rightarrow r \in R$ and for all $x \in \mathcal{V}ar(l)$, the number of occurrences of $x$ in $r$ is less than or equal to the number of occurrences of $x$ in $l$.

**Shared Rewriting** Many implementations of term rewrite systems use directed acyclic graphs (dags) rather than trees for efficiency reasons. Common subterms are structurally shared in a dag. In this way, multiple occurrences of a subterm may be simultaneously reduced to a common term. This reduction relation is called *shared-rewriting* (which is a simple case of term graph rewriting [17] where the graphs are acyclic). In a shared-reduction step, subterms that are covered by the same variable in the left-hand side of the rule are not copied but shared in the resulting dag, even if the right-hand side of the rule has multiple occurrences of this variable. For definitions and examples we refer the reader to [12].

**Combinatory Reduction Systems** A combinatory reduction system is a pair consisting of an alphabet and a set of rewrite rules. The alphabet consists of

1. variables $x, y, z, \ldots$,
2. metavariables, written as $Z_i^k$, where $k$ is the arity of $Z_i^k$,
3. function symbols, each with a fixed arity,
4. an operator for abstraction over variables, written as $[\cdot].$,
5. improper symbols such as $(,)$.

In combinatory reduction systems a distinction is made between *metaterms* and *terms*. Metaterms are the expressions built from the symbols in the alphabet, in the usual way. The set of metaterms is denoted by *MTerms*. Terms are metaterms that do not contain metavariables. Variables that are in the scope of the abstraction operator are *bound*, and free otherwise. We adopt the usual convention: all bound variables are chosen to be different from the free variables.

**Definition 1.** A rewrite rule is a pair of metaterms, written $l \to r$, where $l, r$ are closed metaterms, $l$ has the form $F(s_1, \ldots, s_n)$, the metavariables that occur in $r$ occur also in $l$, the metavariables $Z_i^k$ that occur in $l$ occur only in the form $Z_i^k(x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are pairwise distinct variables.

To extract from rules the actual rewrite relation induced on terms, each metavariable is replaced by a special kind of $\lambda$-term, and in the obtained term all $\beta$-redexes and the residuals of these $\beta$-redexes are reduced (i.e. a development is performed). Formally, to define the rewrite relation we have to consider a notion of substitution using *substitutes* and *valuations*. An $n$-ary substitute is an expression of the form $\lambda x_1 \ldots x_n.t$, where $t$ is a term and $x_1, \ldots, x_n$ are different variables. A substitute can be applied to a $n$-tuple $s_1, \ldots, s_n$ of terms. The result is the term $t$ where $x_1, \ldots, x_n$ are simultaneously replaced by $s_1, \ldots, s_n$. A valuation $\sigma$ is a map that assigns an $n$-ary substitute to an $n$-ary metavariable. It is extended to a mapping from metaterms to terms: given a valuation $\sigma$ and a metaterm $T$, first we replace in $T$ all metavariables by its image in $\sigma$ and then we perform the developments of the $\beta$-redexes created by this replacement.

A context is a term with an occurrence of a special symbol $[\,]$ called hole. A rewrite step is now defined in the usual way: if $l \to r$ is a rewrite rule, $\sigma$ a valuation and $C[\,]$ a context, then $C[l\sigma] \to C[r\sigma]$.

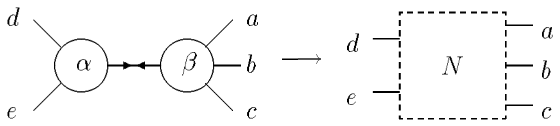
### 2.2 Interaction Nets

Lafont [13] describes a new rewriting paradigm for programming languages and implementation, based on what he calls *interaction nets*: a networked system of interacting agents founded on proof nets for linear logic. We briefly review the paradigm here, and refer the reader to [13, 6] for detailed examples.

**Definition 2.** An interaction net $(\Sigma, IR)$ is specified by the following data:

- A set $\Sigma$ of symbols (*nodes* or *agents*), each with an arity in $\mathbb{N}$, which is the number of edges attached to each node. A *net* on $\Sigma$ is an undirected graph with vertices labelled by symbols $\alpha, \beta \in \Sigma$. The points of contact of edges with nodes are called *ports*. Each agent has a distinguished port, called the *principal port*, which is the point of contact where communication can take place. All the other ports are called *auxiliary* ports.
- A set $IR$ of *interaction rules* which are activated when two symbols are connected on their principal port. There is only one rule for each pair of symbols.

