ELSEVIER

Theoretical Computer Science 190 (1998) 3–39

Theoretical Computer Science

Interaction nets and term-rewriting systems

Maribel Fernández[a,*], Ian Mackie[b]

[a] *DMI - LIENS (CNRS URA 1327), École Normale Supérieure, 45 Rue d'Ulm, 75005 Paris, France*

[b] *LIX (CNRS URA 1439), École Polytechnique, 91128 Palaiseau Cedex, France*

Abstract

Term-rewriting systems provide a framework in which it is possible to specify and program in a traditional syntax (oriented equations). Interaction nets, on the other hand, provide a graphical syntax for the same purpose, but can be regarded as being closer to an implementation since the reduction process is local and asynchronous, and all the operations are made explicit, including discarding and copying of data. 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, deriving syntactical properties of interaction nets from the (well-studied) properties of term-rewriting systems; in particular concerning termination and modularity.

Keywords: Term rewriting; Interaction nets; Termination; Modularity

1. Introduction

Term-rewriting systems provide a general framework for specifying and reasoning about computation. They can be regarded as a universal programming language where different paradigms (functional, logical, parallel, etc.) can be expressed, or as an abstract model of computation (abstract in the sense that they specify actions but not control, for instance they are free from strategies – there is no intentional behaviour implied by the rewrite rules).

Recently, *interaction nets* have been proposed by Lafont [22] as a new paradigm in rewriting, based on rewriting of *networks* rather than terms; hence a graphical syntax. Interaction nets are a generalisation of *proof nets* [12]. Because of the linear logic foundation, they give a more refined view of computation, which is exemplified by their successful use in the study of the dynamics of computation (sharing in the λ-calculus for example [14]). Interaction nets are closer to an implementation than term

*Corresponding author. E-mail: maribel@dmi.ens.fr.

PII S0304-3975(97)00082-0

rewriting systems, since the interaction rules are non-ambiguous and confluent, and the reduction process is local and asynchronous.

The two formalisms in rewriting outlined above have been developed separately, isolating each paradigm from progress in the other. The aim of our work is twofold. 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 term-rewriting framework. 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. This is an important point if we see interaction nets as a programming paradigm (as presented by Lafont [22]).
- 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 of interaction nets to term rewriting, and, in particular, understanding term-rewriting systems in the interaction net framework allows us to apply to term rewriting languages the implementation techniques of interaction nets.

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). More specifically, when comparing interaction nets and term-rewriting systems a number of questions arise naturally:

- Is it possible to translate between the two formalisms in a faithful way? What properties are well behaved under these translations?
- There are a number of different classes of interaction nets. What classes of term-rewriting systems correspond to these? The same question also applies the other way around.
- Some classes of term-rewriting systems do not correspond to any particular class of interaction nets. Can this lead to the definition of new and interesting extensions? For example, interaction nets only capture confluent and sequential computations, so there is no notion of parallel function. Since term-rewriting systems can code such functions (e.g. parallel-or), can we generate a new notion of interaction net where these are captured but still retain the salient features?

In this paper we begin this work by showing how to understand interaction nets in the term rewriting world. The expression "term-rewriting system" is normally used for rewrite systems that deal with 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 takes us away from the world of standard term-rewriting systems. We will consider a generalisation of the first-order systems, introduced by Klop [18] under the name of *combinatory reduction systems*, that combines first-order rewriting with the presence of bound variables. We will use the expression term-rewriting systems in a broad sense, including first-order systems and extensions like combinatory reduction systems and shared-rewriting.

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. As a consequence, we can apply the techniques developed for term-rewriting systems in proofs of termination of interaction nets (confluence holds by construction). Moreover, we will show that many of the modularity results for termination of term rewriting can be reformulated in a simple way for unions of interaction nets.

The study of the reverse translations, encoding term-rewriting systems in the interaction framework, was started in [9] as a first step towards the development of an interaction-net based implementation of term-rewriting systems. In order to encode interesting systems like parallel-or (more generally, non-sequential or non-confluent term-rewriting systems), a generalisation of interaction nets is required. This led to the definition of parallel interaction nets with state which is also reported in [9].

Interaction nets have been used as a tool to study optimal implementations of the λ-calculus [14]. Using these ideas, Laneve [24] extended the notion of optimality to *interaction systems* (a subclass of combinatory reduction systems). It turned out that interaction systems then corresponded to (a class of) interaction nets. Also, interaction nets and other related models that are founded on linear logic [12], like the Geometry of Interaction [13], have been successfully used for the implementation of various λ-calculi [25]. Our work can be regarded as a continuation of this last research line. On the one hand, we study the relations between interaction nets and term-rewriting systems with the hope of making these semantic results and implementation techniques applicable also to term-rewriting systems and to languages that combine term rewriting and λ-calculus. On the other hand, seeing interaction nets as a programming paradigm, our study allows us to apply the programming techniques developed for term-rewriting languages, in particular concerning modularity, to interaction nets. The latter point is the main subject of this paper.

The 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 the sequel. In Section 3 we provide translations of interaction nets into various classes of term-rewriting systems. In Section 4 we study some applications of the translations, in particular modularity of termination. Finally, in Section 5 we conclude our ideas and suggest further directions. This paper is a revised and extended version of the paper [10] presented at CAAP'96.

2. Basic concepts

In this section we recall the formalisms that we will use throughout this paper. We refer the reader to the surveys [6, 19] for further examples of rewrite systems, and to [22] for interaction nets.

2.1. Term-rewriting systems

Term-rewriting systems can be seen as programming or specification languages, or as formulae manipulating systems that can be used in various applications such as program optimisation or automated theorem proving. We recall briefly the definition of first-order term-rewriting systems, and then describe two extensions: shared-rewriting and combinatory reduction systems.

2.1.1. First-order term-rewriting systems

A *signature* $\mathscr{F}$ is a finite set of *function symbols* together with their (fixed) arity. $\mathscr{X}$ denotes a denumerable set of *variables*, and $T(\mathscr{F},\mathscr{X})$ denotes the set of *terms* built up from $\mathscr{F}$ and $\mathscr{X}$.

Terms are identified with finite labeled trees, as usual. The symbol at the root of t is denoted by $root(t)$. *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 is also used to indicate that u is a subterm of t. The strict subterm relationship is denoted by $\lhd$ (then $\rhd$ denotes the superterm ordering). We use $\equiv$ to denote syntactic equivalence of objects.

$\mathscr{V}ar(t)$ denotes the set of variables appearing in t. A term is *linear* if variables in $\mathscr{V}ar(t)$ 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.

Definition 2.1. Given a signature $\mathscr{F}$, a *term-rewriting system* on $\mathscr{F}$ is a set of rewrite rules $R = \{l_i \to r_i\}_{i\in I}$, where $l_i, r_i \in T(\mathscr{F},\mathscr{X})$, $l_i \notin \mathscr{X}$, and $\mathscr{V}ar(r_i) \subseteq \mathscr{V}ar(l_i)$. A term t *rewrites* to a term u at position p with the rule $l \to r$ and the substitution σ, written $t \xrightarrow[l\to r]{p} u$, or simply $t \to_R u$, if $t|_p = l\sigma$ and $u = t[r\sigma]_p$. Such a term t is called *reducible*. Irreducible terms are said to be in *normal form*.

We denote by $\to_R^+$ (resp. $\to_R^*$) the transitive (resp. transitive and reflexive) closure of the rewrite relation $\to_R$. The subindex R will be omitted when it is clear from the context.

The signature $\mathscr{F}$ of a term rewriting system is partitioned into a set $\mathscr{D}$ of *defined symbols*: $\mathscr{D} = \{f \mid root(l) = f \text{ for some } l \to r \in R\}$, and a set $\mathscr{C}$ of *constructors*: $\mathscr{C} = \mathscr{F} - \mathscr{D}$.

In most programming languages based on rewriting, the *constructor discipline* is assumed, that is programs are constructor systems:

Definition 2.2. A *constructor system* is a term rewriting system over a signature $\mathscr{F} = \mathscr{C} \cup \mathscr{D}$ (where $\mathscr{C}$ is the set of constructors and $\mathscr{D}$ the set of defined symbols) with the property that every left-hand side $f(l_1,\ldots,l_n)$ of a rule in R satisfies $f \in \mathscr{D}$ and $l_1,\ldots,l_n \in T(\mathscr{C},\mathscr{X})$. A constructor system is then specified by a triple $(\mathscr{D},\mathscr{C},R)$.

Let $l \to r$ and $s \to t$ be two rewrite rules (we assume that the variables of $s \to t$ were renamed so that there is no common variable with $l \to r$), p the position of a non-variable subterm of s, and μ a most general unifier of $s|_p$ and l. Then $(t\mu, s\mu[r\mu]_p)$ is a *critical pair* formed from those rules. Note that $s \to t$ may be a renamed version of $l \to r$. In this case a superposition at the root position is not considered a critical pair.

A term rewriting system R is

- *confluent* if $t \to^* u$ and $t \to^* v$ implies $u \to^* s$ and $v \to^* s$ for some s,
- *terminating* (or *strongly normalising*) if all reduction sequences are finite,
- *left-linear* if all left-hand sides of rules in R are linear,
- *non-overlapping* if there are no critical pairs,
- *orthogonal* if it is left-linear and non-overlapping,
- *non-duplicating* if for all $l \to r \in R$ 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.

2.1.2. Shared rewriting

Many implementations of term-rewriting 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*; it is a particular case of term graph rewriting (see [3, 16]) where graphs are acyclic. In a shared-reduction step, subterms that correspond to 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.

Formally, in order to define shared-reductions we use marked terms to represent dags, and we define a rewrite relation on marked terms that corresponds to dag rewriting (for more details see [21]).

Definition 2.3. Consider a countably infinite set M of objects called *marks* (M will usually be the set of integers). Let $\mathcal{F}^* = \{f^m \mid f \in \mathcal{F} \text{ and } m \in M\}$ be the set of *marked function symbols*. Similarly, the set of *marked variables* is denoted $\mathcal{X}^*$. We define $mark(X^m) = m$ for $X^m \in \mathcal{F}^* \cup \mathcal{X}^*$. The elements of $T(\mathcal{F}^*, \mathcal{X}^*)$ are called *marked terms*.

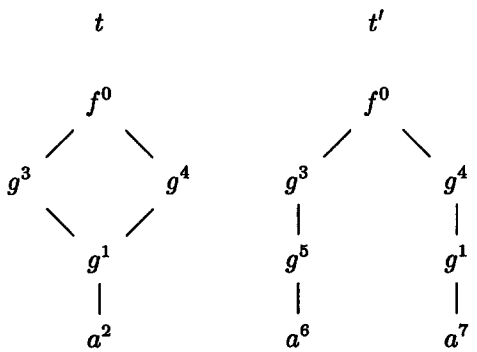
A term $t \in T(\mathcal{F}^*, \mathcal{X}^*)$ is *well-marked* (or a dag) if for every pair of subterms t_1, t_2 of t, $mark(root(t_1)) = mark(root(t_2))$ implies $t_1 \equiv t_2$. The subset of well-marked terms of $T(\mathcal{F}^*, \mathcal{X}^*)$ is denoted by $\mathcal{D}(\mathcal{F}^*, \mathcal{X}^*)$.

