Journal of Programming Languages **5**, 1997, 201–231

A study of two graph rewriting formalisms: Interaction Nets and MONSTR

R. Banach[1] and G.A. Papadopoulos[2]

[1] *Department of Computer Science, University of Manchester, Manchester, M13 9PL, UK*
banach@cs.man.ac.uk
[2] *Department of Computer Science, University of Cyprus, CY-1678, Nicosia, Cyprus*
george@turing.cs.ucy.ac.cy

Two superficially similar graph rewriting formalisms, Interaction Nets and MONSTR, are studied. Interaction Nets come from multiplicative Linear Logic and feature undirected graph edges, while MONSTR arose from the desire to implement generalized term graph rewriting efficiently on a distributed architecture and utilizes directed graph arcs. Both formalisms feature rules with small left-hand sides consisting of two main graph nodes. A translation of Interaction Nets into MONSTR is described for both typed and untyped nets, while the impossibility of the opposite translation rests on the fact that net rewriting is always Church–Rosser while MONSTR rewriting is not. Some extensions to the net formalism suggested by the relationship with MONSTR are discussed, as well as some related implementation issues.

Keywords: Graph rewriting, MONSTR, Interaction Nets

Many different kinds of graph have been studied over the years, and inevitably, people have invented a large number of ways of rewriting them, yielding a vast number of different models of computation. In this paper we study the relationship between two models that bear a superficial resemblance, but that were inspired by very different motivations: Interaction Nets and MONSTR.

Interaction Nets (Lafont 1990, Lafont 1991) evolved from the multiplicative fragment of Linear Logic (From the vast literature on that subject, see the work by Girard (1987), Troelstra (1992), Girard et al. (1995).) The basic idea is that a multiplicative proof object consists of inference steps. The object is represented by a graph, in which the individual inference steps, combining a number of hypotheses to form a conclusion, are represented by agent nodes for which the adjacent edges represent the hypotheses and conclusions. The special nature of a conclusion singles it out, making it **principal**. The dynamics of proof objects is enshrined in the notion of elimination of cuts, whereby the two conclusions meeting in a cut are eliminated by transforming the proof object in the vicinity of the cut. In the world of the representing graphs, two agents joined by a connection which is principal for both of them is the analogue of the cut, and its elimination is a rewrite rule for such graphs. This gives rise to the Interaction Net model of graph rewriting.

MONSTR (Banach 1993) originated from the desire to implement the generalized term graph rewriting language Dactl (Glauert et al. 1988, Glauert et al. 1990) on a distributed parallel machine, the Flagship machine (Watson et al. 1988). The demands of (even an imperfectly adhered to notion of) serializability

0963–9306 © 1997 Chapman & Hall

for Dactl executions necessitated curtailing the expressive power of Dactl rules rather drastically. It was vital for the Flagship machine that the computational model encompassed a reasonable notion of state, despite the predominantly declarative programming models that it was primarily intended for. The MONSTR computational model as it eventually emerged, therefore permitted each rule to include at most one unit of updatable state per rewrite, apart from the root of the rewrite itself, giving two key nodes on the left-hand side of each rule. The similarity to Interaction Nets is clear.

That two models of computation emerging from such diverse backgrounds should both settle on the idea that left-hand sides of graph rewrite rules should consist of two main nodes is intriguing, and is the main spur for this paper. Another motive for the work described here is the question of whether the efficiency considerations that motivated MONSTR translate well into the world of Interaction Nets, whose primary motivations were always much more abstract. To be more precise, both formalisms share some common views regarding what constitutes a 'good' computational model for distributed systems: they both support locality of computation (interaction between two redex nodes during their rewriting is clearly a local activity), likewise rewriting which is asynchronous and free from excessive locking; and they both enjoy formal semantics. Furthermore, while more traditional term (and/or graph) rewriting systems are rather abstract in the sense that the way rewrite rules are formulated puts more emphasis on the 'logic' behind the reduction sequences and less on more operational aspects like implementing execution strategies, MONSTR differs in using explicit annotations capable of also expressing these. Finally, both formalisms can play the role of being intermediate compiler target languages and can be used as implementation models for higher-level (linear or otherwise) programming languages. Thus, a proper study of Interaction Nets and a concrete term graph rewriting system model like MONSTR which bridges the gap between the two is beneficial both in theory and in practice. In particular, we can reason about the rather unusual syntax of Interaction Nets in terms of a more traditional ('directional') syntax employed by term graph rewriting systems. We can also study the runtime behaviour and properties of rewrite rule systems generated for Interaction Nets and exploit them in producing more efficient implementations with respect to locality of computation, garbage collection, etc. Nor should we underestimate the fact that since MONSTR is a basic execution model for reduction machines (Watson et al. 1988), the translation route from Interaction Nets to MONSTR presented in this paper is effectively a parallel (distributed) realization of the Interaction Net formalism.

The rest of the paper is organized as follows. The following section describes MONSTR and some of its more important properties. Section 2 does the same for Interaction Nets, following the treatment of Banach (1995). The two models are brought together in section 3 which describes a translation of typed Interaction Nets into MONSTR. Section 4 shows how untyped Interaction Nets may also be translated. Section 5 presents the translation to equivalent MONSTR rule systems of some concrete Interaction Nets examples and compares the mapping framework with other similar ones. It also discusses some practical ramifications related to efficient implementation. Section 6 discusses some generalizations of the net model based on MONSTR's properties. Section 7 offers concluding remarks and related and further work, including why a corresponding modification of MONSTR emulating the properties of Interaction Nets is not appropriate in the present work.

1 MONSTR

Unlike most typical graph rewriting formalisms such as the ones developed by Barendregt et al., Ehrig et al. and Sleep et al. (see collections of relevant papers in, for example, Ehrig et al. (1991), Sleep et al.

(1993), TCS (1993)), MONSTR was designed with the repercussions of efficient distributed implementation uppermost in mind. This meant tuning the expressiveness of the basic atomic actions of the model to the capabilities of a typical distributed architecture, so as not to overtax the synchronization properties of the latter unduly – something which would lead to a dramatic loss of performance as a result of having to implement a lot of distributed locking.

1.1 MONSTR rewrites

The fundamental objects of MONSTR are **term graphs**. A term graph is a directed graph where the nodes are labelled with symbols, assumed of fixed arity, and each node has a sequence of out-arcs to its child nodes. The nodes and arcs of term graphs are marked to control rewriting strategy, as we will see below. The term graph that represents the instantaneous state of the computation is modified by the application of some rule. Let us look at a rule in action, to see what happens during a **rewrite**:

$$F[Cons[a\ b]\ s{:}Var] \implies \#G[a\ {}^{\wedge}{*}b],\ s := {*}SUCCEED;$$

First the left-hand side (the part before $\implies$) is matched. F is the root node and has two children, the *Cons* node, and the *Var* node. The *Cons* node has two unlabelled children; such undefined nodes may match anything. Note that the pattern is shallow; this is fundamental to MONSTR as large patterns demand large-scale locking to ensure atomicity.

Once a match is located, which must be at an active ($*$-marked) node of the graph, the nodes on the right-hand side are built into the redex area. Thus a once-suspended (#-marked) G node is constructed, with arcs to the existing left-hand side nodes referred to by a and b (so these nodes become shared even if they were not previously). Also the arc to b is a notification arc ($\wedge$-marked). The other new node is the active *SUCCEED* node.

The notation $\implies$ indicates that the root is to be redirected to the node immediately following the $\implies$, i.e. G. Also the *Var* node is to be redirected to *SUCCEED* by the notation $s := SUCCEED$. During redirection, all in-arcs to the respective redirection subjects (i.e. F and *Var*) are replaced by in-arcs to the respective targets (i.e. G and *SUCCEED*). Redirection is the fundamental notion of update in term graph rewriting, being a graph-oriented version of substitution.

The final tasks of a MONSTR rewrite are to make the root inactive (idle, written visibly as ε when necessary); and to activate specified left-hand side nodes (which causes them to be marked active if otherwise unmarked). In the concrete syntax, this is accomplished by mentioning the relevant nodes on the right-hand side of the rule, with a $*$ marking, e.g. b above. We illustrate the action of the rule described above in Fig. 1. In Fig. 1, note how the in-arcs of F now point to G after redirection, and those of *Var* point to *SUCCEED*. In the rewrite illustrated here, we are assuming that the left-hand side nodes F and *Cons* had no further in-arcs, and thus became inaccessible and were garbage collected.

The above assumed that there was a rule which matched, and that the explicitly matched arguments of the root of the redex (i.e. those arguments whose symbol needs to be inspected for pattern matching to succeed, *Cons* and *Var* in our example), are idle. If any of the explicitly matched arguments of the root is not idle then **suspension** occurs, in which the root of the redex becomes suspended on as many of its explicitly matched arguments as happen to be non-idle; i.e. the root node acquires that many suspension markings, and each of the relevant out-arcs becomes a notification arc (i.e. $\wedge$-marked).

If no rule can match regardless of the markings, then **notification** occurs, in which the root becomes idle, and for all its notification in-arcs, the $\wedge$-marking is removed, and the number of suspensions (#s) in

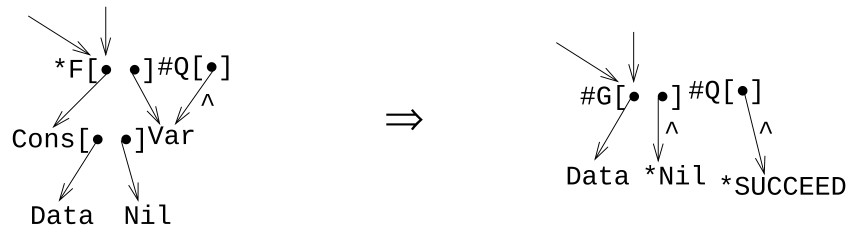

Figure 1: MONSTR rewrite.

the parent node's marking is decremented (with $\#^0 = *$). In this manner subcomputations can signal their completion to their parents (and suspended rewrites can thereby be reawakened).

1.2 MONSTR syntactic restrictions and runtime properties

To make the above ideas into a computational model suited to distributed machines, a number of restrictions are imposed on the syntactic structure of systems so that some useful runtime properties can be rigorously demonstrated. We point out the main ones now rather informally, referring the reader to Banach (1996a), Banach (1997a) for a thorough study (in the context of the formal semantics of MONSTR) of why these are appropriate and what their consequences are.

- All nodes respect the arities of their symbols (within rules; and by means of a simple induction, within all execution graphs).

- The alphabet of symbols is divided into **functions**, **constructors** and **stateholders**. Functions label root nodes of left-hand sides of rules (but not subroot nodes), and function symbols must always have at least one **default rule** which has no explicitly matched arguments, enabling such a rule always to rewrite at runtime, regardless of its arguments. Constructors and stateholders can label subroot nodes of left-hand sides of rules (but not the root nodes). Functions and stateholders (but not constructors) can label left-hand side nodes of redirections, and all redirections must specify an explicit function or stateholder as left-hand side node (one of which must be the root). Thus no attempt is ever made to redirect a constructor at runtime.

