7

PARALLEL GRAPH REWRITING ON LOOSELY COUPLED MACHINE ARCHITECTURES

M.C.J.D. van Eekelen, M.J. Plasmeijer, J.E.W. Smetsers.

Department of Theoretical Computer Science and Computational Models,

University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands,

June 1988.

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 guaranteed 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. Lazy and eager process creation is possible. Synchronous and asynchronous process communication can be modelled. Complicated parallel algorithms can be expressed which can go far beyond divide-and-conquer like applications.

7.1 INTRODUCTION

In the following paragraphs the importance of computational models is addressed. It is explained why Graph Rewriting Systems are suited to model the essential aspects of functional languages and their implementation. However, if one wants to model parallel evaluation Graph Rewriting Systems have to be extended. The motivation for the proposed computational model is given and the context is described in which the model will be used.

Computational models

In general, a programming language is composed out of many language constructs. Examining the semantics of a language carefully it is possible to classify these constructs: some of them can be regarded as the basic concepts of the language, while others are purely syntactic sugar added to the language for programming convenience or for software engineering reasons. In order to understand the facilities offered by a language it is important to know what the essential language constructs are and what they mean. The key question is: what is the ideal basic model of computation for the language? If this model of computation is known it is much easier to reason about the essential properties of the language, the expressive power, the orthogonality of the design, the implementation methods for a given computer architecture and the design constraints for new architectures to support the language.

One problem is that there are, in principle, many computational models which can be used. For instance, any deterministic computation can be expressed on a Turing Machine. But this will not make reasoning more easy because this computational model is too restrictive. 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.

Graph rewriting systems and functional languages

Our prime interests are functional languages and their implementation on sequential and parallel hardware.

Traditionally, the lambda calculus (Church (1932/1933), 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 lacks pattern matching and the notion of sharing of computations. Patterns contain important information for strictness analyzers (Nöcker (1988)). Sharing of computations is essential to obtain efficient implementations on traditional hardware.

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,b)) 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 & Nöcker (1988), van Hintum & van Schelven (1988)). However, on parallel hardware sharing has to be handled with care, so it is not at all simple to make an efficient parallel implementation.

Lazy copying

Explicitly controlled copying can be very useful. In a sequential environment explicit control over the copying process can be used to improve the efficiency of memory management. In a parallel environment communication between processors with local memory always involves copying. With an explicit mechanism for copying the communication can be controlled on the level of the rewriting system itself.

In this paper graph rewriting is extended with a notion of explicit (lazy) copying. When a full copy is made, sharing is lost. 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. In this paper lazy copying will stand for the notion of having the possibility to explicitly denote that a copy or a lazy copy has to be made.

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. Its merits in sequential and parallel environments will be discussed.

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*.

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 shorter redex). If in this context redexes are rewritten in any order, the normal form (if it exists) will always be the same. The worst thing that can happen is that a computation does not terminate. The unicity of normal forms offers the theoretical possibility to reduce redexes in parallel. So, functional languages are often considered to be well suited for parallel computation. Two kinds of parallelism are distinguished: fine grain and coarse grain.

Fine grain parallelism

Although strictness is in general undecidable, it can be approximated by using strictness analyzers which can find enough redexes (grains) that can be evaluated in parallel without causing termination problems.

Fine grain machine architectures try to exploit this parallelism fully. In principle, any strict redex is a candidate for evaluation. Unfortunately, these architectures, such as data flow machines (Gurd *et al.* (1985), Arvind *et al.* (1987)), are very complex and not yet commercially available.

Coarse grain parallelism

Loosely coupled machine architectures, such as Transputer racks, 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 those redexes which yield a large amount of computation are suited to be evaluated in parallel. The complexity of a grain is in general undecidable and furthermore no satisfactory automatic approximation method has been developed.

So in this case it is necessary to have an explicit way of indicating the parallel redexes in a program by using special language constructs. Developing an efficient program starts with some sequential algorithm which is converted by one or more program transformation steps in order to obtain a program containing useful grains. Especially the so called Divide-and-Conquer algorithms are well suited to be treated in such a way. With only a few extra language primitives Divide-and-Conquer algorithms have been implemented efficiently on parallel machines (McBurney & Sleep (1987)). However, this approach is only suited for hierarchical process creation and communication, which is in general too restrictive.

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 7.6). 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.

Parallel graph rewriting

By denoting subgraphs on which reduction processes have to be created, parallelism in graph rewriting can be modelled. Reduction processes which evaluate an indicated subgraph, can be created dynamically in an eager manner (immediately) and in a lazy manner (when needed).

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*.

Parallel graph rewriting with lazy copying

It will be shown that *PC-GRS's* (GRS's with explicit parallelism and lazy coping) have a surprisingly high expressive power. Various process behaviours in different environments can be described in PC-GRS's. In particular, loosely coupled parallel evaluation can be modelled such that any process communication structure can be defined. In order to illustrate the expressive power examples will be given of some non-trivial parallel algorithms.

Structure of this paper

The next section introduces graph rewriting briefly. After that in section 7.3 graph rewriting is extended with lazy coping. In section 7.4 eager and lazy process creation are introduced. The power of the combination of lazy copying and eager and lazy process creation is shown in section 7.5. In particular, the use of the system to model parallel graph reduction on loosely coupled parallel architectures is demonstrated. In section 7.6 comparisons with related work, implementation aspects and directions for future research are given.

7.2 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 nodeid's (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 (in implementations this is realized by overwriting the node or by using an indirection node). The part of the graph which matches the pattern is sometimes called a *redex*.

A *reduction strategy* is a function which makes choices out of the available redexes. A *reducer* is a process which 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. Reducers are deterministic or non-deterministic. A reducer chooses (non-)deterministically one of the redexes which are indicated by the strategy. In this paper only deterministic reducers 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. Note that the root normal form property is in general undecidable. When a reducer terminates its result generally is a root normal form. 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 left-comparing (rewriting systems where multiple occurrences of variables on left-hand-sides are allowed are called *left-comparing* or *non left-linear*). No multiple occurrence of variables implies that it is impossible to pattern match on equivalency of nodeid's (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 which resembles very much the way execution proceeds in lazy functional languages (a full formal definition of this strategy can be found in Smetsers *et al.* (1988)).

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 can be used for term graph rewriting (Barendregt *et al.* (1987a)). Term graph rewriting connects term rewriting systems (TRS) (Klop (1987)) in which no sharing can be expressed, with graph rewriting systems. Term graph rewriting means that a TRS is interpreted (lifted) as an FGRS. The normal forms of the FGRS which are graphs, are unravelled to terms in the TRS world. Via term graph rewriting in (Barendregt *et al.* (1987a)) it is proved that sharing terms is sound, furthermore restrictions are given which ensure completeness of sharing implementations.

FGRS's are also the 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 which are proposed in this paper will be incorporated in a new intermediate language: Concurrent Clean. This language is now being defined (Smetsers *et al.* (1988)).