The basic idea of an interaction rule is given by the following diagram, where the arities of $\alpha$ and $\beta$ are 3 and 4 respectively, and $N$ is a new net (we write nets with dashed lines), which may contain occurrences of the original nodes in the interaction. Note that *the interface is preserved.*

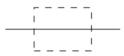



A net is said to be in *normal form* when no interactions are possible. As an almost immediate consequence of this kind of rewriting we have the following features: *confluence*, *local implementation* (the local interface is preserved during an interaction — two agents interacting do so in their own "space" and do not affect any other part of the net), and *asynchrony* (as a direct consequence of the above point, we have the possibility of a truly parallel implementation — no order on the interactions is required since any two agents ready to communicate can do so in any order).

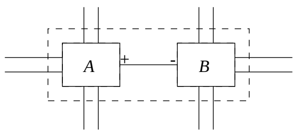
**Classification of Nets** Lafont introduced a type discipline for interaction nets, using constant types (atom, nat, list-nat, ...). For each agent, ports are classified into input and output. An input port will be assigned a type $\tau^-$ and an output port a type $\tau^+$. A net is *well-typed* if input ports are connected to output ports of the same type. In the following, for the sake of simplicity, we will consider only one type. In other words we will only distinguish between input ports (marked with a $-$ sign) and output ports (marked with $+$). If the principal port of an agent is an output port, the agent is a *constructor*, otherwise it is a *destructor*.

For each symbol, the auxiliary ports are divided into *partitions*. This notion was introduced in [13] with the purpose of defining a class of interaction nets that are deadlock free: semi-simple nets. A net is *semi-simple* if it can be obtained by using only the following operations:

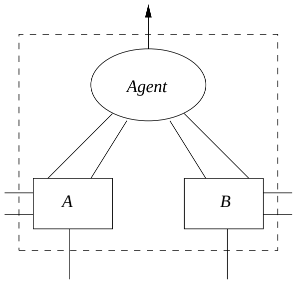
1. LINK, which builds an edge:



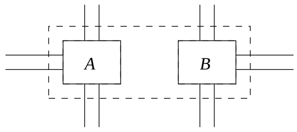
2. CUT, that connects two nets using a single edge:




3. GRAFT, that adds an agent to a set of nets according to its partitions:




4. MIX, juxtaposing two nets:




5. and EMPTY, which constructs an empty net.

Interaction nets where all partitions contain only one port are called *discrete*. As remarked by Lafont [13], in the discrete case a net is semi-simple if it is a graph without cycles. If the operations of MIX and EMPTY are not used, a semi-simple net is a connected graph without cycles (i.e. a tree).

If a negative port exists in a partition, then we will call it an *input partition*, otherwise it will be called an *output partition*. Every agent has at most one output partition (e.g. constructors do not have an output partition, the principal port is an output, and they can have output ports in input partitions). According to this, there are two classes of interaction nets: *dependent interaction nets*, where a positive port appears in an input partition of some agent; and *non-dependent interaction nets*, where every agent has only negative ports in input partitions.

## 3 From Semi-simple Nets to Term Rewriting Systems

In this section we study the relationship between semi-simple interaction nets and term rewriting systems. The general case of interaction nets (i.e. including nets that are not semi-simple) will be addressed in Section 4.

It is known that discrete semi-simple interaction nets correspond to (a restricted class of) first-order term rewriting systems (see [13, 14]). We will define a translation function that transforms a semi-simple interaction net into a combinatory reduction system. As a particular case, we will see that non-dependent semi-simple nets (which include the class of discrete nets) correspond to first-order term rewriting systems with shared-rewriting. In the discrete case we obtain a linear first-order term rewriting system.

### 3.1 Translation

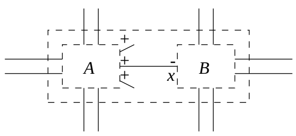
We consider first the case of non-dependent semi-simple nets and then the general case of semi-simple nets. We start by defining a mapping $\Theta : Nets \rightarrow T(\mathcal{F}, \mathcal{X})$

which takes a non-dependent semi-simple net and gives a term. With the help of this function we will translate interaction rules into rewrite rules: the translation of an interaction rule will simply be obtained by translating each member.

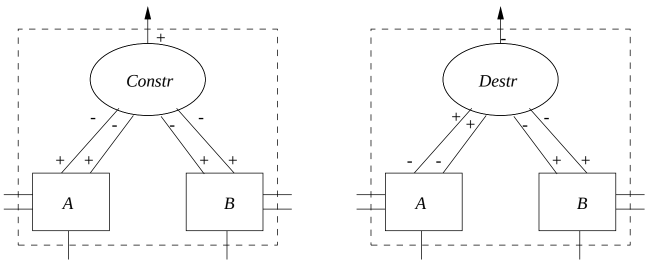
The translation of a net with inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_m$ will be a term $t[x_1, \ldots, x_n]$ where the type of $t$ will be an $m$-tuple. A semi-simple net is defined by a sequence of operations of LINK, CUT, GRAFT, MIX, EMPTY. We will define the translation function by induction on the definition of the semi-simple net.

