Parallel Graph Rewriting on Loosely Coupled Machine Architectures

M.C.J.D. van Eekelen, M.J. Plasmeijer, J.E.W. Smetsers
University of Nijmegen*

Abstract

Graph rewriting models are very suited to serve as the basic computational model for functional languages and their implementation. Graphs are used to share computations which is needed to make efficient implementations of functional languages on sequential hardware possible. When graphs are rewritten (reduced) on parallel loosely coupled machine architectures, subgraphs have to be copied from one processor to another such that sharing is lost. In this paper we introduce the notion of lazy copying. With lazy copying it is possible to duplicate a graph without duplicating work. Lazy copying can be combined with simple annotations which control the order of reduction. In principle, only interleaved execution of the individual reduction steps is possible. However, a condition is deduced under which parallel execution is allowed. When only certain combinations of lazy copying and annotations are used it is guarantied that this so-called non-interference condition is fulfilled. Abbreviations for these combinations are introduced. Now complex process behaviours, such as process communication on a loosely coupled parallel machine architecture, can be modelled. This also includes a special case: modelling multiprocessing on a single processor. Arbitrary process topologies can be created. Synchronous and asynchronous process communication can be modelled. The implementation of the language Concurrent Clean, which is based on the proposed graph rewriting model, has shown that complicated parallel algorithms which can go far beyond divide-and-conquer like applications can be expressed.

1. Introduction

Ideally, a computational model of a language is a formal model as close as possible to both its semantics and its implementation, still it models only the essential aspects of them. In the following paragraphs it is explained why Graph Rewriting Systems (GRS's) are suited to serve as a computational model of functional languages and their implementations. After that, GRS's are extended in order to deal with parallel evaluation.

1.1. Graph rewriting systems and functional languages

Our prime interests are functional languages and their implementation on sequential and parallel hardware. Traditionally, the pure lambda calculus (Church (1932),Church

---

[*] This research is partially supported by the ESPRIT basic research action 3074: Semagraph (Semantics and Pragmatics of Generalised Graph Rewriting)

(1933),Barendregt (1984)) is considered to be a suitable model for these languages. However, in our opinion, some important aspects of functional languages and the way they are usually implemented, cannot be modelled with this calculus. In particular, the calculus itself lacks pattern matching and the notion of sharing of computations. Patterns contain important information for strictness analyzers (Nöcker (1990)). Sharing of computations is essential to obtain efficient implementations on traditional hardware (Fasel and Keller (1986)).

Graph Rewriting systems are based on pattern matching and sharing. We believe that compared to the λ-calculus, graph rewriting systems (Barendregt et al. (1987a),Barendregt et al. (1987b)) are better suited to serve as computational model for functional languages. In the past we have defined and implemented the intermediate language Clean (Brus et al. (1987)) based on graph rewriting systems with a functional evaluation strategy and we have shown that efficient state-of-the-art implementations on sequential hardware can be obtained by compiling functional languages to Clean (Koopman and Nöcker (1988)).

1.2. Parallel evaluation

At any stage during its evaluation a functional program may contain more than one function application that can be rewritten (reducible expression or redex for short). If in this context redexes are rewritten in any order, the normal form (if it exists) will always be the same. This uniqueness of normal forms offers the theoretical possibility to reduce redexes in parallel. Hence, functional languages are often considered to be well suited for parallel computation.

Two kinds of parallelism are distinguished: fine grain and coarse grain. In principle, with fine grain parallelism any redex (grain) is a candidate for parallel evaluation. Fine grain machine architectures try to exploit this parallelism fully. Unfortunately, these architectures, such as data flow machines (Gurd et al. (1985),Arvind et al. (1987)), are very complex and not yet commercially available. For coarse grain parallelism loosely coupled machine architectures, such as Transputer systems, are available on a wide scale. But now one of the major problems is that most reductions of function applications will not contain a sufficient amount of computation compared with the overhead costs caused by the inter-processor communication (grain size problem). Therefore, for these architectures only redexes that yield a large amount of computation are suited to be evaluated in parallel.

In analogy with the concurrent imperative languages, a parallel functional language should provide a way to create concurrent entities (processes) in a program, preferably without violating the functional semantics. Arbitrary communications between processes have to be definable in a general way. Special language constructs have been proposed to make process creation and communication possible (see section 6). Mostly, these constructs are either rather *ad hoc* or have limited expressive power. We are looking for powerful but elegant basic components needed to realize dynamic process creation with arbitrary communication.

1.3. Parallel graph rewriting

In order to deal with parallelism, the graph rewriting model is extended with two issues: a way to control the evaluation order and a way to regulate the distribution of data.

By denoting subgraphs on which reduction processes have to be created, the reduction order in graph rewriting can influenced. Reduction processes which evaluate an indicated subgraph are created dynamically.

A subclass of GRS's in which reducers can be created explicitly will be prefixed with P. So the abbreviation for general graph rewriting systems with explicit parallelism is *P-GRS*.

The distribution of data, which in the context of graph rewriting involves copying of graphs, can be regulated by means of a so called *lazy copying* mechanism. Intentionally, sharing is used to prevent that the same computation is performed more than once. With lazy copying it is possible to make a copy without loosing this advantage. Although in implementations generally some kind of copying/sharing scheme is used, up to now it has never been incorporated in graph rewriting models. For all these reasons we have given a more firm basis to lazy copying by explicitly incorporating it in graph rewriting systems. As we shall see, with lazy copying one can model all the major aspects of data distribution on parallel machine architectures. It is possible to specify whether synchronous or asynchronous inter-processor links are used and also the kind of and the moment at which data is communicated. To handle all these aspects it is unavoidable that the lazy copying mechanism has become very complex. This in contrast with the rather obvious way of creating parallel reducers.

A subclass of GRS's which is extended with lazy copying will be prefixed with C. So the abbreviation for general graph rewriting systems with lazy copying is *C-GRS*.

In this paper it will be shown that arbitrary process structures with various forms of inter-process communication can be modelled in *PC-GRS's* (GRS's with explicit parallelism and lazy copying). In particular, loosely coupled parallel evaluation is defined wherein any process structure can be expressed. In order to illustrate the expressive power examples will be given of some non-trivial parallel algorithms.

1.4. Structure of this paper

This paper has not the intention to give a full formal description of a parallel graph rewriting model. As can be seen in Barendsen and Smetsers (1992) such a description becomes very complex. We believe that going into too many technical details at this stage will not help the reader to understand the fundamental issues of what we want to address.

The next section introduces graph rewriting briefly. After that in section 3 process creation is incorporated in graph rewriting. In section 4 lazy copying is introduced. The power of the combination of lazy copying and process creation is shown in section 5. In particular, the use of the system to model parallel graph reduction on loosely coupled parallel architectures is demonstrated. In section 6 comparisons with related work, implementation aspects and directions for future research are given.

2. Graph Rewriting

In graph rewriting systems (Barendregt et al. (1987b)) a program is represented by a set of rewrite rules. Each *rewrite rule* consists of a left-hand-side graph (the *pattern*), an optional right-hand-side graph (the *contractum*) and one or more redirections. A *graph* is a set of nodes. Each node has a defining node-identifier (the *nodeid*). A *node* consists of a symbol and a (possibly empty) sequence of applied nodeids (the *arguments* of the symbol). Applied nodeid's can be seen as references (arcs) to nodes in the graph, as such they have a *direction*: from the node in which the nodeid is applied to the node of which the nodeid is the defining identifier. Starting with an initial graph the graph is rewritten according to the rules. When the pattern matches a subgraph, a *rewrite* can take place which consists of building the contractum and doing the redirections. A *redirection* of one nodeid to another nodeid means that all applied occurrences of one nodeid are replaced by occurrences of the other. The part of the graph that matches the pattern is sometimes called a *redex*.

