2

GRAPH REWRITING SYSTEMS:
A PROMISING COMPUTATIONAL MODEL

Graph rewriting systems are the key computational model throughout this thesis. So, in this chapter we will first motivate why graph rewriting is so well suited for serving as a model of computation for implementations of functional languages. Then a short introduction is given to generalized graph rewriting as it is defined in the chapters 4 and 5 of this thesis including its extensions which are defined in chapter 7 of this thesis.

An overview of the main results which are presented in this thesis, is given in the rest of this chapter beginning with section 2.3. Motivations are given for the choices which were made. Relations between various topics of this thesis are discussed.

2.1 WHY GRAPH REWRITING?

The reason for choosing graph rewriting as the model of computation for implementations of functional languages is that graph rewriting is built around two basic concepts: pattern matching and sharing. The advantages of these concepts are discussed below.

PATTERN MATCHING

Pattern matching is the basic operation for Term Rewriting Systems (TRS's). A tutorial on TRS's is given in Klop (1987).

TRS's are more general than λ-calculus and combinatory logic because TRS's allow non-Church-Rosser computations to be specified. In TRS terms combinators are very simple rewrite systems without recursion and with patterns with a simple specific structure only. TRS's do not have free variables.

Combinatory logic defined as a TRS:

```
Apply (Apply (Apply S x) y) z    → Apply (Apply x z) (Apply y z)
Apply (Apply K x) y              → x
```

Sometimes a special notation is used not writing down the binary Apply operators:

```
S x y z  → x z (y z)
K x y    → x
```

In TRS's also ambiguous, non-confluent computations can be specified. However, *Regular* TRS's i.e. TRS's with no overlaps between the rules and without multiple occurrences of variables on the left-hand-side, are confluent. So, this class of TRS's is suited for modelling functional languages.

Parallel outermost reduction is a normalising strategy for regular TRS's. Because strategies influence the reduction order and hence the run-time behaviour of a program, it is necessary to be able to reason about them. Strategies are very often only informally defined which makes this reasoning almost impossible. Formal specification and comparison of reduction strategies is not only important for reasoning about these strategies but it is also important for implementing them. In chapter 3 of this thesis some specification methods are compared and a new high level specification method is introduced which simplifies the reasoning about reduction strategies.

TRS's are very close to functional languages because the basic concept is pattern matching. Patterns contain important information for strictness analyzers (Nöcker (1988)). Strictness analysis tries to find opportunities to deviate from the standard reduction strategy in order to achieve a more efficient implementation (e.g. many times eager evaluation can be more efficient than lazy evaluation). The following example illustrates the importance of the use of patterns for strictness analysis.

Patterns may contain important information for strictness analyzers:

```
F x           -> G (Cons x Nil)  ;
G (Cons a b)  -> a               |
G x           -> Nil             ;
```

Using the pattern-match information of `G` a very simple analysis shows that `F` is strict in its first argument.

With combinators or with λ-calculus an analysis which effectively uses the pattern-match information would be very cumbersome. Therefore, we consider TRS's to be better suited than λ-calculus to serve as a computational model for functional languages and their implementations.

TRS's are very close to functional languages and they even broaden the scope of investigation to non-confluent computations. Unfortunately, TRS's lack the explicit notion of sharing of computations which, in our opinion, is an essential notion in the implementation of functional languages.

SHARING

Sharing of computations is essential to obtain efficient implementations on traditional hardware, although it might be a problem in a parallel environment. Machine architectures generally have a memory which has addresses and contents; instructions change the contents of the memory. The memory can be seen as a not connected graph and the instructions can be seen as rewrite rules on the graph. Furthermore, every program which contains non-trivial data structures deals with a certain notion of graph rewriting. So, we feel that it is important to investigate general graph rewriting.