Well-marked terms correspond to dags as follows: A marked symbol x^m in a marked term t corresponds to a node m labeled by x in the dag. If two subterms t_1, t_2 of t have the same mark at the root they must be identical, because there is only one subgraph in the dag for t_1 and t_2.

In a dag t, for each mark m occurring in t there is a unique subterm s of t that satisfies $mark(root(s)) = m$, which is denoted by $t@m$. Note also that t is well-marked

if and only if all its subterms are well-marked. Two occurrences s_1, s_2 of a subterm of a well-marked term t are *shared* in t if and only if $mark(root(s_1)) = mark(root(s_2))$.

Example 2.4. Let f, g, a be function symbols of arity $2, 1, 0$, respectively. The marked term $t \equiv f^0(g^3(g^1(a^2)), g^4(g^1(a^2)))$ is well-marked, and both occurrences of the subterm $t@1 \equiv g^1(a^2)$ of t are shared. Also shared are the occurrences of $t@2 \equiv a^2$. The term $t' \equiv f^0(g^3(g^5(a^6)), g^4(g^1(a^7)))$ is also well-marked, but there are no shared subterms in it. The graphical representations of t and t' are shown in the following diagram:



In order to define shared-rewriting we need the notion of marked substitution and marked rewrite rule. Let e be a function that erases all the marks in a marked term (we denote by $e(t)$ the unmarked term obtained from t by erasing all the marks), and let us say that $t \sim^e t'$ (for $t, t' \in T(\mathscr{F}^*, \mathscr{X}^*)$) if $e(t) = e(t')$. A *marked-substitution* σ is a substitution in $T(\mathscr{F}^*, \mathscr{X}^*)$ such that for all $x_1, x_2 \in \mathscr{X}^*$, if $e(x_1) = e(x_2)$ then $x_1\sigma \sim^e x_2\sigma$. A *marked rewrite rule* is a rewrite rule $l^* \to r^*$ such that l^* and r^* are marked terms with the usual convention: $l^* \notin \mathscr{X}^*$, and $\mathscr{V}ar(r^*) \subseteq \mathscr{V}ar(l^*)$. This implies that the number of copies of a (marked) variable *in the graph* corresponding to r^* is less than or equal to the number of copies of that variable *in the graph* corresponding to l^*.

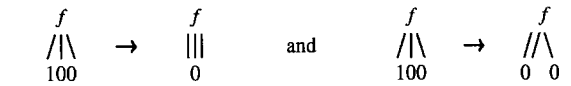
A marked rewrite rule $l^* \to r^*$ is a *marked version* of a rewrite rule $l \to r$ if $e(l^*) = l$ and $e(r^*) = r$.

Now we can define the *shared-reduction relation* $\to_R^s$ induced by a rewriting system R on the set $T(\mathscr{F}^*, \mathscr{X}^*)$ of marked terms:

Definition 2.5. Let t be a marked term, $l^* \to r^*$ a marked version of the rule $l \to r \in R$ and σ a marked substitution. Let $p_1, \ldots, p_n$ be the set of positions in t such that $t|_{p_i} = l^*\sigma$, and let t' be $t[r^*\sigma]_{p_1} \ldots [r^*\sigma]_{p_n}$. Then $t \to^s_{l \to r} t'$. Remark that all occurrences of $l^*\sigma$ in t are reduced simultaneously.

Example 2.6. Consider the rewrite rule: $f(1, x, x) \to f(x, x, x)$. The term $f^1(1^2, 0^3, 0^4)$ reduces to $f^1(0^4, 0^4, 0^4)$ using the marked version $f^1(1^2, x^3, x^4) \to f^1(x^4, x^4, x^4)$, and

to $f^1(0^3,0^3,0^4)$ using $f^1(1^2,x^3,x^4) \to f^1(x^3,x^3,x^4)$. The corresponding dag reductions are:



This example shows that we can choose the degree of sharing of variables at each reduction step, within certain limits: the number of copies of a variable in a rule cannot increase. However, in the following when referring to the shared-rewrite relation $\to_R^s$ induced by a rewrite system R, we will assume maximal sharing of variables, that is, all the occurrences of the same variable in the right-hand side are shared.

It is interesting to note that well-markedness may be lost if arbitrary markings are used in rules (for instance, consider a rule $a^1 \to b^2$ and a rewrite step $f^2(a^1) \to f^2(b^2)$). A simple sufficient condition for preserving well-markedness [21] is to assume that in $t \to_{l\to r}^s t' \equiv t[r^*\sigma]_{p_1} \ldots [r^*\sigma]_{p_n}$ r^* is fresh with respect to t, that is, all the marks in r^* are different from those in t.

It is easy to see that every shared-rewriting sequence corresponds to a term rewriting sequence, but the converse does not hold in general. However, for orthogonal systems every term reduction sequence can be extended to a sequence which does correspond to a shared-rewriting sequence. This is a consequence of a general theorem by Kennaway et al. [16] showing the adequacy of graph rewriting for simulating term rewriting in the case of orthogonal systems.

Property 2.7. *If R is an orthogonal term-rewriting system then for any rewriting sequence*

$$t_0 \to t_1 \to \cdots \to t_n$$

there exists a shared-rewriting sequence

$$t_0' \to^{s^*} t_1' \to^{s^*} \cdots \to^{s^*} t_n'$$

such that $t_0 = e(t_0')$ and for all i, $t_i \to^* e(t_i')$.

Proof. In [16] it is shown that if an orthogonal graph-rewriting system G is finitary (i.e. contains only finite graphs) and acyclic (i.e. graphs in rewrite rules do not contain cycles), and a term-rewriting system R is obtained by unravelling of G (unravelling, in the case of dags, is just the function that erases all the marks) then the unravelling mapping from G to R is adequate.

A shared-rewrite system defined by an orthogonal term-rewriting system R satisfies these conditions. Moreover, adequacy (see [16]) implies, in particular, that if $e(t_0') \to^* t_i$, then there exists t_i' such that $t_0' \to^{s*} t_i'$ and $t_i \to^* e(t_i')$. □

2.1.3. Combinatory reduction systems

Combinatory reduction systems were designed by Klop [18] with the aim of combining the usual first-order term rewriting systems with the presence of bound variables as in the λ-calculus.

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

1. *variables* $x, y, z, \ldots$,
2. *metavariables* Z_i^k each with a fixed arity, where k is the arity of Z_i^k (indexes are often omitted),
3. *function symbols* $f, g, \ldots, F, G, \ldots$, each with a fixed arity,
4. a binary operator $[\cdot]\cdot$ for *abstraction* over variables,
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:

Definition 2.8. The set *MTerms* of *metaterms* over an alphabet Σ is defined as the smallest set such that

- $x \in MTerms$ for every variable x,
- if x is a variable and $s \in MTerms$ then $[x]s \in MTerms$,
- if $s_1, \ldots, s_n \in MTerms$ and f is a function symbol of arity n then $f(s_1, \ldots, s_n) \in MTerms$,
- if $s_1, \ldots, s_n \in MTerms$ and Z^n is a metavariable, then $Z^n(s_1, \ldots, s_n) \in MTerms$.

In the last two cases $n \geqslant 0$, and if $n = 0$ we omit the brackets as usual.

Terms are metaterms that do not contain metavariables.

Variables that are in the scope of the abstraction operator are *bound*, and free otherwise. A (meta)term is *closed* if every variable occurrence is bound. As in the λ-calculus, naming problems can arise. We adopt the usual convention: all bound variables are chosen to be different from the free variables.

We define now the rewrite rules of combinatory reduction systems, which are pairs of metaterms (but they induce a reduction relation on terms, by assigning terms to metavariables as explained below).

Definition 2.9. 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, and 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.

Example 2.10. The β-reduction rule for the λ-calculus is written in the syntax of combinatory reduction systems as

$$\mathsf{app}(\mathsf{lambda}([x]Z(x)), Z') \to Z(Z')$$

where the binary function symbol app represents application and the unary function symbol lambda represents λ-abstraction. Z is a unary metavariable, and Z' a nullary metavariable.

The metavariables in metaterms can be thought of as holes that must be instantiated by terms. In other words, rules act as schemes defining a reduction relation on terms. To extract the actual rewrite relation on terms from the rewrite rules, each metavariable is replaced by a special kind of λ-term, and in the obtained term all β-redexes and the residuals of these β-redexes are reduced (i.e. a development is performed). This operation is well-defined in the λ-calculus since all developments are finite. Formally, to define the rewrite relation we have to consider a notion of substitution using *substitutes* and *valuations*.

Definition 2.11. 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 ($n \geqslant 0$). It can be applied to an n-tuple $s_1,\ldots,s_n$ of terms, and the result is the term t where $x_1,\ldots,x_n$ are simultaneously replaced by $s_1,\ldots,s_n$.

A *valuation* σ is a map that assigns an n-ary substitute to each n-ary metavariable. This is extended to a mapping from metaterms to terms: given a valuation σ and a metaterm t, first we replace all metavariables in t by their images in σ and then we perform the developments of the β-redexes created by this replacement. When making a substitution, we must take care of bound variables as usual.

We can now define the rewrite relation on terms:

Definition 2.12. A *context* is a term with an occurrence of a special symbol [ ] called a hole. A *rewrite step* is defined as follows: if $l \to r$ is a rewrite rule, σ a valuation, and $C[\,]$ a context, then $C[l\sigma] \to C[r\sigma]$.

As an example, we show the rewrite step that corresponds to β-reduction according to the rule given in Example 2.10:

Example 2.13. Let σ be the valuation that maps Z to $\lambda z.f(z,g(z))$ and Z' to the term y. We apply now σ to the left-hand side of the rule given in Example 2.10:

$$\begin{aligned}\mathsf{app}(\mathsf{lambda}([x]Z(x)),Z')\sigma &= \mathsf{app}(\mathsf{lambda}([x](\lambda z.f(z,g(z)))(x)),y)\\ &= \mathsf{app}(\mathsf{lambda}([x]f(x,g(x)),y))\end{aligned}$$

The application of σ to the right-hand side gives

$$\begin{aligned}Z(Z')\sigma &= (\lambda z.f(z,g(z)))(y)\\ &= f(y,g(y))\end{aligned}$$

Hence, according to the previous definition, there is a rewrite step

$$\mathsf{app}(\mathsf{lambda}([x]f(x,g(x)),y)) \to f(y,g(y)).$$

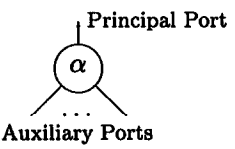
A combinatory reduction system is *left-linear* if it does not contain a left-hand side in which some metavariable has multiple occurrences. It is *non-overlapping* if whenever an instance t of a left-hand side l contains a reducible (strict) subterm u, then u is contained in one of the instantiated metavariables of l. It is *orthogonal* if it is left-linear and non-overlapping.