- The pattern matching requirements of each redex depend solely on the symbol at the root (and so can be delegated to simple hardware). More specifically, each function symbol has a fixed matching template, one level deep, which specifies which of the root's children need to have their symbols inspected to match a non-default rule for the function. Furthermore, a single fixed position within this template can be designated for matching stateholders; the other positions may only match constructors. (This explains the MONSTR acronym: it stands for a **M**aximum of **O**ne **N**on-root **ST**ateholder per **R**ewrite.) No pointer equality testing is permitted except for the matched constructors (and for some special built-ins of which we will have no need in this paper).

- All nodes in rules are **balanced**, i.e. they have exactly as many suspension markings as they have notification out-arcs. (By a simple induction, all nodes in all execution graphs are balanced too.)

- All nodes in rules are either **state saturated** (i.e. if they have one or more notification in-arcs and are idle, then they must be stateholders), or designated for **activation**.
- Any redirection whose right-hand side node is idle and not activated must be a stateholder. (As a consequence of this and the previous point, all nodes in all execution graphs are state saturated.)
- In all rules, the left-hand side node of a redirection should not be activated unless it is also the right-hand side of some other redirection. (In practice this enables the convenient representation of rewriting by packet store manipulations, and particularly the representation of most redirections by packet overwriting.)

By convention, rewriting always starts with a single active node labelled *INITIAL*; and MONSTR provides a rule selection policy which permits non-default rules to be selected before default rules when either would match (`;` is the sequential rule selector in concrete syntax). Note that we have said nothing very specific about garbage collection. The general idea is that active and *Root*-labelled nodes are live, and liveness is propagated down normal arcs and up notification arcs; see Banach (1996a), Banach (1997a) for a more precise discussion. The implicit mark-scan strategy that such a scheme embodies can be considerably simplified when we restrict to a linear subset (see below).

2 Interaction Nets

Interaction Nets were invented for describing fine-grained computations graphically. Their theory builds on prior work in multiplicative Linear Logic that gives the Interaction Net model particularly transparent properties regarding confluence, and to a lesser extent normalization. We use the formulation of Banach (1995) as it is more convenient for the translation that we subsequently give.

Interaction Nets can be viewed as bipartite graphs where the two node kinds are **agent** nodes and **port** nodes. Each agent bears a **symbol**, which determines the number of **port edges** incident on it, and the attributes of those edges. These port edge attributes are: the port edge's **name**; whether it is **principal** or **auxiliary**; and the port edge's **type**. The types come in complementary pairs ($\alpha+$, $\alpha-$), for α drawn from a suitable type alphabet. Exactly one of an agent's ports is principal, and the rest are auxiliary. Finally, we have the all-important port invariant which states that at most two port edges may be incident on a port node, and that they must be of complementary types, say $\alpha+$ and $\alpha-$. Figure 2 illustrates the situation and also introduces the notion of **port connection**, which we will use as required below. Note that we indicate principal port edges using an arrowhead, while auxiliary port edges are unadorned. Also we will suppress some of the detail to avoid clutter in future. We will say that a port connection consisting of two principal port edges is a principal port connection. An Interaction Net rewrite rule has, on the left-hand side, two agents joined by a principal port connection, and with all their auxiliary ports free (i.e. not connected to other port edges). The right-hand side is an arbitrary Interaction Net with the same external interface as the left-hand side, which is to say that part of the rule's data is a bijective mapping between the free port edges of the left-hand side and right-hand side nets, which preserves the types. The only exceptions to the bijective law are **short circuits**, where two free port nodes of the left-hand side with complementary types are allowed to be identified in the right-hand side. Figure 3 shows a picture of a rule. The numbers on the interface port edges define the aforementioned bijection between left- and right-hand sides. The blobs labelled ($2{:}\beta+$, $9{:}\beta-$) and ($3{:}\gamma-$, $4{:}\gamma+$) are the short circuits in an obvious notation. Any fresh port nodes introduced in the right-hand side, i.e. port nodes not belonging to the

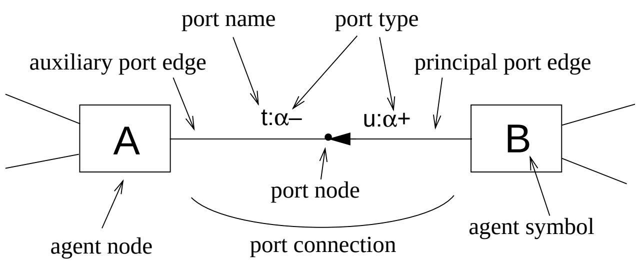

Figure 2: Port connection in an Interaction Net.

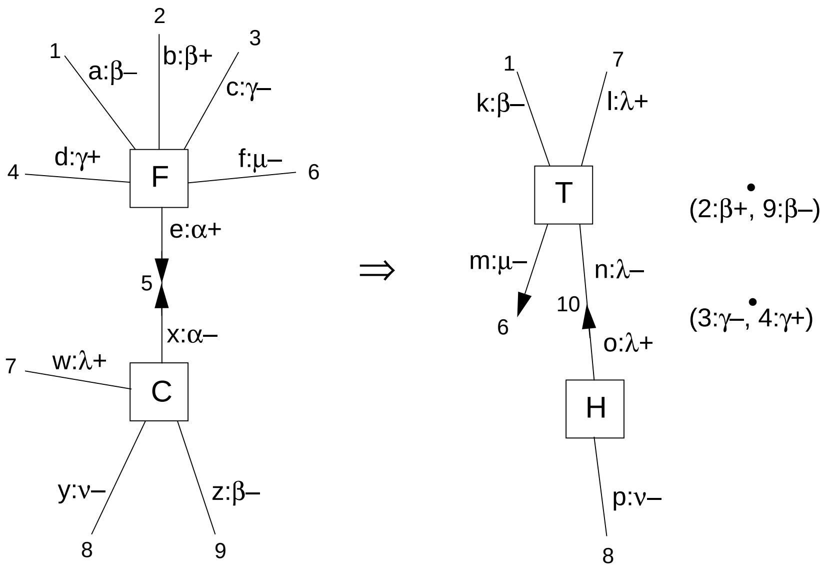

Figure 3: Interaction Net rewrite rule.

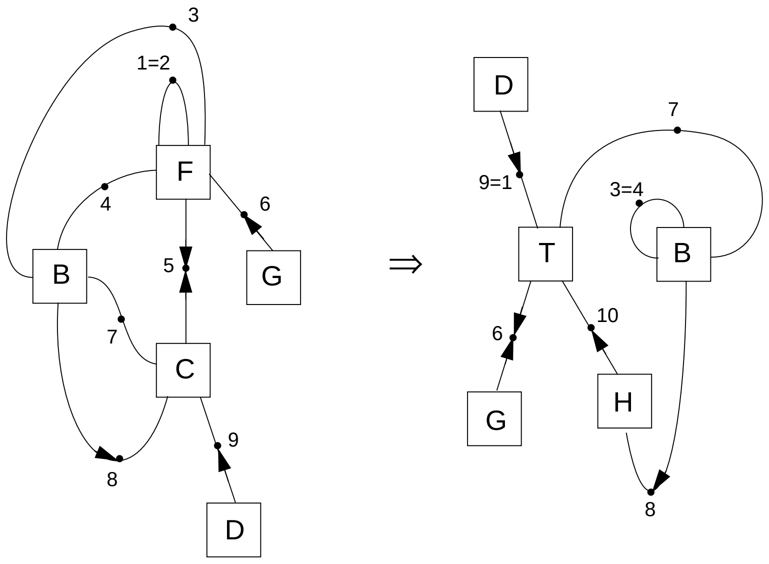

Figure 4: The action of an Interaction Net rewrite rule.

interface, will be called **internal ports** in future (port node 10 in Fig. 3). The operational semantics of such a rule starts by finding a matching of the left-hand side of the rule in the net being rewritten. Then the matched subnet is removed, and replaced with a copy of the right-hand side of the rule. It is easy to see that the type preserving bijective law with short circuits of type-matched pairs, means that the port invariant is preserved by rewrites. Figure 4 shows a rewrite according to the rule introduced previously. The principal/auxiliary distinction on ports, and the fact that rules must feature a principal connection, leads to the study of **deadlock prevention** for Interaction Nets. A deadlock is where there is a cycle of agents, each of whose principal port edges connects with an auxiliary port edge of the next agent. Obviously in such a situation, none of the agents involved can ever rewrite. A rich theory can be developed to ensure that such situations cannot arise, but this is not needed in the present paper: see the cited references for details.

We end this brief exposition of Interaction Nets with some further observations. Since each agent has only one principal port, it can interact with at most one other agent, the one connected to the said principal port, and then only if that agent's corresponding port is itself principal. This means that apart from auxiliary port nodes that they might have in common, any two distinct redexes in an arbitrary net are non-overlapping, and provided for each possible pair of agents on the left-hand side there is exactly one rule, Interaction Net rewriting is Church–Rosser. If moreover, the right-hand sides of rules are smaller than their left-hand sides (as is the case, for example, for the Interaction Net version of LLM cut elimination), then Interaction Net rewriting is also terminating.

The form of rules, and the one principal port restriction for agents, combine to imply the following structure for the life history of a port node: it is created during some rewrite. Perhaps the two port edges incident on it are auxiliary. While an incident port edge is auxiliary, it may be replaced by another port edge, or short circuited, as its owning agent interacts along some different port connection. Once an auxiliary port edge is replaced by a principal port edge, however, the port edge is committed. It can no longer be replaced, except if the other incident edge also becomes principal, and the whole connection becomes the redex of a rewrite, at which point the connection, and the pair of connected agents are garbaged. For any given port, the whole of the above is not compulsory, and the port may undergo only a subsequence of the indicated transformations, but the structure of that subsequence must always fit within the indicated pattern. This fact can be usefully exploited by implementations, as we will see below.

3 From Interaction Nets to MONSTR

The most striking thing about Interaction Nets from an implementation point of view is that the port edges are unoriented. (We disregard the arrowheads of the principal port edges for this purpose.) Usually, implementing an unoriented edge in a concrete data structure requires a pair of oppositely oriented pointers. With this in mind, if concurrent update of the data structure by many rewriting processes is envisaged, then unoriented edges can imply disaster in performance, since one has to avoid race conditions arising from two agents competing to update the same edge from opposite ends. This can involve all the overheads of locking and perhaps of deadlock avoidance. By contrast, MONSTR, with a close eye kept on implementation matters, features only directed arcs, avoiding the problems indicated above.

The similar shape of left-hand sides in the two models is striking. The obvious thing that we would like to do is to relate the two agents connected by a principal and undirected port connection in the left-hand side of a net rule, to the function and stateholder connected by a directed arc in a corresponding MONSTR rule – fortunately it is possible to do this if one exploits the orientedness of the Interaction Net type system to provide an orientation for the principal port connection. Indeed it is possible to go further. Noticing that the agents of an Interaction Net computation are inspected and replaced exactly once each (since each agent participates in exactly one interaction), enables us to represent some agents by constructors rather than the more general stateholders.