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. By giving some FGRS examples the similarity between functional languages and FGRS's is illustrated.

```
Hd (Cons a b)                ->   a                              ;

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

First (Pair x:(Cons a b) y)  ->   x                              ;

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 nodeid's. If these nodeid's are implicit, an ordinary tree structure is assumed. Using them explicitly, 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).

Influencing the order of evaluation

The FGRS's which are used in this paper, are allowed to be annotated. To every node and to every nodeid one or more attributes can be assigned via annotations. Annotations have the form of a string placed between curly braces. The only annotations which are used in this paper, are annotations which influence the order of evaluation. These annotations play an important role because they are parameters of the reduction strategy. The functional reduction strategy takes them into account and therefore they influence the way in which a result is achieved. This is important if one wants to optimize the time and space behaviour of the reduction process. It is assumed that annotations are never used in such a way that they influence the outcome of the computation or the termination of the reduction.

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. This annotation, denoted by `{!}` or `{strict}`, 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 deviate from the default evaluation order which then becomes partially eager instead of lazy. When more than one `{!}` annotation occurs on a right-hand-side, they are effectuated depth-first from left to right.

The `{strict}` annotation is important because, in general, rules with strict arguments can be implemented more efficiently (Brus *et al.* (1987), Plasmeijer & van Eekelen (198-)).

Annotations can belong to a node (node annotation which is placed before the symbol of the node) and to an argument (argument annotation placed in front of the argument). Annotations may occur on the right-hand-side as well as on the left-hand-side of a rule.

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

```
G n                 -> F {!}n                              ;
```

At this occurrence of `F` the evaluation of the argument is forced before `F` is applied.

Example of `{!}` on the left-hand-side:

```
F (Cons {!}n r)     -> *I n (--I n)                        ;
```

At all occurrences of `F` the evaluation of the (sub)argument is forced before applying the `F` rule. This can also be achieved by the following transformed set of rules:

```
F   (Cons n r)      -> F' (Cons {!}n r)                     ;
F'  (Cons n r)      -> *I n (--I n)                         ;
```

In 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 left-hand-side can be explained via transformations to sets of rules with right-hand-side annotations only. Intuitively, the transformation involves introducing an extra internal reduction with an annotated right-hand-side which forces evaluation after some matching but before the rule is applied. The precise transformation for `{!}` can be found in Smetsers *et al.* (1988).

7.3 EXTENDING FGRS'S WITH LAZY COPYING: C-FGRS'S

7.3.1 WHY COPYING?

It is very useful to have explicit graph copying in sequential environments (for reasons of memory management) and in parallel environments (for off-loading a copy to another processor). 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!

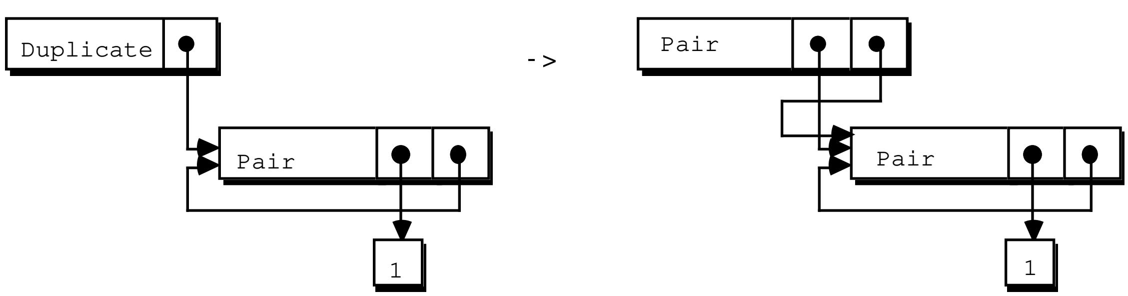
An ordinary rewrite rule:

```
Duplicate x -> Pair x x                                     ;
```

the left graph reduces to the right graph (`@x` is a denotation for a nodeid).

```
@1: Duplicate @2,          ->    @4:  Pair @2 @2,
@2: Pair @3 @2,                  @2:  Pair @3 @2,
@3: 1;                           @3:  1;
```

which 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. In order to make it possible to detect the shared nodes multiple occurrences of the same nodeid on the left-hand-side should be introduced in FGRS's. Then, with many of such left-comparing rules (and a special strategy that handles left-comparing rules: say the *left-comparing functional* strategy) a structure can be copied unless it contains redexes and they have to be copied too. To include that case the reduction strategy has to be changed again. All in all it is very cumbersome to do graph copying within graph rewriting systems, because the copying is not inherent. 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. In other words: copying is not part of the semantics of graph rewriting!

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

7.3.2 EAGER COPYING

To denote a graph g that should be copied, the node identifier which refers to the root of g is attributed with a subscript `c` or `copy`. The `c` subscript can be placed on nodeid's of the right-hand-side only. The copying takes place after the contractum is built and after the root of the redex is redirected to the root of the contractum. All copies of one right-hand-side are instantiated simultaneously.

Copying a graph g implies that an equivalent graph g' is made which has no nodes in common with the original graph g. No reduction takes place. So, for every node of g (also for redexes) there is an equivalent node in g'. Note that copy-equivalency is very much different from reduction-equivalency.

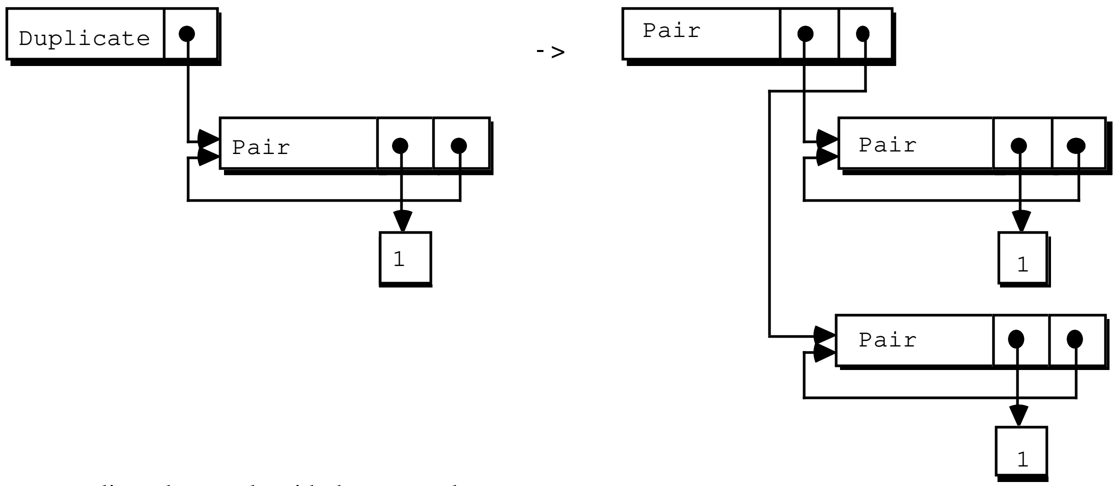
A graph copying example:

```
Duplicate x -> Pair x x_c                                    ;
```

the left graph reduces to the right graph. The new nodeid's are chosen in such a way that the structure is easily seen.

```
@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;
```

which is also illustrated in the following picture:

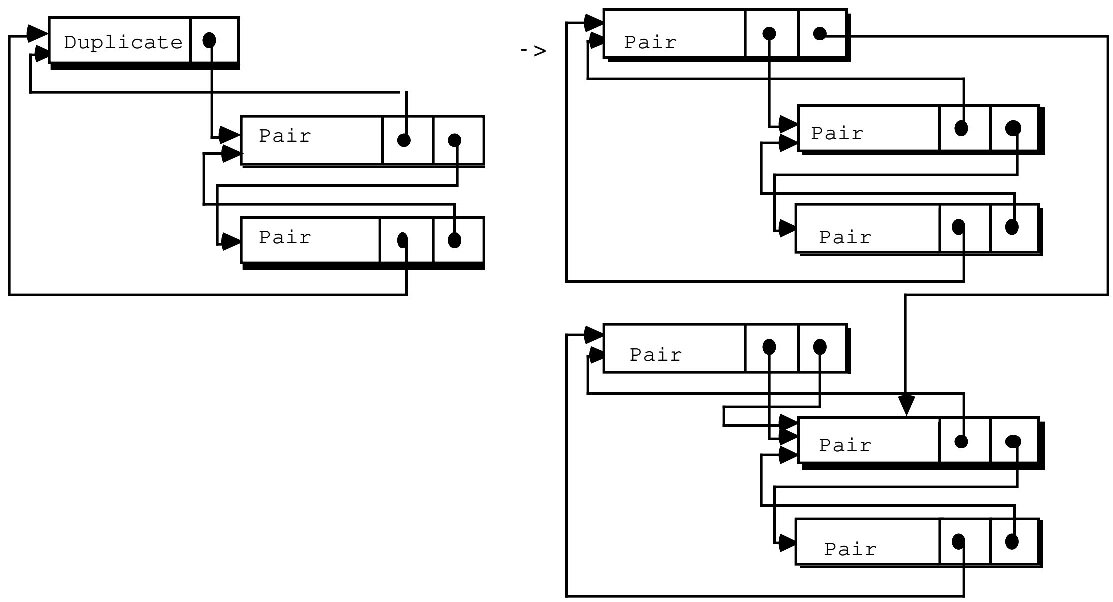


A more complicated example with the same rule:

the left graph reduces to the right graph

