From Term Rewriting to Generalised Interaction Nets

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

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

Abstract. In this paper we present a system of interaction that generalises Lafont's interaction nets by allowing computation in several nets in parallel and communication through a state. This framework allows us to represent large classes of term rewriting systems, genuine parallel functions, non-determinism, communication, sharing, and hence can be used to code features from Standard ML and Concurrent ML.

1 Introduction

Term rewriting systems can be regarded as a multi-paradigm specification language, or as an abstract model of computation (abstract in the sense that they specify actions but not control; they are free from strategies). Lafont's interaction nets are a graphical framework based on net rewriting which is much closer to a real model of computation in that all the operations are made explicit, including discarding and copying of data. Moreover, interaction nets can be regarded as a distributed model of computation since all reductions are local.

In a previous paper [2] we began a programme of research which aims to bridge the gap between these two separate formalisms: we presented a series of translations from interaction nets to various classes of term rewriting systems, and studied various properties that are preserved under the translations. The aim of the present paper is to complete the study of the correspondence between interaction nets and term rewriting systems, by showing translations of term rewriting systems into interaction nets. This is a first step towards the development of an interaction-net based implementation of term rewriting languages.

Due to the intrinsic limitations of interaction nets (they only capture confluent and sequential computations) not all term rewriting systems can be encoded. However, for the class of *match-sequential orthogonal constructor systems* [17] we give a general encoding into interaction nets. We then look at more general classes of term rewriting systems, and investigate what we need to add to interaction nets to allow a full encoding.

We identify several features that we would like to capture that are not included in the interaction net framework. Since interaction nets only capture sequential and deterministic computations, the main additions that we consider concern non-determinism and parallel functions, such as "parallel-or" defined by: $\mathsf{por}\ \mathsf{tt}\ x \to \mathsf{tt}$, $\mathsf{por}\ x\ \mathsf{tt} \to \mathsf{tt}$, $\mathsf{por}\ \mathsf{ff}\ \mathsf{ff} \to \mathsf{ff}$. In addition, we would like to code notions such as communication, so that we can send and receive data between disjoint nets which is required to code languages such as CML [16], and state so that we can code references, as in ML [15].

In this paper we propose a new system of interaction that captures the above, but retains the spirit of Lafont's interaction nets. The generalisation we propose consists of allowing computation in *several interaction nets in parallel*, which can communicate through a *state*. In particular, we still consider a graphical framework and binary interactions, but we permit interactions through the state (a kind of "indirect interaction"), which allows the encoding of all the above notions in a clean and uniform way. As particular cases, we show how to encode the whole class of orthogonal (i.e. left-linear and non-overlapping) constructor systems, which is used in most programming languages based on term rewriting, and also overlapping systems such as the system defining the parallel-or function.

Other models of computation derived from interaction nets with different purposes have been proposed by Laneve [11], who introduced *interaction systems* as combinatory reduction systems that correspond to interaction nets without the linearity condition (but still sequential), and Honda [5] who defined a calculus of concurrent processes in which the connecting edges are not necessarily binary (Lafont's interaction nets can be seen as a particular case, although the framework of concurrent processes is quite different from the graphical framework of interaction nets). In contrast, the generalised interaction nets we propose can represent parallel computations while retaining the essence of Lafont's interaction nets. We define a set of combinators that can be used to manipulate (interact through) the state, thus making possible the encoding of references and communication. By restricted use of these combinators we obtain Lafont's interaction nets as a particular case, and also parallel nets without side effects which inherit all the properties of interaction nets.

Interaction nets have been successfully used for the implementation of various λ-calculi [12]. We study interaction nets from the term rewriting point of view with the hope of making these implementation ideas applicable also to term rewriting systems and to languages that combine term rewriting and λ-calculus. We hope that generalised interaction nets will also allow us to code more general classes of rewrite systems, such as combinatory reduction systems.

The rest of this paper is structured as follows. In the next section we recall the basic concepts used in the rest of the paper. In Section 3 we give the first set of translations from term rewriting systems to interaction nets. In Section 4 we investigate the structures that we need to code general term rewriting systems; in particular this leads to interaction nets with state. In Section 5 we look at several case studies to show how to code additional features into our nets. Finally, in Section 6 we conclude our ideas and suggest further directions.

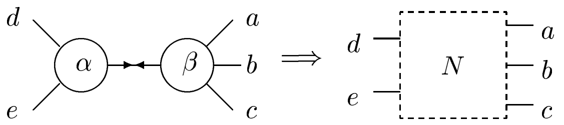
2 Background

We refer the reader to [1, 8] for a detailed account of rewrite systems, and to [10] for interaction nets; here we briefly recall the basic definitions.