Somewhat arbitrarily, we choose to encode agents with principal ports of positive type by MONSTR function nodes, and those with principal ports of negative type by MONSTR constructor nodes. Further, we encode port nodes by stateholders labelled with the symbol *Port*. (We really do need stateholders here, as the *Port* nodes represent synchronization points.) To start with, all port edges are represented by arcs from the agent nodes to port nodes, and all port connections are represented by a pair of in-arcs of a *Port* node – in a representation of an Interaction Net that is rather obvious. For a principal connection, we have to turn the inward-pointing pair of arcs into a single arc; furthermore, we have to do this in a manner which respects the independence of the two port edges. Fortunately, the typical life history of a port edge sketched in the previous section helps, as both edges are following similar trains of activity.

At the point that a function node is created, it is created active. It matches its child. If this is a constructor representing an agent, a MONSTR rewrite representing an Interaction Net rewrite takes place. If not, and the function sees only a *Port* node, it suspends waiting to be notified of a change of state. Conversely, the constructor representing the negative type agent is created along with an additional *Assign* function whose job is to redirect the constructor's principal port node to the constructor itself, thereby making the constructor visible to any waiting function node, should there indeed be one there now or at some point in the future. Specifically, the *Assign* activates the constructor, which then notifies any parent suspended on it. In fact it is clear that with the desired behaviour of the *Assign* function, the original arc from the constructor to the *Port* node becomes superfluous, and can be dispensed with. It is easy to see that the protocol works as required, and in particular that it allows auxiliary port edge representatives to be replaced at will. (Obviously, if a principal port connection is being created all at once within the right-hand side of some rule rather than dynamically, then the protocol can be optimized away.) The replacement of agents and of their port edges works by garbage collection. In a rewrite, the nodes representing the right-hand side of the rule are created and connected to the relevant *Port* nodes; meanwhile the left-hand side agent nodes lose all live references and thus become garbaged.

We now give the translation formally. The reader may be slightly concerned that we do not also define formally the language we are translating into. This would, however, require some substantial work which, in addition to making the paper unacceptably long, would probably also sidetrack us from the main, rather practical, pace we have set.

First we write down a generic Interaction Net rule D_{IN}:

LHS: Agents: F, with auxiliary ports $f1 : \alpha 1^\circ \ldots fi : \alpha i^\circ \ldots fn : \alpha n^\circ$
C, with auxiliary ports $c1 : \beta 1^\circ \ldots ci : \beta i^\circ \ldots cm : \beta m^\circ$
where $^\circ$ is either $+$ or $-$ in each case.

RHS: Agents: $E1 \ldots Ei \ldots Er$, with principal ports $\overline{q}1 : \overline{\delta}1^\circ \ldots \overline{q}i : \overline{\delta}i^\circ \ldots \overline{q}r : \overline{\delta}r^\circ$
and with auxiliary ports $q1_1 : \delta 1^\circ_1 \ldots q1_{j1} : \delta 1^\circ_{j1} \ldots q1_{t1} : \delta 1^\circ_{t1}$ (of $E1$) ...
$qi_1 : \delta i^\circ_1 \ldots qi_{ji} : \delta i^\circ_{ji} \ldots q1_{ti} : \delta i^\circ_{ti}$ (of Ei) ...
$qr_1 : \delta r^\circ_1 \ldots qr_{jr} : \delta r^\circ_{jr} \ldots q1_{tr} : \delta r^\circ_{tr}$ (of Er)
Internal ports: $p1 \ldots pi \ldots ps$
Short circuits: $(x1 : \gamma 1-, y1 : \gamma 1+) \ldots (xi : \gamma i-, yi : \gamma i+) \ldots (xu : \gamma u-, yu : \gamma u+)$

where in the above there is an onto mapping

$$\theta : \{\overline{q}1 \ldots \overline{q}r\} \cup \{q1_1 \ldots qr_{tr}\} \cup \{x1 \ldots yu\} \twoheadrightarrow \{f1 \ldots fn\} \cup \{c1 \ldots cm\} \cup \{p1 \ldots ps\} \;.$$

where θ^{-1} is $1-1$ on $\{f1 \ldots fn\} \cup \{c1 \ldots cm\}$, and each $\theta^{-1}(pi)$ is of cardinality 2. (This just expresses the port invariant for the right-hand side.)

In the MONSTR translation, ***bold*** items will correspond to symbols or parts of symbols mapped from the components of the above generic rule, while *italic* items will stand for constants of the translation. In general, we use font change to identify pieces that correspond in the Interaction Nets and MONSTR rules. The MONSTR rule D_{M} that translates the above rule is:

$$\begin{aligned}
&\boldsymbol{F}[\,\boldsymbol{C}[\,\boldsymbol{c1} \ldots \boldsymbol{ci} \ldots \boldsymbol{cm}\,]\ \boldsymbol{f1} \ldots \boldsymbol{fi} \ldots \boldsymbol{fn}] \implies *OK,\\
&\quad \boldsymbol{p1}{:}Port, \ldots, \boldsymbol{pi}{:}Port, \ldots, \boldsymbol{ps}{:}Port,\\
&\quad \boldsymbol{e1}{:}m1\ \boldsymbol{E1}[\boldsymbol{\overline{q}1}\ \boldsymbol{q1_1} \ldots \boldsymbol{q1_{j1}} \ldots \boldsymbol{q1_{t1}}\,], \ldots,\\
&\quad \boldsymbol{ei}{:}mi\ \boldsymbol{Ei}[\,\boldsymbol{\overline{q}i}\ \boldsymbol{qi_1} \ldots \boldsymbol{qi_{ji}} \ldots \boldsymbol{qi_{ti}}\,], \ldots,\\
&\quad \boldsymbol{er}{:}mr\ \boldsymbol{Er}[\,\boldsymbol{\overline{q}r}\ \boldsymbol{qr_1} \ldots \boldsymbol{qr_{jr}} \ldots \boldsymbol{qr_{tr}}\,],
\end{aligned}$$

where if $\boldsymbol{\overline{q}i}$ $:\overline{\delta}i+$ (i.e. $\boldsymbol{Ei}$ is an agent of positive type principal port, and thus $\boldsymbol{Ei}$ is a function symbol),
then $\boldsymbol{\mu i}$ $= *$,

else if $\overline{\boldsymbol{q}}\boldsymbol{i} : \overline{\delta}i-$ (i.e. $\boldsymbol{Ei}$ is an agent of negative type principal port, and thus $\boldsymbol{Ei}$ is a constructor symbol), **then** $\boldsymbol{\mu i} = \boldsymbol{\varepsilon}$, $\overline{\boldsymbol{q}}\boldsymbol{i}$ is absent from the arguments of $\boldsymbol{Ei}$, and we also have

$$*Assign[\overline{\boldsymbol{q}}\boldsymbol{i}\ \boldsymbol{ei}]$$

fi

$$\begin{aligned}
&*\,Assign[\boldsymbol{x1}\ \boldsymbol{y1}], \ldots ,\\
&*\,Assign[\boldsymbol{xi}\ \boldsymbol{yi}], \ldots ,\\
&*\,Assign[\boldsymbol{xu}\ \boldsymbol{yu}];
\end{aligned}$$

where in the above, the map θ is interpreted as syntactic identity, i.e. if $\theta(\overline{\boldsymbol{q}}\boldsymbol{4}) = \boldsymbol{c9}$ say, then $\overline{\boldsymbol{q}}\boldsymbol{4}$ **is identical to** $\boldsymbol{c9}$, giving the connectedness of the corresponding term graph according to the syntactic conventions of MONSTR.

In addition we need the following suite of rules:

$$\begin{aligned}
&Assign[v{:}Port\ a] \implies *OK, v := *a;\\
&Assign[v\ a] \implies \#Assign[{}^{\wedge}*v\ a];\\
&\boldsymbol{F}[p{:}Port\ \boldsymbol{f1} \ldots \boldsymbol{fi} \ldots \boldsymbol{fn}] \implies \#\boldsymbol{F}[{}^{\wedge}p\ \boldsymbol{f1} \ldots \boldsymbol{fi} \ldots \boldsymbol{fn}];\\
&\boldsymbol{F}[p\ \boldsymbol{f1} \ldots \boldsymbol{fi} \ldots \boldsymbol{fn}] \implies \#\boldsymbol{F}[{}^{\wedge}*p\ \boldsymbol{f1} \ldots \boldsymbol{fi} \ldots \boldsymbol{fn}];
\end{aligned}$$

Here is the translation of the specific example we had previously:

$$\begin{aligned}
&F[C[c7\ c8\ c9]\ f1\ f2\ f3\ f4\ f6] \implies *OK,\\
&\quad p10{:}Port,\\
&\quad e1{:}T[f1\ c7\ p10], *Assign[f6\ e1],\\
&\quad e2{:}*H[p10\ c8],\\
&\quad *Assign[c9\ f2], *Assign[f3\ f4];
\end{aligned}$$

In Fig. 5 we show what happens in the application of this rule to the translation of the net we treated earlier, after all the *Assign*s have done their work. (We note that if multiple redirections were available in MONSTR – as they are in Dactl, of which MONSTR is a sublanguage – then the *Assign*s would not be necessary; we could match a larger pattern and do all the required redirections performed by the *Assign*s in one fell swoop, modulo considerations of locality, of course.)

4 Untyped Interaction Nets

It turns out that much of the theory of Interaction Nets can be carried through without the presence of a type system such as we exploited above (see Banach (1995)). This is because the principal/auxiliary property of ports is already a kind of crude but effective type system. It is therefore interesting to see if a reasonable translation can be concocted without the simplifying influence of orientedness. In this section we show that one can; in fact we describe two schemes, the second of which builds on properties of the first.

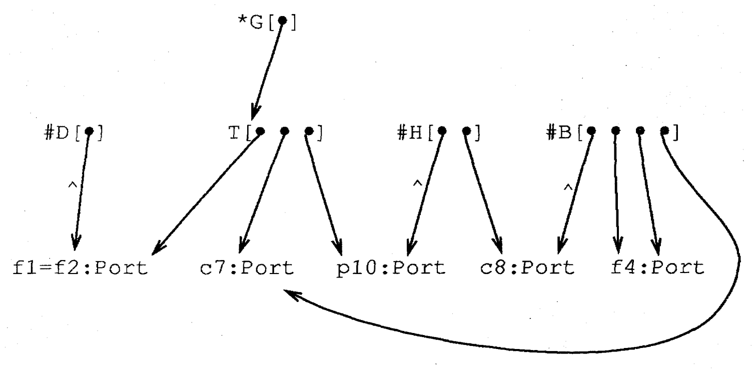

Figure 5: Application of MONSTR rule.