A *reduction strategy* is a function that indicates one or more of the available redexes. A *reducer* is a process that reduces redexes which are indicated by the strategy. The result of a reducer is reached as soon as the reduction strategy does not indicate redexes anymore. A reducer chooses either deterministically or non-deterministically one of the redexes that are indicated by the strategy. In this paper only deterministic reducers (i.e. reducers which make their choices deterministically) are used. A graph is in *normal form* if none of the patterns in the rules match any part of the graph. A graph is said to be in *root normal form* when the root of a graph is not the root of a redex and can never become the root of a redex. The root normal form property is in general undecidable (Plasmeijer and van Eekelen (1993)). Even if a graph has only one unique normal form, this graph may be reduced to several root normal forms depending on how far the subgraphs are reduced.

An important subclass of graph rewriting systems is the class which is defined by the following restrictions:

- all graphs are connected;
- every rule has exactly one redirection which is a redirection from the root of the pattern to the root of the contractum (or when there is no contractum, to the root of a subgraph indicated in the pattern);
- no rule is comparing (rewriting systems where multiple occurrences of variables on left-hand-sides are allowed are called *comparing* or *non left-linear*). No multiple occurrence of variables implies that it is impossible to pattern match on equivalency of nodeids (sharing). In fact, a left-hand-side is always a graph without sharing (like a term).
- a special reduction strategy is used: the *functional* reduction strategy. Reducing graphs according to this strategy resembles very much the way execution proceeds in lazy functional languages (a formal definition of this strategy for TRS's can be found in Toyama et al. (1993).

This class will be called: *Functional Graph Rewriting Systems* (*FGRS's*). In an FGRS every rewrite implies that the root of the redex is redirected to another graph. Every node that after the rewrite is not connected to the root of the graph, is considered to be non-existent (*garbage*).

FGRS's serve as basis for Clean. Clean is an experimental functional language based on graph rewriting (Brus et al. (1987)). The language is designed to provide a firm base for functional programming. In particular, Clean is suitable and used as an intermediate language between functional languages and sequential machine architectures. Every Clean program is an FGRS.

Although the proposed extensions are also meaningful in more general graph rewriting systems, throughout the rest of this paper it will be assumed that FGRS's are used. In all examples the Clean syntax will be used. The extensions to graph rewriting proposed in this paper are incorporated in a new intermediate language: Concurrent Clean (Nöcker et al. (1991)). For an intuitive understanding of what follows it is not necessary to know all details of FGRS's. Some general knowledge about graphs and functional languages will be sufficient. A few examples of FGRS's:

```
Hd      (Cons a b)              →   a                                 ;

Fib     0                       →   1                                 |
Fib     1                       →   1                                 |
Fib     n                       →   +I (Fib (--I n)) (Fib (-I n 2))   ;

Second  (Pair x y:(Cons a b))   →   y                                 ;

Ones                            →   x: Cons 1 x                       ;
```

Every expression is actually a graph consisting of nodes. Each node contains a symbol and a possibly empty sequence of argument nodeids. If these nodeids are implicit, an ordinary tree structure is assumed. Using them explicitly before a colon, one can define any graph structure. The last rule in the example is a typical graph rewrite rule containing a cycle in the right-hand-side. In many cases, the functional graph rewrite rules can intuitively be seen as ordinary function definitions. Each function has one or more alternatives which are distinguished by patterns on the left-hand-side of the definition. Symbols other than function symbols are called *constructors* because they are usually used as data structures (i.e. constructs for defining new data types). For practical reasons some types are assumed to be predefined, such as INT or BOOL. Furthermore, some functions for arithmetic are assumed to be defined on these types, such as ++I (i.e. integer increment) or *I (i.e. integer multiplication).

3. FGRS's with Dynamic Process Creation: P-FGRS's

The creation of parallel reduction processes, also called *reducers*, can be seen as a special case of influencing the order of evaluation. In FGRS's the reduction order is changed by means of annotations. These annotations, which have the form of a string placed between curly braces, can be assigned to both nodes and nodeids. When the reduction strategy encounters an annotation it changes its default reduction order which will influence the way in which a result is achieved. Changing the reduction order is important if one wants to optimise the time and space behaviour of the reduction process.

In sequential FGRS's only one annotation is defined indicating that the reduction of the annotated argument of a symbol (function or constructor) is demanded. From an operational point of view, this annotation, denoted by {!}, will force the evaluation of the corresponding argument before it is tried to rewrite the graph according to a rule definition of the symbol. Note that these annotations may make the reduction strategy partially eager instead of lazy. In formal reasoning about programs with {!} annotations on the left-hand side it will always be true that the annotated argument will be in root normal form when the corresponding rule is applied. The semantics of annotations on the right-hand side can be explained via transformations to sets of rules with left-hand side annotations only. Intuitively, the transformation involves introducing an extra internal reduction with an annotated left-hand side which forces evaluation before the rule is applied.

Example of a rule with a {!} on the right-hand side:

```
Gen    n            →  Cons n (Gen {!}(++I n))          ;
```

This rule is transformed into:

```
Gen    n            →  Gen' n (++I n)              ;
Gen'   n   {!} m    →  Cons n (Gen m)              ;
```

3.1. Process creation

A single sequential reducer repeatedly chooses one of the redexes which are indicated by the reduction strategy and rewrites it. Interleaved reduction can be obtained by incarnating several sequential reducers which reduce different parts of the same graph. By using {!} annotations it is possible to influence the order in which the redexes are reduced by a single reducer.

Now a new annotation is introduced: {!!}, to indicate that a new sequential reducer has to be created with the following properties:

- the new reducer reduces the corresponding graph to root normal form after which the reducer dies;
- the new reducer can proceed interleaved with the original reduction process;
- all rewrites are assumed to be indivisible actions;
- if for pattern matching or reduction a reducer needs access to a graph which is being rewritten by another reducer, the first reducer will wait until the second one has reduced the graph to root normal form.
- for determining its redexes it uses the functional reduction strategy parameterized with {!} annotations.

Considered operationally, if a {!!} annotation is encountered in the right-hand side by a reducer, a new reducer is created after the redirection has been done (and if there is copying, also after the copying (see also section 4). If a {!!} annotation is specified on the left-hand side, a new reducer is created just before the original reducer would reduce the corresponding function application. In reasoning about programs with {!!} annotations on the left-hand side it will always be true that the annotated argument will have been reduced (by another reducer) to root normal form when the corresponding rule is applied. The meaning of {!!} on a left-hand side can be explained via transformations to sets of rules with right-hand side annotations only.

Example of {!!} on the right-hand side:

```
Fib   0    →    1                                          |
Fib   1    →    1                                          |
Fib   n    →    +I ({!!} Fib (-I n 1)) ({!!} Fib (-I n 2))     ;
```

Another way of looking at {!!} annotations is that they influence the overall reduction order. In this view, {!!} annotations are parameters of the overall reduction strategy. This overall reduction strategy will then indicate possibly more than one redex (every process may have a redex). The global reducer will make a non-deterministic choice out of the redexes indicated by the global strategy. So, in this way parallel reduction is modelled via (non-deterministic) interleaved execution of the individual reducers. In section 5 we will see how real parallel evaluation can be made possible.