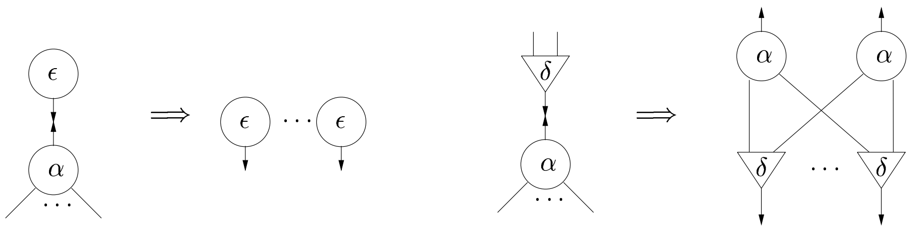
Term rewriting systems. A *signature* $\mathcal{F}$ is a finite set of *function symbols* together with their (fixed) arity. $\mathcal{X}$ denotes a denumerable set of *variables*, and $T(\mathcal{F}, \mathcal{X})$ denotes the set of *terms* built up from $\mathcal{F}$ and $\mathcal{X}$. The *subterm* 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. A term is *linear* if each variable occurs at most once.

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

Interaction nets. An interaction net (Σ, IR) is specified by a set Σ of symbols (*agents*), each with an arity (degree) $n \in \mathbb{N}$ ($n \geq 1$) which is the number of edges attached to it, and a set IR of *interaction rules*. A *net* on Σ is an undirected graph with nodes labeled by symbols in Σ. The points of contact of edges with nodes are called *ports*. Each agent has a distinguished port labeled with an arrow, called the *principal port*, where interaction can take place. All the other ports are called *auxiliary ports*. Interaction rules are activated when two agents are connected on their principal port. There is at most one rule for each pair of agents. The following diagram shows the general form of an interaction rule, where N is a any net. Note that the interface is preserved.



As examples, we show the rules of two ubiquitous agents, namely the *erase* (ϵ), which deletes everything it interacts with, and the *duplicator* (δ), which copies everything. These are represented by the following diagrams, where α is any node.



We say that a net is in *normal form* when no interactions are possible. By construction, the net rewriting process is *confluent*. Moreover, since the local interface is preserved during interaction there is scope for a truly parallel implementation.

3 From Term Rewriting Systems to Interaction Nets

Interaction nets provide an alternative view of rewriting that is easier to work with (since it is graphical), and easier to implement (since they are more explicit with respect to copying/discarding of data, and the reduction process is local). But term rewriting is a very rich field, with well established theories and results such as type systems, termination proof techniques, etc. The study of the relationships between these two formalisms allows results to be transferred from one framework to the other. In [2] we showed the following mappings from classes of interaction nets to rewrite systems:

Class of Interaction Nets	*Class of Rewrite Systems*
Discrete Semi-simple Nets	First-order Term Rewriting Systems (linear and non-overlapping)
Non-Dependent Semi-simple Nets	First-order Term Rewriting Systems (left-linear and non-overlapping)
Dependent Semi-simple Nets	Combinatory Reduction Systems (left-linear and non-overlapping)

In this section we continue the study of the above correspondence by showing how to translate in the opposite direction. We focus on *orthogonal term rewriting systems*.

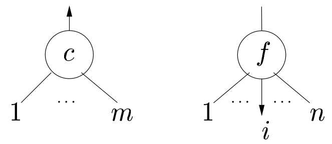
It is not possible to encode an arbitrary system of this class into interaction nets. The first problem is that there is no uniform translation for rules that express relations between *constructors*, like: $Pred(Succ(x)) \to x$, $Succ(Pred(x)) \to x$, since agents can only interact on their principal ports. However, if we consider the class of term rewriting systems that is used in most functional programming languages, constructor systems, then this problem does not arise. A *constructor system* is a term rewriting system with the property that the signature can be partitioned into a set $\mathcal{C}$ of constructors and a set $\mathcal{D}$ of defined functions, and every left-hand side $f(t_1, \ldots, t_n)$ of a rule satisfies $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in T(\mathcal{C}, \mathcal{X})$. A constructor system is then specified by a triple $(\mathcal{D}, \mathcal{C}, R)$, where R is the set of rewrite rules of the system.

It is still not possible to encode an arbitrary function definition of this class as an interaction net: the limitation of binary interactions does not allow us to encode non-sequential functions like the well-known BP function of Berry and Plotkin. Note that this is an orthogonal constructor system, but it is not sequential. Interaction nets can only code function definitions where patterns can be examined in some sequential order. This property is called *match-sequentiality*, and is decidable for orthogonal constructor systems [17].

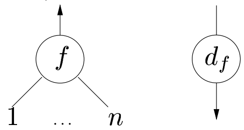
In the rest of this section we show that orthogonal match-sequential constructor systems can be encoded as interaction nets, by showing a translation from this class of rewrite systems to (non-dependent) interaction nets. We then show that the left-linearity condition can be dropped if we consider a call-by-value strategy of evaluation for the term rewrite system.