Referential transparency obviously gives many opportunities for sharing computations. So, in the context of functional programming various implementation methods with different ways of sharing are investigated (Wadsworth (1971), Turner (1979a), Johnson (1984) and many others). So if we take sharing seriously, it has to be incorporated in the model of computation which is the basis for an intermediate language between functional language and machine architecture. Then the model of computation will be close to the implementation and the issues concerning sharing can be discussed and investigated within the model.

Graph rewriting systems extend TRS's with a general notion of sharing. We believe that compared to the λ-calculus, combinatory logic and term rewriting systems, graph rewriting systems are best suited to serve as computational model for functional languages. Therefore, graph rewriting systems were chosen to be the basic model of computation in the Dutch Parallel Reduction Machine Project (Barendregt *et al.* (1987c)) and in the U.K. Flagship project (Glauert *et al.* (1987c)).

It has been shown that with intermediate languages based on this model of graph rewriting (Clean, Concurrent Clean, Dactl1) efficient implementations of functional languages are possible on various sequential and parallel machine architectures (Brus *et al.* (1987), van Eekelen *et al.* (1988), Kennaway (1988b), an overview is given in Glauert *et al.* (1988)).

2.2 INTRODUCTION TO GENERALIZED GRAPH REWRITING

In this section generalized graph rewriting as defined in chapters 4 and 5 of this thesis is introduced informally. Also the extensions which are introduced in chapter 7, are briefly explained. A more rigorous treatment can be found in those chapters.

TERMINOLOGY

In graph rewriting systems a program is represented by an initial graph and 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, one of which is distinguished as being the *root* of the graph. Each node has a defining node-identifier (the *nodeid*). A *node* consists of (*contains*) 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 generally realized by overwriting the node, possibly by an indirection node).

In abstract reduction systems anything that can be rewritten is a redex. In graph rewriting (and in other models with structured terms such as λ-calculus, combinatory logic and TRS's) mostly the term *redex* is reserved for that part of the structure that actually matches a rule. A graph is then said to be in *normal form* if it contains no redex. Furthermore, a graph is in *root normal form* when the root of a graph is not the root of a redex and it is sure that it can never become the root of a redex. Note that whether a graph has such a root normal form is in general undecidable. When a reducer terminates, its *result* is that part of the final graph which is accessible from the root. A 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.

A rule is *left-comparing* or *non left-linear* if a variable occurs more than once in the left-hand-side. A graph rewriting system is *left-comparing* or *non left-linear* if it contains a left-comparing rule.