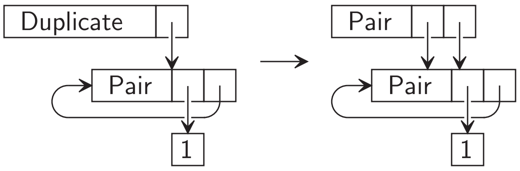
4. FGRS's with Lazy Copying: C-FGRS's

A notion of graph copying is necessary if one wants to express explicitly the distribution of data in parallel environments. One would expect however that it is already possible to express graph copying in graph rewriting systems. Although this is indeed the case, it is rather complex. A function has to be defined which duplicates its argument. Evidently, the following definition only produces two pointers to the argument but it does not duplicate the argument itself.

A graph sharing example:

Consider the following rule:	The graph:	Reduces to the graph:
Duplicate x → Pair x x ;	@1: Duplicate @2	@4: Pair @2 @2,
	@2: Pair @3 @2,	@2: Pair @3 @2,
	@3: 1	@3: 1;

Here @x is a denotation for a nodeid. This rewrite step is illustrated in the following picture.



The only way to access the structure of the argument is to use pattern matching. The only way to duplicate a constructor is to match on it on the left-hand side and to create a new node with the same constructor on the right-hand side. Such a rewrite rule is needed for every constructor that may appear. Furthermore, on the right-hand side the graph structure of the argument has to be duplicated. To detect sharing, multiple occurrences of the same nodeid on the left-hand side should be admitted in FGRS's. Then, with many of such left-comparing rules (and a special strategy that handles left-comparing rules) a structure can be copied. The rules that define copying, are themselves part of the system which makes it difficult to reason about them because the copying gets intertwined with the rest of the evaluation. As a consequence, if such a structure contains redexes that have to be copied too, the reduction strategy has to be changed again in order to prevent that the strategy indicates these redexes.

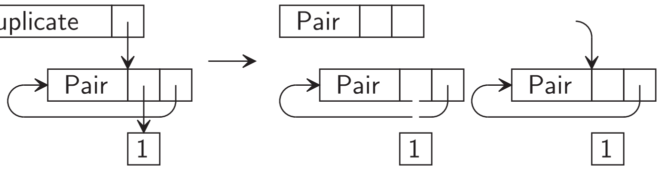
So, extending the semantics of FGRS's with a special mechanism to explicitly copy graphs (possibly containing redexes) would considerably increase the expressive power of these graph rewriting systems.

4.1. Eager copying

To denote a graph g that should be copied, the node identifier that refers to the root of g is attributed with a subscript c. The c subscript can be placed on nodeids of the right-hand side only. The copying takes place after the contractum is built but before the root of the redex is redirected to the root of the contractum. All copies of one right-hand side are made simultaneously. Copying a graph g implies that an equivalent graph g' is made which has no nodes in common with the original graph g. During the copying no rewriting takes place. So, for every node of g (also for redexes) there is an equivalent node in g'. A graph copying example:

Consider the following rule:	The graph:	Reduces to the graph:
Duplicate x → Pair x x_c ;	@1: Duplicate @2,	@4: Pair @2 @12,
	@2: Pair @3 @2,	@2: Pair @3 @2,
	@3: 1;	@3: 1,
		@12: Pair @13 @12,
		@13: 1;

This is illustrated in the following picture.



This way of copying is also called *eager copying* in contrast to lazy copying which is defined in the following sections.

4.2. Lazy copying

Consider a graph containing redexes. The extension of graph rewriting with explicit copying introduces the possibility to copy this graph including all its redexes. We had already the possibility of sharing the graph. Unfortunately, there is nothing in between. Note that duplication of work can be avoided by maintaining the sharing with the original graph as long as the corresponding function applications have not been evaluated. If after the evaluation to root normal form the copying is continued, the graph is duplicated after the work is done. But, it can also be useful to break up the sharing. Take for example a function application that delivers a large structure after relatively few reduction steps. If a graph containing such a function application is submitted to another processor then it is preferable not to reduce this application before the submission.

Copying with the choice of maintaining or breaking up the sharing is called *lazy copying*. A node on which copying will be stopped temporarily, is called a *deferred node*. To denote a deferred node it is attributed with a subscript d (which is denoted by attaching the d-attribute to the symbol of that node).

Lazy copying implies that when a copy action hits a deferred node, the copying itself is deferred. The applied occurrence of the nodeid of the deferred node of which the (now deferred) copy was being made, will be administered as being a *copying deferred nodeid.* When a deferred node is in root normal form, the node will not longer be deferred. The actual copying may continue, but, as we shall see, this will only happen when this copy is demanded. Nodeids of which the contents need to be known for matching, are according to the functional reduction strategy first reduced to root normal form. When a copying deferred nodeid is reduced this will trigger the continuation of the copying.

A lazy copying example:

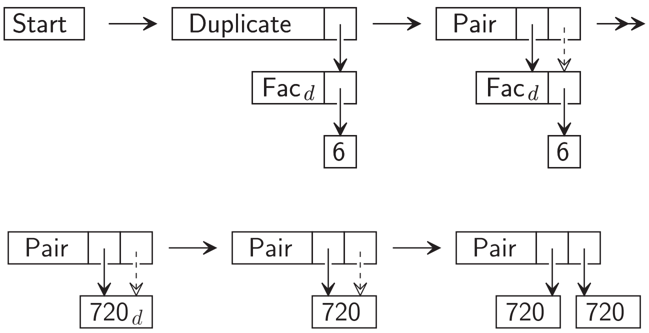
Start		→	Duplicate (Fac$_d$ 6)	;
Duplicate	x	→	Pair x x$_c$	;
Fac	0	→	1	|
Fac	n	→	*I n (Fac (−−I n))	;

The following rewrites occur:

@1: Start;	→	@2: Duplicate @3, @3: Fac$_d$ 6;	→	@4: Pair @3 @3$_c$, @3: Fac$_d$ 6;	↠
@4: Pair @5 @5$_c$, @5: 720$_d$;	↠	@4: Pair @5 @5$_c$, @5: 720;	→	@4: Pair @5 @15, @5: 720, @15: 720;	

The nodeid attribute c in the graph is used to denote that that the nodeid is a copying deferred nodeid. Note that the c attribute was inherited when the node @3 was redirected to @5 which corresponded with the reduction of the node. Do not confuse the c attribute in the graph with the c attribute in the rules which denotes that a copy action has to be started. The deferred attribute of the node @5 is taken away when it is recognized that the node is in root normal form.

The rewrites are also shown in the following picture (dashed arrows indicate copying deferred nodeids).