**Definition 3.** Let $(\Sigma, IR)$ be an interaction net, and $\mathcal{F}$ be a set of function symbols containing the agents in $\Sigma$, the constant *empty*, and a binary symbol $P$ for pair formation (we assume that $\Sigma$ does not contain agents called $P$, *empty*). $P$ is assumed to be associative, so we use the flat notation $P(x_1, \ldots, x_n)$ for any $n$. To simplify the notation we will use some informal abbreviations: for instance, given a term $t = P(t_1, \ldots, t_n)$, $\pi_i(t) = t_i$ and $t - \pi_i(t) = P(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n)$. Here we assume that when only one element remains, $P(t_1) = t_1$. A sequence $u_1, \ldots, u_n$ will be denoted by $\vec{u}$.

1. The translation of an empty net is given by: $\Theta(\text{EMPTY}) = \mathit{empty}$
2. The translation of a link is a variable: $\Theta(\text{LINK}) = x$. We assume that fresh variables are obtained on demand.
3. The translation of a net constructed from two nets $A$, $B$ by a MIX operation is the pair formed by the translations of $A$ and $B$:
$$\Theta(\text{MIX}(A, B)) = P(\Theta(A), \Theta(B))$$
4. The translation of a net constructed from two nets $A$, $B$ by a CUT operation




is: $\Theta(\text{CUT}(A, B)) = P(\Theta(B)\{x \mapsto \pi_i(\Theta(A))\}, \Theta(A) - \pi_i(\Theta(A)))$
In the discrete case we have: $\Theta(\text{CUT}(A, B)) = \Theta(B)\{x \mapsto \Theta(A)\}$
5. The translation of a GRAFT depends on the class of agent:




(a) If it is a constructor: $\Theta(\text{GRAFT}(\mathit{Constr}, A, B)) =$
$$P(\mathit{Constr}(\overrightarrow{\pi_i(\Theta(A))}, \overrightarrow{\pi_i(\Theta(B))}), \Theta(A) - \overrightarrow{\pi_i(\Theta(A))}, \Theta(B) - \overrightarrow{\pi_i(\Theta(B))})$$

In this formula we have taken into account the fact that in the case of a non-dependent net the only output port of a constructor is the principal port, and that $A$ and $B$ can be nets with multiple output ports. In that case the translations of $A, B$ will be tuples, and to obtain the translation of the GRAFT we have to select the corresponding elements of the tuples. This is abbreviated by $\overrightarrow{\pi_i(\Theta(A))}$, $\overrightarrow{\pi_i(\Theta(B))}$. The rest of the outputs of $A$ and $B$ (i.e. $\Theta(A) - \overrightarrow{\pi_i(\Theta(A))}, \Theta(B) - \overrightarrow{\pi_i(\Theta(B))}$) are still outputs of the net resulting from the GRAFT, as we can see in the formula above. When $A$ and $B$ have only one output port each, the formula reduces to:

$$\Theta(\mathrm{GRAFT}(Constr, A, B)) = Constr(\Theta(A), \Theta(B))$$

(b) If it is a destructor: $\Theta(\mathrm{GRAFT}(Destr, A, B)) =$

$$\overrightarrow{P(\Theta(A)\{y_i \mapsto \pi_i(Destr(x, \overrightarrow{\pi_i(\Theta(B))}))\}}, \Theta(B) - \overrightarrow{\pi_i(\Theta(B))})$$

In this formula $x$ represents the principal port of $Destr$, and $\vec{y_i}$ represent the input ports of $A$. Again we can see that in the simplest case (discrete net) this formula gives the natural translation:

$$\Theta(\mathrm{GRAFT}(Destr, A, B)) = \Theta(A)\{y \mapsto Destr(x, \Theta(B))\}$$

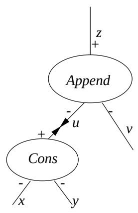
In the degenerate case of a destructor without output ports (see example below) we apply the same formula with a dummy output.

Note that if the interaction net is not discrete a variable can occur more than once in the image of $\Theta$. In this case in order to obtain a behaviour similar to the one of the net, multiple occurrences of a variable have to be shared in the right-hand side (see the examples below).