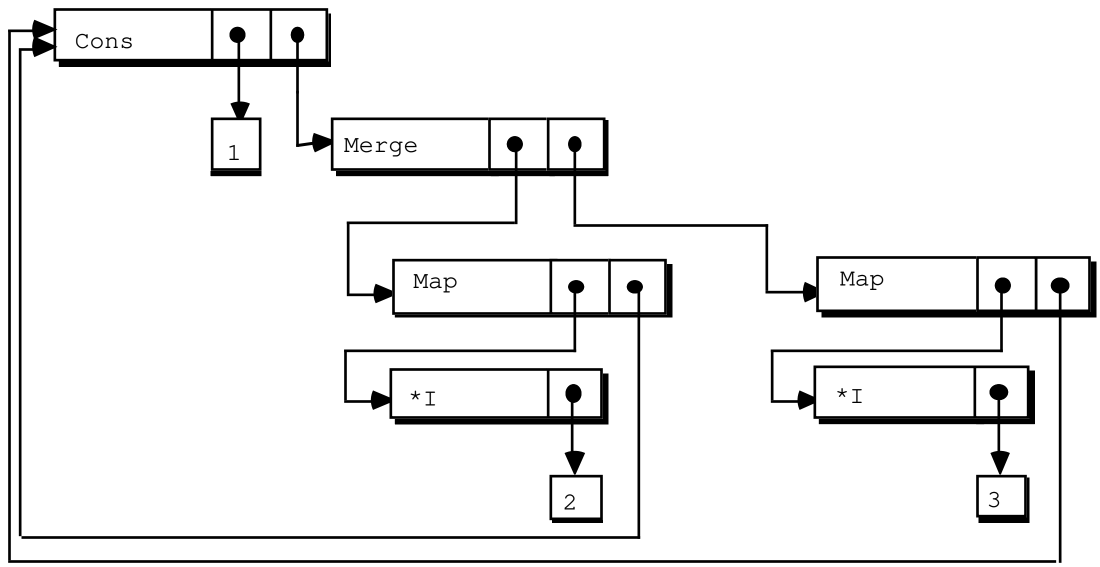
REWRITING

In standard graph theory, a graph in a general graph rewriting system (as defined in chapter 5) is a form of directed graph in which each node is labeled with a symbol, and its set of out-arcs is given an ordering. In general graph rewriting systems nodes are denoted by their names, i.e. their nodeid's. The denotation of a graph can be regarded as a tabulation of the contents function which gives the contents of every node of the graph.

Denotation of a cyclic graph which might occur during execution of example 6.3 in chapter 6:

```
@1: Cons @2 @3,
@2: 1,
@3: Merge @4 @5,
@4: Map @6 @1,
@5: Map @7 @1,
@6: *I @8,
@7: *I @9,
@8: 2,
@9: 3;
```

A pictorial view on this graph:



To get an idea of what general graph rewriting is we will discuss the main differences with term rewriting.

Of course, the objects are graphs so all kinds of sharing (including cyclic structures) can be expressed. The variables in the graphs stand for nodeid's which among other things means that the natural meaning of left-comparing is not just a test for syntactical equivalence but a test on actually sharing the same structure. Furthermore, on the right-hand-side of a rule the new structure may contain nodeid variables which do not occur on the left-hand-side. For these variables new nodeid's must be invented while building the actual contractum.

Graphs need not be connected. When the computation is finished, the final result is the subgraph that is connected to the root. This gives rise to the elimination of nodeid's that are not connected to the root anymore (*garbage collection*). These nodeid's can be reused instead of inventing new nodeid's. Unconnected patterns in the rules give rise to a kind of context dependent rewriting where a part which is connected to the root may be rewritten if a non-connected part of the graph contains some symbol for instance. For tracing purposes this can be very useful.

Rewriting is done via redirections. A redirection of the root of the redex to another nodeid more or less corresponds to rewriting in term rewriting systems. However, multiple redirections which are performed in parallel, and non-root redirections give rise to complex changes of the graph structure.

The actual nodeid's of a graph that is rewritten are also called *global* nodeid's. Explicit redirection of a global nodeid has as a consequence that all references to the original global nodeid have to be changed. So also references in the rewrite rules to global nodeid's have to be redirected. Hence global nodeid's can be viewed as *global* variables (they have a global scope), where nodeid variables are *local* variables (they have a meaning only within a single rule).

EXTENDING THE REWRITE SEMANTICS

In chapter 7 of this thesis the following extensions are discussed and used extensively. In order to make it possible to discuss the relations between the chapters the extensions are briefly explained here.

Influencing the order of evaluation

We can allow graph rewriting systems to be annotated in order to influence the sequential order of evaluation. To every node and to every nodeid one or more attributes can be assigned via annotations. 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.

In this thesis only one sequential annotation is defined indicating that the reduction of the annotated argument of a symbol (function or constructor) is demanded. This annotation 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 such annotations may make the reduction strategy deviate from the default evaluation order which then becomes partially eager instead of lazy. When more than one such annotation occurs on a right-hand-side, they are effectuated depth-first from left to right.

These annotations play an important role because they are parameters of the reduction strategy. The 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 result of the computation or the termination of the reduction.

In reasoning about programs with these 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.

Explicit parallelism

As said before, in general there will be several redexes in a 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 by using annotations it is also possible to influence the order in which the redexes are reduced by a single reducer.