4.3. Operational semantics of lazy copying

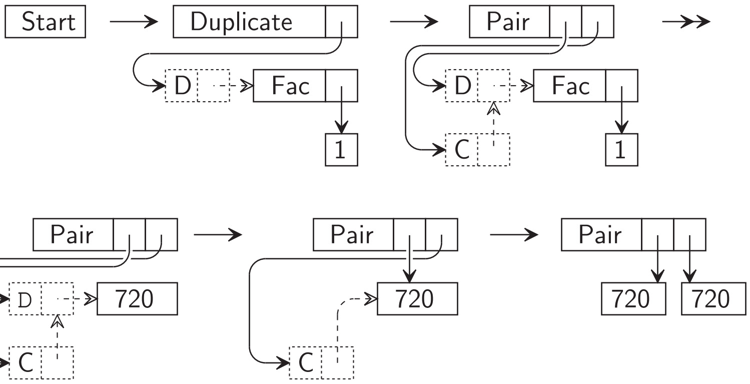
To explain the semantics of lazy copying we introduce two special kinds of indirection nodes: a D(eferred) node which indicates that the function it is pointing to has the deferred attribute, and a C(opy of such a deferred node) node. This node indicates that the graph it is pointing to still has to be copied: the copying is deferred. If on a right-hand side nodeid n is attributed with the c subscript, all nodes accessible from n have to be copied such that the new graph structure is copy-equivalent with the old one. However, if the copy action hits on a D-node, a C-node which refers to the D-node is created and the subgraph to which the D-node refers is not copied. If the copy action hits on a C node, a new C node is created which has the same argument as the original C node. After the copying has been performed this way, this rewrite is finished and reduction continues as usual. D and C-nodes can be rewritten via the following *internal* reduction rules:

```
D    {!} x    →    x      ;
C    {!} x    →    x_c    ;
```

The strict annotations provide that the "deferred" or "not yet copied" property of a function application is is inherited by all intermediate function results until finally a root normal form is reached. Hereafter the reducer is able to apply the special rewrite rules for D and C which will make these D- and C-nodes disappear. If the previous example is considered again, it should be more clear what the semantics are:

```
@1: Start;           →   @2: Duplicate @i,   →   @4: Pair @i @j,     ↠
                         @i: D @3,               @i: D @3,
                         @3: Fac 6;              @j: C @i,
                                                 @3: Fac 6;
@4: Pair @i @j,      →   @4: Pair @5 @j,     →   @4: Pair @5 @15,
@i: D @5,                @j: C @5,               @5: 720,
@j: C @i,                @5: 720;                @15: 720;
@5: 720;
```

This is also illustrated in the following picture.



Note that when the deferred copy turns out to be not needed by the reduction strategy, the C rule will never be executed, so the copying will not be continued.

Properties of lazy copying

An interesting aspect of lazy copying is that normal forms do not contain defer or copying deferred attributes. Every subgraph of a graph in normal form is trivially in root normal form. Evaluation of nodes to root normal form eliminates the defer attributes. Evaluation to root normal form and/or the attempt to access a node will cause the deferred copying to continue.

With the following rule:	this will be the normal form of Start:	
Start → x: Pair 1 x ;	@1: Pair 1 @1;	|| a cycle.
Start → x: Pair$_d$ 1 x ;	@1: Pair 1 @1;	|| a cycle.
Start → x: Pair 1 x$_c$;	@1: Pair 1 @11, @11:Pair 1 @11;	|| a cycle with a copy of it: || a once unravelled cycle.
Start → x: Pair$_d$ 1 x$_c$;	@1: Pair 1 @11, @11:Pair 1 @1$_c$;	|| every copy contains a copying deferred || nodeid which leads to more unravelling || yielding an infinite normal form!

Lazy copying does influence the normal forms in the graph world. Sharing may be broken up when a cycle is copied which contains deferred nodes. The result will be partly unravelled with respect to a full copy. A typical example is given by the latter rewrite rule of the previous example. In C-FGRS's the normal form is also influenced by the order of evaluation (and hence by annotations). If the deferred nodes are not reduced before an attempt to copy them is made, the result will be partly unravelled. A typical example is given below.

Consider the following rules:

Start → r: A (F x),
x: B y,
y: {!}I$_d$ z,
z: E x;

F x → x$_c$;
I x → x;

The normal forms with and without {!} are respectively:

@1 : A @13,	@1 : A @13,
@13 : B @15,	@13 : B @15,
@15 : C @13;	@15 : E @113,
	@113 : B @15;

Note that an extra rule had to be introduced in order to delay the copying. The unravelling of the normal forms of a rule system with lazy copying will always be the same as the unravelling of the normal forms of the same rule system without lazy copying. In other words unravelling is invariant with respect to lazy copying. This is an interesting property for the implementation of functional languages and for term graph rewriting as in Barendregt et al. (1987a). It seems that it enables the proof of the soundness and completeness of implementations which use sharing and copying via term graph rewriting. Lazy copying and term graph rewriting is a very promising topic for further research.

With the copy indication and the defer indication lazy copying is introduced in graph rewriting systems. By introducing the possibility to use subtle combinations of sharing and copying this greatly improves the expressive power of graph rewriting systems. Furthermore, in the next section it will be shown that lazy copying can also be the basis for communication in a parallel environment.

5. The Descriptive Power of PC-FGRS's

In this section the power of the PC-FGRS's is illustrated by showing how with certain combinations of process creation and lazy copying various kinds of process behaviours can be modelled. There are several kinds of behaviours one may be interested in, such as fine and coarse grain parallelism, all kinds of process topologies (hierarchical and non-hierarchical process topologies), synchronous and asynchronous communication between processes, etcetera. At first glance it may seem easy to specify these behaviours in PC-GRS's, since there is the possibility to create reducers dynamically. However, note that a rewriting step is considered to be indivisible and without this assumption reasoning about rewriting systems is in general not possible. Still, of course, one would like to be able to create reducers of which the rewriting steps can be performed in *parallel* instead of *interleaved*. However, it should be clear that, without any restrictions, parallel rewriting causes problems. Imagine that a copy of a subgraph is made while another reducer is working on that subgraph. Problems may also arise when redirections are performed in parallel. Probably there will not be a problem when two reducers are running on subgraphs which have no node in common and no reference to each other.

To call a reducer a *parallel* reducer with respect to another reducer it has to be proven that the constraint that a rewrite step is an indivisible action can be weakened. More precisely, it has to be proven that the corresponding rewrite steps cannot interfere with each other and therefore may be *considered* as being indivisible so that they actually can be performed in parallel. This condition that has to be proven is also called the *non-interference* condition. Hence, the claim that parallel computations can be expressed in our model can only be justified by proving that, under specific conditions, certain reducers are parallel reducers with respect to certain other reducers.

As in the introduction, we consider loosely coupled parallel machine architectures (each processor has its private memory) as the most interesting class of architectures. It should be clear that the kind of architecture the reduction is performed on influences the rewriting model. For instance, in a shared memory machine graph copying may be superfluous.

5.1. Modelling parallel rewriting on loosely coupled parallel architectures

A loosely coupled parallel computer is defined as a multiprocessor system that consists of a number of self-contained computers, i.e. processors with their private memory attached to each other by a sparsely connected network. An important property of such system is that for each processor it is more efficient to access objects located in its own local memory than to use the communication medium to access remote objects. In order to achieve an efficient implementation it is necessary to map the computation graph to the physical processing elements in such a way that the communication overhead due to the exchanging of data is relatively small. Therefore, the computation graph is divided into a number of subgraphs (*grains*) which have the property that the intermediate links are sparsely used. Unfortunately, it is undecidable how much work the reduction of a subgraph involves. Furthermore, there are no well-established heuristics for dividing a graph into grains. So, this partition of the graph cannot be performed automatically. Therefore, in the program it has to be explicitly indicated what is expected to represent a large amount of reductions relative to the expected communication overhead. In this way the program can be tuned to a particular parallel machine architecture.

The annotations and indications in the PC-FGRS have to be used in such a way that non-interference can be proven for reducers which might be executed on different processors. To avoid the need for a proof for every PC-FGRS, methods of annotating and indicating will be developed. Using these methods will guarantee that parallel execution of groups of reducers is allowed.

