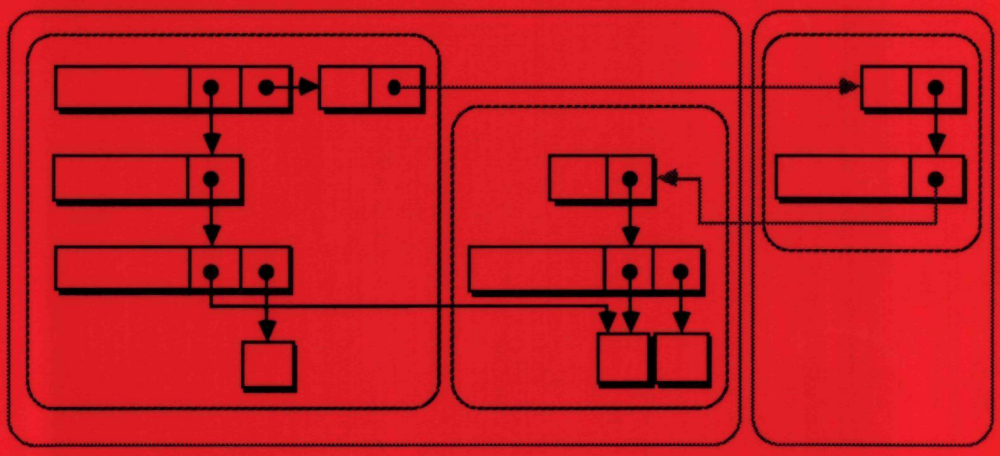
PARALLEL GRAPH REWRITING

—

Some Contributions to its Theory, its Implementation and its Application



Marko van Eekelen

PARALLEL GRAPH REWRITING

—

Some Contributions to its Theory, its Implementation and its Application

een wetenschappelijke proeve op het gebied van de Wiskunde en Natuurwetenschappen.

Proefschrift

ter verkrijging van de graad van doctor aan de Katholieke Universiteit te Nijmegen,
volgens besluit van het college van decanen in het openbaar te verdedigen op
vrijdag 2 december 1988 te 13.30 uur precies

door

Marius Cornelis Johannes Dionisius van Eekelen

geboren op 5 december 1956 te Bergen op Zoom.

Krips Repro Meppel

Promotor:	Prof. dr. H.P. Barendregt
Co-referent:	Dr. ir. M.J. Plasmeijer

Aan mijn vader

Manuscriptcommissie:

Prof. dr. J.W. Klop,	Centrum voor Wiskunde en Informatica, Amsterdam & Vrije Universiteit, Amsterdam;
Prof. dr. M.R. Sleep,	University of East Anglia, Norwich, United Kingdom;
Prof. dr. S.D. Swierstra,	Rijksuniversiteit Utrecht.

Dankbetuiging:

Op deze plaats wil ik iedereen bedanken die op zijn of haar manier aan dit proefschrift heeft bijgedragen. Zonder jullie directe en indirecte hulp zou dit proefschrift nooit tot stand zijn gekomen.

Acknowledgements:

I would like to express my gratitude to everyone who has contributed one way or another to this thesis. Without your direct and indirect assistance this thesis would never have been produced.

PREFACE

Historical background

The main part of this thesis is a collection of papers which are the result of the research performed by the author as a member of the computer science department of the Nijmegen University. The research has been partly sponsored by the Dutch Parallel Reduction Machine Project.

In 1984 this project was set up by the government of The Netherlands. The project was led by Prof. Henk Barendregt and involved teams at the universities of Amsterdam, Utrecht, and Nijmegen. The group at Nijmegen was led by Rinus Plasmeijer with the author of this thesis. The aim of the PRM-project was to investigate the feasibility of building a parallel reduction machine for the efficient evaluation of functional languages.

A related project in the United Kingdom, the Flagship project, was following the path of fine-grain execution, developing dataflow ideas and experience with the ALICE project: a multi-processor project of Imperial College London. The Dutch project investigated coarse-grain parallelism on fairly conventional loosely coupled multiprocessor architectures. In common with the team of the University of East Anglia (Prof. Ronan Sleep, John Glauert and Richard Kennaway) which was part of the Flagship project, the Nijmegen group had a strong interest in developing an intermediate language based on a computational model which should reflect the essential aspects of both functional languages as their implementation. It was recognized that the choice of the computational model was the most critical decision to make. It would highly influence the needed compilation effort and the final efficiency of the obtained code.

During a meeting in Oosterbeek (in the Netherlands) a subgroup consisting of Henk Barendregt, Rinus Plasmeijer and the author of this thesis decided that, in principle, Term Rewriting Systems would be best suited as the computational model to use. The patterns which are the basis of Term Rewriting Systems, were expected to contain essential information necessary for efficient implementation. Sharing of terms was felt to be essential in order to obtain efficient implementations on sequential hardware. However, the sharing of terms would be a problem in a parallel environment. We hoped that this problem could be solved in the future. To model this sharing of terms it was decided that therefore graphs had to be used instead of terms. Following meetings at a workshop on the island of Ustica, and at the second Conference on Functional Programming Languages and Computer Architecture at Nancy, September 1985, collaborative work was undertaken by the Nijmegen group and the UEA team.

After some very beneficial initial consultations with Prof. Jan-Willem Klop, a model of graph rewriting was developed within a sound theoretical framework. Spin-offs included a clear theory of Term Graph Rewriting and the intermediate languages Lean, Dactl1 and Clean, all published in 1987. All these intermediate languages have the same underlying model of graph reduction.

The Dutch Parallel Reduction Machine Project ended in 1987. Its main results are summarized in Barendregt *et al.* (1987c). An international evaluation committee with representatives of universities and industry stressed the high quality of the research meeting international standards and having important practical applications.

The authors work which is reflected in this thesis, focusses on the level of the intermediate language and its underlying model of computation.

Contents of this thesis

Chapter 1 gives an introductory overview of the fields of functional programming and models of computation. It can be skipped by readers who are familiar with these topics.

In chapter 2 the interconnections between the topics of the following chapters are explained and it is motivated why Graph Rewriting Systems are a promising model of computation.

This thesis is essentially a collection of papers: the chapters 3, 4, 5, 6 and 7 are reprints of co-authored papers which are also published elsewhere. The papers are unchanged: only the layout is changed in order to make it more or less uniform and the references are merged in chapter 8. There was a great temptation to revise the papers (e.g. add the improved efficiency figures of the Clean compiler; remove inaccuracies; merge introductions etcetera). This is not done however in order to avoid doing the work twice in different contexts: an integrated overview of the field of functional programming and parallel graph rewriting will already appear as a textbook for students (Plasmeijer & van Eekelen (198-)).

In chapter 3 (van Eekelen & Plasmeijer (1986)) we introduce a new high level specification method for rewriting strategies, which simplifies the reasoning about the correctness of a specification of a reduction strategy. This method is being used for specifying strategies and with a minor change for specifying the semantics of Concurrent Clean. The paper was presented by the author of this thesis at the Workshop on Graph Reduction at Santa Fe, New Mexico.

Chapter 4 is also published in the Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE) at Eindhoven (Barendregt *et al.* (1987a)). It gives some basic soundness and completeness results of a graph rewriting class which is used to implement term rewriting. This makes it possible to identify restrictions guaranteeing correctness of implementations of Term Rewriting which use sharing.

Chapter 5 is a revised version of Barendregt *et al.* (1987b) which was presented by the author of this thesis at the PARLE conference at Eindhoven. It defines generalized graph rewriting which can also be applied in general declarative and even in non-declarative environments. This chapter will also appear in the special issue of the Journal of Parallel Computing with revised versions of selected papers of the PARLE conference (Barendregt *et al.* (1988)).

Chapter 6 has also been published in the Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, USA (Brus

et al. (1987)). It describes a functional intermediate language based on graph rewriting with which it is shown in practise that functional languages can be efficiently implemented.

Chapter 7 broadens the scope of parallelism in functional graph rewriting to general loosely coupled parallel evaluation. This chapter is also published as internal report of the University of Nijmegen (van Eekelen *et al.* (1988)) and it is also submitted for publication elsewhere. It will form the basis of actual implementations of functional languages on parallel computer architectures like e.g. a Transputer Rack or the DOOM machine of Philips Research Laboratories at Eindhoven.

TABLE OF CONTENTS

1

FUNCTIONAL PROGRAMMING AND COMPUTATIONAL MODELS

In this introductory chapter an overview is given of the field of models of computation with respect to functional programming languages. The importance of defining a model of computation is explained. The correspondence between paradigms in programming languages and computational models is discussed. The importance of functional languages is addressed. Furthermore some basic models of computation which are used traditionally for describing the behaviour of implementations of functional languages, are introduced very briefly.

More information on paradigms, functional languages, computational models, graph rewriting, implementation methods, intermediate languages, compilation schemes and abstract and concrete machine architectures can be found in Plasmeijer & van Eekelen (198-).

1.1 PARADIGMS OF PROGRAMMING: LANGUAGES AND MODELS

LANGUAGES

A language is a programming language if its syntax and semantics are formally defined and implementable. Such a formal definition can be based on any formal description method e.g. denotational, algebraic, categorical or operational. In particular, a compiler and an interpreter are themselves formal definitions of the semantics, where another programming language is used as the formal operational description method. In the case of a compiler generally even two other programming languages are used. The basis of all implemented formal descriptions of semantics is the operational semantics of the machine language which by itself is realized in hardware via a particular machine architecture.

Every program which accepts input induces its own special purpose programming language. Some examples of special purpose languages are the set of commands for an editor, the Unix shell and a data base query language. Usually no model of computation is given for a special purpose language. The meaning of such languages is mostly defined via user manuals.

General purpose languages do not focus on a specific (class of) algorithm(s). Being general purpose they have a general view on the world. Such a view can be formalized yielding a model of computation of the language.

MODELS

A general purpose programming language is usually 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. It is sufficient to examine the semantics of these essential language constructs since the semantics of other language constructs can be expressed in terms of the basic constructs by removing syntactical sugar. A computational model tries to capture the essential aspects of a programming language in a formal model in order to be able to reason about them. However, in principle for a specific language there are many computational models which can be used. For instance, any deterministic computation can be expressed on a Turing Machine (Turing (1936)). But this will not greatly simplify the reasoning about the correctness of a particular program written in a specific language because the Turing machine generally is too far away from the semantics of a specific language.

Furthermore, a computational model tries to capture the essential aspects of (a class of) implementations of the programming language.

A model of computation tries to capture the essential aspects of a language by making some abstractions. A *model of computation* (or a *computational model*) of a programming language is a formal model as close as possible to both semantics and implementation still modelling only the essential aspects of them. If such a model of computation is known it is much easier to reason about the correctness of specific programs, 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.

It is of course very difficult to find such an ideal model which models every essential aspect of a programming language. In fact the classical (sequential and imperative) programming languages all have the same model of computation: the Turing machine. Clearly this model can only describe the very basic concepts of those languages. Lately, newly defined programming languages all come with a model of computation. The new language and the model of computation are closely related: either the language is built on top of the model or the other way around or sometimes they are even developed together. When defining a model of computation for a language one often has to choose between what does the language mean (the semantical aspects) and how is it done (the behavioural aspects). Many times this leads to a language having both a denotational and an operational semantics. These two semantics may be based on different models of computation which together fully specify what the language is about.