```
@1: Duplicate @2,          ->   @4 :  Pair @2 @12,
@2: Pair @1 @3,                 @2 :  Pair @4 @3,
@3: Pair @1 @2;                 @3 :  Pair @4 @2,
                                @12:  Pair @14 @13,
                                @13:  Pair @14 @12,
                                @14:  Pair @12 @12;
```

Note that the copying takes place after the root of the ordinary redex is redirected to the root to the contractum. The reduction is also 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.

7.3.3 WHY LAZY COPYING?

Take a graph containing redexes. The extension of explicit copying to graph rewriting introduces the possibility to copy this graph including all its redexes. There also is the possibility of sharing the graph. Unfortunately, there is nothing in between.

However, 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, also it can be useful to break up the sharing. Take for example a function application that delivers a large structure after relatively few reduction steps. When several parts of the program only need certain parts of this structure, then in terms of memory management it can be more efficient to copy the function application instead of sharing the large structure all the time.

Copying with the choice of maintaining or breaking up the sharing is called *lazy copying*.

Apart from the benefits of lazy copying in a sequential environment, lazy copying will serve as the basis for the communication of processes in a parallel environment.

7.3.4 LAZY COPYING

A function application on which copying will be stopped temporarily, is called a *deferred function application*. To denote a deferred function application the corresponding node is attributed with a subscript `d` or `defer`. Such a node is called a *deferred node*. Because every node has an explicit symbol, it is syntactically convenient to attach the attribute of the node to the symbol of the 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 this will only happen when this copy is demanded. The actual copy of a deferred node will not be deferred.

Nodeid's of which the contents need to be known for matching, are according to the functional reduction strategy first reduced to root normal form. Then, before the nodeid is accessed, a copy will be made. So, read access via a copying deferred nodeid never occurs.

A lazy copying example:

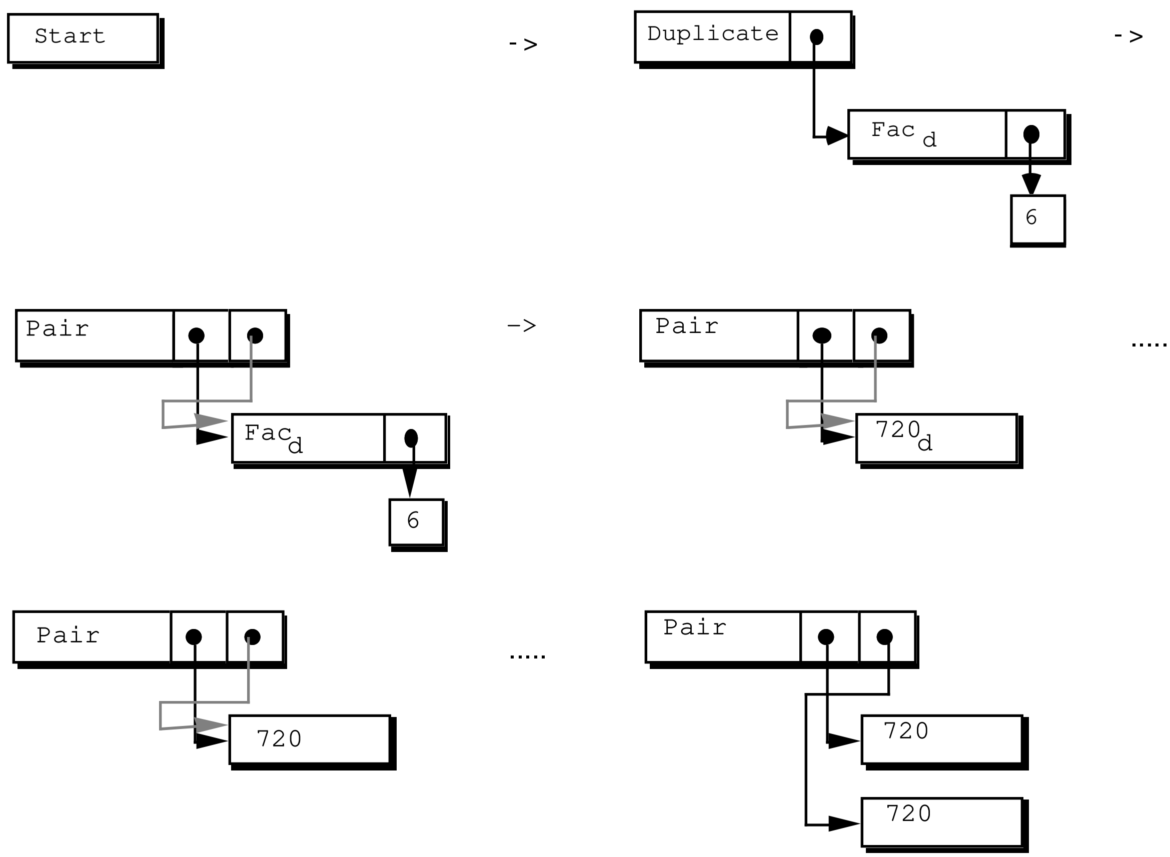
```
Start          -> Duplicate (Fac_d 6)          ;

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,       ->>  @4:  Pair @5 @5_c,     ..
@3:  Fac_d 6;                 @5:  720_d;

@4:  Pair @5 @5_c,       ->   @4:  Pair @5 @15,
@5:  720;                     @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:



The fact that copying is deferred will be an attribute of an applied occurrence of a nodeid in the graph, this attribute is always inherited when the nodeid is redirected to another nodeid. However, when there are multiple copies they have to be distinguished. Multiple copies are distinguished by numbering the attributes.

A lazy copying example with multiple copies:

```
Start          -> Duplicate (Fac_d 6)              ;

Duplicate x    -> Triple x x_c 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:  Triple @3 @3_c1 @3_c2,    ->>  @4:  Triple @5 @5_c1 @5_c2,    ..
@3:  Fac_d 6;                       @5:  720_d;
```

```
@4:  Triple @5 @5c1 @5c2,          ..  @4:  Triple @5 @15 @5c2,          ..
@5:  720;                              @5:  720,
                                       @15: 720;

@4:  Triple @5 @15 @25,
@5:  720,
@15: 720,
@25: 720;
```

Both copies are deferred. They are distinguished via numbers. Eventually, in the normal form two copies occur.

The copying deferred attribute number is not written down when there can be no confusion but actually it is always there. Every copying deferred attribute is actually a number. When a copy action has to be deferred, a new number is taken as the unique identification of this copy. So, two applied occurrences of the same nodeid can have the same attribute number (meaning they stand for the same copy of the nodeid), but it is impossible that two different nodeid's have the same attribute number (the same copy of two different things is contradictory).

As was already said, the attribute number is always inherited when a copying deferred nodeid is redirected to another nodeid. When a not copying deferred nodeid is redirected to a copying deferred nodeid, all replaced occurrences stand for one and the same copy all having the attribute number of the copying deferred nodeid. Also when a copying deferred nodeid is bound on the left-hand-side and used several times on the right-hand-side, all occurrences stand for the same copy.

During a copy action all occurrences of attribute numbers are replaced by new attribute numbers because a new copy of these nodeid's is demanded. Of course, the attribute number which stands for the copy action that is being done, is not replaced. Furthermore, when an attribute number is copied several times, all copies are replaced by the same new attribute number.

At first sight the fact that copying deferred attributes are numbered and that nodeid's with the same attribute number must yield a pointer to the same copy, might seem hard to implement. In the following paragraph the operational semantics of lazy copying is given. It also shows that lazy copying is in fact very easy to implement via special indirection nodes. The actual nodeid's of these indirection nodes represent the attribute numbers of the copying deferred nodeid's.