2.2. Interaction nets

The interaction net paradigm was introduced by Lafont in [22] as a new rewriting framework for programming, founded on proof nets of linear logic [12]. These nets are very appealing from a computational point of view. On the one hand, they are a very simple, graphical-rewriting system which enjoys properties such as confluence, and on the other they bring out the parallelism in the rewriting process making them well-suited as a basis for parallel implementations. Here we will briefly review the paradigm. The reader will find additional examples in the articles by Lafont [22] and Gay [11].

Definition 2.14. An interaction net (Σ, IR) is specified by the following data:

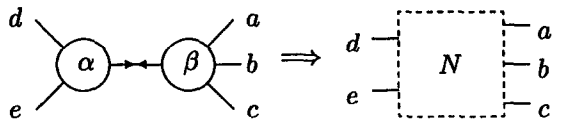
- A set Σ of *symbols* (or agents), each characterised by a label, and an arity $n \in \mathbb{N}$ $(n \geqslant 1)$ which is the number of ports it has. Each agent has a distinguished port, called the principal port, where interaction can take place. All the other ports are called *auxiliary* ports. Agents are represented graphically as follows, where we indicate the principal port by an arrow in conformity with Lafont; note that ports are ordered.



  A net on Σ is an undirected graph whose vertices are agents in Σ, and whose edges join different ports in the same or in different agents. The words agent and node are often used as synonyms. A net may be empty, or consist just of edges without agents. Ports that are not connected to other ports in the net are called free. Free ports are marked with edges that have a free extreme, as in the diagram above. Then each node has as many incident edges as the arity of the agent.
  The interface of a net is the (ordered) set of free extremes of edges (in particular, for a net consisting only of an edge, the interface contains the two extremes of the edge).
- A set IR of *interaction rules* which are net rewriting rules where the left-hand side is a net consisting of two agents connected on their principal ports, and the right-hand

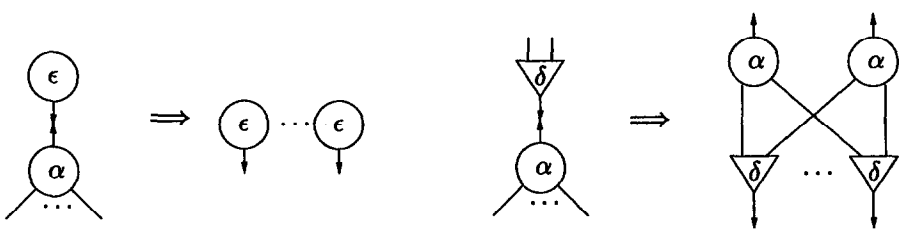
side is an arbitrary net with the only constraint that it must have the same interface as the left-hand side. There is at most one rule for each pair of agents.

The following diagram shows the general form of an interaction rule, using agents α and β of arity 3 and 4, respectively. The right-hand side N is any net, which may contain occurrences of the agents in the left-hand side (we represent nets with dashed lines). Note that *the interface* is preserved; there are equal numbers of free ports before and after the interaction. We use names (a,b,c,d,e) to indicate the correspondence between the free ports in the left- and right-hand sides of the rule, but we will often omit them when there is no ambiguity.



A net rewrite step on a net W, called an *interaction*, replaces in W a pair of agents connected on their principal ports (i.e. an occurrence of a left-hand side of an interaction rule) by the corresponding right-hand side, plugging the edges in the interface of the right-hand side to the corresponding ports in W. We write interactions as $W \Rightarrow W'$.

To give an example of interaction nets, we show the interaction rules of two ubiquitous agents, namely the *erase* (ε), of arity 1, which deletes everything it interacts with, and the *duplicator* (δ), of arity 3, which copies everything. These are represented by the following diagrams, where α is any node:



The first rule shows that the interaction deletes the node α and places erase nodes on all the free edges of the node. For the second rule, we see that the α node is copied, and all its free edges are too. Further examples of interaction nets are given throughout the paper, making use of the above agents and rules.

A net is in *normal form* when no interactions are possible. We say that an interaction net (Σ, IR) is *terminating* if all sequences of interactions in nets on Σ are finite.

As an almost immediate consequence of this definition of net rewriting we have the following features:

- *Confluence.* The restriction of interaction only on the principal port of an agent, and the constraint that there is at most one rule for each pair of agents, suffice to give

the strongest notion of confluence: if $N \Rightarrow N_1$ and $N \Rightarrow N_2$ (N_1 different from N_2) then there exists a net N_3 such that $N_1 \Rightarrow N_3$ and $N_2 \Rightarrow N_3$.

- *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 network.
- *Asynchrony.* As a direct consequence of the above points, we have the possibility of a parallel implementation – no order on the interactions is required since any two agents ready to interact can do so in any order.

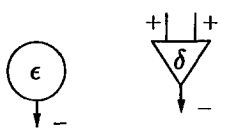
Interaction nets can be regarded as a generalisation of proof nets for multiplicative linear logic, and indeed, this is their origin. Roughly, the relationship is given by setting the set of symbols Σ to be the logical symbols; the principal port of each symbol is the conclusion and the auxiliary ports are the premises of the rule for that symbol; and the rewrite rules are specified by the rules for *cut-elimination* for multiplicative linear logic. We refer the reader to [23] for a complete presentation; see also [25] for another approach.

In the following we will use *interaction nets* or *interaction net systems* as synonyms.

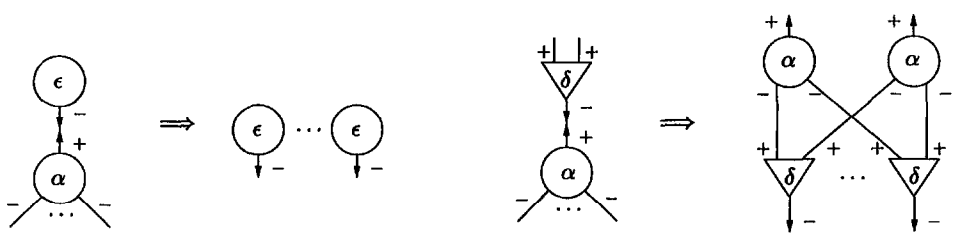
2.2.1. Classification of nets

Lafont [22] introduced a type discipline for interaction nets, using a set of constant types (atom, nat, list-nat, ...). For each agent, ports are classified between *input* and *output*. An input port will be assigned a type τ^- and an output port a type τ^+. 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 $+$).

For example, for the agents ε and δ we consider the following typings:



Assuming α has a positive principal port, the nets in the interaction rules above can be typed as follows:



Definition 2.15. Agents can be divided into *constructors* and *destructors*: if the principal port of an agent is an output port, the agent is a constructor, otherwise it is a destructor.

In the previous example, ε and δ are destructors, whereas α is a constructor.

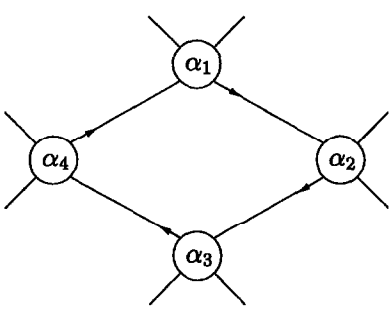
The division between constructors and destructors originates in the logical system that inspired the formalism of interaction nets: destructors and constructors are respectively associated with left and right introduction rules of logical operators.

For each agent, the auxiliary ports are divided into *partitions* (the notion of partition has also its origins in the sequent calculus that inspired the formalism).

Definition 2.16. Each agent $\alpha \in \Sigma$ has a principal port and a (possibly empty) set of auxiliary ports which are 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.

The partitions given by Lafont for ε and δ are as follows: ε, which does not have any auxiliary port, has one partition, which is empty (see [2] for a detailed discussion of the meaning of empty partitions); for δ both auxiliary ports are in the same partition (δ has one partition containing two output ports).

The notion of partition was introduced in [22] with the purpose of defining a class of interaction nets, called *semi-simple nets*, that are deadlock free, that is, a class of nets such that vicious circles of principal ports, as depicted in the diagram below, cannot be created during computation:

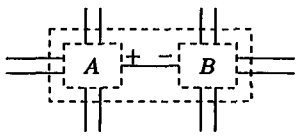


Definition 2.17. A net is called *semi-simple* if it can be constructed using only the following operations:

1. LINK, which builds an edge:

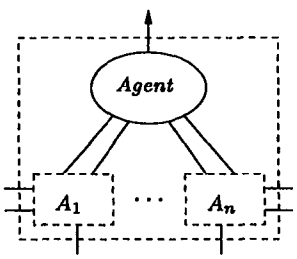
As mentioned before, an edge is a particular case of net. It can be typed by assigning opposite signs to the extremes.

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



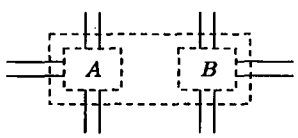
In particular, when A and B are just an edge, we obtain an edge.

3. GRAFT, that adds an agent to a set of nets according to its partitions. The principal port remains free, and all the ports belonging to the same partition of the agent are connected to the same net, but each partition is connected to a different net, as shown in the following diagram:



We assume that the agent has n partitions, where $n \geqslant 0$. $A_1, \ldots, A_n$ are semi-simple nets, with interfaces such that each auxiliary port in the ith-partition of the agent can be connected to the corresponding net A_i (in particular these nets may be just edges, and may contain more free ports in the interface, that will remain free after the GRAFT is made). Types have to be respected in order to obtain a well-typed net.

4. MIX, juxtaposing two nets:



5. and EMPTY, which constructs an empty net.

A semi-simple net is then defined by a sequence of operations, and in the following we assume that they are well-typed.

For example, the nets in the interaction rule for ε are semi-simple: assume that the partitions of α are unitary (i.e. contain only one port), then the left-hand side is obtained by making a CUT of the nets GRAFT(ε, EMPTY) and GRAFT(α, LINK,..., LINK), and the right-hand side is obtained by making MIX of GRAFT(ε, EMPTY),..., GRAFT(ε, EMPTY).

In fact, the construction of the left-hand side of an interaction rule can always be done in this way, as the following lemma shows.

Lemma 2.18. *Left-hand sides of interaction rules are always built by a CUT of two GRAFTS on LINKS (possibly combined by MIX) or EMPTY nets.*

Proof. By definition of interaction rule, the left-hand side is a net consisting of a pair of agents connected on their principal ports. Hence it is a net built by a CUT of two nets that consist just of one node each. Then these subnets are built by GRAFTS made on LINKS, combinations of LINKS by MIX, or EMPTY nets, according to the arities and partitions of the agents. □

Interaction nets where the partitions of all agents are unitary are called *discrete*. As remarked by Lafont [22], in the discrete case a net is semi-simple if it is a graph without cycles; and if the operations of MIX and EMPTY are not used, then it is a connected graph without cycles (i.e. a tree). In the general case, a semi-simple net contains no vicious circle.