Translating Match-sequential Orthogonal Constructor Systems. Constructor systems contain rules of the form $f(l_1, \ldots, l_n) \to t$ where $l_1, \ldots, l_n \in T(\mathcal{C}, \mathcal{X})$ and are called the *patterns* of the rewrite rule. In a match-sequential rewrite system, given a function f of arity n, there is a sequence $(i_1, \ldots, i_k)$, $0 \leq k \leq n$, of *indexes* of patterns such that pattern-matching can be safely done in the order indicated by the sequence. The case of an empty sequence of indexes ($k = 0$) corresponds to a function defined by a rule of the form $f(\vec{x}) \to t$ (we use vectors to denote sequences, which can be empty).

By adding new function symbols to the signature, we can assume without loss of generality that all the patterns have depth less than or equal to 1. Moreover, because the system is match-sequential, we can transform each function definition so that each function has at most one index (see [7] for a detailed description of these transformations). Hence, in the following we will consider constructor systems with left-linear rules of the form $f(\vec{x}, c(\vec{z}), \vec{y}) \to t$, where $c \in \mathcal{C}$, $\vec{x}, \vec{y}, \vec{z} \in \mathcal{X}$, and the position of $c(\vec{z})$ corresponds to the index i of f. We will translate a constructor system over the signature $(\mathcal{D}, \mathcal{C})$ into an interaction net system with agents:



for each $c \in \mathcal{C}$ of arity m, and $f \in \mathcal{D}$ of arity n with index i. In addition, for each $f \in \mathcal{D}$ of arity n with an empty sequence of indexes (this implies that no pattern-matching is needed for f) we consider the agents:



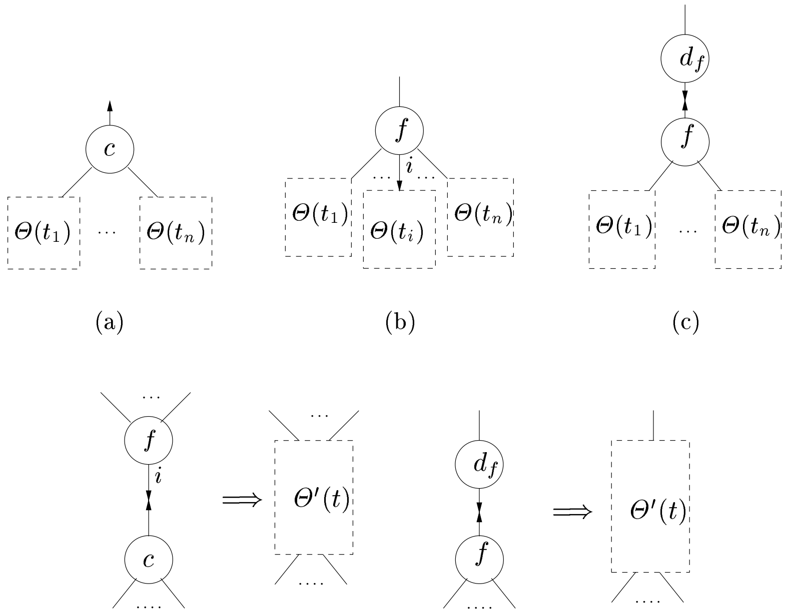
The dummy agent d_f is needed to code the rules for f as binary interactions, as we see below. Note that in this case the output port of f is the principal port, as in the case of constructors.

We now define (by induction) a translation function Θ that takes a term t in the signature $(\mathcal{D}, \mathcal{C})$ and produces a net with the above set of agents.

1. If $t \in \mathcal{X}$ then $\Theta(t)$ is just an edge.
2. If $t = c(t_1, \ldots, t_n)$ and $c \in \mathcal{C}$, $\Theta(t)$ is the net given in Fig. 1(a).
3. If $t = f(t_1, \ldots, t_n)$ and $f \in \mathcal{D}$ with index i, $\Theta(t)$ is the net in Fig. 1(b).
4. If $t = f(t_1, \ldots, t_n)$ and $f \in \mathcal{D}$ with an empty list of indexes, then t will be translated using the dummy function d_f. $\Theta(t)$ is the net in Fig. 1(c).

If t is a linear term we are done, otherwise we define $\Theta'(t)$ as the net obtained from $\Theta(t)$ by connecting the edges corresponding to occurrences of non-linear variables in t to δ nodes (duplicators).

To complete the encoding of a constructor system into an interaction net, we show the translation of the set of rewrite rules. We distinguish two cases. A rule of the form $f(\vec{x}, c(\vec{z}), \vec{y}) \to t$ defining an n-ary function f with index i, is translated as the interaction rule given in Fig. 1(d), where $\Theta'(t)$ is obtained from



(a) (b) (c)

(d) (e)

Fig. 1.

the net $\Theta(t)$ by connecting the edges that correspond to non-linear variables in t to δ nodes (duplicating), and adding ϵ nodes (erasing) for the variables of the left-hand side that do not appear in t. If $l \to r$ is right-linear and $\mathcal{V}ar(l) = \mathcal{V}ar(r)$ then $\Theta'(r) = \Theta(r)$. A rule of the form $f(\vec{x}) \to t$ will be coded as a binary interaction using d_f, as shown in Fig. 1(e), where $\Theta'(t)$ is defined as above.