7.3.5 Operational Semantics of Lazy Copying

In this section the operational semantics of eager and lazy copying is explained informally. The formal definition is given in Smetsers *et al.* (1988).

In order to explain the semantics introduce two special kind of indirection nodes are introduced: a D(eferred) node: this node 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 `copy` 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. The nodeid of the C-node represents the attribute number of the copying deferred nodeid. After the copying has been performed this way, this rewrite is finished and reduction continues as usual. The *internal* reduction rules of the D and C-nodes are the following:

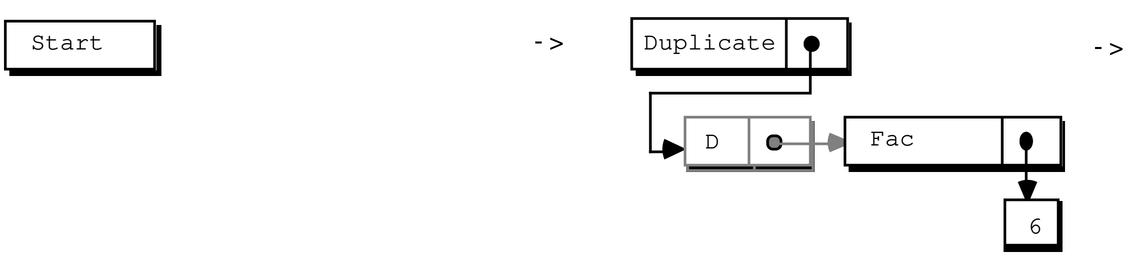
```
D {!} x       -> x                                  ;
C {!} x       -> xC                                 ;
```

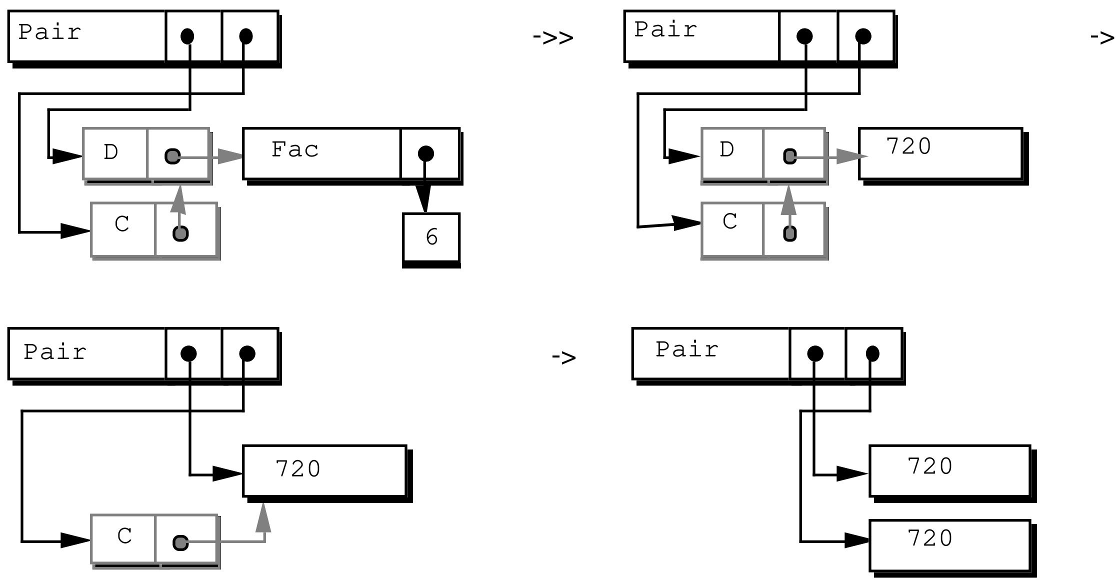
Note that in this way the property that a function is "deferred" or "not yet copied" is inherited by all function results until finally a root normal form is reached. Hereafter the reducer is able to apply the special rewrite rules for D and C which will make these nodes disappear. Sharing a C-node represents having the same attribute number and so it will lead to sharing of the same copy after this indirection node is vanished. The mechanism of sharing and redirecting of indirection nodes implements the attribute number administration.

If the previous example is considered again, it should be more clear what the semantics are:

```
@1:  Start;            ->        @2:  Duplicate @i,      ->
                                 @i:  D @3,
                                 @3:  Fac 6;

@4:  Pair @i @j,       ->>       @4:  Pair @i @j,        ->
@i:  D @3,                       @i:  D @5,
@j:  C @i,                       @j:  C @i,
@3:  Fac 6;                      @5:  720;

@4:  Pair @5 @j,       ->        @4:  Pair @5 @15,
@j:  C @5,                       @5:  720,
@5:  720;                        @15: 720;
```

which 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.

7.3.6 DISCUSSION

Lazy copying can be used in two extreme ways: the case that all nodes are deferred and the case that none of the nodes is deferred. This first form of copying will be called *fully lazy copying*. In the second form always an equivalent copy of the original graph is made immediately (eager copying).

An interesting aspect of lazy copying is that normal forms do not contain defer or copying deferred attributes. In a normal form every subgraph 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.

Normal forms:

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:` $\texttt{Pair}_{\texttt{d}}$ `1 x ;`	`@1: Pair 1 @1;`	‖ a cycle.
`Start -> x: Pair 1` $\texttt{x}_{\texttt{c}}$`;`	`@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 @11;	again: a once unravelled cycle.
Start -> x: Triple$_d$ 1 x$_c$ x$_c$;	@1: Triple 1 @11 @1$_{c1}$, @11:Triple 1 @11 @1$_{c2}$;	every copy that is done, 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 below.

With the following rules:			the normal form is:	without `copy` & `defer` the normal form is:
Start	->	r: A x$_c$, x: B y, y: I$_d$ z, z: C x;	@1 : A @12, @12: B @14, @14: C @22, @22: B @14;	@1: A @2, @2: B @4, @4: C @2;
I x	->	x;		

Note that the right-hand-side of the `Start` rule contains a cycle which can only be copied partly because the `I` node is deferred. The `BC`-cycle is once unravelled when it is copied, without the `copy` and `defer` indications the normal form would be just the cycle.

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.

With the following rules:			the normal form is:	without the `{!}` the normal form is:
Start	->	r: A (F x), x: B y, y: {!}I$_d$ z, z: C x;	@1 : A @13, @13: B @15, @15: C @13;	@1 : A @13, @13 : B @15, @15 : C @113, @113: B @15;
F x	->	x$_c$;		
I x	->	x;		

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 lazy copying is invariant under unravelling. 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 section 7.5 it will be shown that lazy copying can also be the basis for communication in a parallel environment.

7.4 EXTENDING FGRS'S WITH DYNAMIC PROCESS CREATION: P-FGRS'S

In this section graph rewriting will be extended with a way to create more reducers. Together with the extension of the previous section this completes the proposed extensions to graph rewriting.

As said before, in general there will be several redexes in the graph. One 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. As has been explained in section 7.2 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: `{!!}` or `{process}`, 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.

The `{process}` annotation influences the overall order of evaluation because a new reducer proceeds interleaved with the other reduction processes. In this paper all reducers are assumed to use the same strategy. More precisely, they all use the functional strategy parametrized by `{strict}` annotations (see also section 7.4.3).

If the `{!!}` annotation appears on the right-hand-side processes are created eagerly (see section 7.4.1), if the annotations appear on the left-hand-side processes are created lazy (see section 7.4.2).

7.4.1 EAGER PROCESS CREATION

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). This is called eager process creation because for every `{!!}` annotation on the right-hand-side a process is created before the next rewrite can be done.

Example of eager process creation:

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

The second operand of the integer addition will be calculated by a new reducer. The original reducer can proceed with the addition, calculating the first operand. Note that both reducers work on a subgraph sharing information at node `n`. Due to the recursive definition of `Fib` a whole tree of reducers will be dynamically created in this way.

7.4.2 LAZY PROCESS CREATION

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.