PARADIGMS

A *paradigm* in programming is an intuitive view on what the essence of programming is. In the computer science community the traditional imperative paradigm has prevailed for many years. Until recently, proposals for new programming languages based on other paradigms did not really catch on because the implementation techniques to translate them efficiently to the imperative machine architectures were not developed enough. The most important proposed new

paradigms for programming languages are object-oriented, dataflow and declarative. In the following paragraphs each of the paradigms will be discussed.

Imperative

Von Neumann machine architectures (Burks *et al.* (1946)) all use the imperative paradigm. In the imperative paradigm there is a memory containing data and this memory also contains an ordered sequence of commands (*instructions*) and a locus of control which indicates which instruction has to be executed next. The instructions may or may not change any part of the memory. For many years, using this paradigm implementation and programming techniques were developed, refined and optimized. Many imperative programming languages were introduced generally leading to higher levels of abstraction in programming. This trend towards higher levels of abstraction is enabled by improved implementation techniques and by increased computational power of the hardware.

An imperative program (written in Modula-2 (Wirth (1982)) calculating an array with the first N Fibonacci numbers:

```
VAR
   fib: ARRAY [0..N-1] OF INTEGER;
   i:   CARDINAL;
BEGIN
   fib[0] := 1; fib[1] := 1; i := 1;
   WHILE i < (N-1)
   DO
      i := i + 1;
      fib[i] := fib[i-1] + fib[i-2]
   END
END
```

The basic computational model for Von Neumann machine architectures is introduced by Turing in 1936: the Turing machine (Turing (1936)). This model of computation is used for complexity theory and other theoretical issues (e.g. the Halting Problem (Lewis & Papadimitriou (1981)).

However, there are some small problems with the computer systems based on the imperative paradigm. It is hard to find a piece of software without any bugs. Computer scientists have learned to live with the *software crisis*, and they accept that most software products are unreliable, unmanageable and unprovable. Although hardware is much more reliable than software, most hardware systems appear to be designed in a hurry and even well-established processors now and then go down because of undocumented race-conditions. Clearly software and hardware systems have become very complex. So it seems to be understandable that these systems contain bugs. A good, orthogonal design costs many, many, man years of research and development. The good news is that hardware becomes cheaper and cheaper (thanks to the Very Large Scale Integration) and speed can be bought for prices never dreamed of. Of course it never goes fast enough. Yet it seems that the maximum speed that can be obtained with present day technology in Von Neumann architectures is beginning to reach its limit.

The two key problems that the computer science community has to solve are how to make reliable and user-friendly software at low costs and how to increase processing power at low costs. Researchers are looking for solutions for these problems: investigating software

engineering techniques, to deal with problems related to the construction of very large software programs; designing new proof techniques to tackle the problems in proving the correctness of systems; developing program transformation techniques, to transform the specification of a problem to a program which solves it, designing new (parallel) computer architectures using many processors (up to thousands or more) to increase execution speed.

One approach which eventually may help to find a solution for the key problems, is based on the idea that these problems are fundamental problems which cannot be solved unless a totally different approach is taken and hardware and software is designed with a completely different model of computation in mind. We believe that this idea is true and that the solution lies in non-imperative paradigms.

An imperative programming style has the following drawbacks.

- It consists of a sequence of commands of which the dynamic behaviour must be known in order to understand how such a program works. Particularly the assignment causes problems, because it changes the value (and often the meaning) of a variable. Evaluating the same expression in succession may produce different answers. Reasoning about the correctness of an imperative program is therefore very difficult .
- In addition, due to the relatively low expressive power of the present high-level programming languages, programs become large and therefore hard to understand.
- Because of the command sequence, algorithms are more sequential than necessary. Therefore it is hard to detect which parts of the algorithm can or cannot be executed concurrently.

So, we believe that the software crisis and the speed problem are inherent to the nature of imperative programming languages and the underlying model of computation.

Object-oriented

The introduction of multiprocessing on a single machine, led to a new paradigm for parallel languages, which is also used for programming parallel machines:

In the *object-oriented* (or *actor-based*) paradigm a program describes the behaviour of a system in terms of its constituents, the objects. In object-oriented programming languages each object has some internal data and the ability to act on (change) these data. Objects may have an internal activity of their own. Objects only interact by sending messages to each other. One could say that each object is a process controlled by an imperative subprogram communicating via message passing with other processes.

An object-oriented program (written in POOL2 (America (1988))) that generates all prime numbers using the sieve method:

```
IMPL UNIT Sieve
USE File_IO
GLOBAL driver := Driver.new ()
```

```
CLASS Driver
VAR first := Sieve.new ()
BODY FOR i FROM 2 DO first ! input (i) OD
YDOB
END Driver

CLASS Sieve

VAR myprime, current : Int
    next : Sieve

METHOD input (n : Int) : Sieve
BEGIN current := n;
      RESULT SELF
END input

BODY ANSWER (input);
     myprime := current; %% the first input is a prime number
     standard_out ! write_Int (myprime, 0) ! new_line ();
     next := Sieve.new ();
     DO ANSWER (input);
        IF current // myprime ~= 0
        THEN next ! input (current)
        FI
     OD %% forever
YDOB
END Sieve
```

The main parts of the program are the class `Driver` of which there is only one object (created with `Driver.new ()`) and the class `Sieve` of which there is in principle an infinite number of objects (every sieve starts a new sieve). The `driver` generates all natural numbers greater than 2. The class `Sieve` contains a method `input` which if called (by other objects) and answered saves the argument in a local variable. Every `Sieve` has an activity of its own which is defined in the body. Its prime number is printed on the standard output file. It starts a new `Sieve` and it loops forever sieving out multiples of its primenumber.

The most commonly used basic computational model for object-oriented machine architectures is the Calculus of Communicating Systems (Hoare (1985)). This model of computation is used for complexity theory and other theoretical issues (e.g. fairness, proof theory). Process Algebra (Baeten (1986)) might be an alternative when one is more interested in describing the behaviour of processes.

Modularisation is an important technique which is used in the development of large programs. In the object-oriented paradigm a program is modularised by the introduction of the objects. One can imagine that in parallel machines different objects might live on different processors.

The activity of a single object is usually defined in an imperative language. In that case the object-oriented language essentially inherits the disadvantages of the imperative paradigm. In chapter 7 of this thesis we present some promising fundamental research in combining the object-oriented view with functional languages.

Dataflow

In the *dataflow* paradigm the data moves to the instructions instead of the other way around. An instruction may start executing as soon as its data is available. So, the flow of data determines the execution order. This approach has gained a lot of interest because of envisaged fast parallel hardware designs.

Dataflow languages or *single assignment* languages are languages in which variables are assigned only once: i.e. a variable is undefined until it gets a value and then it can never change afterwards. Single assignment languages are made to fit the dataflow model of computation.

A dataflow program (written in SISAL (Glauert (1978)) calculating an array with the first N Fibonacci numbers:

```
fibnumbers :=
   FOR
      INITIAL
         fib1 := 1; fib2 := 1
      REPEAT
         fib1, fib2 := OLD fib2, OLD fib1 + OLD fib2
      WHILE
         fib2 < N
      RETURNS ARRAY OF fib2
   END FOR
```

Also imperative and functional languages can be translated more or less efficiently to dataflow languages (Veen (1985)). It seems that functional languages are better suited to be efficiently translated to dataflow than imperative languages.

To exploit the advantages of the dataflow model special machine architectures are necessary which realize the inherent parallelism. Unfortunately, actual dataflow architectures (Gurd *et al.* (1985), (Arvind *et al.* (1987)) are very complex and not yet commercially available.

Declarative

In the *declarative* paradigm a desired computation is expressed in a static fashion as a list of declarations and an expression to be evaluated. A program is considered to be an executable specification.

A declarative language has the following advantages which are common in any mathematical notation.

- Mathematics is static (no assignments, side effects): a function will always give the same answer if it is applied on the same arguments.
- There is consistency in the use of names (like in $x^2 - 2x + 1$). Variables do not vary, they stand for a, perhaps not yet known, constant value throughout their scope.
- An expression always has the same meaning independent of the history of the computation (*referential transparency*).
- Because equal expressions are always and everywhere interchangeable, declarative languages are convenient to reason about.

The class of languages using the declarative paradigm can be split up in grammar, logic and functional languages.

Grammar languages

Grammar languages are based on the idea that a program essentially just parses its input and produces a result accordingly. The programmer specifies in a grammar what the input may look

like and which semantic actions must be taken. In the field of compiler development these languages have lifted software development to a higher level and they have greatly increased the programming productivity.

A grammar program (written in enhanced Extended Affix Grammars (Meijer 1986)) that appends two lists into another list:

```
append (>empty, >list, list>): ;
append (>elt⊕list1, >list2, elt⊕list3>): append (>list1, >list2, list3>).
```

The > signs define the flow of the arguments: before the argument they denote an input parameter, after the argument they denote an output parameter.

Although all grammar languages are based on grammars they do not all have the same computational model. The kind of grammars that are used, identifies the computational model: e.g. affix grammars (Koster (1971)) or graph grammars (Nagl (1979)).

Grammar languages originated as special purpose languages but it has been proven to be worthwhile to investigate whether they can also serve as general purpose languages. Problems occur in userfriendliness, debugging facilities and programming environments.

Logic languages

Logic languages are based on the idea of a program being a set of rules which state that predicates are true if certain conditions are met. These rules are multi-directional. So, you can ask whether a predicate is true with certain values for specific logical variables but you can also ask for which values of these variables the predicate holds.

A logic program (written in PROLOG (Clocksin & Mellish (1984))) that appends two lists into another list:

```
append ([], List, List).
append ([Elt|List1], List2, [Elt|List3]):- append (List1, List2, List3).
```

The logic program is very much like the grammar program. The main difference is the multi-directionality of the logic program. The logic program will give a result for the first parameter if the other two are specified but it will also give a result for the third parameter if the first two are specified. This difference is perhaps not really essential but it is typical for the general view on programming.

In logic languages evaluation means investigating whether a formula in first-order predicate logic can be true. The evaluation method uses unification and is based on the resolution method of Robinson (Robinson (1965)). In fact the languages are quite close to the model by contrast with the imperative languages and the Turing machine model. Logic languages can often be seen as a subset of the general model defined by certain restrictions and extended with some specific constructs for efficiency and software engineering purposes.

The Japanese Fifth Generation Computers project has had a great impact on the world-wide interest for logic languages. Logic languages are now widely used in the context of database systems. In the context of logic languages data is often called *knowledge*.

Functional languages

In a *functional language* every program is just a collection of function definitions. Each function can be seen as a kind of program which accepts input (its arguments) and produces output (its result). The concept of a function is one of the fundamental notions in mathematics.

Some functional programs (written in Miranda™* (Turner (1985)):

Fibonacci numbers:

```
fibs thisfib nextfib = thisfib : fibs nextfib (thisfib + nextfib)

take n (fibs 1 1)
```

A functional program for sieving prime numbers can be found in chapter 7.