Correctness of the translation. We only give a brief sketch of the proof. It consists of two parts. First we show that a sequence of shared-rewrite steps (see [9] for details on shared-rewriting) $s \to^+ t$ using rules in R can be simulated by a sequence of interactions from $\Theta'(s)$ using the translation of R. Then we use a result of Kennaway *et al.* [6] that shows that in the case of orthogonal systems, shared-rewriting can simulate standard rewriting. This completes the proof.

Non-linearity and Overlappings. We address first the problem of encoding match-sequential non-overlapping constructor systems *with non-linear left-hand sides.* Let us consider a simple example: $f(c(x), x) \to t$. This rule can be applied only to terms of the form $f(c(t_1), t_2)$ where t_1 and t_2 are equal. The translation function defined previously is not correct for non-left-linear rules: in order to mimic term reduction, we have to code the function that tests syntactical equality of terms as an interaction net. We can represent such a function by an agent *eq* with 3 ports, corresponding to the two arguments and the result, and we will arbitrarily choose the left argument to be the principal port. Since *eq* can only interact with constructors, this forces a *call-by-value strategy* on our encoding.

To simplify the presentation we consider only rewriting on ground terms (terms without variables), and assume that normal forms are constructor terms. The encoding of terms is done in exactly the same way as before. For the rules we just give an example. Figure 2 shows the translation of the system containing the rule $f(c(x), x) \rightarrow t$ in a signature f, c, a, b where $a, b, c \in \mathcal{C}$, a is nullary, b is binary, and c is unary. The generalisation is straightforward.

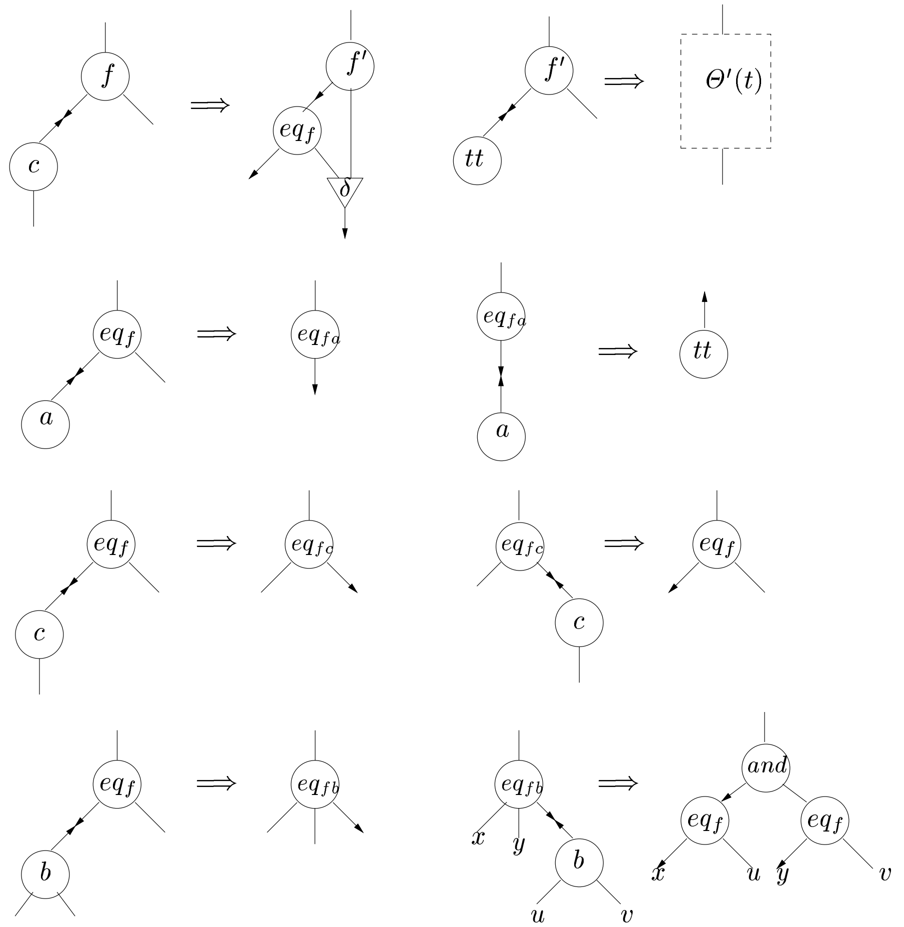


Fig. 2.