Note also that our translation function takes as input a sequence of operations that construct a semi-simple net. The sequence of operations needed to build a given semi-simple net is not uniquely defined. It is possible in general to change the order in which the operations of CUT are done, and the same for GRAFTS. A change in the order of CUTS does not affect the result of the translation, but a change in the order of GRAFTS can change it.

We now give some examples to illustrate the definition of $\Theta$.

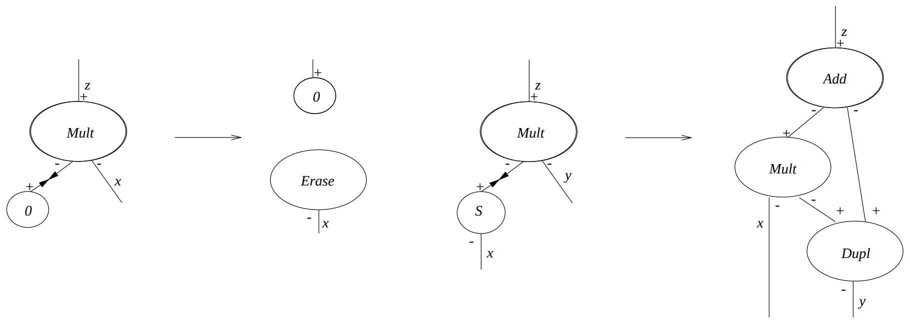
*Example 1.* Consider first an interaction net for lists, specified by the constructors *Cons* and *Nil* and the destructor *Append*. The net




is a semi-simple net obtained by first creating the LINKS $z$ and $v$, and the GRAFT of *Append* on them (which gives the term $z\{z \mapsto Append(u, v)\} \equiv$

$Append(u,v))$, and then making a CUT with the net represented by $Cons(x,y)$, which gives: $Append(u,v)\{u \mapsto Cons(x,y)\} \equiv Append(Cons(x,y),v)$.

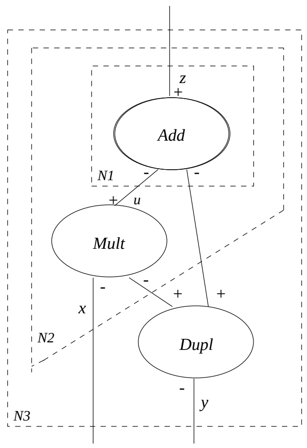
For an example with a non-discrete net, consider the system of interaction rules to add and multiply natural numbers given in [13]. The agents $S$ and $0$ are constructors, and $Add$ and $Mult$ are destructors. In the standard definition of multiplication using addition, the second argument is not used when the first is 0, and it is used twice otherwise. In the interaction net presentation this requires an agent for erasing and another for copying (duplication). This is done with the agents $Erase$ and $Dupl$ (the latter with two output ports in the same partition). We show only the interaction rules for multiplication:




The translations of the terms in the left-hand sides of the rules are similar to the previous example (two GRAFTS and a CUT for each left-hand side). More interesting is the translation of the right-hand sides. For the first rule, the net in the right-hand side is the result of a MIX operation on the nets obtained by grafting 0 and $Erase$:

$\Theta(MIX(GRAFT(0), GRAFT(Erase, LINK(y)))) = P(0, Erase(x))$

For the second rule, the net in the right-hand side is constructed by three GRAFTS (and LINK operations), as shown in the following picture:

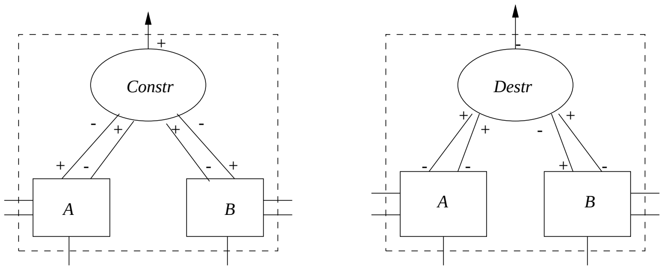



According to the definition of $\Theta$:

$\Theta(N_1) = Add(u, v_2)$
$\Theta(N_2) = \Theta(N_1)\{u \mapsto Mult(x, v_1)\} = Add(Mult(x, v_1), v_2)$
$\Theta(N_3) = \Theta(N_2)\{v_1 \mapsto \pi_1(Dupl(y)), v_2 \mapsto \pi_2(Dupl(y))\} =$
$\quad Add(Mult(x, \pi_1(Dupl(y))), \pi_2(Dupl(y)))$

Next we consider the general case of semi-simple nets, which includes dependent nets. We will use terms with *bound variables* to encode nets where agents have positive ports in input partitions. By abuse of the language, we still call the translation function $\Theta$, although its image now will be *MTerms*, the set of metaterms associated to the signature $\mathcal{F}$ of Definition 3, and interaction rules will be translated as combinatory reduction rules. The translation function is again defined by induction. The cases of LINK, EMPTY, MIX, and CUT are similar to the previous definition, so we will only present the case of a GRAFT.