4.1 The first translation

The main novelty, of course, is that Interaction Net rules are now truly symmetrical in the two agents that occur on the left-hand side. Therefore neither can be deemed to be ‘in charge’ of the rewrite as previously, and we must look for a more symmetrical solution. One way is to allow the port node of the principal port connection itself to take control. We replace the simple *Port* node at the centre of a port connection of the previous translation by a gadget consisting of a *Rew* function node with two *Port* node children, as shown in Fig. 6. The gadget is connected to the two respective agent nodes, either using an *Assign* function when the port in question is a principal port, or by a conventional port edge coming from the agent when the port is auxiliary. One of each is shown in the figure. The basic idea is that *Assign* redirects the relevant child of *Rew* to an agent node, causing one notification to *Rew*, and when *Rew* has been notified twice, it has two agent children and so can model the relevant Interaction Net rewrite. We leave it to readers to convince themselves that the gadget works properly, i.e. that it controls rewrites in a suitable way, even in the presence of short circuits (which are now undirected of course). In effect, the gadget is a kind of composable concurrent data structure, in today’s terminology.

For the schematic rule translated in the previous section, but with the types now elided, we find the following new translation. For each agent symbol X we will need two MONSTR symbols, $\boldsymbol{X}$ and $Rew\_\boldsymbol{X}$, the former a constructor to be matched by *Rew*, the latter a function incorporating *Rew*’s memory of the first agent it matched.

$$Rew[p{:}Port\ a] \implies *OK, p := *a;$$
$$Rew[\boldsymbol{F}[\boldsymbol{f1} \ldots \boldsymbol{fn}]\ \boldsymbol{c}] \implies *Rew\_\boldsymbol{F}[\boldsymbol{c}\ \boldsymbol{f1} \ldots \boldsymbol{fn}];$$
$$Rew[a\ b] \implies \#\#Rew[^{\wedge}{*a}\ ^{\wedge}{*b}];$$

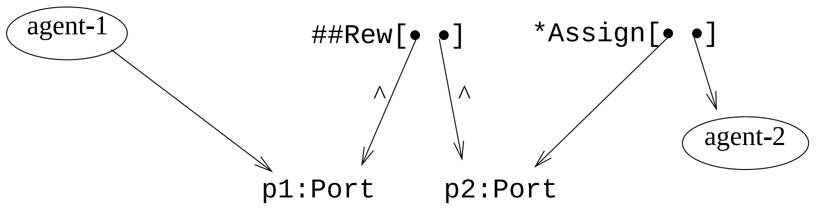

Figure 6: *Rew* function node with two *Port* node children.

(Similarly $Rew[C[\boldsymbol{c1} \ldots \boldsymbol{cm}]\,\boldsymbol{f}] \implies \ldots$. We do not give the symmetrically paired rules.)

$$\begin{aligned}
&Rew\_\boldsymbol{F}[p{:}Port\ \boldsymbol{f1} \ldots \boldsymbol{fn}] \implies \#Rew\_\boldsymbol{F}[{}^{\wedge}p\ \boldsymbol{f1} \ldots \boldsymbol{fn}];\\
&Rew\_\boldsymbol{F}[\boldsymbol{C}[\boldsymbol{c1} \ldots \boldsymbol{ci} \ldots \boldsymbol{cm}]\ \boldsymbol{f1} \ldots \boldsymbol{fi} \ldots \boldsymbol{fn}] \implies *OK,\\
&\quad \boldsymbol{rew1}{:}\#\#Rew[{}^{\wedge}\boldsymbol{p1}.1{:}Port\,{}^{\wedge}\boldsymbol{p1}.2 : Port], \ldots,\\
&\quad \boldsymbol{rewi}{:}\#\#Rew[{}^{\wedge}\boldsymbol{pi}.1{:}Port\,{}^{\wedge}\boldsymbol{pi}.2{:}Port], \ldots,\\
&\quad \boldsymbol{rews}{:}\#\#Rew[{}^{\wedge}\boldsymbol{ps}.1{:}Port\ {}^{\wedge}\boldsymbol{ps}.2 : Port],\\
&\quad \boldsymbol{e1}{:}\boldsymbol{E1}[\boldsymbol{q1}_{\boldsymbol{1}} \ldots \boldsymbol{q1}_{\boldsymbol{j1}} \ldots \boldsymbol{q1}_{\boldsymbol{t1}}], *Assign[\boldsymbol{\overline{q}1}\ \boldsymbol{e1}], \ldots,\\
&\quad \boldsymbol{ei}{:}\boldsymbol{Ei}[\boldsymbol{qi}_{\boldsymbol{1}} \ldots \boldsymbol{qi}_{\boldsymbol{ji}} \ldots \boldsymbol{qi}_{\boldsymbol{ti}}], *Assign[\boldsymbol{\overline{q}i}\ \boldsymbol{ei}], \ldots,\\
&\quad \boldsymbol{er}{:}\boldsymbol{Er}[\boldsymbol{qr}_{\boldsymbol{1}} \ldots \boldsymbol{qr}_{\boldsymbol{jr}} \ldots \boldsymbol{qr}_{\boldsymbol{tr}}],\ *Assign[\boldsymbol{\overline{q}r}\ \boldsymbol{er}],\\
&\quad *Assign[\boldsymbol{x1}\ \boldsymbol{y1}], \ldots, *Assign[\boldsymbol{xi}\ \boldsymbol{yi}], \ldots,\\
&\quad *Assign[\boldsymbol{xu}\ \boldsymbol{yu}];
\end{aligned}$$

where the previous mapping θ now becomes a genuine bijection

$$\theta : \{\overline{q}1 \ldots \overline{q}r\} \cup \{q1_1 \ldots qr_{tr}\} \cup \{x1 \ldots yu\} \leftrightarrow \{f1 \ldots fn\} \cup \{c1 \ldots cm\} \cup \{p1.1 \ldots ps.2\},$$

interpreted as syntactic identity in the MONSTR rule (this change being provoked by the replacement of the original internal *Port* nodes by gadgets having two such *Port* nodes, of course).

The key rule in our specific example now becomes:

$$\begin{aligned}
&Rew\_F[C[c7\ c8\ c9]\ f1\ f2\ f3\ f4\ f6] \implies *OK,\\
&\quad rew{:}\#\#Rew[{}^{\wedge}p10.1{:}Port\ {}^{\wedge}p10.2{:}Port],\\
&\quad e1{:}T[f1\ c7\ p10.1], *Assign[f6\ e1],\\
&\quad e2{:}H[c8], *Assign[p10.2\ e2],\\
&\quad *Assign[c9\ f2], *Assign[f3\ f4];
\end{aligned}$$

Figure 7 illustrates the situation after the rewrite of the example using the rule above, but before all the ‘plumbing’ rewrites have completed. To economize on space in this figure, we abbreviate *Assign* to *Ass*,

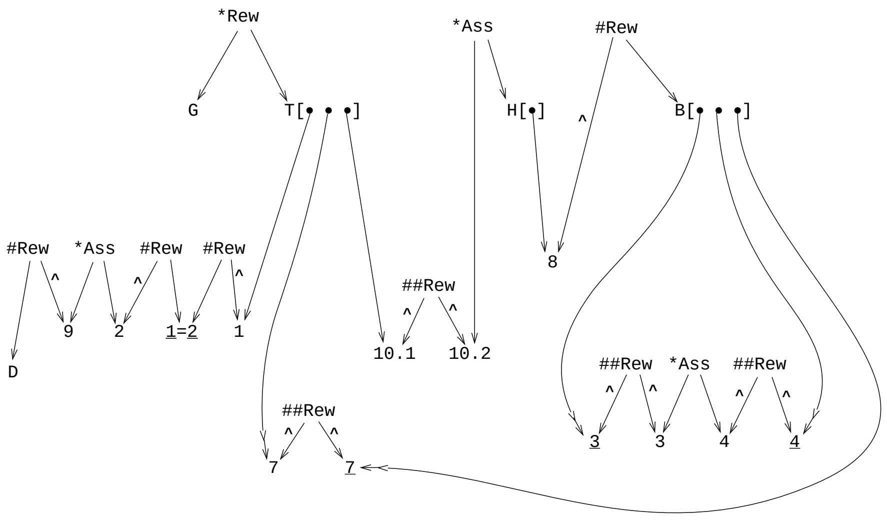

Figure 7: After rewriting.

and the various port nodes to just numbers, with underlined numbers standing for the other *Port* nodes in the gadgets of the rewrite's interface. The notation $\underline{1} = \underline{2}$ indicates that *Port* nodes $\underline{1}$ and $\underline{2}$ were short circuited at some previous point and are now one node.

4.2 The second translation

The second translation is inspired by the first rule for *Rew*. This rule is needed to ensure the *Rew*/ *Port*/ *Port* gadget works properly when two of them have been short circuited. Note that it behaves much like the rule for *Assign*. This suggests that we model each rewrite as the resolution of a short-circuit-like competition between the two principal port edges involved in the principal connection. In such a scheme, *Rew* functions are allocated per principal port edge, rather than per port node as previously, and the *Assign*s and additional *Port* nodes of the first translation become superfluous. The allocation of *Rew*s per principal port edge effectively creates them lazily, since when a port node is first created, there is no need for either incident port edge to be principal. The previous translation creates the *Rew*s eagerly, and thus is less efficient. We now need just one *Port* node per port node as before. The rules for *Rew* are

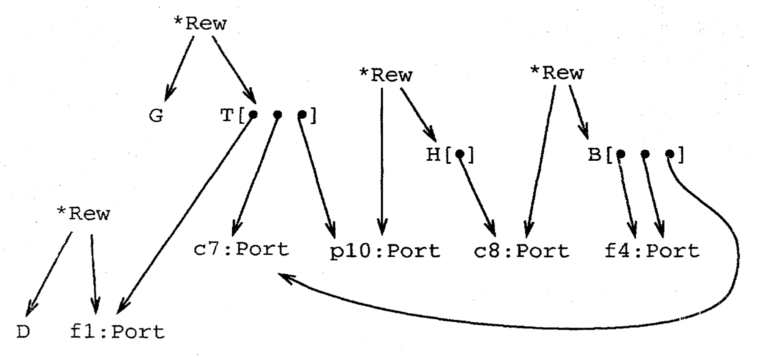

Figure 8: The situation just after the rewrite.

unchanged, but the right-hand sides of the *Rew_F* rules are different. Here is the key generic rule (other rules are unaltered).

$$
\begin{aligned}
&\mathit{Rew\_}\boldsymbol{F}[\boldsymbol{C}[\boldsymbol{c1}\ldots\boldsymbol{ci}\ldots\boldsymbol{cm}]\ \boldsymbol{f1}\ldots\boldsymbol{fi}\ldots\boldsymbol{fn}] \implies *OK,\\
&\quad \boldsymbol{p1}{:}Port,\ldots,\\
&\quad \boldsymbol{pi}{:}Port,\ldots,\\
&\quad \boldsymbol{ps}{:}Port,\\
&\quad \boldsymbol{e1}{:}\boldsymbol{E1}[\boldsymbol{q1}_1\ldots\boldsymbol{q1}_{j1}\ldots\boldsymbol{q1}_{t1}],*Rew[\overline{\boldsymbol{q1}}\ \boldsymbol{e1}],\ldots,\\
&\quad \boldsymbol{ei}{:}\boldsymbol{Ei}[\boldsymbol{qi}_1\ldots\boldsymbol{qi}_{ji}\ldots\boldsymbol{qi}_{ti}],*Rew[\overline{\boldsymbol{qi}}\ \boldsymbol{ei}],\ldots,\\
&\quad \boldsymbol{er}{:}\boldsymbol{Er}[\boldsymbol{qr}_1\ldots\boldsymbol{qr}_{jr}\ldots\boldsymbol{qr}_{tr}],*Rew[\overline{\boldsymbol{qr}}\ \boldsymbol{er}],\\
&\quad *Assign[\boldsymbol{x1}\,\boldsymbol{y1}],\ldots,*Assign[\boldsymbol{xi}\,\boldsymbol{yi}],\ldots,\\
&\quad *Assign[\boldsymbol{xu}\,\boldsymbol{yu}];
\end{aligned}
$$

The mapping θ is as in the typed case. We quote the main rule of the running example in the second translation:

$$
\begin{aligned}
&Rew\_F[C[c7\ c8\ c9]\ f1\ f2\ f3\ f4\ f6] \implies *OK,\\
&\quad p10{:}Port,\\
&\quad e1{:}T[f1\ c7\ p10],\quad *Rew[f6\ e1],\\
&\quad e2{:}H[c8],\quad *Rew[p10\ e2],\\
&\quad *Assign[c9\ f2],\quad *Assign[f3\ f4];
\end{aligned}
$$

A picture of the situation just after the rewrite and having completed the *Assign*s but not the *Rew*s, appears in Fig. 8.