The problem of overlapping patterns is more delicate. Rules like $f(c(x)) \rightarrow a$

and $f(c(x)) \to b$ are perfectly acceptable, although the system is not confluent. Interaction nets are confluent by definition; only one interaction rule can be defined for each pair of agents, preventing the encoding of rules like the ones above. Of course, an overlapping system may be confluent, an interesting example being the class of *weakly-orthogonal* rewrite systems: left-linear systems that have only trivial critical pairs of the form (t, t). The translation function defined before can be directly applied to weakly-orthogonal constructor systems that are match-sequential: in this case the only possible overlappings are at the root position, between a left-hand side l and another left-hand side instance of l (note that the assumption of match-sequentiality is crucial here); then it is sufficient to translate only the most general rules that create a critical pair.

4 Generalised Interaction Nets

Thus far the most general class of term rewriting system that we have been able to code completely in interaction nets is the class of match-sequential, orthogonal constructor systems. Here we generalise the interaction net framework so that all orthogonal constructor systems, and also overlapping systems can be encoded.

We add the notion of a *state* to the interaction net framework. That is to say that we consider the interaction system together with a state that we can allocate, update and read values. In particular, we see generalised interaction nets as parallel interaction nets that communicate via the state.

The main reason for this choice is that having a notion of state allows us to have *interaction by indirection*. We maintain the binary interaction (with a reference node) but can have multiple pointers to a single net via the state (this is crucial for capturing sharing of values). By keeping the restriction to binary interactions, our generalised interaction nets inherit the properties of interaction nets (asynchrony, locality, confluence) when restricted to computations in one net. The interactions between different nets are restricted to interactions through the state.

To explain how this works we define several interaction rules that are examples of basic combinators for state manipulation. These operations are identified using the same nomenclature as for references in ML [15]. In Fig. 3 we introduce the nodes that we are going to use, and give the corresponding interaction rules for each operation in Fig. 4. These are all explained as follows.

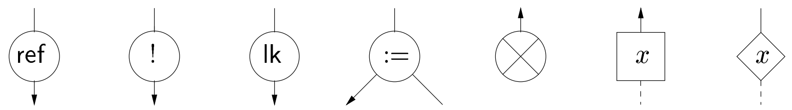


Fig. 3. Nodes

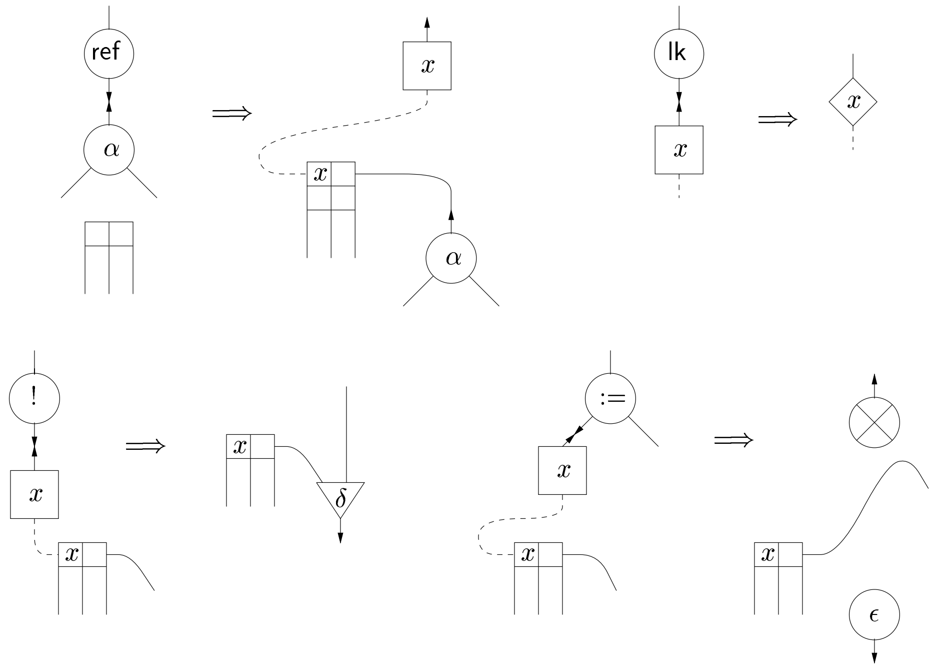


Fig. 4. Interaction rules with state

Allocation of a new cell (ref N). This operation creates a new memory location, say x, and assigns to it an initial net N. The interaction produces a new node which we draw as a square holding the memory location. We draw a dashed line to show that this node is "pointing to" the state.

Read (dereferencing $!x$). This operation reads out a copy of the net in memory location x. Note that we use the interaction net machinery to copy the contents using the δ node.

Look (lk). This behaves in a similar way to read, except the net in the location is *not* copied. An indirection node, drawn as a diamond containing a pointer to the memory location, then allows interactions through the state. Of course only a limited kind of interaction can be done, which we specify below. Remark that there is no principal port for this indirection node; it is there solely to delimit the dashed line from the edges of the interaction net.

Update (assignment $x := N$). This operation assigns the new net N to the memory location x and deletes the old net in memory (using the ϵ node). Remark that the correctness of some systems may require that we evaluate the net N to normal form before making the assignment and deleting the old net. This would mean that we only allow values in the state.

