LINEAR BEHAVIOUR OF TERM GRAPH REWRITING PROGRAMS

Richard Banach
CS Department, University of Manchester

George A. Papadopoulos
CS Department, University of Cyprus

Abstract

The generalised term graph rewriting computational model is exploited to implement concurrent languages based on Girard's Linear Logic (LL). In particular a fragment of LL is identified which is able to serve as a "process calculus" and on which the design of a number of languages can be based. It is then shown how this fragment can be mapped onto equivalent sets of graph rewriting rules that both preserve the functionality of the LL connectives and also exploit the properties of linearity for efficient implementation on a distributed architecture. Notions such as channels, production and consumption of messages, and N-to-N communication between agents, are interpreted in the world of (term) graph rewriting. This work serves two purposes: i) to extend the notion of Term Graph Rewriting as a generalised computational model for the case of linear concurrent languages, and ii) to act as an initial investigation towards a fully linear term graph rewriting model of computation able to be implemented efficiently on distributed architectures.

Keywords: *Programming Languages for Distributed Execution; Linear Concurrent Programming; Term Graph Rewriting.*

Introduction

Term graph rewriting systems (TGRSs) first emerged in their currently recognisable form in Barendregt *et al.* ([5]). The impetus for this work was the desire to establish a precise framework within which lower level issues appertaining to the implementation of (among others) functional and logic languages could be reasoned about mathematically. A number of groups, in particular in Nijmegen, East Anglia and London, have taken up TGRSs as a useful vehicle at the implementation level. The ability of TGRS to accommodate the, often divergent, needs of a number of language families such as concurrent logic ([13]) and functional ([10]) justifies its nature as a *general purpose* computational model, and languages based on TGRSs are suitable as *compiler target* (intermediate) languages. In particular, a number of specific TGR languages have been developed, all closely related, among which we may mention Clean ([14]), DACTL ([9]), and MONSTR ([2]). The last of these, MONSTR, can be viewed as a subset of DACTL, at least in the syntactic sense, and it is the "machine language" of the distributed architecture Flagship ([11]). MONSTR will be of considerable concern to us in this paper, being the target of our translation of a fragment of Girard's Linear Logic (LL). More recently, the relationship of TGR to both other paradigms for graph rewriting and to other issues in computer science has been explored, and the uses to which TGR has been put have expanded. The collection Sleep *et al.* ([18]) gives an accurate reflection of contemporary interest in this area.

Girard's LL ([8]) arose as a result of making logic more sensitive to the way formulae were produced and consumed during reasoning, eg. in the sequent calculus. The multiplicative fragment in particular, enforces the constraint that each formula is produced exactly once, and is consumed exactly once. This restriction has obvious computational repercussions, and forces distinct pieces of a computation to interact in only the simplest and cleanest of ways, giving scope for cheap approaches to resource management. This realisation has spread rapidly through a number of areas of computer science, and has spawned research into the "linear" subsets of many already extant models of computation. The logic and functional programming models have unsurprisingly been prime candidates for such work and in this paper we concentrate on a fragment of LL that is used in such languages ([6,12,15,16,19]).

What is perhaps more surprising, is that the implementation problems of distributed parallel TGR, crystallized in the design of the MONSTR language, threw up many questions whose answers turned out to be closely related to the criteria forced by linearity, and in a piquant historical coincidence, did so at around the same time. See for instance the work by Watson and Watson ([20]), Watson *et al.* ([21]), and Banach and Watson ([4]). The smoothness of the translation we present in this paper is concrete evidence for this claim.

It should be emphasised here that the purpose of the present work is not to describe in detail the implementation of any specific concurrent linear language in MONSTR; this is one of the additional issues addressed in the extended version of this paper. Instead, we identify a number of key programming techniques based on linearity and we show how they can be mapped onto equivalent sets of TGRS rules. In the process we (hopefully!) succeed in: i) justifying a number of important decisions that were taken in the design of Dactl and MONSTR, ii) illustrating how the concepts of linearity can be expressed in TGRS by means of the notion of a *stateholder*, and iii) providing evidence that the TGRS model can also be viewed as a generalised computational model for linear concurrent languages.

The rest of the paper is organised as follows. The next section introduces the MONSTR computational model. This is followed by a description of a fragment of LL on which a number of linear languages are based ([6,12,19]). The main part of the paper presents the mapping of this fragment onto operationally equivalent sets of MONSTR rewrite rules. For lack of space, we present the key concepts by discussing illustrative examples. A more formal translation scheme will be presented in the full version of the paper. The paper ends with some conclusions and suggestions for further work.