Divide-and-Conquer evaluation

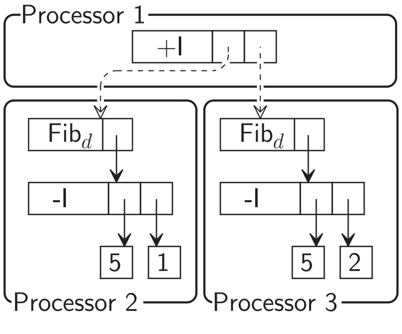
An obvious method to get safe parallelism is to create a reducer on a copy of an indicated subgraph. Such a copied subgraph has the property that it is *self-contained,* i.e. the root of the subgraph is the only connection between the subgraph and the rest of the graph. This will make it possible that the copied subgraph is reduced in parallel on another processor. When it is reduced to root normal form the result will be copied back to the father processor. So, copying is performed twice: one copy is made of the task for the off-loading of the task and one copy is made of the result to communicate it to the father.

A self-contained subgraph will be regarded as a *virtual processor* because it has the property that it may be reduced on another processor. It is easy to prove that on a self-contained subgraph it is allowed to weaken the interleaving restriction to parallelism: the self-contained subgraph can only be accessed by other reducers via the root and the semantics of P-FGRS's does not allow reducers to access a node on which another reducer is running.

Example of a divide-and-conquer algorithm:

$$\begin{array}{lllr}
\mathsf{Fib\ 0} & \rightarrow & \mathsf{1} & | \\
\mathsf{Fib\ 1} & \rightarrow & \mathsf{1} & | \\
\mathsf{Fib\ n} & \rightarrow & \mathsf{+I\ left}_c\ \mathsf{right}_c, & \\
 & & \mathsf{left:}\ \{!!\}\ \mathsf{Fib}_d\ (-\mathsf{I\ n}_c\ 1), & \\
 & & \mathsf{right:}\ \{!!\}\ \mathsf{Fib}_d\ (-\mathsf{I\ n}_c\ 2) & ;
\end{array}$$

The {!!} annotations combined with the copy and defer indications specify that both calls of Fib can be evaluated in parallel. The graph on which each process runs is self-contained because the root of the graph on which a process is started, is built with copies of subgraphs as arguments. The father reducer may already start with copying the result but this copying will be immediately deferred. It can continue each time when an argument of +I is in root normal form.The following picture illustrates the virtual processor structure after one reduction of Fib 5.



This way of modelling divide-and-conquer algorithms relies on the fact that the subgraph to be reduced is self-contained and that after the reduction to root normal form, the result is also self-contained. Unfortunately, self-containedness is an undecidable property, for, if a lazy copy of a certain graph is made this graph may contain deferred nodes. But, as will be shown in the next section, it is possible to use a graph property that is on the one hand weaker than the property of self-containedness and, on the other hand, strong enough not to violate the non-interference requirement.

Modelling loosely coupled evaluation

A method which makes it possible to model process behaviours that are more general than divide-and-conquer, must provide a way to define arbitrary connections between processes and processors. The lazy copy scheme introduced in section 4 provides a way to make a self-contained copy on a lazy manner. Such a lazy copy is a self-contained subgraph with the exception of copying deferred nodeids, which are references to deferred nodes in the graph. These deferred nodes will be copied *later* if they are in root normal form and needed for the evaluation. So, copying deferred nodeids are natural candidates for serving as inter-connections between parallel executing processes because they induce further copying when they are accessed. Therefore, communication between parallel processes can be realized via copying deferred nodeids. In this context copying deferred nodeids are also called *communication channels* or just *channels.*

A subgraph is *loosely connected* if channels (copying deferred nodeids) are the only connections between the subgraph and the rest of the graph. Also a loosely connected subgraph is regarded as a virtual processor because it has the property that it may be reduced on another processor. From now on the notion virtual processor stands for loosely connected subgraph. Several processes (reducers) can run on such a virtual processor. Processes running on the same virtual processor are running interleaved. So, there is interleaved multiprocessing on each virtual processor. Processes running on different virtual processors run in parallel. The semantics of copying deferred nodeids implies that channels have the following properties. The flow of data through a channel is the reverse of the direction of the copying deferred nodeid in the graph. Since channels are (applied) nodeids, they can be passed as parameters or copied. Now, suppose that a parallel process is reducing a loosely connected subgraph. This process may need the reduction of a channel connected to another processor. Of course, this channel cannot be reduced by the demanding process. It has to be reduced by another process running on the virtual processor which contains the graph whereto the channel refers. Now, the demanding process will be suspended until the result has been calculated by the process running on the other processor. A channel can be used to retrieve an (intermediate) result in a demand-driven way, i.e. as soon as the result of a sub-reduction is needed a request for the result is made. This request will be answered if the corresponding result is in root normal form. Note that the channel vanishes after the result has been returned. Because the copying is lazy new channels may have come into existence.

The question is now when the non-interference condition is fulfilled for reducers running on different virtual processors such that they can run in parallel instead of interleaved. The non-interference condition is satisfied if it can be guarantied throughout the execution of the program that when a parallel reducer is demanding information from a channel which refers to another virtual processor,

- this subgraph is either in root normal form (such that it can be copied lazily to the demanding process) or,
- there is a process running on the other virtual processor which is reducing the subgraph if it is not yet in root normal form (such that the demanding process will wait until the information has been reduced to root normal form).

Virtual processors which satisfy these conditions are called *loosely coupled virtual processors*.

It is possible to show that this allows the weakening of the restriction of interleaving to parallelism with respect to the loosely coupled virtual processors: parallel reducers running on different virtual processors work on different loosely connected subgraphs. They can only access subgraphs on other processors via copying deferred nodeids (channels). The demanding reducer will wait if the information is not in root normal form because in that case another process is reducing the information. If the information is in root normal form a lazy copy is made. In that case the resulting graph, i.e. the original graph of the demanding reducer together with the copy that has been made, is also loosely connected.

A method to create loosely coupled virtual processors

The obvious way to guarantee that virtual processors are loosely coupled, is to create a reducer on every deferred node. Hence, when a deferred node is created, at the same time also a process is started which reduces the deferred node. So, when via a copy a channel will be created to the node, the node will already be in root normal form or a reducer is still reducing it to root normal form. First, we introduce two abbreviations {e} and {i} that can be put on a node n. Example:

```
Fib n   →   +I left right,
            left: {i} Fib (−I n 1),
            right: {e} Fib (−I n 2)    ;
```

The {e} abbreviation (e for external) will create a new loosely coupled virtual processor together with an external reducer which reduces the corresponding loosely connected subgraph in parallel. To realize this, a channel to a lazy copy of the subgraph is made and a process is created to reduce this copy. The channel provides that a (lazy) copy of the result is returned if its value is demanded on other processors. In particular a lazy copy of the result is returned to the father process if it demands its value. The {i} abbreviation (i for internal) will create a new internal reducer on the same virtual processor which reduces the corresponding subgraph interleaved with the other processes on the same virtual processor. A deferred node to this subgraph is created which provides that a (lazy) copy of the result is returned if its value is demanded on other virtual processors (since all virtual processors are created via lazy copies, this demand will come via a channel). To realize this, a deferred node to the indicated subgraph is made and a process is created to reduce it. The {e} and {i} abbreviations may be used on the same positions as annotations. When an {e} or {i} abbreviation is put on a nodeid, this is equivalent with putting it on the node the nodeid belongs to. For each occurrence on a node a simple program transformation is made as follows.