A rule is semi-simple if when free ports have been grouped according to the partitions in the left member, the right member becomes semi-simple. For example, the rules for ε and δ are semi-simple. Semi-simple nets are closed under reduction by semi-simple rules.

Following Laneve [24], if a negative port (i.e. an input port) exists in a partition we will call it an *input partition*, otherwise it will be called an *output partition*. Hence, an input partition may contain some output ports, whereas an output partition contains only output ports. According to this, there are two classes of interaction nets:

1. *dependent interaction nets*: if a positive port appears in an input partition of some agent,
2. *non-dependent interaction nets*: if every agent has only negative ports in input partitions.

3. From interaction nets to term-rewriting systems

In this section we study the encodings of interaction nets into term rewriting systems. First we consider semi-simple nets, and then the general case of interaction nets.

3.1. Translation of semi-simple nets

It is known that *discrete* semi-simple interaction nets correspond to (a restricted class of) first-order term rewriting systems (see [22, 24]). 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) are mapped to first-order term-rewriting systems. In the discrete case we obtain a linear first-order term-rewriting system.

As in [24], we will assume that constructors do not have any output partition (the principal port is an output port, and they may have output ports in input partitions). Destructors may have one output partition like in the case of δ, or none, like in the case of ε. Summarising, we assume that every agent has at most one output partition and a number (maybe 0) of input partitions, that may or may not contain positive ports. These assumptions allow us to give a smoother translation of semi-simple nets (dependent or non-dependent) into term-rewriting systems. Of course, this is a restriction on the class of nets under study, but this class is sufficiently rich in that it can capture all computable functions (and includes the whole class of interaction systems [24]).

We consider first the case of non-dependent semi-simple nets and then generalise the translation to deal with output ports in input partitions (dependent nets).

3.1.1. Non-dependent nets

We start by defining a mapping

$$\Theta : Nets \to T(\mathscr{F}, \mathscr{X})$$

which takes a non-dependent semi-simple net and gives a term. More precisely, since a semi-simple net is defined by a sequence of operations of LINK, CUT, GRAFT, MIX, EMPTY, Θ takes as input a sequence of operations that build a 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.

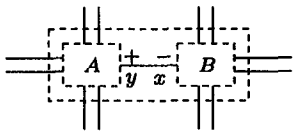
The translation of a net with an interface consisting of inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_m$, will be a term $t[x_1, \ldots, x_n]$ representing an m-tuple. The translation function Θ is defined by induction on the definition of the semi-simple net.

Definition 3.1. Let (Σ, IR) be an interaction net, and $\mathscr{F}$ be a set of function symbols containing the agents in Σ, the constant empty, a binary symbol P for pair formation, and unary symbols π_i for projections (we assume that Σ does not contain agents called P, π_i, 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 abbreviations: given a term $t \equiv P(t_1, \ldots, t_n)$, $\pi_i(t) = t_i$ and $t - \pi_i(t) = P(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n)$. In other words, the expressions of the form $\pi_i(P(t_1, \ldots, t_n))$ and $P(t_1, \ldots, t_n) - \pi_i(P(t_1, \ldots, t_n))$ used in the metalanguage have to be replaced by the corresponding definition (the terms t_i and $P(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n)$, respectively). Here we assume that when only one element remains, $P(t_1) = t_1$. A sequence $u_1, \ldots, u_n$ will be abbreviated by $\vec{u}$.

1. The translation of an empty net is simply the constant empty: $\Theta(EMPTY) = \text{empty}$.
2. The translation of a link is a variable: $\Theta(LINK) = x$. 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: $\Theta(MIX(A,B)) = P(\Theta(A), \Theta(B))$.

Note that A and B may in turn be translated as pairs, which means that we can obtain tuples of arbitrary length. When $\Theta(N)$ is a tuple, we assume (without loss of generality) that $\pi_i(\Theta(N))$ corresponds to the ith output in the interface of N.

4. The translation of a net constructed from two nets A, B by a CUT operation connecting y and x:



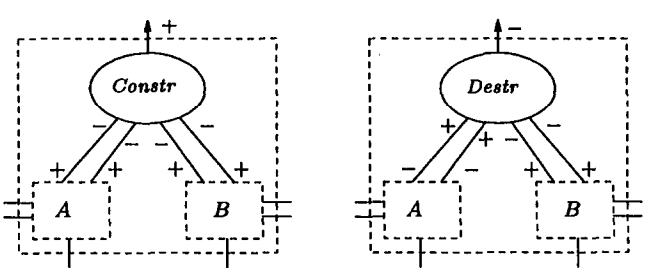
is: $\Theta(CUT(A,B)) = P(\Theta(B)\{x \mapsto \pi_i(\Theta(A))\}, \Theta(A) - \pi_i(\Theta(A)))$, where we assume that y is the ith output port in the interface of A, and x is the input port in B connected to y.

Note that in this formula we used a metalanguage with substitution and abbreviations like $\Theta(A) - \pi_i(\Theta(A))$. In fact, $\Theta(CUT(A,B))$ is the term that we obtain from the expression $P(\Theta(B)\{x \mapsto \pi_i(\Theta(A))\}, \Theta(A) - \pi_i(\Theta(A)))$ after making the operations in the metalanguage (substitution and replacement of abbreviations by their definitions).

In the formula above, we have taken into account the fact that A can be a net with multiple output ports. In that case the translation of A will be a tuple, and to obtain the translation of the CUT we have to select the corresponding element $\pi_i(\Theta(A))$ to "plug" in the translation of B. The rest of the outputs of A (i.e. $\Theta(A) - \pi_i(\Theta(A))$) are still outputs of the net resulting from the CUT (recall that even if A has only one output port, we use the notation $\pi_i(\Theta(A))$ with $i = 1$, identifying a tuple of length one with its only element).

In the discrete case we have: $\Theta(CUT(A,B)) = \Theta(B)\{x \mapsto \Theta(A)\}$ as particular case of the previous formula.

5. The translation of a $GRAFT$ depends on whether the agent that is added is a constructor or a destructor (to simplify the formulas, we will assume that the agent has two partitions; the generalisation is straightforward). The result of grafting the agent is a net of one of these forms:



To define the translation of a $GRAFT$, we consider these two cases separately.

(a) If the agent is a constructor:

$$\Theta(GRAFT(Constr,A,B)) = P(Constr(\overrightarrow{\pi_i(\Theta(A))}, \overrightarrow{\pi_i(\Theta(B))}), \Theta(A) - \overrightarrow{\pi_i(\Theta(A))}, \Theta(B) - \overrightarrow{\pi_i(\Theta(B))}).$$

In this formula we have taken into account the fact that, according to our assumptions, 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

$$\Theta(GRAFT(Constr,A,B)) = Constr(\Theta(A),\Theta(B)).$$

If the constructor has arity one, then we simply obtain $\Theta(GRAFT(Constr)) = Constr$.

(b) If the agent is a destructor:

$$\Theta(GRAFT(Destr,A,B)) = P(\Theta(A)\{y_j \mapsto \overrightarrow{\pi_i(Destr(x, \overrightarrow{\pi_i(\Theta(B))}))}\}, \Theta(B) - \overrightarrow{\pi_i(\Theta(B))}).$$

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

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

In the degenerate case of a destructor without output ports (see Example 3.2, part 2 below) we apply the same formula with a dummy output, in other words, we take

$$\Theta(GRAFT(Destr,B)) = P(Destr(x, \overrightarrow{\pi_i(\Theta(B))}), \Theta(B) - \overrightarrow{\pi_i(\Theta(B))}).$$

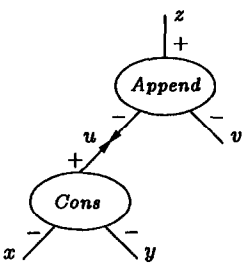
The translation function extends to interaction rules in the natural way, i.e. by applying Θ to each side of the rule (giving consistent variable names to the edges in both sides). The translation of an interaction net system (Σ, IR) is a term-rewriting system on the signature $\mathscr{F}$ defined above, with a set $\Theta(IR)$ of rewrite rules that contains the translations of the rules in IR and the additional rules $\Pi = \{\pi_i(P(x_1,\ldots,x_n)) \to x_i\}$ defining the projections.

In the definition of the translation function Θ we did not consider the case of a CUT of two LINKS (since it gives again a LINK).

As we already mentioned, our translation function takes as input a sequence of operations that construct a semi-simple net and gives a term as result. 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. This, however, does not cause any problem since the properties of the translation function that we will show below do not depend on the particular sequence of operations used to build a net.

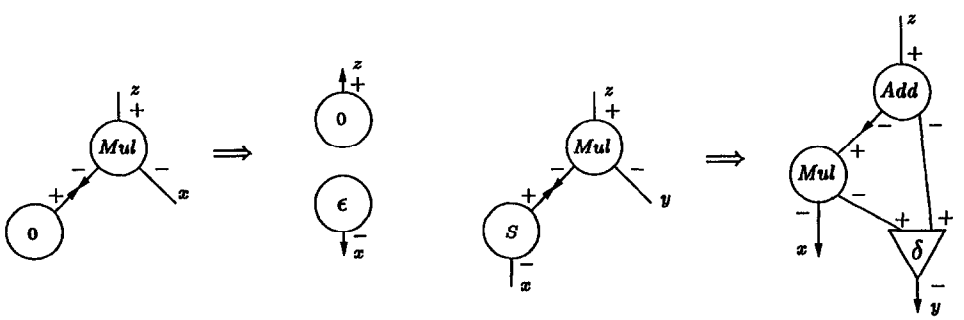
We now give some examples to illustrate the definition of Θ.

Example 3.2. (1) Consider 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 $Append(u,v)\{u \mapsto Cons(x,y)\} \equiv Append(Cons(x,y),v)$.

(2) For an example with a non-discrete net, consider the system of interaction rules to add and multiply natural numbers given in [22]. The agents S and 0 are constructors, and Add and Mul 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 erasing and copying (duplication), which is done with the agents ε and δ. We show only the interaction rules for multiplication:

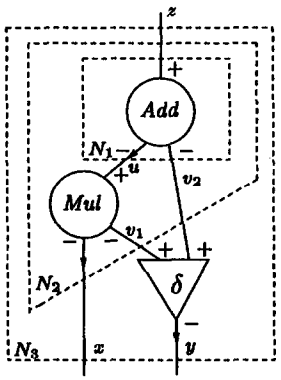


The translations of the terms in the left-hand sides of the rules are similar to the previous example. More interesting is the translation of the right-hand sides. For the first rule, the net in the right-hand side is the result of a *MIX* operation on the nets obtained by grafting 0 and ε. Formally,

$$\Theta(MIX(GRAFT(0), GRAFT(\varepsilon, EMPTY))) = P(0, \varepsilon(x)).$$

Note that since ε is a destructor without output, we used the special case of the formula in Definition 3.1, part 5b.