MONSTR

MONSTR arose as a result of the attempt to reconcile the desire for an intermediate language with rewriting-based semantics, with the reality of a parallel machine such as Flagship ([11,21]) where the primitive atomic actions were in principle

of much smaller granularity than atomic rewrites of arbitrary size. The result was the term graph rewriting language MONSTR, for which the implementation problem did not make excessive demands on the architecture's semantics.

The operational semantics of MONSTR deal with the transformation of term graphs. These are graphs in which the nodes are labelled with fixed arity symbols; each node having a number of out-arcs determined by its symbol, while itself being the target of an arbitrary number of in-arcs. The nodes and arcs of MONSTR graphs are further decorated with certain markings. If a node is marked with '`*`', then it is active and can serve as the root of a redex. If it is marked with '`#`^{n}', then it is suspended waiting for `n` notifications (see below), and then and only then `n` of its out-arcs are marked with the notification mark '`^`'. (This correspondence between `#`^{n} and `n` `^`'s is called *balancedness*, a vital invariant of MONSTR graphs.) The only other possibilities are that nodes and arcs are unmarked (i.e. idle).

Computation proceeds by arbitrarily selecting an active node `t` in the execution graph and attempting to find a rule that matches at `t`. The easiest case to describe is when there is no such rule, whereupon notification takes place: The active marking is removed from `t` and a "notification" is sent up along each `^`-marked in-arc of `t`. When this notification arrives at its (necessarily) `#`^{n} -marked source node `p`, the `^` mark is removed from the arc, and the `n` in `p`'s `#`^{n} marking is decremented, preserving balancedness (`#`^{0} is replaced by `*`, so suspended nodes eventually wake when all their subcomputations have notified).

The other case, when a rule matches at `t`, needs a little more background for its description. The symbols that label nodes are statically divided into three classes: *functions*, *constructors*, and *stateholders*. Functions label roots of LHSs of rules but are not allowed to occur at subroot positions; they may be redirected (see below). Constructors and stateholders can only occur at subroot positions of LHSs of rules; and while stateholders may be redirected, constructors may not. Here is a MONSTR rule:

```
F[ Cons[ a b ] x:Var ] => #G[ a ^*b ] , x := *SUCCEED
```

The LHS and RHS are separated by '`=>`'. On the left is the pattern with root labelled by function `F`, whose arguments are a constructor labelled `Cons` with two unspecified arguments, and a stateholder `x` labelled by `Var`. The notation makes it plain that `F`-labelled nodes have two out-arcs, and that if this rule is to match at `t`, then the subgraph immediately descending from `t` must be of the specified form.

For future reference we note that when rules are separated by '`;`' they are pattern matched in sequence (whereas a group of rules separated by '`|`' are matched in an arbitrary order). With this proviso, the first rule that is found to match is used.

Suppose the rule indeed matches at active `t`. The RHS of the rule specifies new pieces to be glued into the execution graph, namely firstly, a node labelled by `G`, created suspended and with the same arguments as the LHS `Cons` node, the second of which is activated (i.e. has its marking changed to `*` if previously idle), and such that `G` is waiting for this second argument to notify; and secondly, a node labelled by `SUCCEED`, created active. The `=>` further indicates that the LHS root `F` is to be redirected to `G`, and `x := *SUCCEED` indicates that the node matched to `x` is to be redirected to the new `SUCCEED` node. Redirection itself consists of swinging all in-arcs of the LHS node of the redirection (eg. `F`) so that they point to the RHS node of the redirection (eg. `G`). All redirections are performed simultaneously.

According to the above, no node in a MONSTR execution graph is ever destroyed. The rewriting model does not prescribe what nodes are ever to be removed from an execution graph because the rather arbitrary way in which arcs of the graph may cause it to be connected up, means that we can not statically determine which nodes are useless. Destruction of nodes is therefore done by garbage collection. Apart from specially designated system roots, active nodes are live, as are nodes accessible from a live node along an idle arc, as are also nodes which can access a live node along a notification arc. All others are garbage. This definition turns out to be sound.

In order that executions have various desirable properties, MONSTR imposes a further collection of restrictions. All functions must have a default rule (i.e. one that does no pattern matching at all in the LHS), so that every active function node can rewrite. (Correspondingly, all active constructors and stateholders can only notify.) Pattern matching by functions is only allowed to be one level deep, and each function has a statically fixed set of positions at which non-default rules must pattern match; at most (a statically fixed) one of these is permitted to match a stateholder. A further restriction is imposed that any notification arc must point to a node which is either non-idle or a stateholder (arcs are said to be *state saturated*). Finally, if an argument needed for matching by a rewrite is found to be non-idle, the rewrite suspends until the subcomputation notifies in the normal way, although this is a feature of MONSTR semantics that we will not need below.