**Definition 4.** The translation of a GRAFT depends as before on the class of agent that is being added. There are two cases:




In the case of a constructor $\Theta(\text{GRAFT}(Constr, A, B)) =$

$P(Constr(\overrightarrow{\pi_i([\vec{x}]\Theta(A))}, \overrightarrow{\pi_i([\vec{y}]\Theta(B))}), [\vec{x}]\Theta(A) - \overrightarrow{\pi_i([\vec{x}]\Theta(A))}, [\vec{y}]\Theta(B) - \overrightarrow{\pi_i([\vec{y}]\Theta(B))})$

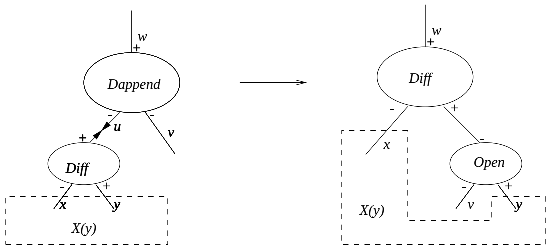
In this formula we have taken into account the fact that the input partition of *Constr* connecting to $A$ (resp. $B$) can have some positive ports. The translation of $A$ (resp. $B$) is used in the negative port of the partition of *Constr*, the positive ports of the same partition are not represented as arguments of *Constr*, since they correspond to bound variables that will typically appear in $\Theta(A)$ (resp. $\Theta(B)$). In case no information about the net $A$ (resp. $B$) is available, the translation of $A$ (resp. $B$) in the formula above is just $X(\vec{x})$ (resp. $Y(\vec{y})$). Also, as in the previous definition, $A$ and $B$ can be nets with multiple output ports. Then the translations of $A, B$ are tuples, and to obtain the translation of the GRAFT we select the corresponding elements.

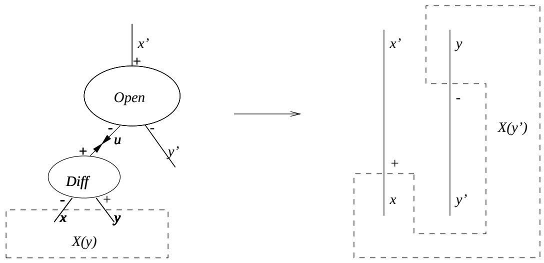
In the case of a destructor we have to consider that the input partition of *Destr* connecting with $B$ can contain positive ports. Then $\Theta(\text{GRAFT}(Destr, A, B)) =$

$$P(\overrightarrow{\Theta(A)\{y_i \mapsto \pi_i(Destr(x, \overrightarrow{\pi_i([\vec{z}]\Theta(B))}))\}}, [\vec{z}]\Theta(B) - \overrightarrow{\pi_i([\vec{z}]\Theta(B))})$$

where, as in Definition 3, $x$ represents the principal port of *Destr*, and $\vec{y_i}$ represent the input ports of $A$. The variable $z$, representing the port marked with a negative sign in $B$, will typically appear in $\Theta(B)$.

*Example 2.* Let us recall the interaction system for appending difference lists given by Lafont [13]. The interaction rules defining *Dappend* are:




Note that in these rules we have emphasised the fact that the ports $x$ and $y$ of *Diff* are in the same partition, connected to a net from which we do not have any information (hence its translation is $X(y)$).

Let us translate the first interaction rule. The net in the left-hand side is obtained by a CUT of two GRAFTS, one translated as $Dappend(u, v)$ and the other as $\mathit{Diff}([y]X(y))$. The net in the right-hand side is the result of the composition of two GRAFTS. The first one is translated as $X(open(v, z))$, and its composition with the second, using the constructor case of Definition 4, gives $\mathit{Diff}([z]X(open(v, z)))$. Then the combinatory reduction rules corresponding to the interaction rules are:

$$Dappend(\mathit{Diff}([y]X(y)), V) \to \mathit{Diff}([z]X(open(V, z)))$$
$$open(\mathit{Diff}([y]X(y)), Y') \to X(Y')$$

It is easy to prove that the translation function preserves normal forms and rewrite sequences. More precisely:

**Theorem 5.** *Let $N, N'$ be nets in a semi-simple interaction net system. If $N$ is a normal form then so is $\Theta(N)$, and if $N \to N'$ then $\Theta(N) \to^+ \Theta(N')$.*

*Proof.* By induction on the definition of the semi-simple net $N$. For CUT and GRAFT the translation of a part of $N$ can appear several times in $\Theta(N)$, and then an interaction step in $N$ may correspond to several rewrite steps in $\Theta(N)$.