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 Θ:

$$\Theta(N_1) = Add(u, v_2),$$
$$\Theta(N_2) = \Theta(N_1)\{u \mapsto Mul(x, v_1)\} = Add(Mul(x, v_1), v_2),$$
$$\Theta(N_3) = \Theta(N_2)\{v_1 \mapsto \pi_1(\delta(y)), v_2 \mapsto \pi_2(\delta(y))\}$$
$$= Add(Mul(x, \pi_1(\delta(y))), \pi_2(\delta(y))).$$

In this example we can see that if the interaction net is not discrete then a variable can occur more than once in the image of Θ. In this case shared-rewriting mimics reductions on the net (see remark below).

Theorem 3.3. *Let (Σ, IR) be a non-dependent semi-simple interaction net. The term rewriting system $\Theta(IR)$ on $\mathcal{F}$ is a constructor system, and it is orthogonal.*

Proof. It is easy to see that, by Lemma 2.18, the translation of the left-hand side of an interaction rule is a term of the form $\alpha(\beta(x_1, \ldots, x_n), y_1, \ldots, y_m)$ (recall that P and π_i in the definition of Θ are part of the metalanguage). The system $\Theta(IR)$ contains the translations of the interaction rules IR, and the projection rules Π. Since the translations of left-hand sides of interaction rules are linear terms, the system is left-linear. Moreover, since they do not contain P and π_i, and since each agent is either a constructor or a destructor, and there is at most one interaction rule for each pair of agents, we obtain a constructor system without critical pairs. □

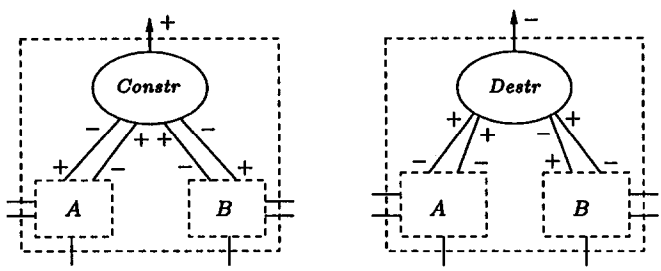
Remark 3.4. We can define in the same way a translation function Θ^* from nets to marked terms (dags), and consider the translation of an interaction net (Σ, IR) as a shared-rewriting system $\Theta^*(IR)$. The definition of Θ^* is similar to that of Θ, but in the cases of *CUT* and *GRAFT* we have to add marks so that all occurrences of $\Theta(A)$ and $\Theta(B)$ are shared.

Next we consider the general case of semi-simple nets, which includes dependent nets.

3.1.2. *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 Θ, although its image now will be *MTerms*, the set of metaterms associated to the signature $\mathcal{F}$ of Definition 3.1, 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 3.5. The translation of a *GRAFT* depends as before on whether the agent that is being added is a destructor or a constructor. Again, to simplify the formulas we will assume that the agent has two partitions. There are two cases:



1. If it is a constructor:

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

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 of the tuples.

2. If it is a destructor, we have to consider that the input partition of *Destr* connecting with B can contain positive ports. Then the translation of the *GRAFT* is

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

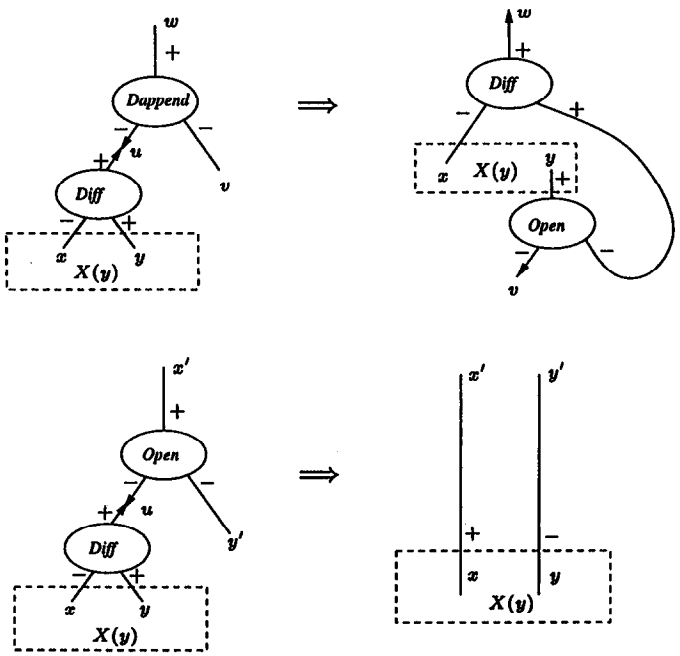
where, as in Definition 3.1, x represents the principal port of *Destr*, and $\vec{y}$ are the input ports of A connected with the destructor. The variables $\vec{z}$, representing the negative ports in B that are connected to the destructor, will typically appear in $\Theta(B)$.

Note that the formulas for the translation of non-dependent nets can be obtained as a particular case of these when no positive port appears in an input partition (hence no bound variables).

Next we give an example to illustrate the definition of Θ in the case of dependent nets.

Example 3.6. Let us recall the interaction rules for appending difference lists given by Lafont [22]. The agents are *Cons*, *Diff*, *Dappend*, and *Open*, where *Cons* and *Diff* are the constructors and *Dappend* and *Open* the destructors. *Diff* has two auxiliary ports in the same partition: one is positive and the other negative (hence the system is dependent).

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 $X(y)$). *Open* has two partitions, with one port each.

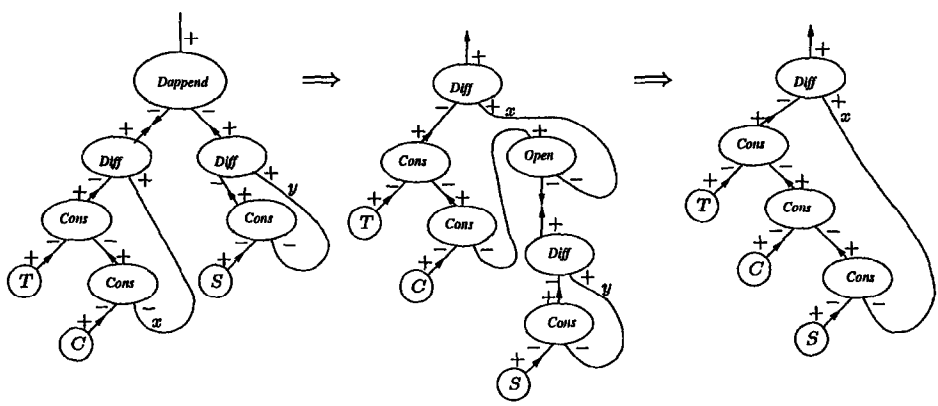
Let us translate the first interaction rule. The net in the left-hand side is obtained by a *CUT* of two *GRAFTS*, one translated as $Dappend(u,v)$ and the other as $Diff([y]X(y))$. The net in the right-hand side is the result of the composition of two *GRAFTS*. The first one is translated as $X(Open(v,z))$, and its composition with the second, using the constructor case of Definition 3.5, gives $Diff([z]X(Open(v,z)))$. Then the translation of the first rule according to Definition 3.5 is the combinatory reduction rule

$$Dappend(Diff([y]X(y)),V) \to Diff([z]X(Open(V,z))).$$

In the same way the translation of the second rule is

$$Open(Diff([y]X(y)),Y') \to X(Y').$$

There is a correspondence between sequences of interactions in a net and sequences of rewrite steps in its translation. For example, the sequence



is translated as

$$Dappend(Diff([x]Cons(T,Cons(C,x))),Diff([y]Cons(S,y)))$$
$$\to Diff([x]Cons(T,Cons(C,Open(Diff([y]Cons(S,y)),x))))$$
$$\to Diff([x]Cons(T,Cons(C,Cons(S,x))))$$

which is a rewrite sequence using the combinatory reduction rules above.

Note again that the translation of an interaction net system is an orthogonal system.

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

Theorem 3.7. *Let N,N' be nets in a semi-simple interaction net system. If $N \Rightarrow N'$ then $\Theta(N) \to^{+} \Theta(N')$. If N is a normal form, then so is $\Theta(N)$.*

Proof. By induction on the definition of the semi-simple net N.

1. If $N = EMPTY$ or $N = LINK$ then N is a normal form, and so is $\Theta(N)$.
2. If $N = MIX(A, B)$ then $\Theta(N) = P(\Theta(A), \Theta(B))$, and the properties follow by induction.
3. If $N = CUT(A, B)$ then $\Theta(N) = P(\Theta(B)\{x \mapsto \pi_i(\Theta(A))\}, \Theta(A) - \pi_i(\Theta(A)))$.

   The reductions inside A and B are preserved by induction. If there is an interaction between the agent in A and the agent in B connected by the CUT (in this case these agents will be replaced by a new net M according to the right-hand side of the interaction rule), then $\Theta(N)$ contains a redex: the translation of the interaction rule can be applied to $\Theta(B)\{x \mapsto \pi_i(\Theta(A))\}$, and the redex will be replaced by $\Theta(M)$ with the corresponding substitution. This results in a term t that coincides with $\Theta(N')$ modulo $\rightarrow_\Pi$, more precisely: $\Theta(N) \rightarrow t \rightarrow_\Pi^* \Theta(N')$. Hence $\Theta(N) \rightarrow^+ \Theta(N')$.

   If N is a normal form, so are A and B, and hence $\Theta(A)$ and $\Theta(B)$ are in normal form by induction. Moreover, if the CUT does not create a redex in the net N, then no rewrite rule applies to $\Theta(N)$ either (because all the rewrite rules that correspond to interaction rules cannot be applied, and the projections do not apply because they were already applied in the translation function: they are in the metalanguage).
4. If $N = GRAFT(Constr, A, B)$ or $N = GRAFT(Destr, A, B)$ then the property follows by induction, since there is no CUT on the principal ports (the agent being grafted has its principal port free).

Note that in the case of a CUT or a $GRAFT$ 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 $\Theta(N)$. □

Remark 3.8. For shared-rewriting systems we can prove in the same way, using the function Θ^* of Remark 3.4:

If $N \Rightarrow N'$ then $\Theta^*(N) \rightarrow^{s+} \Theta^*(N')$. If N is a normal form, then so is $\Theta^*(N)$.

The proof is similar to the previous one: in the case of a CUT we also have $\Theta^*(N) \rightarrow^s t \rightarrow_\Pi^* \Theta^*(N')$, and the same for a $GRAFT$ (only one step $\rightarrow^s$ suffices because using Θ^* repeated subterms are shared).