The above constitutes a brief sketch of MONSTR and its abstract execution model; for a more detailed account see [2]. At the implementation level, term graphs may be represented by "*packets*" in memory, a packet being a data structure containing principally, a marking, a symbol, and a sequence of pointers to child packets. Thus each packet represents a node and its out-arcs. Notification arcs are in fact represented by reversed pointers (the balancedness invariant is vital for this), and there is thus some additional machinery needed to cope with the fact that a node can be the target of an arbitrary number of notification in-arcs. This representation of notification arcs is also in sympathy with the definition of liveness/garbage given above. Redirection is efficiently implemented by overwriting the LHS packet with the RHS packet; all in-arcs automatically point to the RHS packet when this is done. Garbage collection may take place at any time convenient to the implementation and is not mentioned further below.

When the packet store is distributed, as in a distributed store parallel machine, the shallowness of the pattern matching helps make rewriting tractable. The usual strategy is to move the active root to any stateholder if necessary (since there can be no more than one of them, this is possible), collect copies of needed constructors (since these are not redirectable, they will not change subsequently in the packet store), and then complete the rewrite. The suspension mechanism mentioned above conceals most of the lower level machinations from higher levels of abstraction. The details of these tricks are thoroughly discussed in [2].

From the perspective of higher levels of abstraction, "linear" models of computation correspond closely to the principal features of MONSTR, neatly eliminating some awkward special cases that MONSTR otherwise has to take into account. The single stateholder argument of a MONSTR function in a rewrite

maps nicely to the interaction of exactly two entities in a single computational step of a linear model; the fact that both function and stateholder can be redirected in a single computational step reflects the symmetric role played by function and data in linear models. The mapping of a fragment of multiplicative LL to MONSTR illustrated in the fourth section is a good example of this. The whole phenomenon is not surprising since the forces underlying the designs of MONSTR and of linear models of computation are fairly similar at the implementation level.

A fragment of Linear Logic as a "Process Calculus"

In order to express one of the most important concepts of LL, that of accounting, most linear languages ([6,12,19]) are based around the notion of *channels* and *messages*. In particular, the traditional notion of a variable shared among a number of processes and instantiated to some value is replaced by the notion of a channel shared among a number of agents which are able to *post* messages to that channel as well as *consume* them. Concurrency is then achieved by allowing the asynchronous or synchronous communication between a number of agents by means of posting or receiving messages through shared channels. The way a particular message that was posted to a channel is selected by some agent is specified using *methods*.

A method can consume messages (which are possibly required to match specific patterns), and can suspend (if necessary) until the required messages have been posted there, thus achieving the required synchronisation between concurrently executing agents. Upon commencing execution, methods can create further messages and channels. Depending on the characteristics of each particular linear language, an agent may be able to post to a channel, in addition to messages, methods as well as other channels.

The linear behaviour of such a framework stems from the fact that by "reading" a message from a channel, an agent causes its deletion. On the other hand, if the same message is posted to a channel more than once by one or more agents, it will appear there precisely that many times. Furthermore, depending again on a language's characteristics, other features are also offered such as specifying whether a particular agent should execute once or multiple times, examining the messages posted to a channel without removing them, modelling higher-order linear concurrent programming, etc.

A number of linear concurrent languages have been designed around the above notions, where the underlying logic supported is essentially the ($\otimes$, &, $\multimap$, $\exists$, $\forall$) fragment of LL. In particular, the basic syntax of such a language is as follows (where `A`, `X`, `T`, `C`, and `M` denote agents, channels, terms, constraints and messages respectively and terms and messages comprise an appropriate combination of data terms and channels):

Idle agent (Unit, Termination) **`1`**

This agent does nothing. It denotes termination of the program.

Create new channel (Hiding) **`∃X.A`**

Create a new channel `X` local to agent `A` ('∃' is the existential quantifier).

Send a message **`X:M`**

Send message `M` to channel `X`, where ':' is the channel operator.

Linear implication (method) **`[!] ∀Y.H ⊸ A`**

where `H` ::== Idle agent | Procedure Call | Linear Composition | `X:M`

For the latter case (where `H` is `X:M`) the agent receives message `M` from channel `X` and then behaves like `A` where free occurrences of `Y` are substituted by `M` ('∀' is the local universal quantifier). Note that unless the reusability operator '!' is used, the agent will execute just once. Upon receiving `M`, the message is deleted from the channel `X`.

Blocking ask (matching method) **`∀X.C -> A`**

Wait until (if ever) the constraint `C` is satisfied and then execute agent `A`. For the purposes of this work we only consider primitive constraints.