5 Examples and comparison with similar work

In this section we provide the MONSTR code for a number of concrete programming examples, covering both the cases of typed and untyped nets; this code has run successfully on the Dactl interpreter ((Glauert

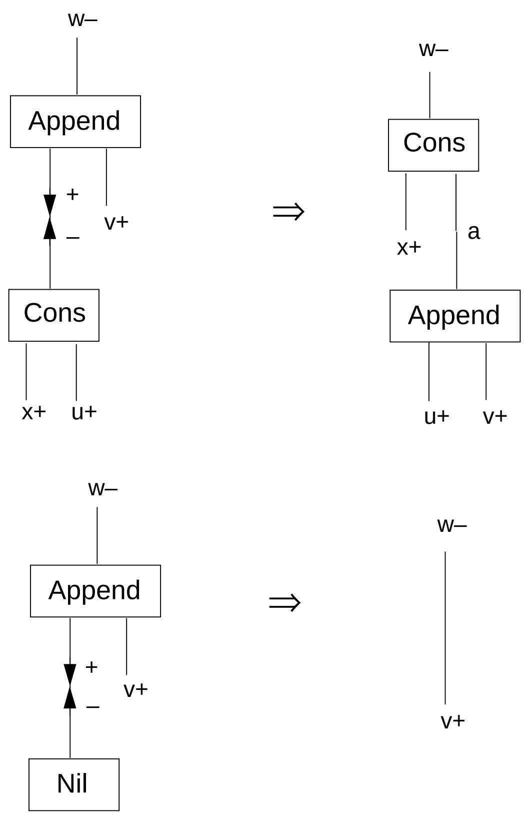

Figure 9: Visual representation of *Append* in Interaction Nets.

et al. 1988, Glauert et al. 1990)). We also compare the resulting MONSTR code with that produced when programs written in other computational models, similar to Interaction Nets, are likewise translated to sets of MONSTR rewrite rules. This comparison further highlights some features of MONSTR and also puts each into a wider perspective.

5.1 Typed examples

We start with the typed version of the unavoidable *Append*. In order to make it easier for the reader to understand how the generic Interaction Nets to MONSTR translation presented in section 4 is used to generate the MONSTR program, we show below both the 'visual' and textual representation of *Append* in Interaction Nets. The visual representation is in Fig. 9. (The convention we adopt in this paper is that the principal ports of the 'functions' have a positive sign and those of the 'constructors' have a negative one; the opposite convention would of course be equally valid provided it were used consistently, and indeed that is the one adopted in Lafont (1990), Lafont (1991).) The textual equivalent of Fig. 9 now follows, where the type of the principal port is shown first in the symbol declarations.

type *atom, list*

symbol *Cons:list-;atom+,list+*
Nil:list−
Append:list+;list+,list−

Cons[x Append(v,t)] >< Append[v,Cons(x,t)]
Nil >< Append[v,v]

It should now be easier for the reader to understand how the following MONSTR rule system is derived.

$$
\begin{aligned}
&Append[Cons[x\ u]\ w\ v] \implies *OK,\\
&\quad a{:}Port,\\
&\quad e1{:}Cons[x\ a],\\
&\quad e2{:}*Append[u\ a\ v],\\
&\quad *Assign[w\ e1];\\
&Append[Nil\ w\ v] \implies *OK,\\
&\quad *Assign[w\ v];\\
&Append[p{:}Port\ w\ v] \implies \#Append[{}^\wedge p\ w\ v];\\
&Append[p\ w\ v] \implies \#Append[{}^\wedge * p\ w\ v];
\end{aligned}
$$

A typical MONSTR query involving the above program is shown below. (We recall that *INITIAL* denotes the first piece of graph to be attempted for reduction in a MONSTR/Dactl program.)

$$
\begin{aligned}
&INITIAL \implies p{:}Port,\\
&\quad *Append[l1\ p\ l2],\\
&\quad l1{:}Cons[1\ Cons[2\ Nil]],\\
&\quad l2{:}Cons[3\ Cons[4\ Nil]];
\end{aligned}
$$

In the above rule for *INITIAL*, it is easy to see the two arguments of *Append* and how the code ought to work. It is the optimization described earlier of the following, where the right-hand side of the initial graph is constructed in the longwinded way.

$$
\begin{aligned}
&INITIAL \implies p{:}Port,\\
&\quad *Append[l1\ p\ l2],\\
&\quad l1{:}Cons[1\ p11], *Assign[p11\ Cons[2\ p12]],\\
&\qquad *Assign[p12\ Nil],\\
&\quad l2{:}Cons[3\ p21], *Assign[p21\ Cons[4\ p22]],\\
&\qquad *Assign[p22\ Nil],\\
&\quad p11{:}Port, p12{:}Port, p21{:}Port, p22{:}Port;
\end{aligned}
$$

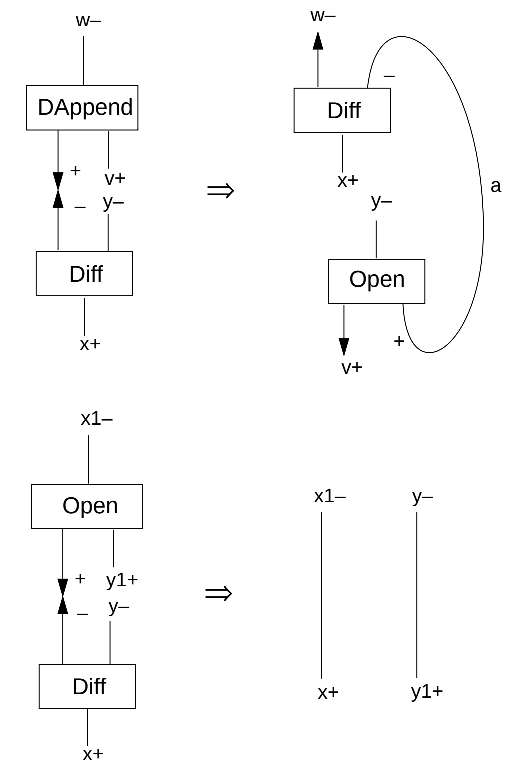

Figure 10: Visual representation of *DAppend*.

Note that, as Fig. 9 shows, the result of appending the two lists will appear in the second argument of *Append*.

The next program is a variant of *Append*, referred to as *DAppend*, which makes use of difference lists. Aside from the obvious usefulness of once more demonstrating the mapping procedure from Interaction Nets to MONSTR which we have described, the example illustrates some features of Interaction Nets related more to concurrent logic (Shapiro 1989) than to functional programming. We discuss this issue at greater length later. As for the case of ordinary *Append*, we show both the visual (in Fig. 10) and the textual representation of *DAppend*.

type *d_list*

symbol *Diff:d_list−;list+,list−*
DAppend:d_list+;d_list+,d_list−
Open:d_list+;list+,list−

Diff[x,y] >< DAppend[Open(t,y),Diff(x,t)]
Diff[x,y] >< Open[y,x]

Note the use of the auxiliary agent *Open* to bypass the constraints imposed by Interaction Nets in the formation of rules. The equivalent MONSTR code follows.

```
DAppend[Diff[y x] w v] ⟹ *OK,
   a:Port,
   e1:Diff[a x], *Assign[w e1],
   e2:*Open[v y a];
DAppend[p:Port w v] ⟹ #DAppend[^p w v];
DAppend[p w v] ⟹ #DAppend[^*p w v];
Open[Diff[y x] x1 y1] ⟹ *OK,
   *Assign[x1 x],
   *Assign[y y1];
Open[p:Port x1 y1] ⟹ #Open[^p x1 y1];
Open[p x1 y1] ⟹ #Open[^*p x1 y1];
```

A typical query involving *DAppend* could be formulated as follows (we do not bother with the unoptimized case this time). As before, the expected result will be produced along *DAppend*'s second argument.

```
INITIAL ⟹ w:Port,
   *DAppend[a w v],
   a:Diff[y Cons[1 Cons[2 y]]],
   v:Diff[z Cons[3 z]],
   y:Port, z:Port;
```

5.2 Untyped examples

In order to illustrate via the use of concrete examples how the untyped version of Interaction Nets programs is translated to MONSTR by means of the methodology developed in section 5, we make use once more of *Append*. The reader can refer again to Fig. 9, but the plus and minus signs on ports should now be ignored. The equivalent MONSTR code is given in Fig. 11; for brevity we use only the second (optimized) version of the translation scheme as described in section 5.2.