Append two lists into another list:

```
append []           list = list
append (elt:list1) list2 = elt : (append list1 list2)
```

One of the greatest advantages of functional programming languages is that they are based on a sound and well understood mathematical model, the *λ-calculus* (Church (1932/1933)). In terms of denotational semantics one could say that functional programming languages are sugared versions of this λ-calculus. This computational model is introduced more or less at the same time as the Turing model. The power of these models is the same (Turing (1937)). Beside the λ-calculus there are other related computational models which can be seen as a basis of functional programming languages, namely *combinatory logic* (Schönfinkel (1924), Curry (1930)) and *rewriting systems* (Klop (1987)). These computational models are very important because they have a great influence on the specification of the semantics of functional programming languages and on their implementations. In section 1.3 the traditional models are discussed and compared. In chapter 2 rewriting models will be discussed. Recently many researchers started to investigate sequential and parallel machine architectures especially suited for these models of computation. These architectures are called *reduction machines*.

The main advantages that are offered by functional languages, are a great expressive power, relatively easy correctness proofs and a relatively high suitability for parallel evaluation. Therefore functional languages are a very important research topic. An overview of the field of functional programming is given in Field & Harrison (1988).

1.2 FUNCTIONAL PROGRAMMING LANGUAGES

In Backus (1978) it is pointed out that the solution for the software problems should be searched in finding a new programming discipline. He proposed to investigate *functional programming languages*, also called *applicative programming languages* (sometimes the notion "functional" programming languages is used for languages which support higher order functions and the notion "applicative" programming languages for languages which have application as the overall

*Miranda is a trademark of Research Software Limited.

basic concept; outside the functional programming community the notion "functional" is widely used as a synonym of useful).

To be able to program with functions a suitable rich set of basic functions has to be defined and then they have to be used to define new functions in terms of these. Thus a whole library of useful functions may be built. Some of those library functions are probably built upon layers of others.

The question arises, whether more is needed than just a repertoire of basic functions and the ability to combine them in order to define a library of functional programs? Fortunately, there is a mathematical thesis, known as the *Church's thesis* (Church & Rosser (1936)), which states that the class of *computable functions* is exactly the same as the class of *recursive functions*. This class of recursive functions is exactly the class of functions you can get by combining some basic functions via primitive recursion, composition and minimalisation.

In functional languages the programmer can only define functions which compute values uniquely determined by the values of their arguments. Consequently, many of the familiar concepts of conventional programming languages are missing in purely functional languages. Most important, assignment is missing. So is the heavily used programming notion of a variable, something which holds a value that is changed from time to time by an assignment. Rather, the variables that exist in purely functional language are used like in mathematics to name and refer to a yet unknown constant value. Once the value is known it cannot be altered anymore: in mathematics a variable does not vary.

FUNCTIONAL PROGRAMMING IN AN IMPERATIVE LANGUAGE

Perhaps a functional programming style is important, like avoiding goto's. But are new languages really needed? Imperative languages also have functions, so why not just use the functional subset of e.g. C, Algol or Modula. Well, even if only the functional subset of these languages would be used (this means leaving out the assignment) these languages are not as suitable as the new functional programming languages. The reason is that functions in the imperative language are often not treated as first-class citizens. This is a fact of life, not a fundamental problem. In some languages a function cannot be an argument of a function, only values can be arguments. In other languages functions cannot yield a function as result. Sometimes these restrictions are present because one did not know how to make an (efficient) implementation of such functions. Furthermore, the available type systems of the classical imperative languages makes it impossible to fill the gap without a complete redesign of these languages. Also nice features of functional programming languages such as infinite data structures (see the following section) are not possible in imperative languages because of the evaluation order which is used in these languages.

BASIC CONCEPTS OF FUNCTIONAL PROGRAMMING LANGUAGES

In this section the most common concepts of functional languages are introduced. When not explicitly stated otherwise, all examples are written in the functional programming language Miranda.

Function definitions

A program contains an expression to be computed and a collection of function definitions written in the form of recursive equations.

Some simple function definitions:

```
increase x    =    x + 1
square x      =    x * x
squareinc x   =    square (increase x)
constant      =    7
```

In the definitions of these functions, x is a *formal parameter* or *formal argument*. It is essentially like a *bound* variable in mathematical logic. Its scope is limited to the equation in which it occurs (whereas the other names introduced above have the whole program as scope).

The basic operation in functional programming languages is *function application*. Because function application is so fundamental in functional programming languages, the operator is generally not written down explicitly. Application is simply denoted by juxtaposition.

For example, in case the function `square` is applied to the value 3, we simply write down:

```
square 3
```

In some languages however, application has to be written explicitly with a special binary function.

For example:

```
Apply (square, 3)
```

The expression on which a function is applied (in the example the value 3) is called the *actual parameter* or *actual argument*. The notation where the application operator is hidden is called the *applicative style* and the notation in which the application is explicitly present as binary function is called the *functional style*.

As usual in programming languages, one can denote and manipulate objects of certain predefined types. The following basic data types, with appropriate basic operations, are usually available.

Data types:	**Operations:**	**Notation of constant values:**
numbers	+, -, *, ...	1, 34, 12, ...
truth values	and, or, ...	True, False.
characters	=, <, ...	'a', 'c', ...

The execution of a functional program simply consists of the evaluation of an initial expression in the context of the function definitions in the program. Evaluating means repeatedly performing

reduction or *rewriting steps*. In each reduction step (indicated by an "→") a function application in the expression is replaced (*reduced, rewritten*) according to its definition (by the right-hand-side of the equation), substituting the formal arguments by the corresponding actual arguments. The subexpression that is rewritten is called a *redex* (*reducible expression*). The reduction process stops when none of the function definitions can be applied anymore (there are no redexes left). Then the initial expression is in its most simple form, the *normal form*. This result of the evaluation is printed.

For instance, given the function definitions above (the environment), the expression `squareinc constant` can be evaluated (reduced) as follows. The expression which will be rewritten, is underlined.

```
squareinc constant      →   square (increase constant)
                        →   square (constant + 1)
                        →   square (7 + 1)
                        →   square 8
                        →   8 * 8
                        →   64
```

Higher order functions

Compared with the traditional imperative languages, which normally allow also the declaration of functions, functional programming languages have a sound view on the concept of a function: functions are treated as first-class citizens.

A *first-order* function is a function which can only have basic types as argument and as result. A *higher order* function is a function which can have a function as argument or as result. Languages which support higher order functions in a general way are also called *higher order* languages. Functional languages are higher order languages.

The possibility to yield a function as result makes it unnecessary to consider functions with more than one argument. A function with n arguments can be constructed by a function with one argument that returns a function that can be applied to the next argument, and so on.

In general a function definition has the following form:

```
function-name arg1 arg2 ... = expression
```

In these definitions each function actually has only one argument. The convention is used that function application is associative to the left, so the parentheses are left out. But actually this must be read as:

```
.. ( (function-name arg1) arg2) ... = expression
```

This way of considering all functions as functions of one argument is called *Currying*. At first sight, Currying perhaps looks a bit strange, but keep in mind that the basic operation in functional programming languages is binary function application, which is hidden by the applicative notation using juxtaposition. With explicit application the definition would be:

```
.. Apply (Apply (function-name, arg1), arg2), ... = expression
```

Currying enables a familiar notation to be used, but furthermore it enables "the application of functions with any number of arguments", which gives an additional descriptive flexibility. This encourages a programmer to write "parametrized functions".

A simple Curried definition:

```
plus x y = x + y   || the type of plus is: num → num → num. The → 's are right associative,
                   || so given an object of type num, the higher order function plus produces
                   || an object of type: num → num
```

So, also the following definition is allowed:

```
inc = plus 1       || the type of inc is: num → num
```

This enables the following reductions:

```
inc 3   →   plus 1 3   →   1 + 3   →   4
```

Patterns

In functional programming languages functions are defined by a series of equations. In the left-hand-side of an equation one can specify that the equation in question can only be applied if the actual arguments of the function are of a certain shape. The execution mechanism must be given the "intelligence" to match actual parameters with the patterns used in the definition of a function so that it can decide which equation to use. Generally, the equations are tried in the textual order they are specified, from top to bottom. Within one equation the arguments are tried from left to right. The patterns may consist of expressions with variables (e.g. `n`, values (e.g. `0`) and data constructors (e.g. the infix `:`, which is used to represent a list; see also the next section on data structures). This facility, which is called *pattern-matching*, offers an alternative to conditional expressions. It often leads to clearer and more concise definitions.

Patterns and pattern-matching:

```
fac 0 = 1
fac n = n * fac (n - 1)

hd (a:b) = a
tl (a:b) = b

cond True  x y = x
cond False x y = y
```

'0' and 'n' are the patterns of the first two rules (a simple variable as pattern indicates that it does not matter what is on that position) Reducing '`fac 7`' will result in a call to the pattern-matching facility, which decides that only the second rule is applicable.

The functions `hd` and `tl` require a non-empty list as actual argument. The head of the list is mapped on `a`, the tail of the list is mapped on `b`. Again these variables indicate that the contents of the list are irrelevant.

A conditional can easily be defined. The choice between "then" and "else" part will depend on the actual value of the first argument as indicated by the pattern.

Data structures

Lists are a rather important data structure in many functional programming languages. Lists, like any data structure in a functional programming languages, are constructed with the help of so called *data constructors*. Such a constructor can be seen as a kind of *identification* or *tag* which

uniquely identifies (and is part of) a "record" of a particular type. The *list-constructor* is usually named "Cons" (prefix notation) or ":" (infix notation). The type of a list could recursively be defined as (usually it is predefined):

```
list *   ::= Cons * (list *)      | Nil
```

List-types are denoted by `list *` or `[*]`. A list contains two elements and the list-constructor `Cons`, or, a list is an empty list denoted by `Nil` or `[]`. If the list is non-empty it contains a list element of a certain type * as head while the tail of the list is again a list of the same type. One could say that the head and the tail are "glued" together with help of the list-constructor. Lists are declared either by enumeration or by (recursive) definitions. The syntax we use for lists, is illustrated by the following examples.

Lists, explicit infix notation:

```
1:(2:(3:(4:(5:[]))))            || the list of numbers from 1 up to 5
True:(False:(False:[]))         || a list of three booleans
[]                              || denotes the empty list
1:2:3:4:5:[]                    || ":" is right associative
```

For convenience of the programmer there is a shorthand notation for lists, using square brackets.

Lists, shorthand notation:

```
[1, 2, 3, 4, 5]                 || the same list of numbers as above
[True, False, False]            || again the same list with 3 booleans
0 : [1, 2, 3]                   || same as [0, 1, 2, 3]
```

One of the powerful features of functional programming languages is the possibility to declare infinite data structures via recursive definitions.

Definition of an infinite list equal to [1, 1, 1, 1,]:

```
ones = 1 : ones
```

`ones` is a recursive function yielding an infinite list

Programs having infinite data structures as result do not terminate of course. For practical reasons such a result is printed as soon as possible. For an infinite list this means that, from "left to right", the elements of the list are printed as soon as they are in normal form.

The possibility of defining infinite data structures puts some restrictions on the evaluation strategy which is explained in the following section.