Parallel (Linear) Composition **`A⊗A`**

Both agents execute concurrently.

Procedure Call **`P(T1,…,Tn)`**

where `P` is a procedure name and `T1` to `Tn` are terms.

Choice **`R1&R2`**

Each one of `R1`, `R2` is respectively of the form `p ⊸ A`, `q ⊸ B`, so (`p ⊸ A & q ⊸ B`) reduces to either `A` or `B` depending on whether it is `p` or `q` respectively that it is supplied; `p` and `q` are themselves of the form `X:M` or `p⊗p`.

Thus by applying the LL principle of "formula as resource and proof as computation" one can view the linear operators as the building blocks for constructing sequents, i.e. configurations of concurrently executing agents. In the following section we present a number of examples which are used to illustrate the use of the particular fragment of LL we are considering in this paper but also to describe the mapping of the LL connectives to equivalent sets of MONSTR rewrite rules.

A Term Graph Rewriting Modelling of Linear Logic Agents

Here we discuss how the functionality of the LL fragment described in the previous section can be modelled in terms of operations on term graphs, and more generally how a run-time configuration for a number of LL agents executing concurrently and communicating by means of exchanging messages can be represented as a term graph comprising a number of active and suspended graph nodes synchronised by access to a number of stateholders. In the next subsection we present such a MONSTR run-time configuration and we show the strong relationship between stateholders and channels as well as the power of the former. We then present by means of suitable examples the effectiveness of our model in implementing a variety of programming paradigms.

A MONSTR run-time configuration for channels

One of the most important aspects of implementing a LL language is how to represent the notions of channels and sending and receiving messages. Activity at run-time evolves around channels: messages posted to a channel remain in a suspension queue until claimed by some method posted to that channel. Also, methods posted to a channel consume messages posted to the channel, suspending if necessary until the

required message has appeared there. The methods are then deleted from the queue of methods associated with that particular channel except if the reusability operator has been used. Note that in a distributed implementation of such a language, the load distribution of a program uses the channels as points of reference, and methods and messages associated with a channel are moved to the processor where the channel in question is located ([15,19]).

We recall from the second section that this is exactly how a distributed implementation of MONSTR would behave; in particular when a rule that is candidate for reduction involves the use of constructors and a stateholder, both the function and the constructors are moved (respectively copied) to the processor where the stateholder is held. We highlight here some features of MONSTR which have proved invaluable in the design of a LL language to MONSTR mapping strategy:

- Atomic overwriting of a stateholder, allowing a number of concurrently executing processes to compete for control of it.
- Repeated updating of a stateholder (modelled by a sequence of overwrites of the representing packet), allowing the modelling of side-effects such as destructive assignment.
- Ability to organise workload locality around the stateholders involved in the computation.
- Arbitrary format of a stateholder node in terms of number of descendant arcs and type of arguments which, as shown later on, allows among others the modelling of higher-order LL features.

We now describe a MONSTR configuration which is able to model the run-time behaviour as described above. A channel is, obviously, a stateholder. Posting of messages to this channel is modelled by means of the following MONSTR rule system (see below for the significance of the `GARBAGE_COLLECT` constructor):

```
Send[c:Channel_Empty message] => *GARBAGE_COLLECT,
                                 c:=*Channel_Full[message];
Send[c message] => #Send[^c message];
```

The first rule matches an empty stateholder and posts its message there while the second one suspends if the stateholder already contains a value. Any number of `Send` processes, all of which share the stateholder `Channel_Empty`, can attempt to reduce in parallel; however, only one of them will be able to post its message. The rest of them will suspend waiting for some other process to "consume" the message and free the stateholder again. Which one of the `Send` processes will succeed in "posting" its message is of course non deterministic, thus modelling correctly the asynchronous posting of messages to a channel. Note that the `Send` processes suspended on some stateholder (channel) effectively form the message suspension queue associated with that stateholder. The consumption of the message held in the stateholder by a corresponding "receive" process causes the waking of the `Send` processes suspended on it which will compete again for the next posting. Actual adherence to the aforementioned operational semantics is guaranteed by MONSTR's ability to update stateholders atomically.