Unit ($\otimes$) used to code the result of assignment.

We emphasise that we still maintain the restriction of binary interaction rules, but allow many pointers to the same memory cell (indicated by a dashed line). We now have the possibility to define interaction rules that operate through the state, via the indirection node. We adopt a notational convention for the latter to simplify the diagrams, as shown in Fig. 5(a). Rather than writing an interaction via the state we write the interaction rule with a dashed line and the name of the location. Figure 5(b) shows the interaction rules that we allow.

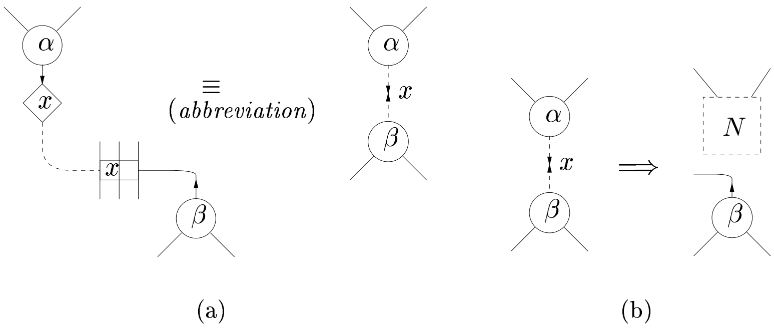


Fig. 5.

It is important to note that in this kind of rule we preserve the interface of the node α interacting through the state, hence we are not able to change the contents of the memory; just pattern match. This is the feature that allows us to get extra sharing in this framework.

We end this section with a few remarks about our nets. We have extended the framework of interaction nets to allow computation in several nets in parallel. These parallel interaction nets can communicate through the state, which requires that we have a global entity. Hence our interaction system does not preserve locality of interaction nets. However, this is only the case for the specific memory operations. One of the main reasons for introducing state was to break the confluence property of interaction nets, to allow non-deterministic computation. There is a simple condition on our interaction system that ensures that confluence is maintained, and that is that there is only one assignment to each memory location. A simple way to avoid side-effects would be to restrict to only using the ref and lk operations on the state.

There is a natural extension of Lafont's type system [10] for our generalised interaction nets, where we use ref types of ML. It is then possible to check that all rewrite rules preserve types.

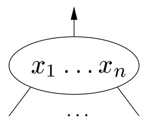
We remark that we have only presented some examples of combinators for state manipulation. For many applications one might need additional, or modified, rules; for example, to code reference counting, mutual exclusion, etc. Also it may be necessary to impose some order of evaluation on the rules to ensure the desired behaviour, in particular for assignment. In the examples that follow, we omit these details. Of course, for interaction nets that do not use the state, we preserve all the properties of Lafont's interaction nets. Note also that garbage collection forces evaluation of discarded computations that may contain assignments.

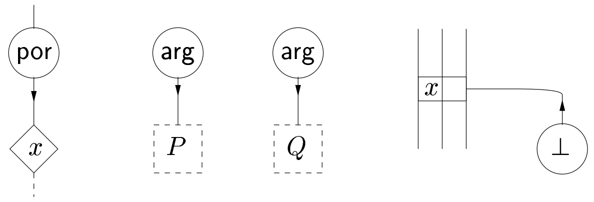
4.1 Translation of non-match-sequential systems

We begin with an example of how this new interaction framework can code a richer class of systems by giving the coding of the parallel-or function defined above. Let us assume that the two arguments of the por are compiled as nets P and Q which are connected to nodes called arg. A por node is then introduced which takes only one argument, which is a memory location, say x. The initial configuration of nodes is given by the diagram in Fig. 6(a). The node $\perp$ is introduced to represent an undefined value in the state, and should not be confused with a non-terminating computation. The rewrite rules for the nodes introduced are given in Fig. 6(b).