Loosely coupled machine architectures, such as Transputer racks, are available on a wide scale. But 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, 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.

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).

The process annotation indicates 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. If the process annotation appears on the right-hand-side processes are created eagerly, if the annotations appear on the left-hand-side processes are created lazy.

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. Although in implementations generally some kind of copying/sharing scheme is used, up to now it has never been incorporated in graph rewriting models. With an explicit mechanism for copying in the model the communication can be controlled on the level of the rewriting system itself.

In chapter 7 of this thesis 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 general the phrase '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. By introducing the possibility to use subtle combinations of sharing and copying this greatly improves the expressive power of graph rewriting systems. Furthermore, in chapter 7 it will be shown that lazy copying can also be the basis for communication in a parallel environment.

Classes of GRS's

A subclass of general GRS's in which the order of evaluation can be influenced via annotations will be prefixed with A. So the abbreviation for general graph rewriting systems with these annotations is *A-GRS*.

A subclass of general 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*. Because generally the sequential control annotations are also used when the parallel annotations are used, classes which use both annotations are also just prefixed with P.

A subclass of general 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*.

2.3 HIGH LEVEL SPECIFICATION WITH MULTI-LEVEL REWRITING SYSTEMS

With multi-level rewriting systems (defined in chapter 3) high level specifications of reduction strategies can be made. With this method specifications can be constructed which are relatively easy to prove. As is shown in chapter 3 transformations on these specifications can be elegantly performed in order to obtain certain properties.

In Goos & van Latum (1987) this method is applied successfully for various strategies. They have specified the strategies and they have proved several properties.

This method of specification can easily be extended to graph rewriting systems as defined in chapters 4 and 5. This is achieved by using GRS's where TRS's are used in chapter 3.

In Smetsers *et al.* (1988) a slightly adapted version of the method is being used to specify the operational semantics of Concurrent Clean (a language based on the extensions of graph rewriting systems which are given in chapter 7). Since this specification adopts the syntax of Clean (a functional language based on graph rewriting defined in chapter 6), such a specification also yields a directly executable (slow) interpreter for Concurrent Clean.

2.4 TERM GRAPH REWRITING

Term graph rewriting (defined in chapter 4) connects term rewriting systems in which no sharing can be expressed, with graph rewriting systems. *Term graph rewriting* means that a TRS is interpreted (lifted) as a GRS. The normal forms of the GRS which are graphs, are unravelled to terms in the TRS world. Via term graph rewriting it is proven that sharing terms is sound. Furthermore restrictions are given which ensure completeness of sharing implementations. Using C-GRS's (to denote sharing as well as copying; see section 2.2 for a brief introduction) and PC-GRS's (to denote copying with explicit parallelism); see sections 2.2 and 2.7) alternative ways of lifting TRS's can be investigated. This might lead to proving the correctness of parallel implementations of more general (not necessarily regular) TRS's. Term graph rewriting is a very promising topic for further research.

Different ways of lifting TRS's lead to the investigation of special classes of GRS's. These classes have the following restrictions of general graph rewriting systems in common:

- all graphs are connected and global nodeid's do not occur in the rules;
- 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 a nodeid indicated in the pattern;
- no left-comparing rules which implies that it is impossible to pattern match on equivalency of nodeid's (sharing). This is a difference with TRS's: the corresponding property of TRS's can not be lifted to GRS's.

Because of the last restriction which implies that left-hand-side are actually more like terms, the classes are called *Term ... Rewriting Systems* where ... captures the extra restrictions that are put on the right-hand-sides.

In a *Term Tree Rewriting System* (TTRS) right-hand-sides are dag's (directed acyclic graphs) and furthermore all variables on the right-hand-side have copy indications as defined in chapter 7. If TRS's are lifted by adding these copy indications, then with trees as initial graphs this class of GRS's exactly corresponds to term rewriting which is shown in chapter 4 with somewhat different terminology.