Each occurrence of:	will be substituted by:
n : {e} Sym $a_1 \ldots a_n$	n : I x_c, x : {!!} I_d y_c, y : Sym $a_1 \ldots a_n$

A reducer is created by the {!!} annotation, it will reduce a node which contains the identity function of a lazy copy of the annotated node Sym $a_1 \ldots a_n$. The node on which the reducer is started, is itself deferred and a channel is immediately created to it via the copy in the new definition of the node n.

Each occurrence of:	will be substituted by:
n: {i} Sym $a_1 \ldots a_n$	n : {!!} I_d x, x: Sym $a_1 \ldots a_n$

A reducer is created on a deferred node. All sharing is maintained. The nodeids x and y in the substitution rules stand for nodeids not used elsewhere in the rewrite rule, and I is just the identity function: I x → x. The indirection nodes are only created to see to it that the copies are made correctly. In the following they are considered to be internal nodes.

It can be proven that, when using the {e} and {i} abbreviations only (i.e. neither other defer or copy attributes nor other process annotations), it is guarantied that each subgraph supplied with an {e} denotes a loosely coupled virtual processor. So, the proposed abbreviations provide a method to create loosely coupled virtual processors which allows real parallel execution.

5.2. Examples of the use of the proposed method

In this section some small examples are given illustrating the expressive power of the method for loosely coupled evaluation.

Non-hierarchical process topology

With the {e} abbreviation a parallel (sub)reduction can be created introducing a new virtual processor. With the creation of internal processes by using {i}, multiprocessing can be realized on each virtual processor. The only way to refer to such an internal process is via its channel. If such a channel node is passed (via a lazy copy) to another virtual processor, a communication channel between this processor and the reducer on the original processor is established. In this way any number and any topology of communication channels between processes and processors can be set up. For instance, it is possible to create a cycle of virtual processors. An example of this is given at the end of this section.

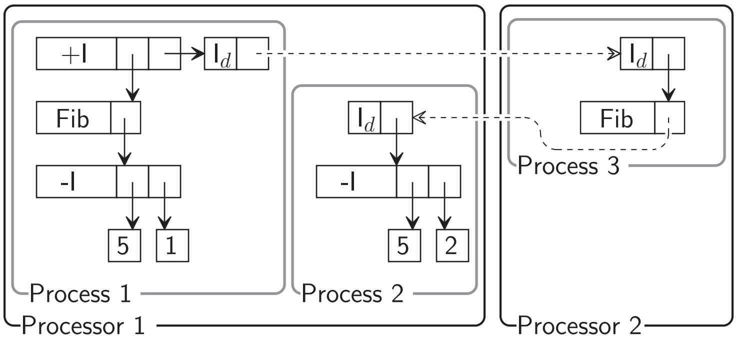
In the following example a simple non-hierarchical process topology is demonstrated. It serves the purpose of explaining how such process topologies can be expressed (it does not realistically implement the Fibonacci function). The Fib example uses a non-hierarchical process structure (which is very unconventional for Fib). The second call of Fib will be executed on another virtual processor but the argument of that call is reduced internally on the virtual processor that also does the first call of Fib.

```
Fib 0  →  1                      |
Fib 1  →  1                      |
Fib n  →  +I (Fib (−I n 1)) m,
          m: {e} Fib o,
          o: {i} −I n 2          ;
```

This is equivalent to:

```
Fib 0  →  1                      |
Fib 1  →  1                      |
Fib n  →  +I (Fib (−I n 1)) m,
          m : I x_c,
          x : {!!} I_d y_c,
          y : Fib o,
          o : {!!} I_d z,
          z : −I n 2             ;
```

So the following process topology is obtained (a snapshot of the program execution of Fib 5 is given):



In the picture it is shown how the graph is distributed over two virtual processors. Channels are dashed. Note that the direction of the flow of data through a channel is the reverse of the direction of the corresponding reference in the graph. In the following, internal indirection nodes are not shown in pictures and their defer indications are added to the nodes they refer to.

Asynchronous virtual processor communication with streams

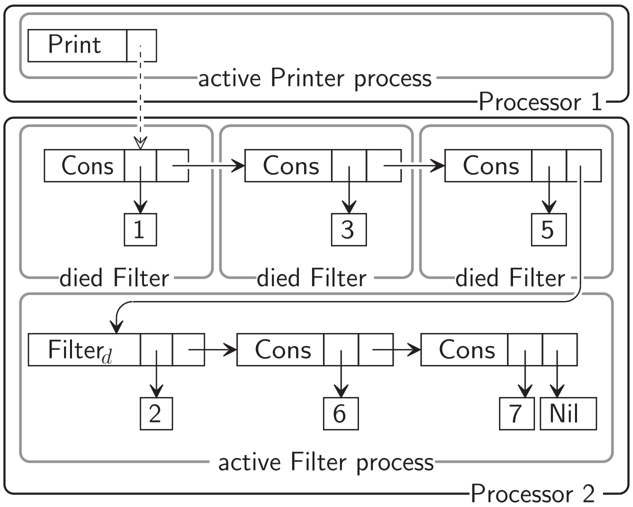
It is possible to model asynchronous communication between virtual processors, i.e. a virtual processor is already computing the next data before the previous data is communicated. To achieve this a family of internal processes has to be created connected to the communication channel between the processors. Each process computes a partial result which can be sent across the channel. Just before a process delivers the partial result (and dies) it creates a new process chained via a new channel to the delivered result. This new member of the family will compute the next partial result on the same way. For convenience sake, such a cascaded family of processes is often regarded as being one (asynchronously) sending process with some family name. The chain of channels is then regarded to be one channel. The total result which is copied, is sometimes called a *stream*. Note that this kind of stream is capable of sending over more than one node (a *burst*) at the same time. Furthermore, these streams can contain cyclic graphs such that cycles can be sent to another processor.

A virtual processor may contain several of such families each producing a stream via a chain of channels. In the case of the following filter example the virtual processor contains exactly one such process: Filter. It sends a stream via the channel to the process Print:

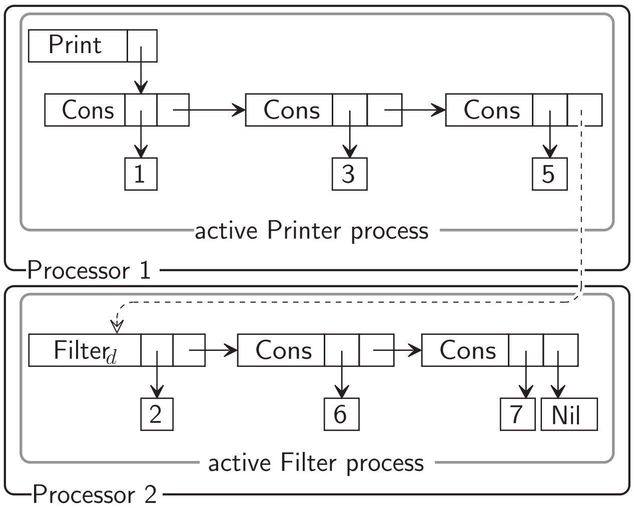
```
Start list             →    Print s,
                            s: {e} Filter list 2              ;

Filter Nil pr          →    Nil                               |
Filter (Cons f r) pr   →    IF     (=I (MOD f pr) 0)
                                   (Filter r pr)
                                   (NewFilter f r pr)         ;

NewFilter f r pr       →    Cons f rest,
                            rest: {i} Filter r p              ;
```

The main virtual processor creates a new virtual processor on which the Filter process is started. The channel s is the communication channel between the two processors. The function Filter removes from its first argument, which is a list, all the elements which are divisible by the number n. A part of the stream becomes available as soon as Filter has computed an element of the result list and a new interleaved Filter process has been created. It may start already computing the next element of the stream before the first is asked to be communicated. The partial stream result is a list containing the first element and a new channel reference to the new filtering process.