Example of `{!!}` on the left-hand-side:

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

This is equivalent to the transformed set of rules given below.

```
   Fib 0              ->   1                                          |
   Fib 1              ->   1                                          |
   Fib n              ->   ParPlusI (Fib (-I n 1)) (Fib (-I n 2))     ;
ParPlusI  n m         ->   ParPlusI' {!!}n {!!}m                      ;
ParPlusI' {!}n {!}m   ->   +I n m                                     ;
```

It should be clear that with some internal reductions now in general two new reducers are created for each of the operands of the integer addition. The extra `{!}` annotations are introduced to ensure that the reduction strategy will not reduce `ParPlusI'` before the processes on the arguments are finished.

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 transformations for `{!!}` on a left-hand-side are similar to the ones for `{!}`. They can be found in Smetsers *et al.* (1988).

7.4.3 DISCUSSION

The process annotation proposed in this section is very straightforward and quite similar to other proposals (see section 7.6). Note the analogy between the `{!}` and `{!!}` annotations. They both influence the reduction order. They both make the evaluation partially eager instead of lazy. The only aspect in which `{!!}` differs from `{!}`, is that a graph annotated with `{!!}` is reduced by a new reducer while the original reducer can proceed with its reduction scheme.

Another way of looking at `{!!}` annotations is that they influence the overall reduction order. In this view, `{!!}` annotations are parameters of the global reduction strategy (just as `{!}`). The global 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. A reference interpreter for PC-FGRS's which is being developed at the University of

Nijmegen adopts this view. There is no essential difference, but in the context of this paper, the process view with deterministic reducers is preferred. This will simplify the reasoning about locally weakening in the semantical restriction of interleaving to parallelism.

Note that a deadlock of processes arises when processes are demanding each others results on a cycle in the graph.

An example where deadlock may arise:

```
Start          ->   x: {!!} A y,
                    y: {!!} B x        ;

A C            ->   C                  ;

B C            ->   C                  ;
```

Note that actually the graph is in normal form.

However if we analyze the rule system that is specified above, it is not surprising at all that the evaluation may end in a deadlock sitution, namely, the rules define a process structure wherein two processes are waiting for each others results: a classical deadlock. The possible occurence of deadlock is inherent to systems in which one can describe arbitrary process communication. So, just as one has to be careful to avoid non-termination due to the {!} annotations, one has to be careful to avoid deadlock due to the {!!} annotations.

7.5 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 may cause a disaster. 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. One can imagine that, in general, the actual effect of parallel rewrites will highly depend on the kind of implementation.

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 actually can be performed in parallel because they cannot interfere with each other and therefore may be *considered* as being indivisible. 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. Assumptions have to be made on the kind of machine architecture the reduction is performed on. As argued in the introduction, we consider loosely coupled parallel machine architectures (each processor has its private memory) as the most interesting class of architectures. Therefore the possibility to model rewriting on this kind of architectures is treated in detail in the next sections. The suitability to use the system to model rewriting on other architectures is briefly discussed in section 7.5.3.

7.5.1 MODELLING REWRITING ON LOOSELY COUPLED PARALLEL ARCHITECTURES

A loosely coupled parallel computer is defined as a multiprocessor system which consists of a number of self-contained computers, i.e. processors with their private memory, which are connected 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 automatically be performed. 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 reduced on different processors.

In order 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. The methods differ in expressive power with respect to process creation and communication. They range from divide-and-conquer process behaviour to remote lazy process creation.

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.

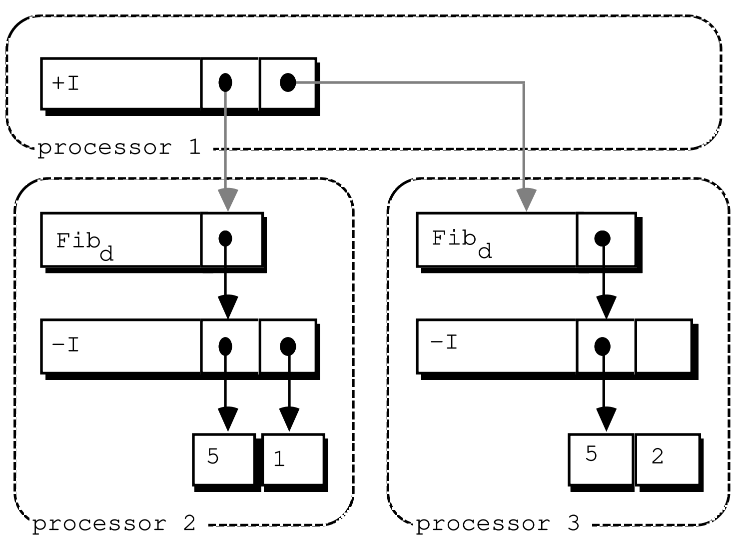
A self-contained subgraph will be regarded as a *virtual processor* because it has the property that it may be reduced on another processor. A reducer is also called a *process.*

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:

```
Fib 0    ->   1
Fib 1    ->   1
Fib n    ->   +I left_c right_c,
              left:  {!!}Fib_d (-I n_c 1),
              right: {!!}Fib_d (-I n_c 2)                 ;
```

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 is already started with copying the result but this is immediately deferred. The copying of the result 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`:



An alternative hierarchical process structure is obtained if the father reducer would reduce the left argument by itself. This woould have been easily achieved by leaving out all annotations and indications in the definition and application of `left` and by replacement of `{!!}` by `{!}`.

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. However, this method of modelling can only be used for this kind of algorithms.

Modelling loosely coupled evaluation

A method which makes it possible to model process behaviours which are more general than divide-and-conquer, must provide a way to define arbitrary connections between processes and processors. So, self-contained subgraphs as used in the previous section are in general too limited.

The lazy copy scheme introduced in section 7.3 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 nodeid's, 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 nodeid's are natural candidates for serving as interconnections between parallel executing processes because they induce further copying when they are accessed. Therefore, communication between parallel processes will be realized via copying deferred nodeid's. In this context copying deferred nodeid's are also called *communication channels* or just *channels*.

A subgraph is *loosely connected* if channels (copying deferred nodeid's) are the only connections between the subgraph and the rest of the graph. Note that this implies that a self-contained subgraph is loosely connected if its root is a channel. A loosely connected subgraph is called a *virtual processor* because it has the property that it may be reduced on another processor. 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.

Note that the definition of virtual processor in this section differs from the definition which was given in the previous section. In all following sections the new (more general) definition will be used.

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. 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 channel.

The semantics of copying deferred nodeid's 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 nodeid's, they can be passed as parameters or copied. However, when the subgraph to which the channel refers to is needed, the process will be suspended until the result is calculated by a 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 subreduction 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 is the non-interference condition 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 the following conditions are met. It must be guaranteed 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 lazy-copied 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 prove 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 loosely connected subgraphs. They can only access subgraphs on other processors via copying deferred nodeid's (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.

Unfortunately, it is in general undecidable whether virtual processors are also loosely coupled virtual processors. In general one cannot prove, when a parallel reducer is demanding information from a channel, that this information will either be in root normal form or that there will be a process running on it.

A method to create loosely coupled virtual processors

The obvious method which guarantees that virtual processors are loosely coupled, is by creating 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. Therefore using this method the non-interference condition is guaranteed to be true.

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 processor 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. For each occurrence a simple program transformation is made. More precisely,

Each occurrence of:	will be substituted by:
$\texttt{n : \{e\} Sym a}_1 \texttt{ .. a}_n$	$\texttt{n : I x}_c\texttt{,}$ $\texttt{x : \{!!\} I}_d \texttt{ y}_c\texttt{,}$ $\texttt{y : Sym a}_1 \texttt{ .. a}_n$

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 $\texttt{Sym a}_1 \texttt{ .. a}_n$. 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`.

$\texttt{n : \{i\} Sym a}_1 \texttt{ .. a}_n$	$\texttt{n : \{!!\} I}_d \texttt{ x,}$ $\texttt{x : Sym a}_1 \texttt{ .. a}_n$

A reducer is created on a deferred node. All sharing is maintained.

The nodeid's `x` and `y` in the substitution rules stand for nodeid's not used elsewhere in the rewrite rule.