Some predefined operations on lists are generally available. These are: length of a list (denoted by #), subscription (!) and concatenation (++). The operations are assumed to be predefined in a library for reasons of efficiency and convenience.

Predefined list-operations:

```
# [2, 3, 4, 5]          || length of list,      →   4
[2, 3, 4, 5] ! 2        || subscription,        →   4
[0, 1] ++ [2, 3]        || concatenation,       →   [0, 1, 2, 3]
```

Evaluation

As explained in the previous section, the evaluation of a functional program has the intention to find a final non-reducible expression denoting the same value as the initial expression E by repeatedly performing reduction steps. Because there are in general many redexes in the expression to reduce, one can perform these steps in several ways. This is determined by the so called *reduction strategy* which controls the evaluation. A reduction strategy is sometimes also called an *order* because it is used to indicate the next redex to reduce. There are a couple of important things to know about the ordering of reduction steps.

In functional languages the final result of the computation (the normal form) does not depend on the order in which the redexes were reduced: the normal form is unique (see also section 1.3). However, there are restrictions that have to be put on the evaluation order. The specification of a pattern forces evaluation in order to decide whether or not a particular rule for a function can be applied.

Forced evaluation:

```
fac 0 = 1
fac n = n * fac (n - 1)
```

The expression `fac (1-1)` should of course not match the second rule, but the argument has to be evaluated first. After the evaluation of the argument the rules may be matched in order.

Furthermore, some reduction orders may not lead to the normal form at all (such a computation will not terminate).

Non-terminating reduction: define

```
inf = inf
```

then the following reduction order may be taken:

```
cond True 1 inf  →  cond True 1 inf  →  cond True 1 inf  →  ...
```

In this case another choice would lead to termination and to the normal form:

```
cond True 1 inf  →  1
```

The reduction strategy followed depends on the functional programming languages. In some languages, e.g. Lisp and Hope, the arguments of a function are always reduced before the function itself is considered as a redex. This is called *eager* evaluation and languages which use such an evaluation strategy, are sometimes called *eager* languages. Infinite data structures cannot be handled in these languages, because they are evaluated as soon as they are passed as arguments, which leads to infinite computations.

Nowadays, in many functional programming languages the rewriting is done *lazy*. That is, the value of a subexpression (redexes) is calculated if and only if this value must be known to do the rewriting. Lazy evaluation makes it possible to handle infinite data structures.

Lazy evaluation and infinite lists:

```
hd (tl ones) → hd (tl (1:ones)) → hd ones → hd (1:ones) → 1
```

In eager languages the evaluation of the parameter `ones` would not terminate:

```
hd (tl ones) → hd (tl (1:ones)) → hd (tl (1:(1:ones))) →
```

There is a trade-off in the choice between lazy and eager evaluation: on one hand lazy evaluation gives better expressiveness in the language and on the other hand eager evaluation simplifies the implementation of a language. We have chosen for lazy evaluation because we like to improve the expressiveness of programming languages.

Typing

There are untyped functional languages (Twentel, KRC and SASL) but most functional languages are typed. In untyped languages any kind of functions can be written which is at the same time their advantage and their disadvantage. After all, every function is written with intended argument and result domains. In practise, a programmer will want to restrict himself deliberately to using these functions with elements of the intended domains only. Essentially, the type of a function is precisely this information on the intended domains. The type information can not only be used as a filter to restrict the set of acceptable programs, but it can also be useful for a compiler to produce faster code for specific types.

Two ways of typing are distinguished: explicit typing and implicit typing.

Explicit typing (or type *checking*) means that the compiler or the interpreter checks the type which is explicitly given by the programmer. Type checking occurs e.g. in the languages FP, HOPE, PONDER, ML and Miranda. There are many different algorithms for type checking in functional languages, differing in the kind of types they allow.

Implicit typing (or type *inference* or type *deduction*) means that the compiler or interpreter of the language will try to infer the types of a program. So in principle, the programmer does not need to supply any type information.

From a software engineering point of view it can be argued that explicit typing is much better because it forces the user to think about what he really wants to do with the functions he defines. However, for simple functions a lot of (error prone) specifying is asked from the programmer and then it is of course very nice if the types are deduced by the system. Since Miranda (the language we use in our examples) has implicit as well as explicit typing we will discuss implicit typing some more and we will give some comments on this combination of both ways of typing.

The type inference algorithms are usually based on the Curry$^+$ type system of the λ-calculus. In the Curry$^+$ type system it is inherently undecidable to determine the type of a λ-term, because this type system contains a special type deduction rule (EQ: if two terms are β-equal they have the same type) which (in programming terms) means that in order to determine the type of an

expression the expression first has to be fully evaluated. So any practical algorithm based on the Curry$^+$ type system must be an approximation.

Milner (Milner (1978)) has given such an approximating algorithm which is used in Miranda. This algorithm uses the Curry type system (i.e. Curry$^+$ without EQ) and with some small extensions. It can deal with polymorphism i.e. a function can be applied on objects of different types (e.g. the identity function delivers an expression of the same type as its argument). But it can not handle *internal* polymorphism i.e. polymorphism of locally defined functions or polymorphism of arguments in the definition itself.

Milner-untypable function definitions:

```
length::      [*] -> num

length []     = 0
length (a:b)  = 1 + length b


fun::              (* -> num) -> * -> * -> num

fun f list1 list2 = f list1 + f list2


missionimpossible = fun length [1,2,3] "abc"
```

In the right-hand-side of the definition of `missionimpossible` the type system does not allow the actual argument `length` to get two different types in the body of the function `fun`. `length` cannot get type `[num] -> num` on `[1,2,3]` and also type `[char] -> num` on `"abc"`. These types are consistent with the definition of `length` but they are not consistent with the definition of `fun`.

Although the deduction scheme can be improved for some cases, a fully satisfactory type inference algorithm is not yet found.

In Miranda type inference and type deduction are combined. A programmer may specify types but he does not need to do so. For this purpose Miranda uses the type checking algorithm of Mycroft (Mycroft (1984)) which only slightly differs from the type deduction algorithm of Milner. At first sight this is a very nice feature. However, in certain cases it may lead to not very admirable situations.

For instance, suppose a programmer gives the following definition:

```
g x = 1 : g (g 'c')
```

The type deduction algorithm of Milner which is used in Miranda cannot deduce the type of `g` because of the internal polymorphism of `g`. `g` is used polymorphic in the right hand side of its own definition, namely with type instances: `char -> [num]` and `[num] -> [num]`. Hence, an error message will be given by the compiler.

However, when the user explicitly specifies the type of `g` as: `* -> [num]` the type checker has no problems with this case: the specified type is accepted and no error message is produced.

For a programmer it might be very confusing when on one hand a program has a type error if types are deduced while on the other hand if a type is explicitly given the program is typed

correctly. This confusion might be prevented if the language has explicit typing only. The programming environment could then have a facility built in that on explicit request of the user does its best trying to infer the type. This would give the user a clearer view on type inference and type checking.

Examples

The power of functional programming languages is illustrated in two examples which use the general features of functional programming languages.

Sorting a list.

The function `sort` needs a list of any type as argument and delivers as result the sorted list which has the same type as the argument.

```
sort::              [*] → [*]

sort []             = []
sort (a:x)          = sort (smalleq a x) ++ [a] ++ sort (greater a x)
```

The functions `smalleq` and `greater` need two arguments: a list and an element. This element must have the same type as the elements of the list.

```
smalleq::           * → [*] → [*]

smalleq a []        = []
smalleq a (b:x)     = cond (b <= a) (b:(smaller a x)) (smaller a x)

greater::           * → [*] → [*]

greater a []        = []
greater a (b:x)     = cond (b > a)  (b:(greater a x)) (greater a x)
```

Obviously, the type of the elements of the list must be such, that the operations '<=' and '>' are well defined.

Roman numbers.

Roman numbers are built up from the characters M, D, C, L, X, V and I Each of these characters has its own value These characters always occur in sorted order, characters with a higher value before characters with a lower value. Exceptions to this rule are a number of 'abbreviations', given below. The value of a Roman number which contains no abbreviations, can be found by adding the values of the characters that occur in the Roman number (MCCLVI = 1000 + 100 + 100 + 50 + 5 + 1 = 1256). The abbreviations make it less simple to calculate the value of a Roman number because now the value of the character depends on its relevant position in the string. Negative numbers, or the number zero cannot be expressed in Roman numbers. The values of the Roman characters and the values of the common abbreviations are

M	=	1000	CM	=	DCCCC	=	M	- C
D	=	500	CD	=	CCCC	=	D	- C
C	=	100	XC	=	LXXXX	=	C	- X
L	=	50	XL	=	XXXX	=	L	- X
X	=	10	IX	=	VIIII	=	X	- I
V	=	5	IV	=	IIII	=	V	- I
I	=	1						

A Roman number is represented as a string. It is assumed that the specified functions are applied on correct arguments only.

```
|| value defines the value of a Roman digit.

value::       char → num

value 'M'     =    1000
value 'D'     =    500
value 'C'     =    100
value 'L'     =    50
value 'X'     =    10
value 'V'     =    5
value 'I'     =    1

|| rconvert converts a Roman number to a decimal number.

rconvert::              [char] → num

rconvert ('C':'M':x)    =    value 'M' - value 'C'    + rconvert x
rconvert ('C':'D':x)    =    value 'D' - value 'C'    + rconvert x
rconvert ('X':'C':x)    =    value 'C' - value 'X'    + rconvert x
rconvert ('X':'L':x)    =    value 'L' - value 'X'    + rconvert x
rconvert ('I':'X':x)    =    value 'X' - value 'I'    + rconvert x
rconvert ('I':'V':x)    =    value 'V' - value 'I'    + rconvert x
rconvert ( a : x)       =    value a                  + rconvert x
rconvert []             =    0
```

DISCUSSION

The following claims have been made in favour of functional programming.

- Functional languages have a sound mathematical basis, with function definition and application as the essential concepts.
- Because of the lack of side-effects, which leads to referential transparency, program correctness proofs are easier. Proofs can be constructed with classical mathematical techniques like induction. Proofs are compositional i.e. properties of a function which are proven, can be used directly in other functions. Also it makes program transformations easier: a programmer can start with a straightforward solution of a problem and via transformations convert this solution to a more efficient one. Therefore it is less difficult to develop reliable software.
- Mainly because of the use of higher order functions the expressive power of functional programming languages is higher than of conventional languages. Programming is more like mathematically specifying the algorithm. Programs are therefore in generally shorter than their conventional counterparts and thus easier to enhance and maintain.

- A functional programming style is a mathematical style. Therefore a functional language is in particular suited as a specification language. The fact that it is also a programming language can be seen as a convenient additional feature. For instance, in Boute (1986) it is shown that the functional paradigm can be used for the description of digital and analog circuits.

- In the evaluation process of an expression it may occur that there is more then one redex in that expression. So, the evaluation process may continue in different ways. The Church-Rosser property states that this evaluation order is unimportant if the choice does not lead to an infinite evaluation process. The Church-Rosser property is a property of the set of definitions, which forms the program of a functional programming language. The Church-

Rosser property is also valid in (subclasses of) the models that are used for functional languages (see also section 1.3 and chapter 2). Due to the Church-Rosser property, alternative evaluation orders, such as parallel evaluation, will never produce a wrong result, although special care has to be taken in order to avoid non-termination. So, functional languages seem to be very suited for parallel evaluation.