Assume that the list to be filtered contains the natural numbers from 1 to 7. Then the following situations can arise.

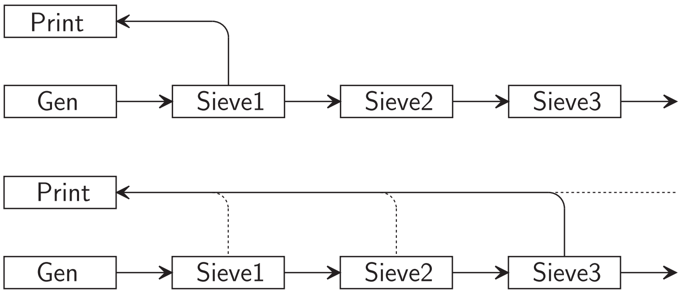


Now the three list elements, root normal forms yielded by successive filter processes, can be shipped with one lazy copy action.



Dynamically changing process topologies

The sieve of Eratosthenes is a classical example which generates all prime numbers. A pipeline of virtual processors is created. On each processor a Sieve process (a family of processes actually) is running. Those Sieves hold the prime numbers in ascending order, one in each Sieve. Each Sieve accepts a stream of integers as its input. Those integers are not divisible by any of the foregoing primes in the pipeline. If an incoming integer is not divisible by the local prime as well, it is send to the next Sieve. A newly created Sieve accepts the first incoming integer as its own prime and outputs this prime and the channel of the next Sieve to a printing processor. After that it starts sieving. A virtual processor called Gen sends a stream of integers greater than one to the first Sieve. The Gen process and every Sieve process proceed in more or less the same way as the Filter process of the previous example. They all are actually families of processes servicing chains of channels. They are regarded as single processes. Every chain of channels is regarded as one channel. So Sieve1 holds 2 as its own prime, Sieve2 holds 3, Sieve3 holds 5, and so on. The printing process one by one receives the channel identifications from these sieves and collects the corresponding primes. Seen through the time this can be illustrated as follows (all arrows indicate flow of data on channels).



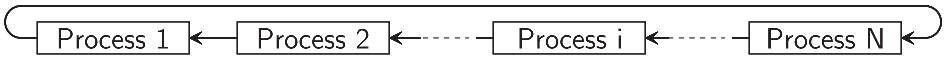
The Sieve program:

```
Start                    →   Print s,
                             s: {e} Sieve g,
                             g: {e} Gen 2                  ;

Sieve (Cons pr stream)   →   Cons pr s,
                             s: {e} Sieve f,
                             f: {i} Filter stream pr       ;

Gen n                    →   Cons n rest,
                             rest: {i} Gen {!} (++I n)     ;

Filter (Cons f r) pr     →   IF    (=I (MOD f pr) 0)
                                   (Filter r pr)
                                   (NewFilter f r pr)      ;

NewFilter f r pr         →   Cons f rest,
                             rest: {i} Filter r pr         ;
```

Note that when the {!} annotation in Gen would be left out, the increments of the integers would not be evaluated by Gen but by the first Sieve. Even worse: because the result of Gen is copied, the Sieve would have to recalculate every new integer by increments starting from 2.

Cyclic process structures

The next example shows how a cyclic process structure, i.e. a number of parallel reducers that are mutual dependent, can be created. This example has been extracted from quite a large program that implements Warshall's solution for the shortest path problem. The full algorithm can be found in Eekelen (1988). First the intended reducer topology is given in a picture.



This reducer structure can be specified directly in the following way.

```
Start                    →   last:CreateProcs NrOfProcs last         ;

CreateProcs 1 left       →   Process 1 left                          |
CreateProcs pid left     →   CreateProcs (−−I pid) new,
                             new: {e} Process pid left               ;
```

CreateProcs is responsible for the generation of all the parallel reducers. This process, which will finally become the first reducer, has initially a reference to itself in order to make it possible to expand it to a cycle of reducers. Each reducer is connected to the next one, i.e. the one with the next pid number, by means of a channel. During the creation of the processes this channel is passed as a parameter called left.

5.3. Properties of the proposed method

With the proposed abbreviations arbitrary process structure can clearly be expressed. Still one has to be careful with their use. Normally the abbreviations will be used to obtain a parallel version of an ordinary sequential program. In general the sequential program has to be transformed to create the wanted processes and process topologies. If the abbreviations of any parallel program are regarded as comments, again a sequential version of the program is obtained. In the given examples such a sequential version would yield the same result as the parallel version. Unfortunately, in general the normal form is not unique. In section 4 it was that the normal form in a C-FGRS depends on the order of evaluation. In section 3 it was explained that the overall reduction strategy of a P-FGRS is non-deterministic. Hence the normal form PC-FGRS will in general depend on the choices made by the reducer. Although the normal form is not unique, the different normal forms which can be produced are related. Modulo unravelling they are the same, i.e. if the normal forms are unravelled to terms, these terms are the same. This is a very important property. The consequence is that the use of PC-FGRS's as a base for the implementation of functional languages or of term rewriting systems is sound. In these cases first the terms are lifted to graphs and after reduction the graph in normal form will be unravelled to a term again. In this way, always the same term will be yielded.

6. Discussion

6.1. Related work

The idea to use annotations (Burton (1987),Glauert et al. (1987),Hudak and Smith (1986) or Kluge (1983),Vree and Hartel (1988)) which control the reduction order is certainly not new. Some of them are introduced on the level of the programming language (Burton (1987),Hudak and Smith (1986),Vree and Hartel (1988)) while others are introduced on the level of the computational model (Glauert et al. (1987),Goguen et al. (1986),Kluge (1983)). They all express that an indicated expression has to be shipped to another (or to some concrete) processor. Most annotations (Hudak and Smith (1986),Goguen et al. (1986),Kluge (1983), Vree and Hartel (1988)) are only capable of generating strict hierarchical "divide-and-conquer parallelism". Non-hierarchical process structures are possible in Burton's proposal. He proposes a call-by-name parameter passing mechanism (which must involve copying of some nodes) between mutual recursive functions. In DACTL (Glauert et al. (1987)), also based on Graph Rewriting Systems there is no overall reduction strategy. This means that the reduction order is completely controlled by the annotations in the rewrite rules. This makes DACTL very suited for fine grain parallelism, but makes it very hard to reason about the overall behaviour of the program. In all proposals copying graphs from one processors to another and back is implicit and cycles cannot be copied. Some annotations (Burton (1987),Hudak and Smith (1986)) are not only used to control parallelism but also to control the actual load distribution. Annotations for load distribution are not yet incorporated in our model, primarily because virtual processors can be freely created on the level of the computational model. However, the specification of load distribution will be investigated in the future.