For non-dependent nets with cycles we can define a translation function in the same spirit as $\Theta$ into *infinitary* first-order term rewriting systems (see [8]). In the same way, arbitrary interaction nets can be coded as infinitary combinatory reduction systems. However, there is another way of translating arbitrary interaction nets into finitary combinatory reduction systems, which we give in the following section.

## 4 Interaction Nets to Combinatory Reduction Systems

In this section we show the general case of translating arbitrary (well-typed) interaction nets into combinatory reduction systems, for which we define a mapping $\Theta : Nets \rightarrow MTerms$.

For each node $\alpha$ in a net $W$ we consider the term $\alpha(x, \vec{y})$ where $x$, the element in the first position, will be the edge connected to the principal port by convention, and $\vec{y}$ the edges connected to the auxiliary ports (we assume that each edge in $W$ has a different name). For each edge not connected to any node we consider the term $I(a, b)$ where $a, b$ are the extremes of the edge in $W$ and $I$ is a binary function symbol. For technical reasons, it is convenient to use a dummy function symbol $N$ to represent nets. The translation of $W$ will then be the term

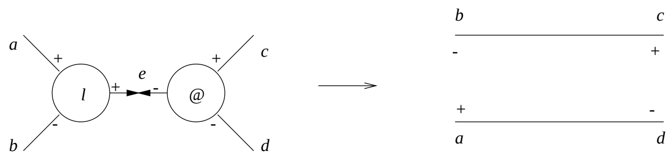
$$N([\vec{z}]\, \overrightarrow{\alpha(x, \vec{y})} \cdot \overrightarrow{I(a, b)})$$

where $\cdot$ is an infix operation that we think of as a set constructor, and all the variables are bound. The empty net will be represented again by the constant *empty*. Interaction rules are translated as pairs of metaterms, obtained by applying $\Theta$ to each member of the rule (and adding a metavariable in the left-hand side so that rules can be applied inside a context). To the resulting system we will add the following rule for $I$:

$$N([x, y]I(x, y) \cdot Z(x)) \rightarrow N([y]Z(y))$$

which indicates that $I$ makes a renaming of variables. The symbol $N$ is used only because in a combinatory reduction system the left-hand sides of rules cannot start with an abstraction (see Definition 1). To show how the translation works we consider a simple example.

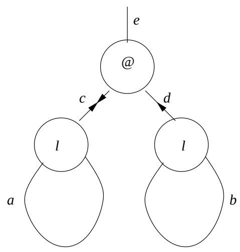
*Example 3.* The linear $\lambda$-calculus is a restriction on the $\lambda$-calculus in that variables must occur *exactly* once. There is a simple interaction net system for this which requires two agents, which we call $l$ and @. To satisfy $\beta$-reduction in the linear $\lambda$-calculus, the (only) interaction rule must be the following:




We apply the translation $\Theta$ to the above interaction rule to generate a combinatory reduction system that will also code the linear $\lambda$-calculus. The translation of the above rule gives:

$$N([e, a, b, c, d]l(e, a, b) \cdot @(e, d, c) \cdot Z(a, b, c, d)) \rightarrow$$
$$N([a, b, c, d]I(b, c) \cdot I(d, a) \cdot Z(a, b, c, d))$$

As a simple example we show the reduction of the net $II$ corresponding to $(\lambda x.x)(\lambda x.x)$:




The translation of this net according to $\Theta$ is:

$$\Theta(II) = N([a,b,c,d,e]@(c,d,e)\cdot l(c,a,a)\cdot l(d,b,b))$$

Applying the rewrite rule above we obtain the following reduction sequence:

$$\Theta(II) \rightarrow N([a,b,d,e]I(d,a)\cdot I(a,e)\cdot l(d,b,b)) \rightarrow$$
$$N([b,d,e]I(d,e)\cdot l(d,b,b)) \rightarrow N([b,e]l(e,b,b))$$

Note that the last term coincides with the translation of the net obtained by reducing the net $II$ with the interaction rule above, hence the rewrite relation is preserved under the translation.

The preservation of the reduction relation and the normal forms under the translation is in fact a general property. Theorem 5 holds also for the translation function defined in this section.

The method presented here can be implemented directly, although for the sake of efficiency it is useful to have a notion of "redex stack" which indicates which elements of the list are ready to interact rather than testing all the elements in the list. This has been done for the implementation of the $\lambda$-calculus as an interaction net in GOI-Tools [15].