As a consequence of this theorem:

Corollary 3.9. *Let (Σ, IR) be an interaction net such that the rewrite system $\Theta(IR)$ on $\mathscr{F}$ terminates. Then (Σ, IR) is terminating.*

The converse also holds, as shown below, because the rewrite systems are left-linear and non-overlapping (which implies that shared-rewriting can mimic standard rewriting). Hence termination of (Σ, IR) implies termination of $\Theta(IR)$. This property will be used in the next section to derive modularity results for interaction nets from the modularity results of term rewriting systems.

Theorem 3.10. *Let (Σ, IR) be a terminating semi-simple interaction net. Then $\Theta(IR)$ terminates.*

Proof. Since modularity of termination of term-rewriting systems has been studied mostly for first-order term-rewriting systems, we prove the theorem for first-order systems (but the proof generalises to combinatory reduction systems as well).

First we show that there is a simple correspondence between a term (or a dag representing a term) and a net. Recall that each function symbol in $\mathscr{F}$ corresponds to an agent in Σ except for P, π_i and *empty*. Given a term $t \in T(\mathscr{F}, \mathscr{X})$ we build a net with a node for each occurrence of a function symbol in t except for P, which is not visible in the net, the projections π_i which select edges in the net, and *empty*, which corresponds to the empty net. To ensure that the net is well-formed in the case of a non-linear term, if a non-linear variable x occurs in a subterm which is not of the form $\delta(x)$, we add duplicator nodes to join the corresponding edges. We denote by N_t the net associated to t.

The term-rewriting system $\Theta(IR)$ is orthogonal by Theorem 3.3, hence, by Property 2.7 we can mimic reductions sequences by shared-reduction ones. We prove by contradiction that $\Theta(IR)$ is terminating:

Let t_0 be a minimal non-terminating term (i.e. all its strict subterms are terminating), and $t_0 \to t_1 \to t_2 \ldots$ an infinite reduction sequence starting from t_0. We consider the corresponding shared-rewriting sequence, and the interaction sequence starting from the net N_{t_0} that corresponds to t_0. This is depicted in the following diagram, where $t_0 \xrightarrow{*}{}^{s} t_1' \xrightarrow{*}{}^{s} t_2' \cdots$ is the shared-rewriting sequence that mimics the sequence $t_0 \to t_1 \to t_2 \cdots$.

$$
\begin{array}{ccccccc}
t_0 & \xrightarrow[l_1 \to r_1]{p_1} & t_1 & \xrightarrow[l_2 \to r_2]{p_2} & t_2 & \xrightarrow[l_3 \to r_3]{p_3} & \cdots \\
 & & \downarrow * & & \downarrow * & & \\
t_0 & \xrightarrow{*}{}^{s} & t_1' & \xrightarrow{*}{}^{s} & t_2' & \xrightarrow{*}{}^{s} & \cdots \\
N_{t_0} & \overset{*}{\Longrightarrow} & N_{t_1'} & \overset{*}{\Longrightarrow} & N_{t_2'} & \overset{*}{\Longrightarrow} & \cdots .
\end{array}
$$

Term-rewriting steps using the projection rules are not visible in the interaction sequence: if $t_1' \to_{\Pi}^{s} t_2'$ then $N_{t_1'} \equiv N_{t_2'}$. However, since t is minimal non-terminating, any infinite reduction sequence starting from t contains infinite rewrite steps at the root position, which correspond to an infinite number of interactions in the sequence that starts from N_t, and this contradicts the assumption of termination of IR.

Hence, termination of IR implies termination of $\Theta(IR)$. □

3.2. Summary

The translation function defined above provides evidence of the following mappings between semi-simple nets and term rewriting systems:

Class of semi-simple nets	*Class of term-rewriting systems*
Discrete	First-order term-rewriting systems (linear and non-overlapping)
Non-Dependent	First-order term-rewriting systems (left-linear and non-overlapping)
Dependent	Combinatory reduction systems (left-linear and non-overlapping)

Note that the systems in the same line in the table are not equivalent: each line shows a strict inclusion of a class of nets in a class of term-rewriting systems (there are for instance orthogonal first-order term rewriting systems that cannot be represented in the interaction framework, like the well-known *BP* function of Berry and Plotkin, which is not sequential).

For non-dependent nets with vicious circles, we could define a translation function in the same spirit as Θ into *infinitary* first-order term rewriting systems (for a survey on infinitary term rewriting systems see [17]). In the same way, arbitrary interaction nets could be coded as infinitary combinatory reduction systems. The details of these translations are beyond the limits of the present paper. However, there is another way of translating arbitrary interaction nets into finitary combinatory reduction systems which we present in the following section.

3.3. Translation of general interaction nets

In this section we show an encoding of arbitrary interaction nets into combinatory reduction systems. It is a syntactic encoding of graphs as metaterms; we apply it in particular to nets. Continuing our abuse of notation, we define a mapping

$$\Theta : Nets \to MTerms$$

in the following way: For each node α 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 ($x, \vec{y}, a, b \in \vec{z}$). The empty net will be represented by $N(empty)$.

Interaction rules are translated as pairs of metaterms, obtained by applying Θ to each member of the rule (and adding a metavariable in the left-hand side so that rules

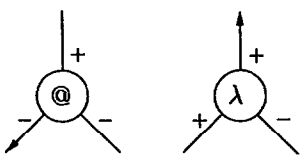
can be applied inside a context). To the resulting system we add the following rule for I:

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

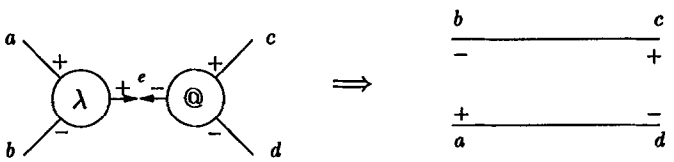
which indicates that I makes a renaming of variables. The symbol N is used only because in a combinatory reduction system the left-hand sides of rules cannot start with an abstraction (see Definition 2.9).

To show how the translation works we consider a simple example.

Example 3.11. The linear λ-calculus is a restriction on the λ-calculus in that variables must occur *exactly* once. There is a simple interaction net system for this which requires two nodes, which we call λ and @, as shown in the following diagram:



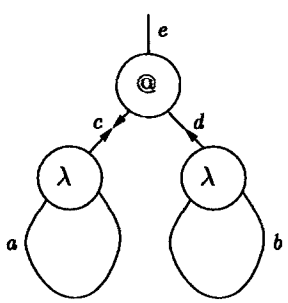
It is straightforward to code the linear λ-terms into this interaction system. To satisfy β-reduction in the linear λ-calculus, the (only) interaction rule must be the following:



We apply the translation Θ to the above interaction rule to generate a combinatory reduction system that will also code the λ-calculus. The translation of the rewrite rule gives

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

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 Θ is

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

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

$$\begin{aligned}\Theta(II) &\to N([a,b,d,e]I(d,a) \cdot I(a,e) \cdot \lambda(d,b,b)) \\ &\to N([b,d,e]I(d,e) \cdot \lambda(d,b,b)) \to N([b,e]\lambda(e,b,b))\end{aligned}$$

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

The preservation of the reduction relation under the translation is in fact a general property:

Theorem 3.12. *Let N, N' be nets in an interaction net system. If $N \Rightarrow N'$ then $\Theta(N) \to^* \Theta(N')$.*

Proof. This syntactic translation Θ represents the left-hand side of a generic interaction rule (between agents α and β, see Definition 2.14) as a metaterm $N([x,\vec{y},\vec{z}]\alpha(x,\vec{y}) \cdot \beta(x,\vec{z}) \cdot Z(\vec{y},\vec{z}))$. An interaction using this rule is possible in a net W if and only if $\Theta(W)$ contains a subterm of the form $\alpha(x',\vec{y}') \cdot \beta(x',\vec{z}')$ where $x', \vec{y}', \vec{z}'$ are bound variables (i.e. $\Theta(W)$ is an instance of $N([x,\vec{y},\vec{z}]\alpha(x,\vec{y}) \cdot \beta(x,\vec{z}) \cdot Z(\vec{y},\vec{z}))$). Hence, if $W \Rightarrow W'$ then $\Theta(W)$ is reducible (using the translation of the interaction rule). After reducing $\Theta(W)$ and applying the rule for I if necessary, we obtain $\Theta(W')$. □

The method presented here can be implemented directly. For the sake of efficiency though 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 λ-calculus as an interaction net in GOI-Tools [25]. We are currently experimenting with an implementation of interaction nets based on these ideas.

4. Applications of the translation functions

Within the framework of reduction systems two properties deserve special attention: confluence, which ensures determinacy, and termination, which ensures that all reduction sequences are finite. Interaction nets are confluent "by construction". This is not the case in general for term rewriting systems, but the ones that code interaction nets according to the previous translations are confluent. Termination however is not guaranteed for arbitrary interaction nets. The aim of this section is to use the translation functions defined above to study the termination of interaction nets.

For the study of termination of term rewriting systems several methods have been proposed (for a detailed account see for example [4, 6, 15]). Moreover, since proving the termination of a term rewriting system is in general a difficult task, modularity results can be very useful.

Definition 4.1. A property P is *modular* for a rewrite system $R = R_1 \cup \cdots \cup R_n$ if

$$P(R) \Leftrightarrow P(R_1) \wedge \cdots \wedge P(R_n).$$

There are different classes of unions $R_1 \cup \cdots \cup R_n$, the simplest case being when $R_1, \ldots, R_n$ have disjoint signatures (i.e. no function symbols are shared). This class of unions is called *disjoint*.

Termination is not modular in general, not even for disjoint unions of first-order term rewriting systems. The following is a famous counterexample, given by Toyama [29]:

$$R_0 = \{f(0,1,x) \to f(x,x,x)\}, \quad R_1 = \{g(x,y) \to x,\ g(x,y) \to y\}$$

Indeed, even if R_0 and R_1 are terminating, in the union there is the following infinite reduction sequence:

$$f(g(0,1),g(0,1),g(0,1)) \to f(0,g(0,1),g(0,1)) \to f(0,1,g(0,1)) \to f(g(0,1),g(0,1),g(0,1)) \ldots$$

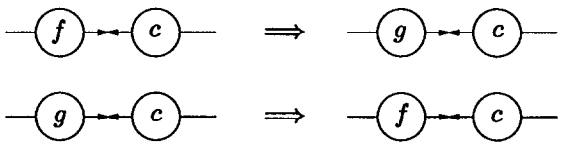
However, there are many important classes of term-rewriting systems that are known to be modular: for instance, disjoint unions of left-linear and confluent term-rewriting systems are modular with respect to termination [30], and so are unions of non-duplicating or shared-rewriting systems when the signatures share only constructors [28, 21, 8]. For these classes of systems it is then possible to prove the termination of a union by proving the termination of each part separately.

In the previous sections we showed that interaction nets can be translated into term-rewriting systems, and a sequence of reductions on nets corresponds to a sequence of reductions on terms. So, termination of the rewrite system obtained by the translation implies termination of the original interaction net system. In order to prove termination of an interaction net, we can then take profit of the termination techniques developed for term-rewriting systems. In particular, in this section we are going to focus on two problems:

1. The modularity of termination of interaction nets. If we show that a class of interaction nets is modular with respect to termination, then we are allowed to decompose a system of interaction rules into simpler subsystems and prove termination of each subsystem separately.
2. The proof of termination of a given (non-decomposable) set of interaction rules.