There also are the following serious disadvantages which however may be taken away by further research.

- Some algorithms seem to be difficult to express in a functional programming style. Although it is in principle not impossible to do, those programs which have a strongly imperative nature (process control, operating systems, concurrently accessed databases) look not elegant. There is some hope for improvement because functional languages are still under development and because one still is learning how to express algorithms better in these languages.
- Efficient implementations are now becoming available (Johnson (1984), Brus *et al.* (1987)) but they are still somewhat slower than implementations of imperative languages. However, space and time efficiency can still be improved by further applying and improving implementation techniques e.g. strictness analysis, avoidance of space leaks and introduction and implementation of arrays with fast creation, access and update. Furthermore, one has to keep in mind that a certain loss of efficiency was also accepted when high level imperative languages were started to be used.
- From the software engineering point of view functional languages still lack several essential aspects:
  - Firstly, there is no well established programming style. Only the first attempts are made towards establishing such a programming style.
  - Secondly, there is no well established proposal for modularity, although several attempts have been made.
  - Thirdly, there are hardly any debugging facilities yet. Although programs tend to contain relatively few errors, debugging cannot totally be avoided. Especially in lazy languages it is difficult for the programmer to be able to get an idea of the exact execution order which influences the real-time behaviour of a program.
  - Lastly, early programmers training and education is just getting started in an experimental way.

The balance between advantages and disadvantages is such that further research on functional programming languages is more than justified.

1.3 TRADITIONAL MODELS OF COMPUTATION FOR FUNCTIONAL LANGUAGES

In this section we will discuss some traditional models of computation for functional programming languages. For these models some properties are given in relation to other models. All these models are capable of representing every computable function. Each model can be used

as computational model for functional languages. Our choice for a non-traditional model is motivated.

GENERAL TERMINOLOGY: ABSTRACT REDUCTION SYSTEMS

The general terminology which is used in functional languages and in all models for these languages is contained in Abstract Reduction Systems (ARS's). ARS's abstract from the precise structure of the objects and from the way reduction takes place.

An *Abstract Reduction System* (*ARS*) is a pair (E,→), where E is a set of *elements* and → is a binary relation on E. The transitive reflexive closure of → is denoted by $\rightarrow^*$ or $\twoheadrightarrow$. A similar definitions or ARS's can be found in Klop (1987).

The intuitive idea is that each element represents a program at some state of execution. If $x \rightarrow y$ then a next step in the execution could be transforming the element x to y.

We say that x can be *reduced* or *rewritten* to y in one *step*. We also say x *reduces* to y and y is called a *one-step reduct* of x. An element x of an ARS is a *redex* (*red*ucible *ex*pression) if there exists an y such that $x \rightarrow y$. A *reduction sequence* of an ARS is a sequence $x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$. The *length* of this sequence is n. A sequence of length 0 is *empty*.

An element x of an ARS is a *normal form* if for there is no y such that $x \rightarrow y$. An element x *has a normal form* if there is an element y such that $x \twoheadrightarrow y$ and y is a normal form.

Given an ARS (E,→), a *(reduction) strategy* for this system is a function S which takes each $x \in E$ to a set S(x) of nonempty finite reduction sequences, each beginning with x. Note that S(x) can be empty.

A strategy S is *deterministic* if, for all x, S(x) contains at most one element. A strategy S is a *one-step strategy* (or *1-strategy*) if for every x in E, every member of S(x) has length 1.

Write $x \rightarrow_S y$ if S(x) contains a reduction sequence ending with y. By abuse of notation, we may write $x \rightarrow_S y$ to denote some particular but unspecified member of S(x). An *S-sequence* is a reduction sequence of the form $x_0 \rightarrow_S x_1 \rightarrow_S x_2 \rightarrow_S \ldots\ldots$. A strategy S is *normalising* if for all x_0 having a normal form any S-sequence $x_0 \rightarrow_S x_1 \rightarrow_S x_2 \rightarrow_S \ldots\ldots$ must eventually terminate with a normal form.

A *reducer* with strategy S is a process which starts with an element x, chooses a reduction sequence of S(x) and repeats this with the end of the chosen reduction sequence as the new element x. A *result* x of a reducer is reached when S(x) is empty. Reducers are deterministic or non-deterministic. A reducer is *deterministic* if a reduction sequence in S(x) is chosen via a function.

An ARS is *confluent* or *has the Church-Rosser property* (*is Church-Rosser*) if for all elements x, y and z for which $x \twoheadrightarrow y$ and $x \twoheadrightarrow z$, there exists a c such that $y \twoheadrightarrow c$ and $z \twoheadrightarrow c$. It can be proven that confluent ARS's have the *unique normal form property* i.e. each element has atmost

one normal form. The unique normal form property implies that different normalising reducers all will have the same result if there is a normal form. This is useful for implementations which can improve efficiency by using other (possibly parallel) reduction orders.

In Klop (1987) many other interesting properties of ARS's can be found.

LAMBDA-CALCULUS

The theory of the λ-calculus (an overview is given in Barendregt (1984)) as a model of computation has been introduced by Church in 1936 (the very same year in which Turing introduced the Turing machine model of computation) and is further investigated since. The basic concepts of λ-calculus are application and abstraction. The set of terms Λ is the following.

λ-calculus: terms

Every constant c of the set of constants C is a term. Every variable v of the set of variables V is a term.
If M and N are terms then M N is also a term. (application)
If M is a term and x is a variable then λx . M is also a term. (abstraction)

Reduction is done by substitution of the variables (like in e.g. `f(x)= x + x; f(2)= 2 + 2`). The reduction relation is defined as follows.

λ-calculus: reduction relation (β-reduction)

$$(\lambda x.M)\ N \rightarrow_\beta M\ [x:=N]$$

where `[x:=N]` denotes substitute N for x. This reduction relation is closed under contexts, e.g. if $M \rightarrow_\beta N$ then also $\lambda x.M\ Z \rightarrow_\beta \lambda x.N\ Z$.

The abstract reduction system corresponding to λ-calculus is of course $(\Lambda, \rightarrow_\beta)$. The substitution mechanism is not at all trivial so in implementations it has to be taken care of in a special way. Not all occurrences of the variable may be substituted (only so called *free* occurrences). Furthermore it might be necessary to change the names of some variables during substitution. Changing the name of a variable is also called α-conversion.

λ-calculus: substitution

$$(\lambda x.x\ (\lambda x.x)\ x)\ c \rightarrow_\beta x\ (\lambda x.x)\ x\ [x:=c] = c\ (\lambda x.x)\ c$$
$$(\lambda x.(\lambda y.y\ x))\ y \rightarrow_\beta (\lambda y.y\ x)\ [x:=y] =_\alpha (\lambda z.z\ x)\ [x:=y] = \lambda z.z\ y$$

Lazy evaluation (leftmost) is normalising for the λK-calculus. Eager evaluation is not normalising for the λK-calculus. In λI-calculus terms which have a normal form, have no non-terminating reduction order. So, eager evaluation is normalising for λI-calculus. Definitions of λK- and λI-calculus can be found in Barendregt (1984).

λ-calculus is confluent. So alternative (possibly parallel) orders of evaluation can never yield a wrong result. Extending λ-calculus with so called delta rules which act on normal forms and use some internal representation to produce a result (e.g. arithmetical functions), is still confluent (Barendregt (1988)).

λ-calculus can be implemented using sharing (Wadsworth (1971)). It is very difficult to do this optimally avoiding copying of redexes always. The following term communicated to us by J.J. Lévy, illustrates the essential issues for this optimality problem:

```
(λz.z (λx.x) z) (λf.(λz.z ((λx.x) z)) (f c))
```

Traditionally, the λ-calculus is considered to be a suitable model for functional languages (Peyton Jones (1988b)). However, certain aspects of functional languages and the way they are usually implemented, cannot be modelled within this calculus. In particular, the λ-calculus lacks explicit recursion, pattern matching and the explicit notion of sharing of computations. With λ-calculus it is impossible to reason about these aspects which, in our opinion, are essential for the languages and their implementation.

COMBINATORY LOGIC

Combinatory logic (Schönfinkel (1924)) is closely related to λ-calculus. In λ-calculus terms a *combinator* is a λ-term without free variables. All recursive functions can be defined with applications of only two basic combinators S and K which are defined below.

SK-combinatory logic: terms

S is a term and K is a term; if M and N are terms then M N is also a term (application).

SK-combinatory logic: the reduction relation follows from the definitions of S and K in λ-calculus: S ≡ λx.λy.λz.x z (y z) and K ≡ λx.λy.x.

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

Of course, it is impossible to make an efficient implementation based on the combinators S and K only. A simple function needs already lots of S's and K's.

A simple algorithm for translating λ-calculus to S- and K-combinators is the following. This is one of the several algorithms which are called *bracket-abstraction* because the translation of a λ-term x to a combinator term is indicated by [x].

A bracket-abstraction algorithm:

```
[λx.x]       ≡   S K K
[λx.y]       ≡   K y
[λx.P Q]     ≡   S [λx.P] [λx.Q]
[P Q]        ≡   [P] [Q]
```

The abstraction algorithm reveals an important intuitive aspect of the combinators. They distribute an argument over the function body; K means the argument is not needed here and S means distribute the argument over both parts of the application. This intuitive idea is applied more clearly in the concept of Director Strings (Kennaway & Sleep (1988)).

David Turner was the first to introduce an implementation of a functional language using combinators as an intermediate language (Turner (1979a)). He did not use the set of two combinators mentioned above but he used an extended set of about 15 combinators and of course some delta-rules. His paper induced much research on which set of combinators was best for

implementation purposes and it also induced research on machine architectures based on such sets of combinators. This research was very colourful with concepts as supercombinators and superdoopercombinators and machines with names such as NORMA and SKIM. However it turned out that using an intermediate language with graph primitives and special compiler techniques it was possible to get at least an order of magnitude increase in efficiency beating on an ordinary VAX the expected performance of special-purpose hardware based on these combinators.

An interesting more recently proposed combinator model is built with the so called *categorical combinators* (Curien (1986)) based on a category theoretical model of the λ-calculus. These combinators encode an environmental implementation of the λ-calculus using eager evaluation.

Leftmost-outermost reduction implemented with sharing is normalising and optimal for combinator systems.

Just as λ-calculus, combinatory logic lacks explicit recursion, pattern matching and the explicit notion of sharing of computations. With combinatory logic it is impossible to reason about these aspects which, in our opinion, are essential for the languages and their implementation.

1.4 CONCLUSION

We believe that the use of the declarative paradigm (and functional languages in particular) might contribute to solving the software crisis.

In our opinion both λ-calculus and combinatory logic are not the best suited models of computation for functional languages and their implementation because they lack recursion, pattern matching and sharing. Therefore, we will consider term and graph rewriting systems in the next chapter.

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.