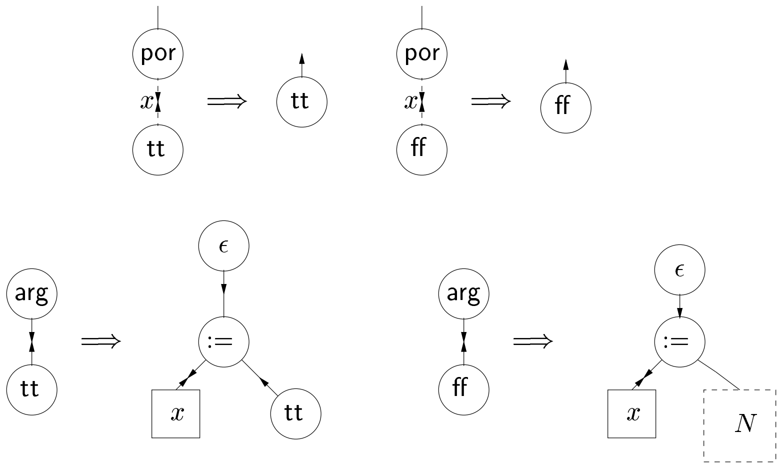
The net N that we have omitted to draw corresponds to the following program which is easily represented as an interaction net using pattern matching through the state: $\mathsf{if}\ !x = \mathsf{tt}\ \mathsf{then}\ \mathsf{tt}\ \mathsf{else}\ (\mathsf{if}\ !x = \perp\ \mathsf{then}\ \mathsf{for}\ \mathsf{else}\ \mathsf{ff})$. The intuition is that if either argument returns tt then this value will be assigned to the state, where the por node can interact with it. If either argument returns ff, then we look at the value already in the state. If the value is tt then we do nothing, if the value is $\perp$ then we place a for node ("false-or") indicating that we have to wait for the other argument. If there is already for in the state then the final value will be false so we can assign this value to the state where the por function can interact. Note that there is no rule for the interaction por-for hence the system will *deadlock* in the case of $\mathsf{por}\ \mathsf{ff}\ \Omega = \mathsf{por}\ \Omega\ \mathsf{ff}$ (here Ω is a non-terminating computation). This example provides the essential ideas that can be used to code non-sequential rewrite systems into nets. We next present in more detail the translation of orthogonal non-match-sequential constructor systems.

Given a constructor system R over the signature $(\mathcal{D}, \mathcal{C})$, we consider in turn the rules defining each $f \in \mathcal{D}$. Without loss of generality we assume f to be defined by rules of the form $f(\vec{C}[\vec{x}]) \to t$, where the patterns C_i are either a variable or a constructor. We code the rules for an n-ary f using an agent arg_i $(1 \leq i \leq n)$ for each argument, which will put in a memory location, say x, the result of evaluating that argument. The memory location x will contain n-tuples of constructor symbols and $\perp$ (initially all $\perp$, and getting more constructors as the arg agents add them). We use new agents to represent the n-tuples; in other words, x contains nets of the form:





(a) Initial configuration



(b) Rewrite rules

Fig. 6. Parallel-or

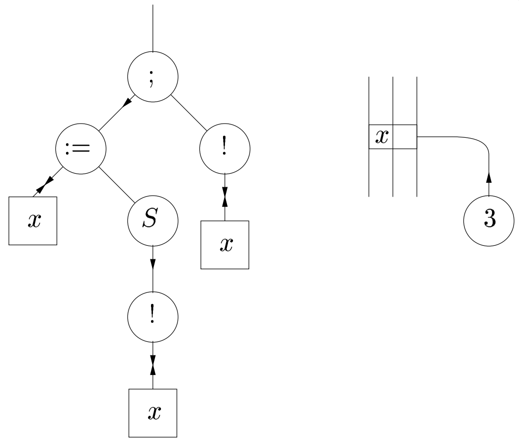
where each x_i is either $\bot$ or a constructor c_j, and the number of auxiliary ports is equal to the sum of the arities of the constructors appearing in the n-tuple.

We use an agent f to represent the function f, which will interact through the state with the n-tuples placed by the arg agents. There will be a set of interaction rules for each rule defining f in R (with a right-hand side that corresponds to the translation $\Theta'(t)$ of the right-hand side t of the rule, as in Section 3). Note that the encoding of parallel-or is an optimised version of this method of translation, where we take profit of the symmetry of the patterns to use only one arg agent, and only one value for ("false-or") to represent the pairs $\bot\mathsf{ff}$ and $\mathsf{ff}\bot$.

5 Case Studies

In this section we show how to encode a series of computational features into our interaction system, some of which go beyond term rewriting systems.

Standard ML References. The basic combinators introduced were given using the notation of ML, hence the coding is direct. However, a call-by-value strategy on the nets is required for the correct (sequential) behaviour. We give a small example, which we hope gives some additional intuitions about how our interaction system works. Consider the program: let val $y =$ ref 3 in $(y := S(!y); !y)$ end. y is a binder, hence we compile the body of the let and bind all occurrences of y to the net representing ref 3. If we perform the interaction of the ref 3, giving a memory location x, and the duplication of the references, we get the following:



To complete this example, we give the rewrite rules for the additional nodes, which are the sequential composition and successor function:

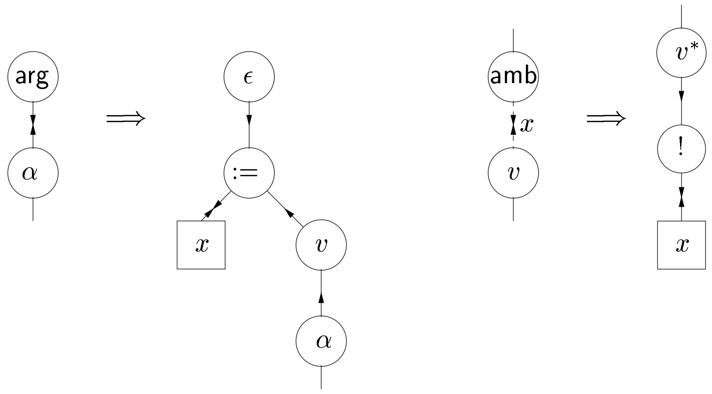
We leave it as an exercise to the reader to use these rules to reduce this term to its normal form, which gives the result 4 as expected.