`I` is just the identity function: `I x -> x;`

The indirection nodes are created to see to it that the copies are made correctly. In the following they are considered to be internal nodes.

When an {e} or {i} abbreviation is put on a nodeid, this is equivalent with putting it on the node the nodeid belongs to.

Examples

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 parallel (sub)reduction can be created and distributed over a number of virtual processors. 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 model a cycle of virtual processors. An example of this is given in one of the example programs of section 7.5.2. 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).

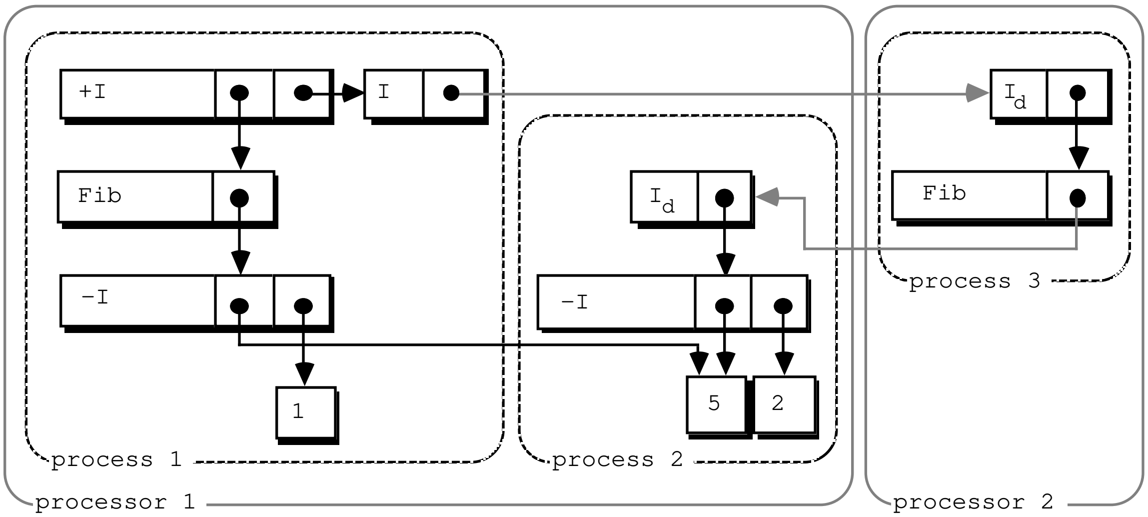
The `Fib` example using a non-hierarchical process structure (which is very unconventional for `Fib`): the second call of `Fib` will be executed on another virtual processor but the argument of that call is reduced internally on the virtual processor that also does the first call of `Fib`.

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

which 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 send 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.

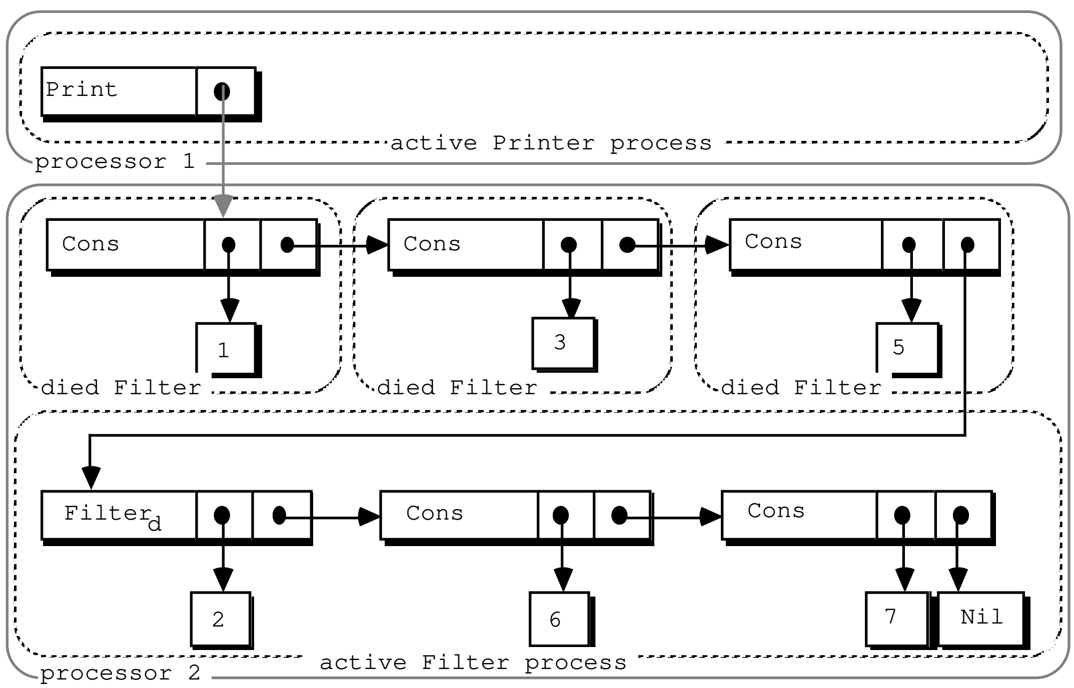
A virtual processor may contain several such families each producing a stream via a chain of channels. In the case of the following filter example the virtual processor contains exactly one such process: `Filter`. It sends a stream via the channel to the process `Print`.

The following example describes an asynchronous communication behaviour with streams:

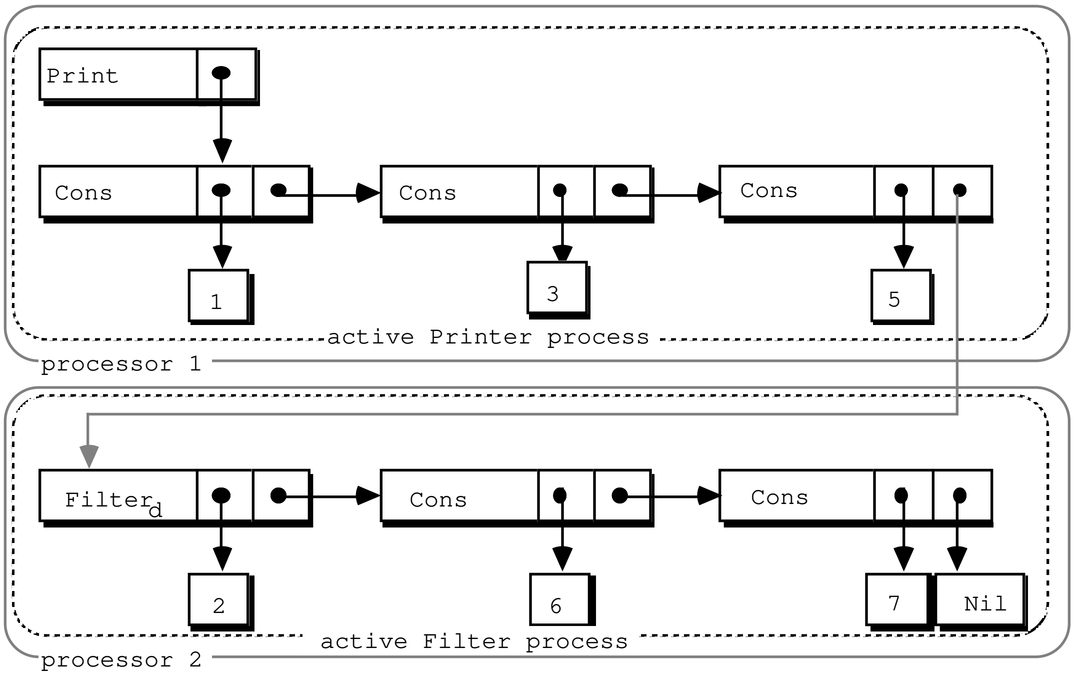
```
Start list                 ->   Print s,
                                s: {e} Filter list 2    ;


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

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

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 is the list containing 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.

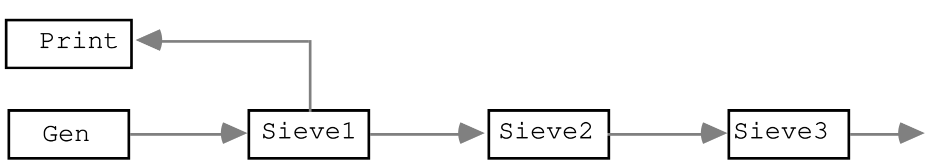


The sieve of Eratosthenes

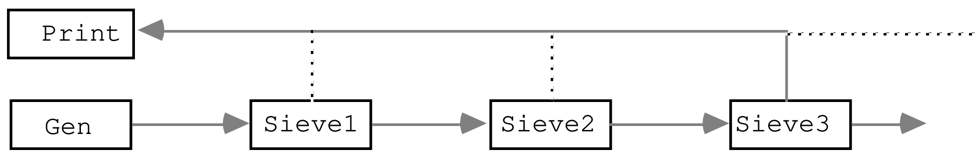
The sieve of Eratosthenes is a classical example which generates all prime numbers. A pipeline of virtual processors is created. On each processor a `Sieve` process (a family of processes actually) is running. Those Sieves hold the prime numbers in ascending order, one in each `Sieve`. Each `Sieve` accepts 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.

This can be represented in a picture as below (all arrows indicate flow of data on channels):



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:



The Sieve:

```
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`.

Extending the method with lazy process creation

With the `{i}` and `{e}` abbreviations processes and processors are created eagerly. However, sometimes it is necessary that processes and processors are created lazy. Since on a real architecture resources are limited, eager process and processor creation may cause serious problems in practise. . Furthermore, eager evaluation may cause infinite computations. With the following abbreviations we extend our method with $\{i_l\}$ lazy process and $\{e_l\}$ lazy processor creation.

Each occurrence of:	will be substituted by:
$n : \{e_l\}\ Sym\ a_1\ ..\ a_n$	$n : I\ x_c,$ $x : LI_d\ y_c,$ $y : Sym\ a_1\ ..\ a_n$
$n : \{i_l\}\ Sym\ a_1\ ..\ a_n$	$n : LI_d\ x,$ $x : Sym\ a_1\ ..\ a_n$

`I` is just the identity function: `I x -> x;`

LI is an identity function on which a process is created when its evaluation is demanded:

```
{!!} LI x -> x;
```

The nodeid's x and y in the substitution rules stand for nodeid's not used elsewhere in the rewrite rule. Just as with eager virtual processor creation, the indirection nodes are created to see to it that the copies are made correctly. In the following they are considered to be internal nodes.

Note that this extended method will create a channel on which a process is created automatically when its result is demanded. So, the non-interference condition which allows the weakening the restriction of interleaving, is fulfilled trivially.

Compare the next example with the previous filter example. The filtering now only takes place on demand of the printing process. Only when the printer wants a new filtered value, the filter calculates the next value.

```
Start list                ->   Print s,
                               s: {e1} Filter list 2        ;


Filter Nil          n     ->   Nil                          |
Filter (Cons f r)   n     ->   IF (=I (MOD f n) 0)
                                  (Filter r n)
                                  (NewFilter f r n)         ;

NewFilter f r n           ->   Cons f rest,
                               rest: {i1} Filter r n        ;
```

7.5.2 CHALLENGES FOR PARALLEL GRAPH REWRITING

Finally, the descriptive power of the proposed methods for loosely coupled evaluation is illustrated by applying them to meet two challenges which have been set for parallel graph rewriting. The classical *Bounded Buffer algorithm* and *Warshall's shortest path algorithm.* The second algorithm has also been used in a large ESPRIT project on parallel architectures and languages (Augusteijn (1985a), Augusteijn (1985b)) to test the expressiveness of parallel languages.

Bounded Buffer

The first challenge to consider is the classical bounded buffer algorithm which is a test in expressing a certain memory and synchronization behaviour. In terms of its result as a function it is equivalent to the identity function.

The wanted memory and synchronization behaviour is the following. Other processes are trying to put elements in the buffer or to take elements out the buffer. At any time not more than `size` elements are in the buffer. When the buffer is full, no element can be put in. When the buffer is empty, no element can be taken out. When the buffer is not full or not empty, elements must be allowed to be put in or taken out.

The memory and synchronization behaviour will be modelled via a family of processes. Together they are executed externally. Putting an element in the buffer is modelled via the list which is an argument of the function `Buffer`. When a reducer has rewritten `GetNextEl` or `Buffer` with the element to put in the buffer at the head of the list, then the element is in the buffer. Taking an element out of the buffer is modelled via the copying of the element of the result list to the `Consume` process.

```
Start                       ->   Consume x,
                                 x: {e} Buffer BufferSize y,
                                 y: {e} Produce                  ;

Buffer size (Cons hd tl)    ->   Cons hd rb,
                                 rb: {i1} Buffer size rc,
                                 rc: {i}  GetNextEl size tl      ;

GetNextEl 0 list            ->   list                            |
GetNextEl n (Cons hd tl)    ->   Cons hd rc,
                                 rc: {i} GetNextEl (--I n) tl    ;

Consume  x                  ->   Print x                         ;

Produce                     ->   Gen 2                           ;

BufferSize                  ->   100                             ;
```

This is the complete program. When an element is taken out of the buffer, a new `Buffer` is lazy created. By rippling through the buffer this new `Buffer` reducer will start a new `GetNextEl` reducer to input an element when `size-1` other elements have been put in.

A parallel version of Warshall's shortest path algorithm

The second challenge that is considered, is a parallel version of Warshall's solution for *the shortest path problem*:

> Given a graph G consisting of N nodes and directed edges with a distance associated with each edge. The graph can be represented by an N x N matrix in which the element at the i-th row and in the j-th column is equal to the distance from node i to node j. Warshall's shortest path algorithm is able to find the shortest path within this graph between any two nodes.

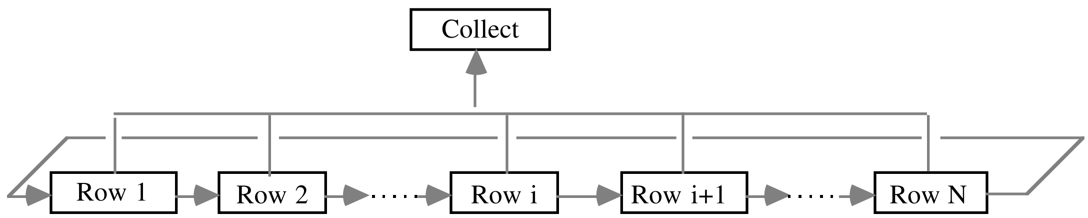
Warshall's shortest path algorithm:

> A path from node j to node k is said to contain a node i if it can be split in two paths, one from node j to node i and one from node i to node k (i≠j & i≠k). Let SP(j,k,i) denote the length of the shortest path from node j to node k that contains only nodes less than or equal to i (0≤i & 1≤j,k & i,j,k≤N).
> So
>
> | | | |
> |---|---|---|
> | SP (j,k,0) | = 0 | if j=k |
> | | = d | if there is an edge from j to k with distance d |
> | | = ∞ | otherwise |
> | SP (j,k,i) | = minimum (SP (j,k,i-1), SP (j,i,i-1) + SP (i,k,i-1)) | |
>
> Define a matrix M as follows: M[j,k] = SP (j,k,i) for some i. The final shortest path matrix can be computed iteratively by varying i from 0 to N using the equations as described above.

Observing the algorithm it can be concluded that during the i-th iteration the updating of the rows of the matrix can be performed in parallel. Therefore a separate process is introduced for each row of the matrix that updates its row during each iteration step. In the i-th iteration all the processes need to have access to row i as well as to their own row. This can be achieved by letting process i distribute its own row as soon as the i-th iteration starts. At first sight it seems to be difficult to express this updating and iterating in a parallel functional language. It will be shown how it can be expressed using the proposed method for loosely coupled evaluation.

As in previous examples, all processes are actually families of processes.