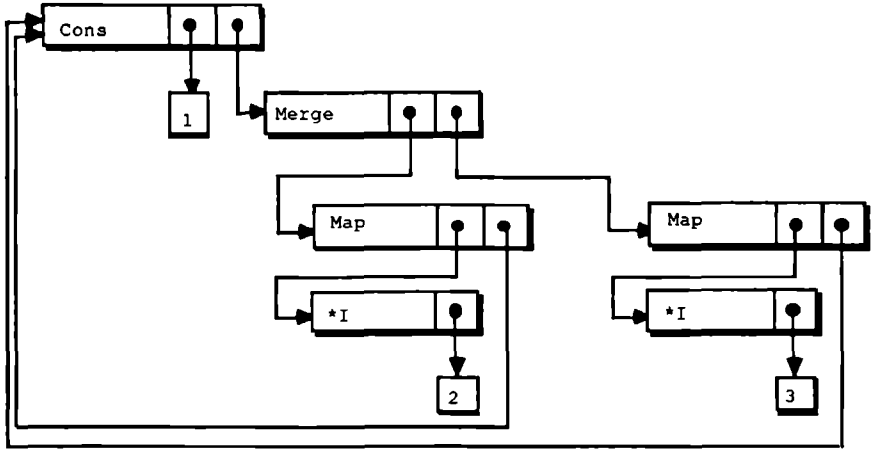
REWRITING

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.

3

SPECIFICATION OF REDUCTION STRATEGIES IN TERM REWRITING SYSTEMS

M.C.J.D. van Eekelen, M.J. Plasmeijer.

Nijmegen University, The Netherlands.

partially supported by the Dutch Parallel Reduction Machine Project.

Abstract.

There is a growing interest in Term Rewriting Systems (TRS's), which are used as a conceptual basis for new programming languages such as functional languages and algebraic specification languages. TRS's serve as a computational model for (parallel) implementations of these languages. They also form the foundation for a calculus for Graph Rewriting Systems (GRS's) In Rewriting Systems reduction strategies play an important role because they control the actual rewriting process. Strategies determine the order of the rewriting and the rules to apply Hence they have a great influence on the efficiency and the amount of parallelism in the computation. In ambiguous or non-deterministic TRS's, they even influence the outcome of the computation. Some of the reduction strategies used in TRS's are extremely complex algorithms. Unfortunately, there is no common formal specification method for reduction strategies yet.

In this paper three formal methods for specifying reduction strategies in TRS's are presented. In the first method the reduction strategy is encoded in the TRS itself The original TRS is transformed to a so called annotation TRS in which the strategy is encoded using functions. This annotation TRS itself may use any normalizing reduction strategy. Unfortunately, compared with the number of rules of the original TRS, the annotation TRS may contain an exponential number of additional rules. In the second method this drawback is prevented, simply by using a priority TRS as annotation TRS The desire to specify a strategy uniformly for all TRS's leads to the third method. A new TRS system is introduced that uses two basic primitives for matching and rewriting and that is build out of three separate TRS's The use of this abstract-interpretation TRS is shown to be the most promising method.

3.1 INTRODUCTION

A Term Rewriting System (TRS) consists of a *term* τ to rewrite and a set ρ of *rewrite rules*. A *reduction strategy* σ is an algorithm which determines in which order the rewritable subterms of the term (the so called redexes i.e. reducible expressions) have to be rewritten and which rules have to be applied. A TRS with a strategy will be called a term *reducer* or a term *rewriter*. A *redex* is a subterm which matches the left-hand-side (LHS) of a rewrite rule. TRS's consist of *variables* and *symbols*. Symbols start with an upper case character. All symbols that occur as left-most symbol of a rewrite rule are *functions*, other symbols are *constructors*.

For instance when we have the following well-known set of rewrite rules:

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

the term Ap (Ap (Ap K S) S) (Ap (Ap K K) K) can be rewritten to Ap S (Ap (Ap K K) K) and finally to Ap S K by applying two times the rule (K). In this example Ap is a function, S and K are constructors and x, y and z are variables.

A thorough introduction to TRS's is given in Klop (1985). TRS based languages like DACTL (Glauert *et al.* (1985)) and Lean (Barendregt *et al.* (1986b)) are used as a computational model for implementations of new languages such as Miranda (Turner (1985)) and OBJ (Futatsugi *et al.* (1985)). Currently various attempts (Barendregt *et al.* (1986a) based on Raoult (1984) and van den Broek & van der Hoeven (1986) based on Ehrig (1979)) are made to extend TRS's to a calculus for Graph Rewriting Systems (GRS's), in order to make them serve as a computational model of (parallel) graph reducing implementations.

A strategy in a TRS can be compared with the control flow in an ordinary imperative programming language. In the example above the strategy recursively takes the left-most redex in the term until the term after rewriting contains no redexes anymore and therefore it is in *normal form*. The outcome of the rewriting process may depend on the strategy followed. In particular this is the case if the TRS is ambiguous.

We distinguish the following forms of *ambiguity*:

- *Non-deterministic* type of rules, i.e. rules of which the LHS's match the same instance, such as

```
Choose x y    →x
Choose x y    →y
```

- *Strongly ambiguous* rules of which (an instance of) a subterm of a LHS is a redex, e.g.

```
G (K n 1)     →1
K x y         →x
```

  or, more subtle

```
F (F x)       →...
```

  which is ambiguous with itself.

For TRS's like these the outcome is depending on the strategy that is followed. Non-deterministic rules such as Choose might be useful but, of course, the strategy must support it by making a non-deterministic choice between the rules. Another example of a useful ambiguous definition is the following TRS which still has the *Church-Rosser* property (i.e. the system has an unique normal form):

```
Or True x          →True
Or x True          →True
Or False False     →False
```

Although in the case that the TRS is Church-Rosser, the outcome of the computation will not depend on the order in which redexes are chosen, it can happen that depending on the strategy

followed the rewrite process may or may not *terminate*. Take for instance Berry's example which has a unique normal form:

```
F x 0 1        →x
F 0 x 0        →x
F 1 1 x        →x
```

Let's assume that of the actual arguments of F in the term there are two arguments which reduce to the indicated normal form while the other argument has no normal form at all. The most efficient strategy would not touch the latter. Unfortunately, on forehand we do not know which argument that is. Forcing argument evaluation if a pattern is specified on the corresponding position in a rule will start a non terminating computation. The only safe way to reach the normal form is to reduce each argument of F once to see if the term has become a redex (the parallel outermost strategy). This strategy is not always the most efficient one.

Another example which shows the importance of strategies is the following program with a pretty familiar appearance:

```
Fac 0    →1                          (1)
Fac n    →* n (Fac (- n 1))          (2)
```

It only has the obvious semantics if the right strategy is chosen. In general the rules are ambiguous because Fac 0 matches both rules. Fac (- 1 1) even matches the wrong rule. However a valid definition of Fac can be obtained if the argument of Fac is evaluated on forehand and the first rule has priority over the second rule. Now Fac (- 1 1) will be reduced to Fac 0 and the priority of the rules guarantees that Fac 0 now matches only the first rule. The reduction strategies in most functional languages are of this type.

Hence we must conclude that, unfortunately, there is no best strategy for all TRS's. Safe strategies are not always in all cases efficient. Efficient strategies are not always in all cases safe. Some algorithms can be expressed more conveniently in one strategy than another. Also one could prefer some strategies for specific reasons, e.g. for a parallel architecture one would like to reduce as much redexes as possible in parallel. One could also imagine, as is proposed in Lean, that several strategies are mixed in one and the same TRS in order to optimize performance and descriptive power. All this gives rise to complex strategies which cannot be explained anymore in terms like "take the left-most redex". Hence there is a need to specify strategies formally.

Before we will specify reduction strategies we must first ask ourselves the question what the properties must be of a good strategy specification. Of course it must specify which redexes are to be rewritten in which order and which of the matching rules have to be applied. If we see the specification as an algorithm, the execution of that algorithm must reduce exactly the same redexes in exactly the same order as the original strategy does. There is however a problem in the case of *parallel* reduction strategies which is caused by the concept time. The question arises if two redexes, which according to the original strategy have to be reduced in parallel, also have to be executed at the same time when we execute the corresponding specification.

This consideration leads to two views on parallel reduction: *synchronous* and *asynchronous*. According to the synchronous view the strategy recursively determines one or more redexes which have to be reduced in parallel after which these redexes are all rewritten in one step. This view is often used in a theoretical framework but in practice there are only very few machines for which this view is applicable (e.g. Magó (1980)). In most distributed environments the strategy algorithm is also implemented distributed and the asynchronous view is therefore more appropriate. In the asynchronous view redexes which are reduced in parallel are actually reduced in any (*possibly* parallel) order while the strategy may already have determined new redexes although not all previously assigned redexes were rewritten. In the asynchronous view we cannot state that the execution of the specification must rewrite *all* redexes in the same order as the original strategy would do. A specification can only model the dynamic behaviour. Though in this paper the specifications are used mainly in a sequential context, they can be easily extended to a parallel environment with the synchronous view as well as using the asynchronous view.

There are many ways in which strategies can be specified. In this paper three methods are introduced and a comparison between them is made. All methods create TRS's which can be reduced with any normalizing strategy.

1. Specification of the strategy by transforming the TRS τ to another one annotation-τ; reducing the latter induces the wanted strategy on τ.
2. Same as 1., but now in annotation-τ there is a priority in the rewrite rules (specificity).
3. The strategy is specified in a separate TRS on a different level.

For each method we will work out as a simple example the TRS consisting of the Curry combinators K and S (notated with explicit application functions). The strategy to be specified will be head reduction, i.e. left-most reduction to head normal form. A term is in head normal form if no reduction can possibly lead to a rewrite of the head of the term. We will refer to the example as Curry's example. For every method Curry's example will be proven correct. These proofs will be using O'Donnell's definition of a TRS simulating another TRS (O'Donnell (1985)). The methods are compared in terms of ease of correctness proving, ease of developing, ease of expressing efficient strategies etcetera.

3.2 TRANSFORMING A TRS TO AN ANNOTATION TRS

3.2.1 DESCRIPTION OF THE METHOD

The method is inspired by the use of strictness annotations in functional programming languages (Miranda e.g.) which serve as compiler directives. The general idea is that we use such annotations, for instance a shriek "!", for labeling those redex(es) in the term that should be rewritten. Instead of directives we use functions. These functions have ordinary semantics which enables us to use the full TRS semantics to reason about their meaning. Therefore the original TRS is transformed to an annotation TRS. All rules in this annotation TRS start with these functions and therefore only those redexes of the original TRS which are "annotated", are also

redexes in the annotation TRS. The annotation TRS itself can be reduced with any normalizing reduction strategy.

A small example: if the rules of the original TRS are:

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

Then these rules are transformed to:

```
! (Ap (Ap (Ap S x) y) z)    →Ap (Ap x z) (Ap y z)        (1)
! (Ap (Ap K x) y)           →x                           (2)
```

This TRS has only one function; the function !. The transformation has as a consequence that functions in the original TRS (Ap in the example) now become constructors in the annotation TRS. Of course, also the term to reduce must be marked. For instance if Ap ((Ap (Ap K S) S) (Ap (Ap K K) K) K) is transformed to Ap (! (Ap (Ap K S) S)) (Ap (Ap K K) K) it has only one redex in the annotation TRS. Due to the applicative order reduction this redex will be reduced giving Ap S (Ap (Ap K K) K) as result.