In a *Term Dag Rewriting System* (TDRS) right-hand-sides are dag's without further restrictions. With these systems it is shown in chapter 4 that sharing implementations of a fairly general class of term rewriting systems are sound and complete. Furthermore, it is shown that sib-normalising strategies (a large subclass of normalising strategies) can be lifted from TRS's to normalising strategies in TDRS's.

In a *Term Graph Rewriting System* (TGRS) right-hand-sides are graphs (possibly containing cycles). Special transformations during lifting of TRS's might make use of cycles. The intermediate language Clean which is introduced in chapter 6, uses TGRS's to implement functional languages.

Infinite data structures can be efficiently implemented with cyclic graphs. A typical example is the solution to the Hamming problem given in section 6.3.3.

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 because it means that when we model TRS's by term graph rewriting the results in the TRS world are not affected by lazy copying.

2.5 GENERALIZED GRAPH REWRITING: LEAN

Lean (defined in chapter 5) is an experimental language for specifying computations in terms of graph rewriting. It is designed to be a useful intermediate language for those language implementations which rely on graph rewriting. An interpreter for Lean is available (Jansen (1987)) which allows mixing of several reduction strategies. The design of Lean has heavily influenced the design of Dactl1 (Glauert (1987c)), which the UK Flagship machine (Watson & Watson (1987)) supports.

The graph rewriting model underlying Lean is of independent interest as a general model of computation for parallel architectures. This model of generalized graph rewriting has a very high expressive power because there are few restrictions on the graph that is transformed and the transformations that can be performed. This induces a trade-off: adding restrictions decreases expressivenes but it may yield important properties for reasoning or implementation (e.g. referential transparency or efficient fine-grain parallelism). So, an important line of research tries to identify restricted subclasses which are tuned for specific properties. This may yield as a spin-off programming languages based on subclasses with specific properties and advantages (see also sections 2.4 and 2.7).

Term dag rewriting systems are an example of an important subclass of generalized graph rewriting. They are used in chapter 4 to model term rewriting systems by term graph rewriting.

Lean and Dactl1 are the first two examples of spin-off languages. The reduction relation of the languages is identical. Lean is used for experiments with generalized graph rewriting. With features such as multiple redirection and global nodeid's also non-functional algorithms such as unification and tracing can be modelled. Dactl1 is tuned for fine grain parallelism by adding fine grain control markings and removing global nodeid's.

Clean which is described in the next section, is another example of such a spin-off language. The language is functional and efficiently implementable.

Concurrent Clean will be yet another spin-off language based on the extensions of the rewrite semantics defined in chapter 7. This language is now being defined (Smetsers *et al.* (1988)). It will be an extension of Clean which is very suited for coarse-grain parallel implementation on loosely coupled machine architectures.

2.6 FUNCTIONAL GRAPH REWRITING: CLEAN

The motivation for the design of Clean was to obtain an efficient implementation of functional languages on traditional sequential hardware. Instead of making a direct implementation of a specific language on a specific target machine it was decided to design an intermediate language based on an appropriate model of computation. This should make it possible to experiment with different kinds of program transformation schemes (SKI-combinators, supercombinators, rewrite rules). Besides, it would make it possible to concentrate on the implementation of a relatively simple system instead of getting lost in the details of a specific functional language. We believed that this intermediate level would not yield a loss of efficiency compared with a direct implementation. Because we wanted to experiment with the intermediate language we demanded that this language could be used as a simple programming language.

In Clean 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). The functional strategy can be used in TRS's and in TGRS's. This is particularly useful in so called *totally overlapping* rule systems in which the only overlaps which are allowed to occur between rules, are overlaps where a complete left-hand-side of a rule is an instance of a left-hand-side of another rule.