The consumption of messages posted to a channel can be modelled by means of "symmetric" MONSTR rule systems similar to the following one:

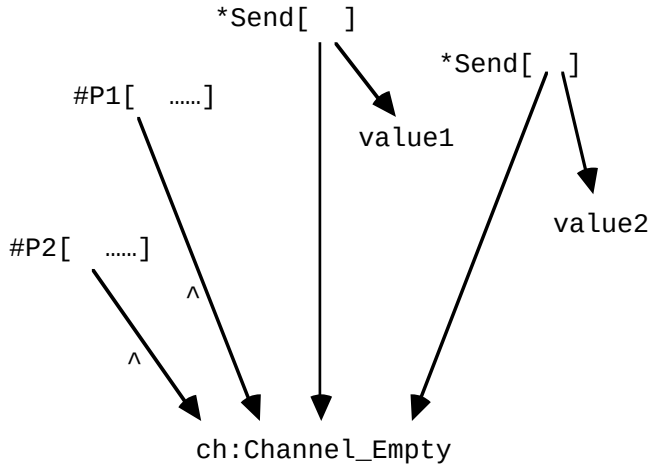
```
Receive[c:Channel_Full[message]] => *GARBAGE_COLLECT,
                                    *Consume[message],
                                    c:=*Channel_Empty;
Receive[c:Channel_Empty ] => #Receive[^c];
```

where we assume the existence of some process `Consume` that "uses up" `message`. Again here, the first rule is responsible for getting the message while the second provides the required synchronisation by suspending if necessary.

In practice the actual MONSTR rule sets produced do not make use of a general `Receive` process, only of a `Send` one. The reason is that the functionality of `Receive` is coded up in the rule sets of the methods associated with a stateholder, which themselves are represented as MONSTR functions.

Again a process representing a method suspends if it attempts to access the value of an empty stateholder, thus achieving the required synchronisation. A number of such processes can be suspended on the same stateholder, effectively modelling a method suspension queue.

The run-time behaviour of a MONSTR graph configuration can be seen diagrammatically as follows:



where `P1` and `P2` represent methods suspended on receiving a suitable message on the stateholder `ch` by the concurrently executing `Send` processes.

Note that all the MONSTR functions produced have no parent nodes in the RHS of any rule that creates them, (in other words the values that they themselves generate are consumed by no one). In fact, the only graph sharing produced is that of stateholders. Thus we could have written alternatively

```
Receive[c:Channel_Full[tuple]] -> *Apply[tuple],
                                  c:=*Channel_Empty;
Receive[c:Channel_Empty ] -> #Receive[^c];
```

using Dactl's '`->`' operator to avoid a superfluous root redirection ([9]). But this is not permitted in MONSTR so we rewrite all functions to the dummy constructor `GARBAGE_COLLECT`, its name indicating what ought to happen to it. In fact to establish that no function node has a parent in the run-time configuration requires a global analysis of the whole ruleset, although an extremely simple one ([1]).

The above run-time configuration can be used to model most of the types of communication using channels: one-to-one (point-to-point), one-to-many (broadcasting) and many-to-one are implemented trivially; one-to-one-of-many (Linda

type) and many-to-many can also be implemented provided that the methods that will consume the posted messages have themselves already been posted to the appropriate channel.

Example programs

In this subsection we show concrete translations of a number of representative LL programs, illustrating how various programming idioms of LL are transformed into equivalent MONSTR rule sets. Note that in practice, before the actual translation is performed, a LL program is translated into a "kernel" form suitable for direct mapping onto MONSTR rewrite rules; for lack of space we do not present these intermediate transformations here and instead we refer the interested reader to the full version of the paper. Nevertheless, the main stages in these transformations are: i) the use of only one level of agent definition and the pulling of all the others to the top level (possibly by means of introducing additional auxilliary agents), ii) the use of multi-clause style instead of the block structured one because the former can be translated directly to an equivalent rewrite rule system, and iii) the deletion of any matching operators and incorporation of the associated functionality into the generated rewrite rule system. We start with the unavoidable `append`:

```
! ∀L1,L2,O.Append(L1,L2,O)
   ⊸ (! ∀M.L1:M ⊸ ( M=[] -> O:@L2
     & ! ∀A,B.M=[A|B] -> ( ∃C.O:[A|C] ⊗ Append(B,L2,C) )
```

The above program uses channel continuations to increase performance. Note that '@' is the "forward messages" operator ([19]) which effectively replaces one channel by another. Its translation to MONSTR is as follows:

```
Append[l1:Channel_Full[Nil] l2 o] => *GARBAGE_COLLECT,
                                     *Forward[o l2],
                                     l1:=*Channel_Empty|
Append[l1:Channel_Full[Cons[a b]] l2 o]
   => *GARBAGE_COLLECT,
      *Send[o Cons[a c:Channel_Empty]], *Append[b l2 c],
      l1:=*Channel_Empty|
Append[l1:Channel_Empty l2 o] => #Append[^l1 l2 o];
```