Representing the process structure in a picture gives (all arrows indicate flow of data on channels):



Initially, all the row processes `Row i` are created and the initial matrix is distributed to these processes. Before `Row i` performs its i-th iteration it distributes its own row to the other processes. This is done in a pipeline, i.e. `Row i` sends its own row to `Row j` via `Row i+1`,..., `Row j-1` and `Row j` (counting modulo N from i to j). Process `Collect` asks all the row processes for sending their final result in the same way as the Print process asked for all prime numbers in the `Sieve` example of section 7.5.1 (in the picture all channels are drawn which at different moments serve this purpose).

All processes are to be created eagerly, so the proposed method will be used without the extensions for lazy process creation. Then, the fairly complicated process graph can be specified directly in the following way:

```
Start                          ->  Collect out,
                                   out: {e} Create                    ;

Create                         ->  out: Row 1 Initmat (Second out)    ;

Row k (Cons row_n Nil) left    ->  Tup (Cons chan1 Nil) chan2,
                                   chan1: {i} Finalrow chan2,
                                   chan2: {i} Iterate k 1 row_n left  |
Row k (Cons row_k restmat) left -> Tup (Cons chan1 {!}(First next))
                                       {!}(Second next),
                                   chan1: {i} Finalrow chan2,
                                   chan2: {i} Iterate k 1 row_k left,
                                   next:  {e} Row {!}(++I k) restmat
                                                             chan2;
```

`Create` becomes the first `Row` process which has a reference to itself in order to make it possible to expand it to a cyclic row of processes. Each `Row` has two internal reducers of which one process will communicate with the other `Rows` during the iteration. The other selects the final result and communicates it to `Collect`. Each newly created `Row` process should be connected to the preceding process by a channel. In order to create a channel from the process `Row i` to the process `Row i+1`, a reference to the first one should be given to the second one. This reference is passed via the parameter `left` of the function `Row`. The result of `Row` is a tuple `(finalmat, rowprocN)` in which finalmat is a list of channels to the `Row` processes. These can be used by `Collect` to retrieve the final result matrix. The second part of the result of `Create`, called rowprocN, is a reference to process `Row N` which should be given to the first one in order to make the cycle of `Row` processes complete.

In this context, the iteration process is very simple:

```
Iterate k i row_k left          ->
        IF (>I i Size)
           (Cons row_k Nil)
           (IF (=I k i)
               (Cons row_k nextit)
               (Cons row_i update)
           ),
        nextit: {i} Iterate k (++I i) row_k (Tail left),
        update: {i} Iterate k (++I i) (Updaterow row_k row_i dist_k_to_i)
                                      (Tail left),
        row_i:        Head left,
        dist_k_to_i: Get row_k i                                        ;
```

Every iteration starts the next iteration as a new incarnation of itself via a new internal process. Before an iteration starts, `Iterate` should output the i-th row. The i-th row is either the row of `Iterate` itself or the row belonging to the left `Row` process. At the end `Iterate` outputs its own row. The rest of the program is straightforward.

7.5.3 MODELLING REWRITING ON OTHER ARCHITECTURES

In this section the possibility to use PC-FGRS's to model rewriting on other kind of architectures is briefly discussed.

Multiprocessing on a single processor

Consider a single processor. Such a processor can be regarded as a special case of a multiprocessor architecture: there is only one processor. Therefore, a PC-FGRS is also suited to model rewriting on such an architecture. Although no real parallelism is possible on a single processor, the possibility to have "multireducing" is important. The classical example is of course an operating system. On a single processor the context switch between reducers can be controlled without problems such that it can be guaranteed that each rewrite step is indeed indivisible in practise.

Parallel architecture with global memory

Consider a multiprocessor architecture with a global memory and a reducer running on every processor. If each rewrite step really would be indivisible there would be no parallelism left. If the indivisibility is not guaranteed one has to be very careful to assure that the modification of the graph by one reducer does not interfere with the rewrite action of another reducer.

One can regard this kind of architecture as a special case of a loosely coupled architecture. The methods introduced for loosely coupled architectures introduced in the previous section, can also be used for architectures with global memories.

If one would like to have maximum benefit of the global memory, the level of PC-FGRS's is too high. An other method is to make detailed assumptions on the way reducers are actually implemented. For instance, one can make an abstraction of the kind of machine code that presumably will be generated (take for example G-machine code (Johnson (1984)) or ABC-machine code (Plasmeijer & van Eekelen (198-)). With this knowledge one can invent some clever locking scheme which assures that, preferably without loosing too much parallelism, wrong results cannot be produced. Another possibility is to search for certain classes of PC-FGRS's for which it can be proven that the reducers can run in parallel without additional locking (Kennaway (1988a)).

Research aimed at identifying situations in which in global memory architectures copying can be an efficient alternative for locking, might be very worthwhile.

Systolic arrays

It is possible to model synchronous communication which occurs in parallel architectures like systolic arrays. In such architectures the processors are synchronized and must communicate at exactly the same moment. This involves communication with a central clocking device. By specifying the clocking device explicitly as a separate process in the system also systolic synchronous communication can be modelled.

7.5.4 DISCUSSION

It is clear that with the proposed abbreviations parallel programming is much easier than without them. They clearly represent the process structure. 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 7.3.6 it was showed that the normal form in a C-FGRS depends on the order of evaluation. In section 7.4.3 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. A typical example is given below.

With the following rules:

```
Start    ->   r: A (F x),
              x: B y,
              y: {!!}Id z,
              z: C x;

F x      ->   xc;

I x      ->   x;
```

the normal form can be:
(doing `I` before `F`)

```
@1:  A @13,
@13: B @15,
@15: C @13;
```

but also it can be:
(doing `F` before `I`)

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

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. Then, always the same term will be yielded.

Although the proposed abbreviations are very promising, perhaps for certain problems other combinations of lazy copying and process creation can be found which for such a particular case guarantee non-interference.

7.6 GENERAL DISCUSSION

Related work

The idea to use annotations (Burton (1987), Glauert *et al.* (1987c), Goguen *et al.* (1986), Hudak & Smith (1986)) or special functions (Kluge (1983), Vree & 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 & Smith (1986), Vree & Hartel (1988)) while others are introduced on the level of the computational model (Glauert *et al.* (1987c), 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 & Smith (1986), Goguen *et al.* (1986), Kluge (1983), Vree & 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.* (1987c)), also based on Graph Rewriting Systems (Barendregt *et al.* (1987b)) 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 & 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 the model, primarily because virtual processors can be freely created on the level of the computational model. Hence, another processor is just created (for instance using `{e}` and `{e1})` when it is needed. However, not only for practical reasons, but also in order to reason about issues like load balancing, we will investigate the specification of load distribution in the future.

Implementation aspects

We already know that efficient implementation of FGRS's is possible on sequential hardware (Brus *et al.* (1987)). Type information (Plasmeijer & van Eekelen (198-)) and strictness analyzers (Nöcker (1988)) play an important role.

An efficient implementation of multiprocessing (interleaved execution) is possible by indicating fixed places at which a process switch may occur. Examples of such places are termination of reduction, suspension of reduction and creation of an intermediate result. Compared with a pure sequential implementation a multiprocessing version will loose a bit of efficiency due to these context switches. Furthermore, each reducer has to check whether or not another reducer is working on its redex.

PC-FGRS's are very suited for implementation on loosely coupled parallel architectures. Most problems which have to be solved of a more 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.

Future work

At the moment we are developing a reference simulator for PC-FGRS's. The ideas introduced in this paper will be incorporated in the language Concurrent Clean. 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 or a DOOM machine (Odijk (1987))). 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.

The combination with other strategies than the functional strategy may be interesting (van Eekelen & Plasmeijer (1986)). For instance, adding reducers following a non-deterministic strategy may be useful for the specification of process control, including scheduling and interrupts.

7.7 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. Multiprocessing 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.

7.8 ACKNOWLEDGEMENTS

We thank Ronan Sleep of the University of East-Anglia for patiently lending us his ear and in particular for his advice to isolate the lazy copying from the process creation. We thank Pieter Koopman of the University of Nijmegen for proof reading and correcting this paper while it was written.