Hence only one of the redexes that is present in the original TRS, is reduced. This is the one which was marked.

In order to continue this process one also has to insert new markings for those redexes which have to be reduced next. When we specify a strategy in this way quite a lot of additional rules have to be added to deal with markings in the right way. Also the outcome of the computation must be the same as in the original TRS, i.e. all marking must disappear at the end. Let's try to make clear what kind of additional rules are needed by looking at our example.

3.2.2 EXPRESSING HEAD REDUCTION FOR CURRY'S EXAMPLE

We start off with the rules of the ordinary TRS:

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

We add exclamation marks to the left-hand-side of the rules of the original TRS in order to promote the original redexes to redexes in the new system.

```
! (Ap (Ap (Ap S x) y) z)    →Ap (Ap x z) (Ap y z)        (1)
! (Ap (Ap K x) y)           →x                           (2)
```

Note that the function ! has only one argument, hence the parentheses around the argument will be sometimes redundant. When there can be no confusion, we will leave them out.

We need a general propagation rule to achieve a left-most search for a redex.

```
! (Ap (Ap (Ap (Ap v w) x) y) z) →Ap (! (Ap (Ap (Ap v w) x) y)) z      (3)
```

The propagation rule takes care of the propagation of the ! for the case that we have a pattern with four Ap's on the left. We also have to specify what happens for all other cases, i.e. when we have less than four Ap's. They can be divided in two classes:

1) The ! may have an argument which contains no redex at all. In this case the !-function must reduce to its argument in order to get the same normal form as in the original TRS. For each non-redex we have to add a rule. This gives us five additional rules.

```
! S                → S                    (4)
! K                → K                    (5)
! (Ap K x)         → Ap K x               (6)
! (Ap S x)         → Ap S x               (7)
! (Ap (Ap S x) y)  → Ap (Ap S x) y        (8)
```

   Note that we could not write a more general rule like

```
! (Ap x y)    → Ap x y
```

   because this rule is ambiguous with the K rule. This would introduce the danger that ! (Ap (Ap K K) K) is reduced to Ap (Ap K K) K instead of being reduced to K.

2) The ! may have an argument of which a subterm is a redex. We call rules that are introduced to handle these cases *envelope-rules*. In this example only one envelope-rule is needed:

```
! (Ap (Ap (Ap K x) y) z)    → Ap x z          (9)
```

The nine rules we have given so far, model exactly **one** step of the head reduction strategy. In order to model the complete reduction to head normal form, the exclamation mark has to be created over and over in a driver rule like

```
* term  → * (! term)
```

In order to let this reduction stop we must encode in the term whether or not a rewrite (according to the strategy on the original TRS) was done. We use a success constructor '+' to denote that a rewrite was done and a failure constructor '-' to denote that it wasn't. We add the following 'driver' rules

```
* (- term)    → term                (10)
* (+ term)    → * (! term)          (11)
```

The strategy stops if on the term no rewrite was done (10). Of course we must also change all the other rules to make them produce a result with the proper success or failure constructor (+/-). Furthermore we must also propagate these constructors back to the beginning of the term so that we can make the decision whether or not to stop the strategy. This is done by adding the following rules:

```
~ (Ap (- x) y)    → - (Ap x y)          (12)
~ (Ap (+ x) y)    → + (Ap x y)          (13)
```

and by changing the propagation rule to:

```
! (Ap (Ap (Ap (Ap v w) x) y) z) →~ (Ap (! (Ap (Ap (Ap v w) x) y)) z)  (3)
```

Finally the complete set of rules is:

```
! (Ap (Ap (Ap S x) y) z)          →+ (Ap (Ap x z) (Ap y z))            (1)
! (Ap (Ap K x) y)                 →+ x                                 (2)
! (Ap (Ap (Ap (Ap v w) x) y) z)   →~ (Ap (! (Ap (Ap (Ap v w) x) y)) z) (3)
! S                               →- S                                 (4)
! K                               →- K                                 (5)
! (Ap K x)                        →- (Ap K x)                          (6)
! (Ap S x)                        →- (Ap S x)                          (7)
! (Ap (Ap S x) y)                 →- (Ap (Ap S x) y)                   (8)
! (Ap (Ap (Ap K x) y) z)          →+ (Ap x z)                          (9)
* (- term)                        →term                               (10)
* (+ term)                        →* (! term)                         (11)
~ (Ap (- x) y)                    →- (Ap x y)                         (12)
~ (Ap (+ x) y)                    →+ (Ap x y)                         (13)
```

The initial term is

```
* (! term)
```

Note that it was also possible to achieve the same result using only one annotation function in stead of three different functions. This was not done for reasons of clarity.

3.2.3 CORRECTNESS PROOF

In this section we first formally define what it means for a strategy to specify another strategy. This definition will also be used in the proofs of the other methods. Then we proof that Curry's example, as it was specified with this method, satisfies the corresponding conditions and we finish this section with some general remarks on proving specifications that are constructed with this method. We also introduce some new terminology for a special kind of TRS that we encounter.

Definition of Specification

In his excellent book (O'Donnell (1985)) Michael O'Donnell defines what it means for a TRS to simulate another TRS. When one considers a strategy of a TRS to be giving a restriction on the reduction relation, then restricting the reduction relation to what is specified by the strategy, his definition can be used directly as the definition for one term rewriter specifying another. This results in the following definition:

Let $<\tau_O, \rightarrow_\sigma>$ and $<\tau_S, \rightarrow_\pi>$ be reducers where $\rightarrow_\sigma$ and $\rightarrow_\pi$ are the reduction relations of τ_O (the original TRS) and τ_S (the specification TRS) restricted to the strategies σ and π, then $\tau_{S\pi}$ *specifies* $\tau_{O\sigma}$ if there exist

an *encoding set* $E \subseteq \tau_S$,
a *decoding function* $d: \tau_S \rightarrow \tau_O \cup \{nil\}$ with $nil \notin \tau_O$,
a *computation relation* $\rightarrow_C \subseteq \rightarrow_S$,

such that

1. $d[E] = \tau_O \,\&\, d^{-1}[N\tau_{O\sigma}] \cap E \subseteq N\tau_{S\pi}$
   where $N\tau_{X\psi}$ is the set of normal forms of a TRS τ_X with respect to the strategy ψ,
2. $\forall \alpha, \beta \in \tau_S \;\; \alpha \rightarrow_C \beta \;\Rightarrow\; d(\alpha) \rightarrow_O d(\beta)$,
3. $\forall \alpha, \beta \in \tau_S \;\; \alpha \,(\rightarrow_{S\pi} \text{-} \rightarrow_C)\, \beta \;\Rightarrow\; d(\alpha) = d(\beta)$
   where $\rightarrow_{S\pi} \text{-} \rightarrow_C$ stands for any $S\pi$ reduction that is not a c reduction,
4. $\forall \alpha \in E, \beta \in \tau_O \;\; d(\alpha) \rightarrow_{O\sigma} \beta \;\Rightarrow\; \exists \delta \in E \;\; \alpha \,(\rightarrow_{S\pi} \text{-} \rightarrow_C)^* \rightarrow_C (\rightarrow_{S\pi} \text{-} \rightarrow_C)^* \delta \;\&\; d(\delta) = \beta$,
5. $\forall \alpha \in N\tau_{S\pi} \;\; d(\alpha) \in N\tau_{O\sigma} \cup \{nil\}$,
6. There is no infinite $(\rightarrow_{S\pi} \text{-} \rightarrow_C)$ reduction path and moreover there exists a bounded function $b: \tau_O \rightarrow \mathbb{N}$ such that $\forall \alpha, \beta \in E \;\; \alpha \,(\rightarrow_{S\pi} \text{-} \rightarrow_C)^m \rightarrow_C (\rightarrow_{S\pi} \text{-} \rightarrow_C)^n \beta \;\Rightarrow\; m < b(d(\alpha)) \;\&\; n < b(d(\alpha))$.

The specification is called *effective* if b, $\rightarrow_C$, d and E are all total computable.

Intuitively, this means that we encode terms into another TRS allowing multiple encodings. The first condition requires that decoding respect normal forms. The specifying TRS has computational reductions which mirror a rewrite in the original TRS and non-computational reductions which are internal book-keeping steps. Conditions 2,3 and 4 require that every original reduction is simulated by any number of book-keeping steps which do not change the encoded expression, followed by exactly one computational reduction to effect the change in the encoded expression. This reduction may again be followed by book-keeping steps. Condition 5 prevents dead ends in the specification. Condition 6 states that the number of book-keeping steps is not allowed to grow without bound, otherwise e.g. all possible original reduction sequences could be simulated before officially choosing one of them, which clearly is not what we want. All conditions together mean that we really simulate the steps of the original TRS and not only return the appropriate normal forms as a result.

For the function d we basically need to describe which term is encoded for all reducts of the elements of the encoding set, but we also have to define which terms are not reducts of elements of E and are therefore decoded to nil. Maybe it is possible to give a precise definition of a TRS simulating another one using a decoding function which is only defined on reducts of elements of E. In the future we would like to do research along this line to find out whether for our purposes it is possible to simplify this definition.

Proof of Curry's Example

We define the encoding set E as $N\tau_{O\sigma} \cup \{ * (! \, t) \mid t \in \tau_O\text{-}N\tau_{O\sigma} \}$, where $\tau_O\text{-}N\tau_{O\sigma}$ stands for the set of all terms of τ_O that are not terms of $N\tau_{O\sigma}$. The set of constructors *,+,-,!,~ is called A. The decoding function d is defined as follows:

$x \in N\tau_{O\sigma}$	$\Rightarrow$	$d(x) = x$	
x matches one of the rules of τ_S and the subterms matching variables do not contain elements of A	$\Rightarrow$	$d(x) = x$	with all applications of elements of A skipped
in all other cases		$d(x) = nil$.	

The computation relation c is given by rules (1), (2) and (9). This leaves us to proof that all conditions are satisfied. When we are not interested in the strategy π of $\tau_{S\pi}$ (i.e. the statement holds for any normalizing strategy of τ_S), we will leave out the strategy suffix π.

1a. $d[E] = \tau_O$
Clear from the definitions.

1b. $d^{-1}[N\tau_{O\sigma}] \cap E \subseteq N\tau_S$
Suppose we have $\alpha \in E$ and $\beta \in N\tau_{O\sigma}$ with $d(\alpha) = \beta$, then $\alpha = \beta$ and because α does not contain any elements of A, $\alpha \in N\tau_S$.

2. $\forall \alpha, \beta \in \tau_S \ \alpha \rightarrow_c \beta \Rightarrow d(\alpha) \rightarrow_O d(\beta)$
Reductions according to the rules (1), (2) and (9) decode into S and K reductions.

3. $\forall \alpha, \beta \in \tau_S \ \alpha (\rightarrow_S \text{-} \rightarrow_c) \beta \Rightarrow d(\alpha) = d(\beta)$
Follows directly from the definition of d.