A few points should be made about the above program. First of all, in order to keep the code size presented at a manageable level and avoid unneccesary details we have deliberately ignored the one level pattern matching restriction of MONSTR. In general a rule of the form

```
P[c:Channel_Full[Pattern] …] => *GARBAGE_COLLECT, …,
                                c:=*Channel_Empty;
```

is actually a combination of

```
P[c:Channel_Full[Pattern] …]
   => *GARBAGE_COLLECT, *P_aux[c Pattern];
P_aux[c …] => *GARBAGE_COLLECT, c:=*Channel_Empty;
```

where `P_aux` performs the multiple level pattern matching. Bearing this in mind, we note that a LL agent definition is compiled into n+1 MONSTR rules where n is the number of clauses defining the agent and there is one more rule implementing the required synchronisation on an empty channel. Note also that each rule matching the stateholder includes a non-root overwrite in its RHS which effectively implements the consumption of the message. Finally, note that if the reusability operator is used then this is implemented in MONSTR by means of recursion.

The above MONSTR program makes use of the function `Forward[y x]` which implements the '@' operator by forwarding messages from channel `x` to channel `y`. This function is defined in MONSTR as follows:

```
Forward[y x:Channel_Full[value]] => *GARBAGE_COLLECT,
                                    *Send[y value],
                                    x:=*Channel_Empty;
Forward[y x:Channel_Empty] => #Forward[y ^x];
```

A possible call to the above `append` program is the following

```
L1:[1|L2:[2 L3:[]]], L4:[3|L5:[4 L6:[]]], Append(L1,L4,O)
```

which in MONSTR is translated as follows:

```
INITIAL => o:Channel_Empty,
           l1:Channel_Empty, l2:Channel_Empty,
           l3:Channel_Empty, l4:Channel_Empty,
           l5:Channel_Empty, l6:Channel_Empty,
           *Send[l1 Cons[1 l2]], *Send[l2 Cons[2 l3]],
           *Send[l3 Nil], *Send[l4 Cons[3 l5]],
           *Send[l5 Cons[4 l6]], *Send[l6 Nil],
           *Append[l1 l4 o];
```

where `INITIAL` denotes a redex that by convention starts off the computation.

MONSTR's insistence that stateholders must be updated atomically allows the modelling of Linda-like generative type of communication ([7]). For instance, the two agents

```
∀L.Receive1(L)
   ⊸ ( ∀M.L:M ⊸ ( ∀C.M=m(a,C) -> Consume1(C)))
∀L.Receive2(L)
   ⊸ ( ∀M.L:M ⊸ ( ∀C,D.M=m(C,D) -> Consume2(C,D)))
```

translated to MONSTR as follows

```
Receive1[l:Channel_Full[M[A c]]] => *GARBAGE_COLLECT,
                                    Consume1[c];
Receive1[l:Channel_Empty] => #Receive1[^l];

Receive2[l:Channel_Full[M[c d]]] => *GARBAGE_COLLECT,
                                    Consume2[c d];
Receive2[l:Channel_Empty] => #Receive2[^l];
```

can compete for consuming the single message `m(a,X)`. The usefulness of such a technique in modelling open and distributed systems is discussed in ([12]).

Finally we show how higher order LL programming techniques can be modelled using stateholders. Consider the following LL agent which connects a number of processes `Proc` in a linear topology by means of `Left` and `Right` communication channels (where '_' is the "black hole" operator which simply consumes any message it receives).

```
! ∀Proc,Left,Right.Map(Proc,Left,Right)
⊸ ( ∀M.Proc:M ⊸ ( M=Nil -> _
                & ∀Pr.M=m(Pr,Nil) -> Pr(Left,Right)
                & ∀Pr,Prs.M=m(Pr,Prs)
                          -> ( ∃Link.Pr(Left,Link) ⊗
                               Map(Prs,Link,Right) ) ) )
```

Note that `Proc` is of the form `m(Pr,Prs)` where `Pr` is a function (process) name and `Prs` is a channel continuation or `Nil`. For lack of space we do not show the whole translation

(which adheres to the principles outlined so far) but instead we concentrate on the modelling of, say, the third case where a new process `Pr` is created and linked to the rest of the processes.

```
Map[proc:Channel_Full[M[pr:REWRITABLE prs]] left right]
   => *Map[prs link right], *Apply_To[pr left link],
      link:Channel_Empty;
```

The above rewrite rule exploits the extensive pattern matching facilities that are supported by Dactl (and hence by MONSTR too) and the so called screwdriver techniques ([9]) which allow the efficient manipulation of the graph itself. In particular, the left hand side matches the graph only if the first part of the message in the channel is a `REWRITABLE` node, i.e. a function name. Then in the right hand side a new function application is generated by means of the screwdriver function `Apply_To` which is defined at the MONSTR level as follows:

```
Apply_To[f:Function_handle arg1 … argn] => *F[arg1 … argn];
```

where `F` is the function name of the respective handle `f`.