## 5 Applications of the Translation Functions

Within the framework of term rewriting systems two properties deserve special attention: confluence and termination. Interaction nets are confluent by construction, but termination is not guaranteed. The aim of this section is to use the translation functions in order to study the termination of interaction nets: since the translations preserve the reduction sequences, we can take profit of the termination techniques developed for term rewriting systems (see e.g. [1, 3, 7]). Even more, we can try to translate the methods proposed for term rewriting systems to have methods to prove termination of interaction nets directly. Because of lack of space, we only consider the problem of the modularity of termination of interaction nets.

If we show that a class of interaction nets is modular with respect to termination, then we can decompose a system of interaction rules into simpler subsystems and prove termination of each subsystem separately. Since more results are known in these respects for first-order term rewriting systems than for combinatory reduction systems, in this paper we concentrate on non-dependent interaction nets. Note that even this class of interaction nets is not modular with respect to termination in general. In the rest of the section we use the results of the previous sections to derive sufficient conditions for modularity.

### 5.1 Modularity of Termination of Interaction Nets

A union of semi-simple non-dependent interaction nets that share only constructor symbols is translated as a union of first-order term rewriting systems with shared-rewriting, where only constructors are shared. Then, as a direct consequence of the modularity of termination for unions of non-duplicating first-order term rewriting systems (or systems with shared rewriting) when it is the case that only constructors are shared (see e.g. [12, 4]) we can deduce:

***Theorem 6.*** *Unions of semi-simple non-dependent interaction nets where only constructors are shared are modular with respect to termination.*

Note that this includes disjoint unions, and that in contrast with term rewriting systems, no restriction needs to be imposed on disjoint unions.

There are some classes of unions of term rewriting systems that are modular even if defined functions are shared. We recall here two modularity results for unions of term rewriting systems with shared defined functions that are easy to apply to interaction nets. We deduce from them two further modularity results for interaction nets.

**Unions of Constructor Systems** A *constructor system* is a term rewriting system with the property that the signature can be partitioned into a set $\mathcal{C}$ of constructors and a set $\mathcal{D}$ of defined symbols, and every left-hand side $f(t_1, \ldots, t_n)$ of a rule in $R$ satisfies $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in T(\mathcal{C}, \mathcal{X})$. Middeldorp and Toyama [16] showed that unions of confluent and *pairwise composable systems* are modular with respect to termination. This result applies directly to non-dependent semi-simple interaction nets: the result of applying $\Theta$ to a union of interaction nets is a union of confluent constructor systems. We only have to rephrase the notion of composability for interaction net systems: two interaction net systems over sets of agents $(\mathcal{D}_1, \mathcal{C}_1)$, $(\mathcal{D}_2, \mathcal{C}_2)$ are composable if $\mathcal{D}_1 \cap \mathcal{C}_2 = \mathcal{D}_2 \cap \mathcal{C}_1 = \emptyset$, and for any $d \in \mathcal{D}_1 \cap \mathcal{D}_2$, the same interaction rules for $d$ appear in both systems. Then, from the previous property of constructor systems we deduce:

***Theorem 7.*** *Unions of composable non-dependent semi-simple interaction net systems are modular with respect to termination.*

**Hierarchical Unions** Hierarchical unions of term rewriting systems are very interesting from a practical point of view because they correspond to the so-called incremental development methodology of algebraic specifications.

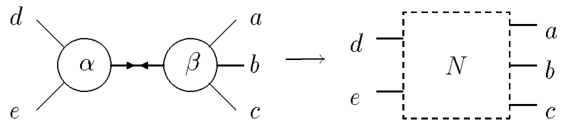
Assume given two rewrite systems over the signatures $\mathcal{F}_i = \mathcal{C}_i \cup \mathcal{D}_i$ $(i = 0, 1)$, each one specifying the set $\mathcal{D}_i$ $(i = 0, 1)$ of *defined functions* with respect to the set $\mathcal{C}_i$ of *constructors* by means of a set $R_i$ of rewrite rules. By definition, $\mathcal{D}_i$ is the set of symbols heading the left-hand sides of rules, the other symbols being in $\mathcal{C}_i$. If $\mathcal{D}_0 \cap \mathcal{D}_1 = \emptyset$ and $\mathcal{F}_0 \cap \mathcal{D}_1 = \emptyset$, but $\mathcal{F}_1 \cap \mathcal{D}_0 \neq \emptyset$, that is, the systems may share constructors, and moreover the defined symbols of $R_0$ may be constructors of $R_1$, then the union of $R_0$, $R_1$ is a *hierarchical union*, denoted by $R_0 + R_1$.