Since more results are known in these respects for first-order term-rewriting systems than for combinatory reduction systems, in this section we are going to concentrate on semi-simple non-dependent interaction nets. Note that even this class of interaction nets is not modular with respect to termination in general. The following is a trivial

counterexample: two interaction net systems containing one interaction rule each, which terminate when considered separately but their union is non-terminating:



4.1. Modularity of termination of interaction nets

As we already mentioned, unions of shared-rewriting systems are modular when it is the case that the signatures share only constructors. The translation of a union of semi-simple non-dependent interaction net systems that share only constructors is a union of first-order term-rewriting systems (with shared-rewriting), where only constructors are shared (if different projection symbols are used in the translation of each system, which we can assume without loss of generality). Then, we can deduce:

Property 4.2. *Unions of semi-simple non-dependent interaction net systems where only constructors are shared are modular with respect to termination.*

Proof. Direct consequence of the modularity of unions of shared-rewriting systems with shared constructors [21, 8], using Theorems 3.7 and 3.10. □

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

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

4.1.1. Unions of constructor systems

Middeldorp and Toyama [27] studied the modularity of termination of unions of constructor systems (recall that in a constructor system, left-hand sides of rules have the form $f(l_1,\ldots,l_n)$ where $f \in \mathscr{D}$ and $l_1,\ldots,l_n \in T(\mathscr{C},\mathscr{X})$). In particular, they considered *composable* unions: two constructor systems $(\mathscr{D}_1,\mathscr{C}_1,R_1)$, $(\mathscr{D}_2,\mathscr{C}_2,R_2)$ are composable if $\mathscr{D}_1 \cap \mathscr{C}_2 = \mathscr{D}_2 \cap \mathscr{C}_1 = \emptyset$, and for any $d \in \mathscr{D}_1 \cap \mathscr{D}_2$, the same rules defining d appear in R_1 and R_2. As shown in [27], unions of pairwise composable systems are not modular with respect to termination, but *confluent composable* constructor systems are.

This result applies directly to non-dependent semi-simple interaction nets: the result of applying Θ 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 $(\mathscr{D}_1,\mathscr{C}_1)$, $(\mathscr{D}_2,\mathscr{C}_2)$ are composable

if $\mathscr{D}_1 \cap \mathscr{C}_2 = \mathscr{D}_2 \cap \mathscr{C}_1 = \emptyset$, and for any $d \in \mathscr{D}_1 \cap \mathscr{D}_2$, the same interaction rules for d appear in both systems. Then, from the previous property of constructor systems we deduce:

Property 4.3. *Unions of composable non-dependent semi-simple interaction net systems are modular with respect to termination.*

4.1.2. 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: they appear naturally when systems are developed in a top-down way.

Assume given two rewrite systems over the signatures $\mathscr{F}_i = \mathscr{C}_i \cup \mathscr{D}_i$ $(i = 0, 1)$, each one specifying the set $\mathscr{D}_i$ $(i = 0, 1)$ of defined functions with respect to the set $\mathscr{C}_i$ of constructors by means of a set R_i of rewrite rules. The set $\mathscr{D}_i$ is defined as the set of symbols heading the left-hand sides of rules, the other symbols being in $\mathscr{C}_i$. Note that R_i is not necessarily a constructor system. If $\mathscr{D}_0 \cap \mathscr{D}_1 = \emptyset$ and $\mathscr{F}_0 \cap \mathscr{D}_1 = \emptyset$, but $\mathscr{F}_1 \cap \mathscr{D}_0 \neq \emptyset$, 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 $R_0 + R_1$. In $R_0 + R_1$, R_0 is the *base*, and R_1 the *enrichment*. There may be several successive enrichments in a hierarchical union.

Example 4.4. A well-known hierarchical system is the following, where the basis defines the addition of natural numbers, in the second level the product of natural numbers is defined using addition, and then factorial is defined using product:

$$\mathscr{C} = \{S, 0\}, \quad \mathscr{D}_0 = \{+\}, \quad \mathscr{D}_1 = \{\times\}, \quad \mathscr{D}_2 = \{!\}.$$

$$R_0 = \begin{cases} 0 + x \to x \\ S(x) + y \to S(x + y) \end{cases} \qquad R_1 = \begin{cases} 0 \times x \to 0 \\ S(x) \times y \to x \times y + y \end{cases}$$

$$R_2 = \begin{cases} 0! \to 1 \\ S(x)! \to S(x) \times x! \end{cases}$$

Let $R_0, R_1, \ldots, R_n$ be term rewriting systems over $T(\mathscr{D}_0 \cup \mathscr{C}, \mathscr{X})$, $T(\mathscr{D}_1 \cup (\mathscr{D}_0 \cup \mathscr{C}), \mathscr{X})$, $\ldots, T(\mathscr{D}_n \cup (\mathscr{D}_{n-1} \cup \cdots \cup \mathscr{D}_0 \cup \mathscr{C}), \mathscr{X})$ respectively. The corresponding *hierarchical union* $R_0 + \cdots + R_n$ has R_0 for *basis*, and $R_1, \cdots, R_n$ for *incremental developments*.

There are several classes of hierarchical unions of term rewriting systems that are known to be modular with respect to termination (see e.g. [5, 20, 8]). In [8] it is shown that termination is modular in hierarchical unions $R_0 + R_1 + \cdots + R_n$ 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. Let us recall the definition of the general recursive scheme.[1]

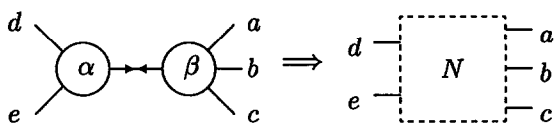
Definition 4.5. Let R be a rewrite system on $T(\mathscr{F},\mathscr{X})$, and assume that f is a new function symbol, i.e. $f \notin \mathscr{F}$. Then we can define f with a finite set of rewrite rules satisfying the following *general (recursive) scheme*:

$$f(l_1,\ldots,l_n) \to v$$

where $l_1,\ldots,l_n \in T(\mathscr{F},\mathscr{X})$, $v \in T(\mathscr{F} \cup \{f\},\mathscr{X})$, and for all subterms of the form $f(r_1,\ldots,r_n)$ in v, $\{l_1,\ldots,l_n\} \rhd_{\text{mul}} \{r_1,\ldots,r_n\}$ where $\rhd_{\text{mul}}$ denotes the multiset extension of the strict superterm ordering.

Hierarchical unions of interaction nets are defined in the same way as hierarchical unions of term rewriting systems. Next we present an interaction net version of the general recursive scheme, with the help of a diagram (to simplify the diagram, we assume that α,β have arities 3 and 4 respectively, the generalisation is straightforward):

Definition 4.6. An interaction rule



where α is a destructor and β is a constructor, satisfies the *general recursive scheme* if each occurrence of the agent α in the net N has one port connected to a, b, or c, another port connected either to a,b,c or to d, and another port connected either to a,b,c or to 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 α 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 [8], and Theorems 3.7 and 3.10, is the following:

Property 4.7. *Hierarchical unions $N_0+N_1+\cdots+N_n$ 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.*

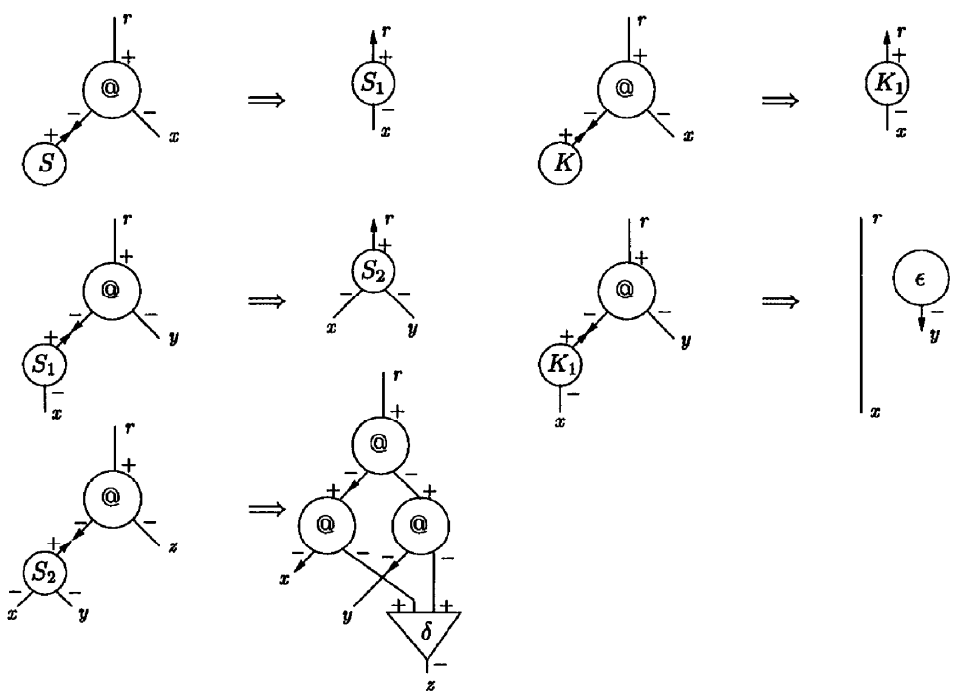
[1] Actually, the general recursive schemes presented in [8] are more general than the one defined in this paper.

4.2. Termination of interaction nets

The problem now is how to prove termination of a given interaction net system (assuming we cannot decompose it anymore!). Since many methods have been developed to study termination of first-order term-rewriting systems, we will take profit of our translations and study termination of the interaction net system directly on its translation. The other alternative would be to derive from each termination proof technique for term rewriting, a termination proof technique for interaction nets, avoiding the translation to term-rewriting systems each time a system of interaction rules has to be analysed. Research in this direction is presented in [7]. Here we will show some examples of termination proofs for interaction nets obtained by translation.

The following is an interaction net definition of Combinatory Logic (this example also shows that non-dependent semi-simple interaction nets are Turing-complete).