Performance Issues and Discussion

The following essentially short performance analysis illustrates the behaviour of a TGRS implementing a linear concurrent program compared with the TGRS generated for the equivalent version of the program written in a non linear concurrent language. In particular, we compare the linear version of `Append` presented in the previous section with the equivalent version written in a state-of-the-art concurrent logic language ([17]) which was translated to MONSTR using the techniques described in [3,13]. The programs were run using the Dactl interpreter ([9]) and exploiting the statistics facilities offered by that implementation. Note that what we compare here are the relative differences in the statistics of the two programs rather than the absolute performance figures.

Perf Params:	R	PC	AvP	MxP	GrN
CL Append	148	75	1.97	4	147 (24 INDs)
Lin Append	76	27	2.81	3	100 (51 INDs)

R: Rewrites; **PC:** Parallel cycles performed; **AvP:** Activations processed per cycle (mean value); **MxP:** Activations processed per cycle (peak value); **GrN:** Graph nodes created

The linear version (`Lin Append`) is in all respects more efficient than the corresponding non linear one. The linear version does not need to reflect on failure (the capture of failing derivations must be coded up by the programmer himself if he so wishes) like the corresponding concurrent logic version (`CL Append`) and hence we can dispense with the associated set of MONSTR rewrite rules that would otherwise be needed ([3]) and the overhead they incur. However, the linear version is also more efficient in memory consumption. The concurrent logic version generates 147 nodes, 24 of which are `IND` ones needed for sharing rewritten packets; this is a standard technique used in implementing languages using graph rewriting ([5,14,19,20,21]). These `IND` nodes cannot be dispensed with since in a concurrent logic program a data structure may indeed be shared by more than one process. The linear version, on the other hand, generates 100 nodes and 51 of them are `IND` nodes which can be dispensed with because it is guaranteed by the linear notion of accounting for resources that only one agent will use a data structure. In the full version of the paper we further explore the effects of linearity in a TGRS program.

Composite data structures and typing

Many of the LL languages proposed support composite data structures and typing ([6,12,19]). Here we can only touch upon this issue and we refer the reader to the full paper for further details. The representation of composite data structures such as arrays and the implementation of associated operations on them can be efficiently supported by means of the `Vectors` module that is supported by the Dactl implementation and allows a variety of accessing methods (lazy, eager, parallel, etc.).

Regarding typing it is very easy in MONSTR (as, indeed, it is in Dactl as well) to impose type restrictions on stateholders. For instance, the following typed version of `Send` refrains from sending a message that is not either an integer or real.

```
Send[c:Channel_Empty message:(INT+REAL)]
   => *GARBAGE_COLLECT, c:=*Channel_Full[message];
Send[c:Channel_Full[Any] message] => #Send[^c message];
Send[Any Any] => *GARBAGE_COLLECT;
```

Note the use of the union operator '+'; in particular a rule with a pattern such as `n:(A+B)` is a candidate for matching if the graph node matches either `A` or `B`. Pattern matching operators are further discussed in [9].

Conclusions

We have investigated the possibility of using term graph rewriting, as expressed in the distributed model MONSTR, to implement linear concurrent languages. In particular, we identified a fragment of LL on which the modelling of a number of linear concurrent languages is based and we showed how it can be implemented in terms of equivalent sets of term graphs. In the process we identified a number of important issues where both LL and MONSTR have adopted similar solutions, for instance in the case of stateholdes (channels).

The MONSTR term graphs produced, exhibit a linear behaviour focused mainly on the "interaction" of a stateholder representing a channel with the root packet representing the consumer method. They also enjoy certain properties such as the fact that graph sharing is constrained only on stateholders. Coupled with a dataflow analysis ([1]), this can assist the underlying implementation significantly, for instance in garbage collection. We view the work presented here as a step towards designing a linear term graph rewriting model.

It is significant to note that many of the techniques discussed in this work have been around in the graph rewriting community for some time ([3,4,11]). In fact, the code produced by our LL to MONSTR methodology is similar to that for other concurrent logic languages ([3]). There, however, the insistence on single-assignment stateholders stemmed from the requirement to keep in line with the non-destructive nature of single-assignment logic variables. The notion of a channel proposed in many LL languages frees the implementor to use the semantics of stateholders more fully.

Indeed, regarding implementation, similar compilers that have been written for mapping a variety of concurrent logic languages onto Dactl and MONSTR ([13,3]) are currently being modified to suit our purposes. The derived code would be able to run on the distributed Flagship machine ([11,20,21]).