6.2. Implementation aspects

Efficient implementation of FGRS's is possible on sequential hardware (Brus et al. (1987),Smetsers (1989)). The ideas presented in this paper are incorporated in the language Concurrent Clean (Nöcker et al. (1991)). Type information (Plasmeijer and van Eekelen (1993)) and strictness information (Nöcker (1990),Nöcker and Smetsers (1990)) play an important role. To investigate parallel programming a simulator for Concurrent Clean has been developed simulating multi-processing. This simulator operates in any sequential C environment. Experiments with this implementation indicate that PC-FGRS's are in principle very suited for implementation on loosely coupled parallel architectures. Most problems that have to be solved are of a general nature: "How can a graph (with cycles) be shipped fast from one processor to another?", "What is the best suited algorithm for distributed garbage collection?", "What happens if one of the processors is out of memory or is completely out of order?". The efficiency of a parallel implementation will strongly depend on the solutions found for these general type of problems. These problems have to be solved for other kinds of concurrent languages too. Perhaps it is possible to adopt existing solutions. But also alternative solutions which take the special behaviour of GRS's into account are thinkable.

6.3. Future work

Besides the concepts introduced in this paper (lazy copying, annotations for dynamic process creation, abbreviations) we will add annotations for load distribution and add predefined rules such that frequently used process topologies (pipelines, array of processes) can easily be defined. Efficient implementation of Concurrent Clean are planned on loosely coupled multiprocessor systems (e.g. a Transputer rack). Developing an efficient implementation will also involve research to load balancing and garbage collection (without stopping all processors).

The theoretical properties of PC-FGRS's will be further investigated. Especially in the context of term graph rewriting new results are envisaged. Using sharing and lazy copying, different ways of lifting term rewriting systems to graph rewriting systems can be investigated. Other strategies than the functional strategy may be interesting. For instance, adding reducers following a non-deterministic strategy may be useful for the specification of process control, including scheduling and interrupts.

7. Conclusions

In this paper two extensions of Functional Graph Rewriting Systems are presented: lazy copying and annotations to control the order of evaluation. The extensions are simple and elegant. The expressive power of a FGRS extended with both notions is very high. Multi-processing can be modelled as well as graph reduction on loosely coupled systems. Arbitrary process and processor topologies can be modelled, as well as synchronous and asynchronous process communication. The introduced abbreviations guarantee that the indicated subgraphs can be evaluated in parallel instead of interleaved. The abbreviations directly correspond with the notion of processes and processors and they are therefore relatively simple to use. The user-friendliness can be increased by creating libraries with functions which can create often used processor topologies like pipelines and arrays of processors. Efficient implementation of the proposed model on loosely coupled parallel architectures should be possible. Actual implementations are started.

PC-FGRS's are very suited to serve as a base for the implementation of functional languages. Sequential functional languages can efficiently be implemented by translating them to FGRS's. The expressive power of the proposed abbreviations in PC-FGRS's and the properties of these systems will now make it also possible to exploit the potential parallelism in the programs successfully.

References

Arvind, Nikhil and S. Rishiyur (1987). Executing a program on the mit tagged token dataflow architecture, *Proc. of Parallel Architectures and Languages Europe (PARLE)*, Eindhoven, The Netherlands, Springer Verlag, LNCS 258 I, pp. 1–29.

Barendregt, H.P. (1984). *The Lambda Calculus: its syntax and semantics*, Studies in Logic 103, North-Holland, Amsterdam.

Barendregt, H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep (1987a). Term graph reduction, *Proc. of Parallel Architectures and Languages Europe (PARLE)*, Eindhoven, The Netherlands, Springer Verlag, LNCS 259 II, pp. 141–158.

Barendregt, H.P., M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer and M.R. Sleep (1987b). Towards an intermediate language based on graph rewriting, *Proc. of Parallel Architectures and Languages Europe (PARLE)*, Eindhoven, The Netherlands, Springer Verlag, LNCS 259 II, pp. 159–175.

Barendsen, Erik and Sjaak Smetsers (1992). Graph rewriting and copying, *Technical Report 92-20*, University of Nijmegen.

Brus, T., M.C.J.D. van Eekelen, M. van Leer, M.J. Plasmeijer and H.P. Barendregt (1987). Clean - a language for functional graph rewriting, *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, Portland, Oregon, USA, Springer Verlag, LNCS 274, pp. 364–384.

Burton, F.W. (1987). Functional programming for concurrent and distributed computing, *The Computer Journal.*

Church, A (1932). A set of postulates for the foundation of logic, *Annals of Math. (2)*, 33, pp. 346–366.

Church, A (1933). A set of postulates for the foundation of logic, *Annals of Math. (2)*, 34, pp. 839–864.

Eekelen, M.C.J.D. van (1988). *Parallel Graph Rewriting, Some Contributions to its Theory, its Implementation and its Application*, Dissertation, University of Nijmegen.

Fasel, J.H. and R.M. Keller (1986). Proceedings of the workshop on graph reduction.

Glauert, J.R.W., J.R. Kennaway and M.R. Sleep (1987). Dactl: A computational model and compiler target language based on graph reduction, *ICL Technical Journal*.

Goguen, J., C. Kirchner and J. Meseguer (1986). Concurrent tern rewriting as a model of computation, *Proc. of Workshop on Graph Reduction*, Santa Fe, New Mexico, Springer Verlag, LNCS 279, pp. 53–94.

Gurd, J.R., C.C. Kirkham and I. Watson (1985). The manchester prototype dataflow computer, *Communications of the ACM*.

Hudak, P. and L. Smith (1986). Para-functional programming: A paradigm for programming multiprocessor systems, *Proc. of 12th ACM Symp. on Principles of Programming Languages*, pp. 243–254.

Kluge, W.E. (1983). Cooperating reduction machines, *IEEE Transactions on Computers*.

Koopman, P.W.M. and E.G.J.M.H. Nöcker (1988). Compiling functional languages to term graph rewriting systems, *Technical Report 88-1*, University of Nijmegen.

Nöcker, E.G.J.M.H. (1990). Strictness analysis based on abstract reduction, *Proc. of Second International Workshop on Implementation of Functional Languages on Parallel Architectures*, University of Nijmegen, the Netherlands, Technical Report 90-16, pp. 297–321.

Nöcker, E.G.J.M.H. and J.E.W. Smetsers (1990). Partially strict data types, *Proc. of Second International Workshop on Implementation of Functional Languages on Parallel Architectures*, University of Nijmegen, the Netherlands, Technical Report 90-16, pp. 237–255.

Nöcker, E.G.J.M.H., J.E.W. Smetsers, M.C.J.D. van Eekelen and M.J. Plasmeijer (1991). Concurrent clean, *Proc. of Parallel Architectures and Languages Europe (PARLE'91)*, Eindhoven, The Netherlands, Springer Verlag, LNCS 505, pp. 202–219.

Plasmeijer, M.J. and M.C.J.D. van Eekelen (1993). *Functional Programming and Parallel Graph Rewriting*, Addison Wesley, University of Nijmegen, The Netherlands.

Smetsers, J.E.W. (1989). Compiling clean to abstract abc-machine code, *Technical Report 89-20*, University of Nijmegen.

Toyama, Yoshihito., Sjaak. Smetsers, Marko. Eekelen and Rinus. Plasmeijer (1993). The functional strategy and transitive term rewriting systems, *in:* Plasmeijer Sleep and Eekelen (eds.), *Term Graph Rewriting*, John Wiley and Sons Inc., to appear.

Vree, W.G. and P.H. Hartel (1988). Parallel graph reduction for divide-and-conquer applications; part i - programme transformations, *Internal Report D-15*, University of Amsterdam.