Communication. A notion of send and receive, both synchronous and asynchronous, on a channel can be coded using assignment. It is also possible to code process forking, which would involve the creation of a new net that would communicate on a channel (a memory location).

Sharing. One of the greatest achievements of interaction nets is the ability to code optimal reduction in the λ-calculus; see [3] for example. However, there is no way of sharing *values* which have to be copied due to the linearity condition of interaction rules. This arises since two pointers to a value (coded with a δ node) forces copying, even if the rule was just "inspecting" the value. This can be overcome in the new framework, by pattern matching through the state.

Non-determinism can be coded as a simple variant of the technique used for parallel-or. A simple example is given by the following rewrite system: amb $x\ y \rightarrow$

x, amb $x\ y \rightarrow y$. The initial configuration is the same as for parallel-or (replace por by amb). The rewrite rules for the agents are given by:



The arg node interacts with any agent and assigns the net connected to that agent to the memory location x, together with a new node v (for "value") to indicate that there is a value in the state. amb simply reads out the value. The node v^* is introduced which interacts with the v node and annihilate each other. Therefore, as soon as a value has been assigned to the location x, the amb agent will read out the result which will be the last value assigned to that location.

6 Conclusions

We have completed the study of the correspondences between interaction nets and first-order term rewriting systems, that we initiated in [2]. We have shown how to encode some restricted classes of constructor systems into interaction nets. To extend this to richer classes of term rewriting systems we were led to extend the paradigm of interaction to include a notion of state, which was shown to allow the coding of general constructor term rewriting systems. In addition, the new interaction system is sufficiently general to code a number of other features, such as references, communication and sharing.

We are currently developing implementations of term rewriting systems based on parallel nets. Most implementations of term rewriting use graph rewriting (see e.g. [6]), a related formalism which also allows sharing of computation. Graph rewriting is more flexible than interaction nets in that left-hand sides are not restricted. However, interaction nets have the advantage of being easier to implement (possibly distributed) since the rewrite rules are local and asynchronous.

Some directions for further investigation are the encoding of: higher-order rewrite systems like combinatory reduction systems (the reverse direction, i.e. coding interaction nets as combinatory reduction systems was studied in [2]), ML (including exceptions and I/O), threads in CML, process calculi (such as the π-calculus [13], and an understanding of π-nets [14]), equational rewriting, and narrowing (for the encoding of logic programming).

This would provide a common basis where we can study models and implementations of systems, for example the combination of λ-calculus with term rewriting systems.

Acknowledgement. The work of Ian Mackie was funded by EC project No. ERB4001GT940674: The Geometry of Interaction and the Implementation of Programming Languages.

References

1. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Methods and Semantics*, volume B. North-Holland, 1989.
2. M. Fernández and I. Mackie. Interaction nets and term rewriting systems (extended abstract). In H. Kirchner, editor, *Trees in Algebra and Programming – CAAP'96*, number 1059 in LNCS, Springer-Verlag, 1996.
3. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proc. ACM Symposium Principles of Programming Languages*, 1992.
4. C. Hankin, I. Mackie, and R. Nagarajan, editors. *Theory and Formal Methods of Computing 94: Proceedings of the Second Imperial College Workshop.* Imperial College Press, 1995.
5. K. Honda and N. Yoshida. Combinatory representation of mobile processes. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, 1994.
6. J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM TOPLAS*, 16(3):493–523, 1994.
7. J. Kennaway. Implementing term rewrite languages in DACTL. *Theoretical Computer Science*, 72:225–249, 1990.
8. J.-W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
9. M. Kurihara and A. Ohuchi. Non-copying term rewriting and modularity of termination. Hokkaido University.
10. Y. Lafont. Interaction nets. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, 1990.
11. C. Laneve. *Optimality and Concurrency in Interaction Systems.* PhD thesis, Università degli Studi di Pisa, 1993.
12. I. Mackie. A λ-evaluator based on interaction nets. In Hankin et al. [4].
13. R. Milner. The polyadic π-calculus: A tutorial. Technical Report 91-180, LFCS, Department of Computer Science, University of Edinburgh, 1991.
14. R. Milner. Action calculi II: π-nets with boxes and replication. Technical report, LFCS, Department of Computer Science, University of Edinburgh, 1993.
15. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
16. J. Reppy. CML: A higher-order concurrent language. *ACM SIGPLAN Notices*, 26(6):294–305, 1991.
17. S. Thatte. A refinement of strong sequentiality for term rewriting systems with constructors. *Information and Computation*, 72:46–65, 1987.