A final point to note is that intermediate compiler target languages such as MONSTR (and Dactl for that matter) can be

used as a basis for comparing the various ways certain important aspects of linearity are handled by different concurrent linear languages such as synchronisation mechanisms. The fact that computational models like MONSTR are very careful to support only those features which can be efficiently implemented by a parallel machine makes this research work even more important.

Acknowledgments

We thank the referees for many constructive comments and criticisms which helped in the preparation of the final version of the paper.

Contact Addresses for Authors

Richard Banach, Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK. E-Mail: banach@cs.man.ac.uk

George A. Papadopoulos, Department of Computer Science, University of Cyprus, 75 Kallipoleos Str., Nicosia, P.O.B. 537, CY-1678, Cyprus. E-Mail: george@turing.cs.ucy.ac.cy.

References

[1] R. Banach, Dataflow Analysis of Term Graph Rewriting Systems, *PARLE'89*, Eindhoven, The Netherlands, June 12-16, LNCS 366, Springer Verlag, pp. 55-72.

[2] R. Banach, MONSTR I — Fundamental Issues and the Design of MONSTR, submitted to *New Generation Computing*, 1993.

[3] R. Banach and G. A. Papadopoulos, Parallel Term Graph Rewriting and Concurrent Logic Programs, *WP&DP'93*, Sofia, Bulgaria, 4-7 May, Bulgarian Academy of Sciences, pp. 303-322, North Holland (to appear).

[4] R. Banach and P. Watson, Dealing with State in Flagship: The MONSTR Computational Model, *CONPAR'88*, Manchester, UK, Sept. 12-16, Cambridge University Press, pp. 595-604.

[5] H. P. Barendregt, M. C. J. D. Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer and M. R. Sleep, Term Graph Rewriting, *PARLE'87*, Eindhoven, The Netherlands, June 15-19, LNCS 259, Springer Verlag, pp. 141-158.

[6] J. Darlington, Y. Guo and M. Köhler, Functional Programming Languages with Logical Variables: A Linear Logic View, Internal Report, DOC, Imperial College, 1993.

[7] D. Gelernter, Generative Communication in Linda, *ACM TOPLAS*, Vol. 7(1), pp. 80-112, 1985.

[8] J-Y. Girard, Linear Logic, *Theoretical Computer Science* Vol. 50, pp. 1-102, 1987.

[9] J. R. W. Glauert, J. R. Kennaway, M. R. Sleep and G. W. Somner, *Final Specification of Dactl*, Internal Report SYS-C88-11, University of East Anglia, Norwich, UK, 1988.

[10] K. Hammond, *Parallel SML: A Functional Language and its Implementation in Dactl*, Ph.D. Thesis, School of Information Systems, University of East Anglia, Norwich, UK, published by Pitman Publishers, 1990.

[11] J. A. Keane, An Overview of the Flagship System, *Journal of Functional Programming*, Vol. 4 (1), pp. 19-45, January 1994.

[12] N. Kobayashi and A. Yonezawa, ACL - A Concurrent Linear Logic Programming Paradigm, *ISLP'93*, Vancouver, Canada, Oct., MIT Press, pp. 279-294.

[13] G. A. Papadopoulos, *Parallel Implementation of Concurrent Logic Languages Using Graph Rewriting Techniques*, Ph.D. Thesis, School of Information Systems, University of East Anglia, Norwich, UK, 1989.

[14] M. J. Plasmeijer and M. C. J. D. van Eekelen, *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley, New York, 1993.

[15] V. Saraswat, A Brief Introduction to Linear Concurrent Constraint Programming, Technical Report, Xerox PARC, April, 1993.

[16] V. Saraswat and P. Lincoln, Higher Order, Linear, Concurrent Constraint Programming, Technical Report, Xerox PARC, July, 1992.

[17] E. Y. Shapiro, The Family of Concurrent Logic Programming Languages, *Computing Surveys*, Vol. 21 (3), 1989, pp. 412-510.

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

[19] C. S. C. Tse, The Design and Implementation of an Actor Language Based on Linear Logic, Thesis Report, MIT, 1994.

[20] P. Watson and I. Watson, Evaluating Functional Programs on the Flagship Machine. *FPLCA'87*, Portland, Oregon, USA, Sept. 14-16, LNCS 274, Springer Verlag, pp. 80-97.

[21] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg and J. Sargeant, Flagship: A Parallel Architecture for Declarative Programming. *15th Annual ISCA*, Hawaii, May 30 - June 2, 1988, pp. 124-130.