The functional strategy effectively disambiguates the rewriting system because the order in which redexes are evaluated is fixed and furthermore the rules are tried for matching in textually topmost order. This is done by forcing evaluations to root normal form in such a way that when a lower rule is actually applied on a redex, no higher rule is applicable on that redex. This property can be seen as a practical approximation of the semantically difficult property of Priority Rewrite Systems (Baeten *et al.* (1987)) that a rule with lower priority is applied only if no higher rule can ever be applied on the subterm in question.

Although the functional strategy is intuitively appealing and important for practical use, it is not a normalising strategy for TTRS's (and hence neither for TDRS's and TGRS's) as is shown by the following example.

A TTRS for which the functional strategy is not normalising:

```
A B C   →   Z
A x D   →   Z
W       →   W
```

With as initial term `A W D` the evaluation of the first argument of `A` is forced because there occurs a non-variable in the pattern. This forced evaluation does not terminate. Nevertheless `A W D` has `Z` as normal form.

The fact that the functional reduction strategy is used, has a great impact on the meaning of a rule system. Therefore, the following terminology is introduced.

Functional Graph Rewriting Systems (FGRS's) is the class of TGRS's in which the functional strategy is used. FGRS's are the basis for Clean. Every Clean program is an FGRS.

It is shown that efficient state-of-the-art implementations on sequential hardware can be obtained compiling functional languages via Clean (Koopman & Nöcker (1988), van Hintum & van Schelven (1988)). Clean is implemented sequentially with reasonable efficiency. Clean has fulfilled the demands which were set for it.

Further research will increase the efficiency of the implementation and it will be investigated how Clean can be extended for e.g. unification without loosing the efficiency. Concurrent Clean will be the extension of Clean suited for parallel implementations on loosely coupled machine architectures based on extended functional graph rewriting systems as is explained in the following section.

2.7 EXTENDING FUNCTIONAL GRAPH REWRITING SYSTEMS: PC-FGRS'S

In chapter 7 we define extensions of special classes of graph rewriting systems which enable the specification of general loosely coupled parallel evaluation in graph rewriting systems in a object-oriented manner.

In that chapter it will be shown that extended FGRS's: *PC-FGRS's* (FGRS's with strict annotations, explicit parallelism and lazy coping) have a surprisingly high expressive power. Arbitrary process and processor topologies can be modelled, as well as synchronous and asynchronous process communication. In particular, loosely coupled parallel evaluation can be modelled such that any process communication structure can be defined. PC-FGRS's will be the basis of the new intermediate language Concurrent Clean (Smetsers *et al.* (1988)).

Although in PC-FGRS's 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. With term graph rewriting always the same term will be yielded.

PC-FGRS's are well 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 PC-FGRS's and the properties of these systems gives high expectations for a better exploitation of the potential parallelism in functional programs.

2.8 CONCLUSION

Graph Rewriting Systems are in our opinion most suited as a model of computation for functional languages and their implementations.

Multi-level rewriting systems are very useful for high level specifications varying from specifying reduction strategies to operational semantics. With term graph rewriting nice results can be obtained on modelling term rewriting with graph rewriting. Generalized graph rewriting is very powerful and of independent interest as a general model of computation for parallel architectures. Based on a restricted subset of generalized graph rewriting Clean is in practise proven to be very suited as an intermediate language for functional languages and sequential machine architectures. Extending graph rewriting with lazy copying and explicit parallelism gives a very promising model for loosely coupled parallel evaluation of functional programs. Its expressive power and its properties will make it also possible in the near future to generally exploit the potential parallelism in functional programs successfully.

Much research on graph rewriting systems still has to be done (taxonomy, typing, strategies, strictness analysis, implementation techniques, parallel evaluation, garbage collection, etcetera). But the results achieved so far are very promising, justifying further research on graph rewriting theory and on identifying special classes of graph rewriting systems. Furthermore, actual experiments with sequential and parallel implementations will be necessary to achieve more experience and to identify key issues.