We recall from section 5 that the driving force in performing reductions in untyped Interaction Nets is the *Rew* function specialized with respect to each program. In fact, all other Interaction Nets agents, such as *Append*, *Cons* and *Nil* in the above example, are 'constructors' (in a way, one could envision all ports in such agents to be of negative sign with the *Rew* ports being of positive sign, thus triggering the reductions). A typical query now takes the following rather longwinded form. (In case the reader finds it hard to see the wood for the trees, this is the appending of [1], [2] to [3],[4] as previously. The advantages

```
Rew[p:Port a] ⟹ *OK, p := *a;
Rew[Append[w v] c] ⟹ *Rew_Append[c w v];
Rew[Cons[x u] f] ⟹ *Rew_Cons[f x u];
Rew[Nil f] ⟹ *Rew_Nil[f];
Rew[a b] ⟹ ##Rew[^*a ^*b];

Rew_Append[Cons[x u] w v] ⟹ *OK,
   a:Port,
   e1:Cons[x a], *Rew[w e1],
   e2:Append[a v], *Rew[u e2];
Rew_Append[p:Port w v] ⟹ #Rew_Append[^p w v];
Rew_Append[p w v] ⟹ #Rew_Append[^*p w v];

Rew_Cons[Append[w v] x u] ⟹ *OK,
   a:Port,
   e1:Cons[x a], *Rew[w e1],
   e2:Append[a v], *Rew[u e2];
Rew_Cons[p:Port x u] ⟹ #Rew_Cons[^p w u];
Rew_Cons[p x u] ⟹ #Rew_Cons[^*p w v];

Rew_Nil[Append[w v]] ⟹ *OK,
   *Assign[w v];
Rew_Nil[p:Port] ⟹ #Rew_Nil[^p];
Rew_Nil[p] ⟹ #Rew_Nil[^*p];
```

Figure 11: Equivalent MONSTR code.

of a typed over an untyped system are obvious.)

$$
\begin{aligned}
&INITIAL \implies p{:}Port, a{:}Port,\\
&\quad *Rew[a\ l1], *Rew[a\ Append[p\ l2]],\\
&\quad *Rew[l1\ Cons[c11\ c1r1]],\\
&\quad *Rew[l2\ Cons[c31\ c3r1]],\\
&\quad *Rew[c11\ 1],\\
&\quad *Rew[c1r1\ Cons[c21\ c2r1]],\\
&\quad *Rew[c21\ 2], *Rew[c2r1\ Nil],\\
&\quad *Rew[c31\ 3],\\
&\quad *Rew[c3r1\ Cons[c41\ c4r1]],\\
&\quad *Rew[c41\ 4], *Rew[c4r1\ Nil],\\
&\quad l1{:}Port, l2{:}Port,\\
&\quad c11{:}Port, c21{:}Port, c31{:}Port, c41{:}Port,\\
&\quad c1r1{:}Port, c2r1{:}Port, c3r1{:}Port, c4r1{:}Port;
\end{aligned}
$$

5.3 Discussion

There are many advantages in mapping different computational models and associated languages onto MONSTR. One is that the latter provides a common point of reference in comparing such models among themselves as well as with MONSTR (Glauert et al. 1988). Another is that one can reason about the behaviour of some model or language by transposing the discussion to the level of MONSTR where the rewriting and interaction (in the general sense) between processes or agents becomes more explicit (Banach et al. 1995, Banach and Papadopoulos 1995b). Finally, MONSTR provides a natural implementation apparatus for a variety of such computational models and languages (Banach 1993, Banach and Papadopoulos 1993, Banach and Papadopoulos 1995a, Banach and Papadopoulos 1995b, Watson et al. 1988).

In the context of the present work, the mapping of Interaction Nets onto MONSTR serves, among others, two purposes:

- to provide an implementation apparatus for Interaction Nets in a distributed environment via MONSTR
- to illustrate how Interaction Net graphs can be transformed into 'ordinary' rewrite-rule based code, thus making the framework developed applicable to a variety of other Interaction Net based rewriting formalisms.

One of MONSTR's rather unusual features is the notion of non-root redirection which is something that can befall an **overwritable** node (as it is called in the terminology of term graph rewriting systems) many times. For instance, it is perfectly acceptable to write a rule performing **destructive assignment** as follows:

$$Assign[v : VAR[old\_val]new\_val] \implies *OK, \quad v := *VAR[new\_val];$$

where *VAR* has been declared as an *OVERWRITABLE* node (like *Port* in the Interaction Nets to MONSTR mapping presented above). We recall that the operations indicated by the above rule (i.e. the rewriting of the redex to *OK* and the redirection of the node *VAR* to a new one whose child is the new value to be assigned) are done as a single atomic action. We also recall that one of the fundamental differences between MONSTR and its predecessor (and in a way 'superset') Dactl, is that the latter allows arbitrary numbers of such non-root overwrites to be performed atomically within the same rewrite. The power of this mechanism, but alas also the futility of trying to implement it efficiently on a real parallel and/or distributed environment, becomes apparent when one notices that multiple and atomic non-root overwrites are more or less comparable to performing atomic unification in the CP-family of concurrent logic languages (Shapiro 1989). ('More or less' means that unification is a two-way pattern matching operation, as opposed to the ordinary one-way matching employed by graph rewrite-rule systems, but it is also done once per variable – the single assignment property – whereas an overwritable MONSTR or Dactl node can be rewritten arbitrarily many times.) The CP-family is a very expressive language model, which, however, was never implemented efficiently and was subsequently abandoned in favour of variants that were weaker but easier to implement.

Nevertheless, allowing multiple non-root overwriting, even of only a single such node (per rewrite rule), is a potentially puzzling feature bearing in mind that the graph rewriting formalism has been used traditionally as an implementation framework for declarative (functional, logic) languages; and although it does not prohibit us from reasoning rigorously about MONSTR (Banach 1996a, Banach 1997a, Banach 1997b), one would ideally like to have some greater insight into this slightly exotic feature. This is provided by diverse examples of its use. In the context of implementing concurrent logic languages via MONSTR (Banach and Papadopoulos 1993), this feature enables a convenient implementation of the *commit* operator. A more interesting and natural interpretation of multiple non-root overwriting arises when it is used in mapping object-oriented and Linear Logic based languages. In both cases, a stateholder plays the role of a channel. In Banach and Papadopoulos (1995b) it represents a **self** channel through which objects receive method invocations and the guaranteed atomicity of the single stateholder redirection is used to implement mutual exclusion between concurrent method invocations and thus achieve data coherence. In Banach and Papadopoulos (1995a), a work more closely related to the present one, a channel plays the role of a communication medium between linear agents. Linearity is achieved by a message posted to some agent being consumed by the latter, freeing the channel for further use, a strategy that can be expressed naturally in MONSTR by means of sequences of non-root overwrites. Since Interaction Nets are so close to Linear Logic, it is interesting to show how a concurrent Linear Logic *Append* would be implemented in MONSTR and compare it with the equivalent Interaction Nets version discussed in detail in section 6.1. The source code could be something like what follows, using the (⊗, &, ⊸, ∃, ∀, !) fragment of Linear Logic and adhering to a language model like the one developed in Tse (1994).

$$
\begin{array}{l}
!\,\forall L1, L2, O. Append(L1, L2, O) \\
\multimap \forall M. L1 : M \multimap (M = [\,] \to O : @L2 \\
\quad \&!\,\forall A, B. M = [U \mid X] \to \exists C. O : [U|Z] \otimes Append(X, L2, Z))))
\end{array}
$$

Note that '@' is a forward operator which effectively replaces one channel by another. The behaviour of the above program can quite easily be made clear by examining the equivalent MONSTR code:

$$
\begin{aligned}
&Append[l1{:}Channel\_Full[Nil]\ l2\ o] \implies *GARBAGE\_COLLECT,\\
&\quad *Forward[o\ l2],\\
&\quad l1 := *Channel\_Empty;\\
&Append[l1{:}Channel\_Full[Cons[u\ x]]\ l2\ o] \implies\\
&\quad *GARBAGE\_COLLECT,\\
&\quad *Send[o\ Cons[u\ c : Channel\_Empty]],\\
&\quad *Append[x\ l2\ c],\\
&\quad l1{:} = *Channel\_Empty;\\
&Append[l1{:}Channel\_Empty\ l2\ o] \implies \#Append[{}^{\wedge}l1\ l2\ o];
\end{aligned}
$$

where *Send* is implemented as follows:

$$
\begin{aligned}
&Send[c : Channel\_Empty\ mess] \implies *GARBAGE\_COLLECT,\\
&\quad c := *Channel\_Full[mess];\\
&Send[c\ message] \implies \#Send[{}^{\wedge}c\ message];
\end{aligned}
$$

and *Forward* is very similar. Note the serial non-root overwriting of the stateholder representing a channel by a succession of rewrites, in order to model the consumption of a resource (in this case messages posted into a channel).

Comparing the above linear *Append* with the Interaction Nets version of section 6.1, one can notice some differences. In the case of the linear *Append*, the communication medium, a channel, is public and a number of concurrently executing agents can have access to it in order to post parts of the list to be appended. Thus a stateholder representing a channel exhibits a rather 'non-monotonic' behaviour in that it switches between an empty and a full state repeatedly. In the case of the Interaction Nets *Append*, the communication medium, i.e. a principal port, is private to the agents that are involved in a rewrite. Thus, the interaction of (precisely) two agents results in a once-only overwriting of the principal ports involved by means of the *Assign* primitive, an effectively monotonic version of *Send*.

Further, note that the atomicity of the rewriting of the stateholder is of paramount importance in the linear *Append* because of its public nature. In the case of the Interaction Nets *Append* this atomicity is of lesser importance since only two agents have access to the stateholder representing a port, and in fact they cooperate in overwriting it either directly (as in the case of typed nets) or indirectly by means of the *Rew* function (in the case of untyped nets).

Interaction Nets ports can be viewed as a limited form of logic variable (allowing for instance concatenation of difference lists in constant time) and thus Interaction Nets can be seen as a deterministic subset of concurrent logic programming (Shapiro 1989). It would thus be of some interest to provide the MONSTR code for a typical concurrent logic *append* such as

$$
\begin{aligned}
&append([u \mid x],y,z) :- append(x,y,z1), z = [u \mid z1] \qquad .\\
&append([\,],y,z) :- z = y.
\end{aligned}
$$

The above program could be implemented in MONSTR as follows:

$$\begin{aligned}
&Append[Cons[u\,x]\;y\;z] \implies *Append[x\;y\;z1{:}Var],\\
&\quad *Unify[z\;Cons[u\;z1]];\\
&Append[Nil\;y\;z] \implies *Unify[z\;y];\\
&Append[x{:}Var\;y\;z] \implies \#Append[{}^\wedge x\;y\;z];\\
&Append[ANY\;ANY\;ANY] \implies *FAIL;
\end{aligned}$$

Note the use of the more elaborate mechanism for variable instantiation, done by means of invoking a *Unify* function. Note also that because the above program exhibits no linearity, the storage for the graph structures that are not referenced in the right-hand side of the rules cannot be automatically reclaimed since these structures may well be referenced by other processes. This is, of course, not the case for either the linear or the Interaction Nets *Append*.

The underlying MONSTR implementation may take advantage of these properties to generate a more efficient runtime infrastructure. For instance, graph reduction language implementations are typically packet based. A packet representing some entity (redex and/or data values) comprises a number of fields with useful information such as the number of other nodes pointing to this entity (used for garbage collection but also load distribution), the number of requests for reducing the entity (if it is a redex) which can be used to give higher priority for reduction to certain redexes over others, etc. In addition, in the case of distributed implementations, shared subexpressions may either have to be recomputed locally in each processor or otherwise bear the penalty of the traffic generated to distribute the results. All these and other issues are simplified considerably due to the nature of Interaction Nets graphs whose properties carry over to the corresponding MONSTR code. Thus, an ‘Interaction Nets MONSTR sublanguage’ would be able to take advantage of those properties for the benefit of the underlying implementation.