There are several classes of hierarchical unions of term rewriting systems that are known to be modular with respect to termination (see e.g. [2, 11, 4]). In [4] it is shown that termination is modular in hierarchical unions $R_0 + R_1 + \ldots + R_n$ in which

- $R_0$ is terminating and non-duplicating (or shared rewriting is used) and
- each incremental development defines a new function symbol using rules that satisfy a *general recursive scheme*.

This result can be easily adapted to the interaction net framework. Hierarchical unions of interaction nets are defined in the same way as hierarchical unions of term rewriting systems, and it is easy to give an interaction net version of the general recursive scheme. We do it with the help of a diagram:

**Definition 8.** An interaction rule




where $\alpha$ is a destructor and $\beta$ is a constructor, satisfies the general recursive scheme if each occurrence of the agent $\alpha$ in the net $N$ has one port connected to (a copy of) $a$, $b$, or $c$, another port connected either to (copies of) $a, b, c$ or $d$, and another port connected either to (copies of) $a, b, c$ or $e$.

A lexicographic version of the scheme also suffices to obtain modularity of termination, i.e. we could have asked that any occurrence of the agent $\alpha$ in the net $N$ had its principal port connected to $a, b$, or $c$, and the other ports connected to $d, e$ respectively.

An immediate consequence of the results presented in [4] is the following:

***Theorem 9.*** *Hierarchical unions $N_0 + N_1 + \ldots + N_n$ of non-dependent semi-simple interaction net systems such that $N_1, \ldots, N_n$ satisfy the general recursive scheme are modular with respect to termination.*

## 6 Conclusions and Further Work

In this paper we have begun the study of the relationship between interaction nets and term rewriting systems. One of the main points of interest that we have shown is that modularity properties of term rewriting systems can be carried over to semi-simple interaction nets in a straightforward and systematic way. To complete these results we intend to study the translation of arbitrary nets to infinitary term rewriting systems.

One of the main goals of this work is to try to get a better understanding of the relationship between interaction nets and term rewriting systems. After showing how to transfer properties between the systems our next step is to transfer the semantics and implementation techniques of one system to the other.

## References


1. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–115, 1987.
2. N. Dershowitz. Hierarchical termination. In *Proc. Conditional and Typed Term Rewriting Systems*, LNCS, 1995.
3. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, vol. B. North-Holland, 1989.
4. M. Fernández and J.-P. Jouannaud. Modular termination of term rewriting systems revisited. In *Recent Trends in Data Type Specification. Proc. 10th. Workshop on Specification of Abstract Data Types*, number 906 in LNCS, 1995.
5. M. Fernández and I. Mackie. From term rewriting systems to generalised interaction nets, 1996. Submitted for publication.
6. S. Gay. Combinators for interaction nets. In C. Hankin, I. Mackie, and R. Nagarajan, eds., *Proceedings of the 2nd Imperial College, Department of Computing, Workshop on Theory and Formal Methods.* Imperial College Press, 1995.
7. J.-P. Jouannaud. Rewrite proofs and computations. In H. Schwichtenberg, ed., *Proof and Computation*, vol. 139 of *Computer and Systems Sciences.* Springer-Verlag, 1995.
8. J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. Transfinite reductions in orthogonal term rewriting systems. *Information and Computation.* To appear.
9. J. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, eds., *Handbook of Logic in Computer Science*, vol. 2. Oxford University Press, 1992.
10. J. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.
11. M. R. K. Krishna Rao. Completeness of hierarchical combinations of term rewriting systems. In *Proc. of FST & TCS'93*, number 761 in LNCS, 1993.
12. M. Kurihara and A. Ohuchi. Non-copying term rewriting and modularity of termination. Hokkaido University.
13. Y. Lafont. Interaction nets. In *Proceedings, 17th ACM Symposium on Principles of Programming Languages*, 1990.
14. C. Laneve. *Optimality and Concurrency in Interaction Systems.* PhD thesis, Dipartmento di Informatica, Università degli Studi di Pisa, 1993.
15. I. Mackie. *The Geometry of Implementation.* PhD thesis, Department of Computing, Imperial College of Science Technology and Medicine, 1994.
16. A. Middeldorp and Y. Toyama. Completeness of combinations of constructor systems. In *Proc. 4th Rewriting Techniques and Applications*, number 488 in LNCS, 1991.
17. M. Sleep, M. Plasmeijer, and M. van Eekelen. *Term Graph Rewriting, Practice and Experience.* John Wiley and Sons Ltd, 1993.