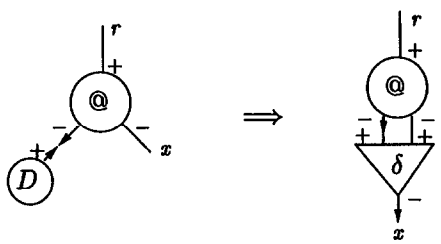
Example 4.8. Combinatory logic is defined by two (higher-order) rewrite rules: $Sxyz \to xz(yz)$, $Kxy \to x$. As an interaction net, it is specified by the agents $@, S, S_1, S_2, K, K_1$ and rules:



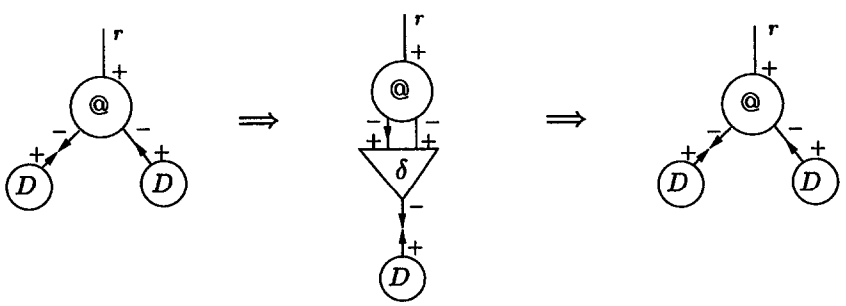
If we try to prove termination of the interaction rules for Combinatory Logic using the recursive scheme defined above, we are stuck with the rule defining the interaction between @ and S_2: @ occurs at the root of the right-hand sides with arguments that are not subterms of the arguments in the left. However, we remark that in this system

application (@) plays a particular rôle. First of all, it serves to define the higher-order rules of Combinatory Logic in a first-order syntax. Once we have a first-order system, we can see @ in two ways: as a defined symbol (then S, S_1, S_2, K, K_1 are constructors), or, more naturally since it was introduced with a predefined meaning, as a primitive (predefined) symbol. The latter is the point of view taken, for instance, in the framework of Curryfied term-rewriting systems (see e.g. [1]), and it is the one we will adopt in the following. With this assumption, the rules of Combinatory Logic satisfy the recursive scheme trivially since they are not recursive; but unfortunately the general scheme does not guarantee termination with the latter approach (i.e. if we consider @ as a primitive symbol), as the following example shows.

Example 4.9. The following rule:



is non-terminating:



The same problem arises of course in the framework of first-order term-rewriting systems: if we add an application symbol to the language, let us call it Ap, and we treat it as a special predefined symbol, then the general scheme does not guarantee termination. This problem was studied in [1], where it is shown that a variant of the general scheme, together with a *typing* condition, suffices to ensure termination of Curryfied term-rewriting systems (which are the extension of first-order term-rewriting systems with a predefined binary application symbol Ap).

We will apply to the interaction net systems above a translation function (for instance, the first we defined since the systems are semi-simple and non-dependent). When applying Θ to these examples, we will use the function symbol Ap as the

translation of @. The rules we obtain for Combinatory Logic are:

$$\begin{aligned} Ap(S,x) &\to S_1(x) \\ Ap(S_1(x),y) &\to S_2(x,y) \\ Ap(S_2(x,y),z) &\to Ap(Ap(x,\pi_1(\delta(z))),Ap(y,\pi_2(\delta(z)))) \\ Ap(K,x) &\to K_1(x) \\ Ap(K_1(x),y) &\to P(x,\varepsilon(y)) \end{aligned}$$

together with the rules defining δ, ε and the projections π_1, π_2. In fact, if we replace S by S_0, K by K_0, and the third and fifth rules by the equivalent rules:

$$\begin{aligned} Ap(S_2(x,y),z) &\to S(x,y,z) \\ S(x,y,z) &\to Ap(Ap(x,\pi_1(\delta(z))),Ap(y,\pi_2(\delta(z)))) \\ Ap(K_1(x),y) &\to K(x,y) \\ K(x,y) &\to P(x,\varepsilon(y)) \end{aligned}$$

then the image of the translation is a Curryfied term-rewriting system, to which we can apply the termination proof technique developed in [1], showing that Combinatory Logic (and also the system with the extra-rule for D) terminates on *typeable terms*. Hence any net whose image is a typeable term will terminate. We can also see that the image of the non-terminating net of Example 4.9 is the term $Ap(D,D)$, which is untypeable, as shown in [1] (in the λ-calculus framework, this net corresponds to the self-application $(\lambda x.xx)(\lambda x.xx)$).

5. Conclusions and further work

In this paper we have presented a study of the relationships between interaction nets and term-rewriting systems. One of the main points of interest that we have shown is that modularity properties and termination proof techniques of term-rewriting systems can be carried over to interaction nets in a straightforward and systematic way. This is important if we see interaction nets as a programming language.

On the other direction, we see the potential of having a more refined understanding of term rewriting by including intentional behaviour. In addition, we would like to carry over the semantics of interaction nets to the term-rewriting world. We anticipate that some variant of the Geometry of Interaction, which has been used successfully for interaction nets before, can be used. The final part of this programme will be the study of the implementation of interaction nets, both directly (net rewriting) and also via the semantics (as done with the Geometry of Interaction Machine [26]). This would give, as for the semantics, a uniform implementation of term rewriting, which can be extended to combinations of term rewriting and λ-calculus.

The study of the correspondences between term rewriting and interaction nets offers a new perspective on both formalisms; we hope that these ideas will open up new

threads of work which will allow well-known results in one formalism to be used in the other.

Acknowledgements

We would like to thank the anonymous referees for many useful suggestions in preparing this version of the paper.

The work of Ian Mackie was funded by EC project No. ERBCHBICT941805: "The Geometry of Interaction and the Implementation of Programming Languages".

References

[1] S. van Bakel, M. Fernández, Strong normalization of typeable rewrite systems, in: B.M.J. Heering, K. Meinke, T. Nipkow, (Eds.), Proc. HOA'93, 1st Internat. Workshop on Higher Order Algebra, Logic and Term Rewriting, vol. 816, Springer, Berlin, 1994, pp. 20–39.

[2] R. Banach, The algebraic theory of interaction nets, Tech. Report UMCS-95-7-2, University of Manchester, 1995.

[3] H.P. Barendregt, M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer, M. Sleep, Term graph rewriting, in: Proc. PARLE, Parallel Architectures and Languages Europe, Lecture Notes in Computer Science, 259-II, pages 141–158. Springer, Berlin, 1987.

[4] N. Dershowitz, Termination of rewriting, J. Symbolic Comput. 3 (1) (1987) 69–115. 1987.

[5] N. Dershowitz, Hierarchical termination, in: Proc. of Conditional and Typed Term Rewriting Systems, Lecture Notes in Computer Science, Springer, Berlin, 1995.

[6] N. Dershowitz, J.-P. Jouannaud, Rewrite systems, in: J. van Leeuwen, (Ed.), Handbook of Theoretical Computer Science: Formal Methods and Semantics, vol. B, North-Holland, Amsterdam, 1989.

[7] M. Fernández, Type assignment and termination of interaction nets, 1996, submitted.

[8] M. Fernández, 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 (ADT'94), Lecture Notes in Computer Science, vol. 906, Springer, Berlin, 1995, pp. 255–272.

[9] M. Fernández, I. Mackie, From term rewriting to generalised interaction nets, in: Proc. PLILP'96, Programming Languages: Implementations, Logics, and Programs, Lecture Notes in Computer Science, vol. 1140, Springer, Berlin, 1996, pp. 319–333.

[10] M. Fernández, I. Mackie, Interaction nets and term rewriting systems (extended abstract), in: H. Kirchner, (Ed.), Trees in Algebra and Programming – CAAP'96, Lecture Notes in Computer Science, vol. 1059, Springer, Berlin, 1996, pp. 149–164.

[11] S.J. Gay, Combinators for interaction nets, in: C. Hankin, I. Mackie, R. Nagarajan (Eds.), Theory and Formal Methods of Computing 94: Proc. 2nd Imperial College Workshop, Imperial College Press, 1995, pp. 63–84.

[12] J.-Y. Girard, Linear Logic, Theoret. Comput. Sci. 50 (1) (1987) 1–102.

[13] J.-Y. Girard, Geometry of interaction 1: interpretation of system F, in: R. Ferro, C. Bonotto, S. Valentini, A. Zanardo, (Eds.), Logic Colloquium 88, North-Holland, Amsterdam, 1989.

[14] G. Gonthier, M. Abadi, J.-J. Lévy, The geometry of optimal lambda reduction, in: Proc. ACM Symp. Principles of Programming Languages, 1992, pp. 15–26.

[15] J.-P. Jouannaud, Rewrite proofs and computations, in: H. Schwichtenberg (Ed.), Proof and Computation, Computer and Systems Sciences, vol. 139, Springer, Berlin, 1995, pp. 173–218.

[16] J.R. Kennaway, J.W. Klop, M.R. Sleep, F.J. de Vries, On the adequacy of graph rewriting for simulating term rewriting, ACM TOPLAS 16 (3) (1994) 493–523.

[17] R. Kennaway, J.W. Klop, R. Sleep, F.-J. de Vries, Transfinite reductions in orthogonal term rewriting systems, Inform. Comput. 119 (1) (1995) 18–38.

[18] J.-W. Klop, Combinatory Reduction Systems, Mathematical Centre Tracts, vol. 127, Mathematischen Centrum, 413 Kruislaan, Amsterdam, 1980.

[19] J.-W. Klop, Term rewriting systems. in: S. Abramsky, D. Gabbay, T. Maibaum, (Eds.), Handbook of Logic in Computer Science, vol. 2, Oxford University Press, Oxford, 1992.
[20] M. Krishna Rao, Completeness of hierarchical combinations of term rewriting systems, in: Proc. FST & TCS'93, Lecture Notes in Computer Science, vol. 761, Springer, Berlin, 1993.
[21] M. Kurihara, A. Ohuchi, Modularity in non-copying term rewriting, Theoret. Comput. Sci. 152 (1995) 139–169.
[22] Y. Lafont, Interaction nets, in: Proc. 17th ACM Symp. on Principles of Programming Languages, 1990, pp. 95–108.
[23] Y. Lafont, From proof-nets to interaction nets, in: J.-Y. Girard, Y. Lafont, L. Regnier (Eds.), Advances in Linear Logic, London Mathematical Society Lecture Note Series, vol. 222, Cambridge University Press, Cambridge, 1995, pp. 225–247.
[24] C. Laneve, Optimality and concurrency in interaction systems, Ph.D. Thesis, Dipartimento di Informatica, Università degli Studi di Pisa, 1993.
[25] I. Mackie, The geometry of implementation, Ph.D. Thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1994.
[26] I. Mackie, The geometry of interaction machine, in: Proceedings 22nd ACM Symp. on Principles of Programming Languages, 1995, pp. 198–208.
[27] A. Middeldorp, Y. Toyama, Completeness of combinations of constructor systems, in: Proc. 4th Rewriting Techniques and Applications, vol. 488, Springer, Berlin, 1991.
[28] M. Rusinowitch, On termination of the direct sum of term rewriting systems, Inform. Processing Lett. 26 (1987) 65–70.
[29] Y. Toyama, Counterexamples to termination for the direct sum of term rewriting systems, Inform. Processing Lett. 25 (1987) 141–143.
[30] Y. Toyama, J. Klop, H. Barendregt, Termination for the direct sum of left-linear term rewriting systems, in: Proc. 3rd Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 355, Springer, Berlin, 1989.