To further clarify some of these points, we show how the MONSTR compiler environment (Banach 1993, Glauert et al. 1990, Watson et al. 1988) could take advantage of the knowledge that the MONSTR code generated originates from Interaction Net code enjoying the usual properties, possibly coupled with knowledge derived from some static dataflow analysis. In particular, it is possible to enhance the MONSTR code with **directives** which indicate to the compiler certain optimizations that can be performed. Consider the (somewhat simplified) left-hand side of the general reduction rule for the case of typed Interaction Nets in section 4, namely:

$$\boldsymbol{F}[\boldsymbol{C}[\boldsymbol{c1}\ldots\boldsymbol{cm}]\;\boldsymbol{f1}\ldots\boldsymbol{fn}] \implies *OK,\ldots;$$

The MONSTR compiler can derive the following variation enhanced with suitable directives:

$$\begin{aligned}
&\boldsymbol{f}\langle GARBAGE\rangle{:}\boldsymbol{F}[\boldsymbol{c}\;\boldsymbol{f1}\ldots\boldsymbol{fn}],\\
&\quad \boldsymbol{c}\langle GARBAGE\rangle\langle PREEVALUATED\rangle\langle MOVETO[\boldsymbol{f}]\rangle : \boldsymbol{C}[\boldsymbol{c1}\ldots\boldsymbol{cm}]\\
&\qquad \to r{:}\langle GARBAGE\rangle\langle CLOSETO[\boldsymbol{f}]\rangle OK,\\
&\qquad \boldsymbol{f} := *r;
\end{aligned}$$

The idea here is that due to the properties of interaction it can be known that the two redex nodes *f* and *c* will disappear after the rewrite and no other agent involved in the Interaction Nets network will ever attempt to access their values in the future. This means that they can be garbage collected (hence the directive *GARBAGE*) but also that it would probably make sense to move the *c* node to the processor where *f* lies (hence the directive *MOVETO[f]*). Furthermore, since *f* will never get accessed, neither will *r:OK* to which it gets redirected. Hence *r* can be garbage collected immediately, i.e. it need never be created. Also we know that once the two Interaction Nets primary ports get engaged in interaction, they are represented by a function and a value. Thus the implementation need not reduce the already evaluated argument of the function (hence the directive *PREEVALUATED*). In this way, the packet structure and code generated by the compiler can be simpler in terms of administrative information held in packet fields, load balancing and garbage collection.

The above analysis may not be possible for MONSTR rules generated from translating other computational models. For instance, recalling one of the rules for a concurrent logic append, namely:

$$\begin{aligned} Append[Cons[u\,x]\;y\;z] \;\Longrightarrow\; & *Append[x\;y\;z1 : Var], \\ & *Unify[z\;Cons[u\;z1]]; \end{aligned}$$

we cannot be sure that the *Cons* structure in the left-hand side of the rule can be garbage collected since it may be shared by other agents, or that this argument is already a *Cons* (it could still be an uninstantiated variable).

6 MONSTR and generalized Interaction Nets

In MONSTR, nothing prevents several function nodes from sharing the same stateholder (a feature we have already referred to). Such overlapping redexes will obviously lead to non-Church–Rosser properties of rewriting in general; therefore, a naive translation of arbitrary MONSTR systems to Interaction Nets systems will be impossible. Instead, in this section we consider briefly how one might generalize the Interaction Nets model to take on board some of the additional expressiveness of MONSTR. This helps in making a sounder comparison of the two systems.

As we pointed out before, the use-once discipline for agents enables us to represent them as constructors. Moreover, the main property of constructors, that they do not change over time, is not used in the Interaction Nets model; agents are not only read-only, they are read-**once**-only. To get something more, we need to generalize the shape of the left-hand sides of rules, or the principal/auxiliary port discipline and its influence on rewriting, or the port invariant. We look at these in order.

In an abstract framework, there is no reason not to allow left-hand sides of rules to be of a more general shape than before. If one simultaneously insists that redexes must consist **only** of principal connections as before, and that these connections include **all** of the principal port edges of the agents involved, then to go beyond what we have already, we must permit agents to have more than one principal port. But we still retain the Church–Rosser property of rewriting since, provided we always have exactly one rule for each possible left-hand side, all distinct redexes are still disjoint. (In the presence of multiple principal ports, the analysis of deadlock prevention might be thought to become more problematic, but the theory of Banach (1993) shows that this is not so if one does things the right way.) We gain some expressiveness thereby, avoiding the need to break rules down into small binary interactions. Figure 12 illustrates such a rule for appending difference lists, adapted from the one presented in section 6.1, but this time requiring one rewrite rather than Lafont’s two to complete the append. We note that a compiler could automatically break such a rule down into binary interactions if required. We note that in the above model, there is still no exploitation of the key properties of constructors. A generalization that does permit such an

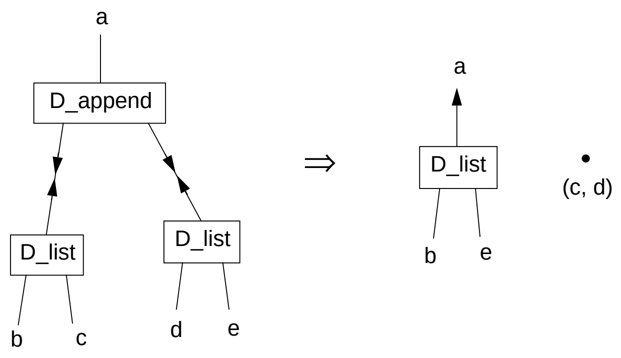

Figure 12: A rule for appending difference lists.

exploitation, instead insists that all agents **removed** from a redex should have all of their principal ports within principal connections of the redex, but now allows inspection but not removal of other agents along auxiliary connections. If these other agents themselves have **no principal ports**, they behave exactly like constructors, being read-only. Figure 13 shows such a rule, where the notation is intended to be interpreted as follows. Agents on the left-hand side having principal ports will have all their principal ports within principal connections of the redex. These agents are to be removed during the rewrite. Agents of the left-hand side having no principal ports are constructors. These are to remain behind when other left-hand side agents are removed. (They cannot, however, be shown on the right-hand side of the rule since all agents of the right-hand side are interpreted by a compiler as being specifications of new agents to be built during rewriting.) So it is intended that the constructor C and its interface nodes d, e, f, g remain when the agents X, Y, Z are removed and replaced in a rewrite using Fig. 13. This model is a nice halfway point between the original Interaction Nets model and MONSTR, incorporating some generalizations, yet still retaining the Church–Rosser property of rewriting, since apart from the immutable constructors, all pairs of distinct redexes are still disjoint. To go beyond this, we need to either relax the criterion that **all** the removed agents' principal ports figure in the principal connections of the redex, or relax the port invariant. We look at these possibilities now.

If we allow agents in the redex to have principal ports that do not connect with other agents of the redex, then we permit overlapping of redexes and non-Church–Rosser behaviour. Figure 14 shows a rule for the *Get* operation of a simple binary semaphore. Clearly the *Free* agent can be competed for along both of its principal ports, thus leading to potentially overlapping redexes, but the *Busy* agent can only be released from the port leading to the *Get* that succeeded, as one would wish.

This model approaches MONSTR's overlapping redexes but with the crucial difference that the sharing structure is constrained by the presence of the port invariant, i.e. since an agent's repertory of port nodes is fixed by its symbol, only a predetermined number of other agents may attempt to interact with it; for Fig. 14, only two agents may ever compete for the semaphore. Emulating dynamically determined sharing in such a framework is almost impossible without cumbersome encoding of lists of sharers. To overcome this final hurdle, we must weaken the port invariant to allow more than two port edges to meet at a port node. This allows arbitrary communities of agents to accumulate at a port node p, but raises a number of

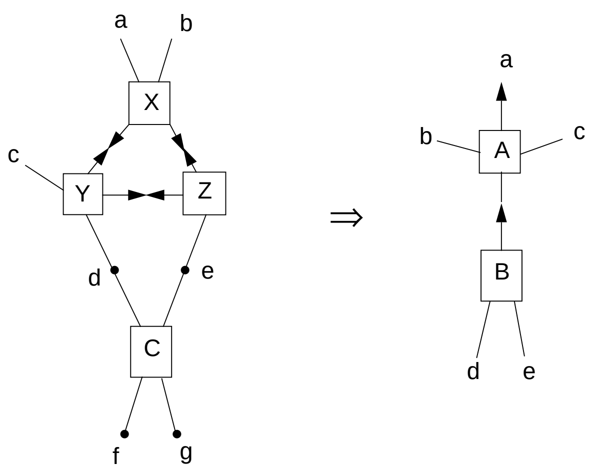

Figure 13: A generalization of the rule in Fig. 12.

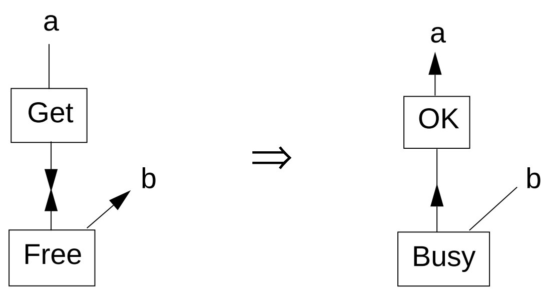

Figure 14: A rule for the *Get* operation of a simple binary semaphore.

fresh issues:

- What now is the allowed form of left-hand side for rules?
- Is the left-hand side still comprised of precisely binary port connections, or are larger collections of agents meeting at a node permitted on the left-hand side?
- Does a port node match (some part of) a left-hand side iff **all** its incident port edges are in the matching from the rule, or is one allowed to have some port edges left over? (In the latter case, presumably the port node in the middle of such a left-hand side edge cannot be regarded as garbage after the rewrite.)
- Does the map from rule left-hand side to redex have to be injective on port edges or is it now allowed to be many–one? (For binary port connections and agents having single principal ports, complementary types force injectivity of the port edge map in a redex.)

A variety of answers to these questions can be contemplated. We will not discuss all the possibilities exhaustively, but content ourselves with the following. Let us insist that left-hand sides still consist of two agents connected by a principal port connection (with perhaps some constructors as discussed above). Let us further assume that the rule system is complete in providing a rule for each possible left-hand side that one can envisage, and that surplus incident edges of a redex's principal connection's port node remain behind after the rewrite. Then minor adaptations of the translations we described above will deal with such a scenario, since agents can compete eagerly to register their willingness to interact at a given port node; the first one to arrive will be sure of finding a cooperating partner as soon as the second one has turned up. (Suitably interpreted, this works for both typed and untyped cases.)

On the other hand, if one does not assume rule completeness, and/or allows an unpredictable numbers of agents to synchronize at a port node in order to interact, then one faces a much harder implementation problem, as agents can no longer be allowed to 'grab' a port node eagerly. Similar issues arise when one wishes to take seriously the synchronization model inherent in process algebras (see Banach et al. (1995)). Thus, while in relaxing the port invariant there are useful and tempting programming models that one can envisage, there are many variations which look innocent enough at the abstract level, but are much more problematic from an implementation viewpoint. We therefore feel that MONSTR's commitment to directed arcs yields a much more reasonable framework when true dynamic sharing and choice are desired within computations.

7 Conclusions and related and further work

In this paper we have studied the relationship between two graph rewriting formalisms, namely Lafont's Interaction Nets and Banach's MONSTR. We have presented a concrete translation from Interaction Nets to MONSTR and we have discussed in detail some important issues pertaining to the reverse mapping. The two formalisms have evolved from rather different perspectives and for different reasons – Interaction Nets as a version of Linear Logic based on normalized Proof Nets and rather abstract, and MONSTR as a stripped-down version of a compiler target language with emphasis on ease of implementation on distributed machines. Thus being able to provide a concrete mapping framework from the former onto the latter yields a number of interesting possibilities.

- It provides an 'implementation apparatus' for Interaction Nets via the compiler target language MONSTR – bear in mind here that Interaction Nets exhibit a high degree of parallelism which can be fully exploited by MONSTR.

- It illustrates how Interaction Net based rule systems can be transformed into 'ordinary' graph rewriting rule based code, for execution using traditional rewriting formalisms and associated architectures (like MONSTR).

- It offers a fresh perspective on some of MONSTR's features by lifting their interpretation to the level of the computational model being mapped (as in the case of multiple non-root overwrites and the rather unusual stateholder object).

- It allows MONSTR to be used as a point of reference and comparison between different computational models by exhibiting their similarities and differences at the MONSTR level. Furthermore, bearing in mind that both formalisms enjoy formal semantics and share common targets (such as a sensitivity for ease of implementation on distributed machines), a direct comparison of the semantics of the two models would certainly be worth exploring.

This paper complements work by the authors (and others), justifying the view of generalized term graph rewriting, and MONSTR in particular, as a good 'generalized computational model' able to accommodate the needs of computational models often divergent in behaviour; needs that range from those associated primarily with reasoning and specification to those more related to implementation issues. In Banach and Papadopoulos (1993) we have shown how concurrent logic languages can be implemented in MONSTR. In Glauert et al. (1988) this is done also for eager and lazy functional languages in a more general setting which is applicable to MONSTR as well. In Banach and Papadopoulos (1995a) we used MONSTR to reason about pi-calculus and in Banach and Papadopoulos (1995b) we showed how MONSTR can be used as an implementation and specification framework for concurrent object-oriented languages. Finally, Banach and Papadopoulos (1995a) studies the possibility of using MONSTR to implement concurrent languages based on Linear Logic but also discusses the definition of a 'linear' MONSTR sublanguage.

In the spirit of the last point, one intriguing possibility that we have not pursued here, is the idea of exploring the relationship between Interaction Nets and MONSTR in the other direction; i.e. using the translation from Interaction Nets to MONSTR to inspire the definition of a sublanguage of MONSTR with some of the special properties of Interaction Nets, particularly the Church–Rosser property. The latter would yield an unusual object, a non-trivial imperative language with the Church–Rosser property, yet featuring arbitrary amounts of concurrency. (Obviously, the concurrency arising from arbitrary numbers of simultaneously active nodes is the thing that makes such a system interesting. Any purely sequential imperative language is trivially Church–Rosser.) It turns out that this is possible, but the proof is rather lengthy, and would unduly distort the balance of this more implementation oriented paper. We content ourselves with pointing out the main idea, and will present the full theory elsewhere (Banach 1997b).

In the typed translation, at any particular moment, every stateholder (say S) in the graph, has at most one parent which is a function node (say F) for which S occurs in F's stateholder position, and such that the rule for F and S does not just reactivate F's matched nodes, or merely resuspend on the stateholder S awaiting a suitable state change. The latter kind of rules do not affect the structure of the graph apart from garbage, and so each stateholder node effectively has at most one function node parent for which the corresponding rewrite 'does something useful'. We say that such a stateholder is at the apex of a **safe**

critical cone (of function nodes). The intuition is that if the executions of a system feature only safe critical cones for all the stateholder instances that arise, then the system has the Church–Rosser property, despite the presence of overlapping redexes involving mutable stateholders. The intuition turns out to be sound, but giving a convincing proof involves a lengthy excursion into the somewhat delicate technical details of MONSTR theory, as the result only holds in the most satisfactory form under transitive coercing operational semantics rather than the standard semantics we used in this paper (see Banach (1997b), Banach (1996b)).

In fact, the safe critical cone property holds also for the first untyped translation, **but not for the second**. The difference shows up at the level of proof of the Church–Rosser property. In the safe critical cone case, the proof can be accomplished by tiling the Church–Rosser diamond with subcommuting squares formed by interchanging individual steps, i.e. the usual strategy. For the system of the second translation, this does not work because the rewrites resolving the race for 'ownership' of a principal connection in different ways do not subcommute. Fortunately, these rewrites themselves create further redexes for rules whose right-hand sides are identical (cf. the right-hand sides for *Rew_Append[Cons …]* and *Rew_Cons[Append …]*). In this way the Church–Rosser property is recovered. We eschew further discussion here.

Finally, we intend to further develop the concept of 'interaction' within the MONSTR framework, the aim being to explore relationships between MONSTR and other computational approaches based on the notions of linearity and interaction (Andreoli et al. 1993, Darlington et al. 1993, Kobayashi and Yonezawa 1993, Tse 1994).

Acknowledgements

The work reported here was done while the first author was on leave at the Computer Science Department of the University of Cyprus. The hospitality of that department, and the financial support of the Royal Society, are gratefully acknowledged.

References

Andreoli, J-M., Ciancarini, P. and Pareschi, R. (1993) Interaction abstract machines, in *Research Directions in Concurrent Object Oriented Programming*, MIT Press, Cambridge, MA, pp. 257–80.

Banach, R. (1993) MONSTR: term graph rewriting for parallel machines, in Sleep M.R., Plasmeijer M.J. and van Eekelen M.C.J.D. (eds), *Term Graph Rewriting: Theory and Practice*, Wiley, New York, pp. 243–252.

Banach, R. (1995) The algebraic theory of interaction nets. Technical Report MUCS-95-7-2, Department of Computer Science, University of Manchester (http://www.cs.man.ac.uk/ csonly/cstechrep/Abstracts/ UMCS-95-7-2.html).

Banach, R. (1996a) MONSTR I – fundamental issues and the design of MONSTR. *Journal of Universal Computer Science*, **2**(4), 164–216 (http://www.iicm.edu/jucs).

Banach, R. (1996b) Transitive term graph rewriting. *Information Processing Letters*, **60**, 109–14.

Banach, R. (1997a) MONSTR II – suspending MONSTR semantics and independence. *Journal of Universal Computer Science*. http://www.icm.edu/jucs

Banach, R. (1997b) MONSTR V – transitive coercing semantics and the Church–Rosser property. *Information and Computation*, submitted.

Banach, R., Balazs, J. and Papadopoulos, G.A. (1995) A translation of the pi-calculus into MONSTR. *Journal of Universal Computer Science*, **1**(6), 339–98. (http://www.iicm.edu/jucs).

Banach, R. and Papadopoulos, G.A. (1993) Parallel term graph rewriting and concurrent logic programs. *WPDP 93*, Sofia, Bulgaria, 4–7 May, pp. 303–22.

Banach, R. and Papadopoulos, G.A. (1995a) Linear behaviour of term graph rewriting programs. *10th ACM Symposium on Applied Computing (SAC 95)*, Nashville, TN, USA, 26–28 February, ACM Press, pp. 157–63.

Banach, R. and Papadopoulos, G.A. (1995b) Term graph rewriting as a specification and implementation framework for concurrent object oriented programming languages. *International Working Conference on Programming Models for Massively Parallel Computers (MPPM 95)*, Berlin, Germany, 9–12 October, IEEE Press, pp. 151–8.

Darlington, J., Guo, Y. and Köhler, M. (1993) Functional programming languages with logical variables: a linear logic view. *5th Symposium on Programming Languages Implementation and Logic Programming (PLILP93)*, Tallinn, Estonia, 25–27 August. Lecture Notes in Computer Science **714**, Springer-Verlag, Berlin, pp. 201–19.

Ehrig, H., Kreowski, H-J. and Rozenberg, G. (eds) (1991) *Fourth International Workshop on Graph Grammars and their Applications to Computer Science*, Bremen, Germany, 5–9 March 1990. Lecture Notes in Computer Science **532**, Springer-Verlag, Berlin.

Girard, J-Y. (1987) Linear Logic. *Theoretical Computer Science*, **50**, 1–102.

Girard, J-Y., Lafont, Y. and Regnier, L. (eds), (1995) *Advances in Linear Logic*. London Mathematical Society Lecture Notes Series **222**, Cambridge University Press, Cambridge.

Glauert, J.R.W., Hammond, K., Kennaway J.R. and Papadopoulos, G. A. (1988) Using Dactl to implement declarative languages. *CONPAR 88*, Manchester, UK, 12–16 September, Cambridge University Press, Cambridge, pp. 116–24.

Glauert, J.R.W., Kennaway, J. R. and Sleep, M.R. (1990) Dactl: an experimental graph rewriting language, in Ehrig H., Kreowski H-J., Rozenberg G. (eds.), *Fourth International Workshop on Graph Grammars and their Applications to Computer Science*, Bremen, Germany, 5–9 March 1990. Lecture Notes in Computer Science **532**, Springer-Verlag, Berlin, pp. 378–95.

Kobayashi, N. and Yonezawa, A. (1993) ACL – a concurrent Linear Logic programming paradigm, International Symposium on Logic Programming (ISLP93), Vancouver, Canada, October, MIT Press, Cambridge, MA, pp. 279–94.

Lafont, Y. (1990) Interaction Nets, in *Seventeenth ACM Symposium on Principles of Programming Languages (POPL 90)*, San Francisco, CA, 17–19 January, ACM Press, pp. 95–108.

Lafont, Y. (1991) The paradigm of interaction. Working Paper. (file://lmd.univmrs.fr/pub/lafont/paradigm1.ps.Z).

Shapiro, E.Y. (1989) The family of concurrent logic programming languages. *Computing Surveys*, **21**(3), 412–510.

Sleep, M.R., Plasmeijer, M. J. and van Eekelen, M.C.J.D. (eds) (1993) *Term Graph Rewriting: Theory and Practice*. Wiley, New York.

TCS (1993) Special issue of selected papers of the International Workshop on Computing by Graph Transformation, Bordeaux, France. *Theoretical Computer Science*, **109** (1–2).

Troelstra, A.S. (1992) *Lectures on Linear Logic*. CSLI Lecture Notes 29, Chicago University Press, Chicago.

Tse, C.S.C. (1994) The design and implementation of an actor language based on Linear Logic. Thesis Report, MIT.

Watson, I., Woods, V., Watson, P., Banach, R., Greenberg, M. and Sargeant, J. (1988) Flagship: a parallel architecture for declarative programming, *15th International Symposium on Computer Architecture*, Hawaii, 30 May–2 June, ACM/IEEE Press, pp. 124–30.