4. $\forall \alpha \in E, \beta \in \tau_O \ d(\alpha) \rightarrow_{O\sigma} \beta \Rightarrow \exists \delta \in E \ \alpha (\rightarrow_S \text{-} \rightarrow_c)^* \rightarrow_c (\rightarrow_S \text{-} \rightarrow_c)^* \delta \ \& \ d(\delta) = \beta$
Suppose we have $\alpha \in E$, $d(\alpha) \rightarrow_{O\sigma} \beta$; then $d(\alpha) \notin N\tau_{O\sigma}$, hence α is of the form * (! ι) with $t \notin N\tau_{O\sigma}$. So $d(\alpha) = t$ and t reduces in $\tau_{O\sigma}$ to β. We have to find a $\delta \in \tau_S$ such that $d(\delta) = \beta$ **and** $\alpha (\rightarrow_S \text{-} \rightarrow_c)^* \rightarrow_c (\rightarrow_S \text{-} \rightarrow_c)^* \delta$; $t \notin N\tau_{O\sigma}$, so t has a head-redex r in τ_O, hence t is of the form Ap^1 (... Ap^n (Ap (Ap (Ap S x) y) z) arg_n) ... arg_1 or Ap^1 (... Ap^n (Ap (Ap K x) y) arg_n) ... arg_1. Consequently, the form of β is also known (apply S or K rule). Take δ = * (! β) then trivially $d(\delta) = \beta$.

Let us first assume that r is a S redex. If $n > 0$ then the only rule that is applicable on α is rule (3) which is not a computational rule. After one step this reduces to ~ (Ap^1 (! (Ap^2 (... Ap^n (Ap (Ap (Ap S x) y) z) arg_n) ... arg_1) and again if $n-1 > 0$ only rule (3) is applicable. So after n steps the result is ~ (Ap^1 (~ (Ap^2 (... ~ (Ap^n (! (Ap (Ap (Ap S x) y) z)) arg_n) ... arg_1). These n steps are the internal steps of τ_S preceding the computational step which is applying rule (1), resulting into ~ (Ap^1 (~ (Ap^2 (... ~ (Ap^n (+ (Ap (Ap x y) (Ap x z)) arg_n) ... arg_1). Now the only possible reductions are n applications of rule (13), which gives us * (+ β), which can only reduce to * (! β) = δ.

The other case we look at, is r being a K redex. We now have to be careful because there can be two computational rules corresponding to a K reduction. If $n \geq 1$, then we can follow the same reasoning as in the S case. We have n-1 applications of rule (3) on application of rule (9) and n-1 applications of rule (13), followed by one application of rule (11); if n=0, then we can only apply directly the computational rule (2) and we have only one internal step from * (+ β) to * (! β).

5. $\forall \alpha \in N\tau_S \ d(\alpha) \in N\tau_{O\sigma} \cup \{nil\}$
Take $\alpha \in N\tau_S$ then either α is also a term of $N\tau_{O\sigma}$ and then $d(\alpha) = \alpha$ and $d(\alpha) \in N\tau_{O\sigma}$ or α is not a term of $N\tau_{O\sigma}$ and then $d(\alpha)$ is nil because α is not a redex in τ_S.

6. There is no infinite $(\rightarrow_S \text{-} \rightarrow_c)$ reduction path and moreover there exists a bounded function b: $\tau_O \rightarrow \mathbb{N}$ such that $\forall \alpha, \beta \in E \ \alpha (\rightarrow_S \text{-} \rightarrow_c)^m \rightarrow_c (\rightarrow_S \text{-} \rightarrow_c)^n \beta \Rightarrow m < b(d(\alpha)) \ \& \ n < b(d(\alpha))$.

In the proof of 4 it is shown implicitly that there is no infinite internal reduction path. We define b as b(t) = 3 * (the number of Ap's on the spine) + 1. Take $\alpha, \beta \in E$ then β can be an element of $N_{O\sigma}$ or not: $\beta \notin N\tau_{O\sigma} \Rightarrow \beta = *\,(!\, d(\beta))$ and in the proof of 4 we saw that both m and n are less than the number of Ap's on the spine in respectively α and β. So certainly $m < b(d(\alpha))$ and $n < b(d(\beta))$; analogously to the reasoning in 4 one can easily show that $\beta \in N\tau_{O\sigma} \Rightarrow \alpha\,(\rightarrow_S \text{-} \rightarrow_C)^{m1} \rightarrow_C (\rightarrow_S \text{-} \rightarrow_C)^{n1}\,(*\,(+\,\beta))\ (\rightarrow_S \text{-} \rightarrow_C)^{m2}\,\gamma\,(\rightarrow_S \text{-} \rightarrow_C)^{n2}\,(*\,(-\,\beta))$ where γ is a term not containing ! and n1, n2, m1 and m2 are all less or equal to the number of Ap's on the spine of α, β and γ respectively. Proving this we use the fact that rules (1), (2), (3), (4), (5), (6) and (9) cover all possible terms ! t with $t \in \tau_{O\sigma}$.

The specification is *effective* because b, $\rightarrow_C$, d and E are all total computable.

Remarks

Several parts of the proof heavily relied on the specific rules, the specific structure of the term and of course on the specific strategy. We see no way to easily extend these proofs to other reducers.

Note that starting with terms * (! t) every possible reduct had at most one redex. We call a term with such a property *linear*, because the reduction path is linear. A TRS where all terms are linear is called a *linear* TRS. For those who think this is confusing considering the concept of left-linearity, i.e. having repeated variables on the left-hand-side, we suggest *left-comparing* as a possibly better name in stead of left-linear. It does not have the relation to polynomial terms (linear, quadratic etc.) and moreover it names the essential aspect of left-linearity which is that arguments have to be identical in order to be able to apply the rule. Linear TRS's have good prospects for efficient sequential execution, and left-comparing TRS's can cause trouble during execution when used with infinitely growing arguments.

3.2.4 EVALUATION OF THE METHOD

Although it is possible to convert the original TRS to an annotation TRS in which the strategy is explicitly encoded, the method has severe drawbacks.

First of all our annotation TRS has many more rules than the original TRS. Unfortunately the number of extra 'non-redex' rules per function can become exponential: the number of constants, that can occur, to the power of the width of the pattern. This results in too many rules. Some of these rules are very awkward such as the envelope rule. Furthermore the rules in the annotation TRS not only depend on the strategy but also on the original TRS. Consequently the correctness of the strategy can not be proven for all TRS's at once, but it must be proven for every TRS separately. Furthermore it is rather tedious work to give such a proof.

Concluding we can state that though it is very well possible to express a strategy in an annotation TRS, the number of rules and their complexity makes this method not very suitable for practical use.

3.3 TRANSFORMING THE TRS TO AN ANNOTATION TRS WITH PRIORITY RULES

3.3.1 DESCRIPTION OF THE METHOD

The method we will describe in this section is very closely related to the way strategies are expressed in Dactl (Glauert *et al.* (1985)). It differs from the previous method in that we will use a different kind of annotation TRS namely an annotation TRS with priority rules (Baeten *et al.* (1986)), a so called priority rewriting system (PRS). The difference with an ordinary TRS is that whenever a rule matches a given redex, it may only be chosen if none of the rules with higher priority can ever be applicable on the (internally rewritten) term.

3.3.2 EXPRESSING HEAD REDUCTION FOR CURRY'S EXAMPLE

We start again with the rules of the ordinary TRS

```
Ap (Ap (Ap S x) y) z        →Ap (Ap x z) (Ap y z)        (A)
Ap (Ap K x) y               →x                           (B)
```

and we add exclamation marks to the rules

```
! (Ap (Ap (Ap S x) y) z)    →Ap (Ap x z) (Ap y z)        (1)
! (Ap (Ap K x) y)           →x                           (2)
```

Our propagation rule is much simpler now:

```
! (Ap x y)                  →Ap (! x) y                  (3)
```

The reason for this is that if a redex matches the third rule **and** one of the other two rules, the topmost rule (**not** rule 3) is taken because It has higher priority. We indicate this by putting an arrow in front of rules with decreasing priority. Rules with the same priority are indicated by adding a bar in front of them. Rules without any indication are not affected by the priority mechanism.

The non-redex rules are extremely simple now:

```
! x                         →x                           (4)
```

Anything that does not match one of the other rules, is not a redex. That's it! The somewhat strange envelope-rule of the previous section is also not necessary any more.

So we have elegantly modelled one step of this strategy. In order to model the strategy completely we still must encode whether or not a subterm was successfully rewritten. This is modelled by exactly the same changes and extra rules as in the previous section. Note that in general we must be careful with adding rules because our PRS could cause that they will never be applied. In this case we have no problems with that issue.

The complete set is now:

```
↓ | ' (Ap (Ap (Ap S x) y) z)     →+ (Ap (Ap x z) (Ap y z))      (1)
↓ | ' (Ap (Ap K x) y)            →+ x                           (2)
↓   ' (Ap x y)                   →~ (Ap (' x) y)                (3)
↓   ' x                          →- x                           (4)
    * (- term)                   →term                          (5)
    * (+ term)                   →* (' term)                    (6)
    ~ (Ap (- x) y)               →- (Ap x y)                    (7)
    ~ (Ap (+ x) y)               →+ (Ap x y)                    (8)
```

Again, the initial term must be of the form

```
* (' term)
```

Note that only the third and the fourth rule really rely on the order in which rules are matched

3.3.3 CORRECTNESS PROOF

In order to prove this specification we first define what the semantics of a PRS is Then we will show that in this case the semantics of the PRS is equivalent to the semantics of a TRS without priority. This TRS will turn out to be equivalent to the one we constructed using the previous method.

Semantics of a PRS

A PRS has a unique semantics (is *well-defined*) if it has a unique sound and complete rewrite set. Unfortunately our underlying TRS is not strongly normalizing nor bounded So we have to use the stabilization lemma formulated in Baeten *et al* (1986) We have to prove that starting with $R^{\Omega} = R_{max}$ (R_{max} being the set of all possible rewrites of τ_S i e. all possible instances of all rules not considering the priority), there exists a $\underline{n}$ for which $R^{\underline{n}} = R^{\underline{n+1}}$, where $R^{\underline{n+1}}$ is defined as $(R^{\underline{n}})^C$ with $(R)^C$ being the set of all rewrites that is correct with respect to R A rewrite $t \rightarrow_r s$ is *correct* if there is no internal R-reduction $t \rightarrow^* t'$ to an r'-rewrite $t' \rightarrow_{r'} s' \in R$ with $r' > r$ (r' has higher priority than r) R is *sound* if all rewrites which are an element of R, are correct w r t R R is *complete* if it contains all possible rewrites of R_{max} which are correct w r t R

Proof

Because this method resembles the previous one, we can take all definitions the same except for the computation relation c which will be given by rules 1 and 2 only But before we can even start to say something about this solution, we must prove that this PRS has a unique semantics.

If we restrict ourselves to those terms t for which $d(t) \neq nil$, then one can easily see that the term has at most one redex for which several rules with different priorities can be applied Of course, the intended semantics is that the rule with the highest priority is applied Since there is always at most one redex, there are no internal reductions of more than zero steps Yet there are trivial zero-step reductions using other instantiations for the variables. So we start with the set R_{max} and compute $(R_{max})^C$ We prove that it is sound and complete, hence $((R_{max})^C)^C = (R_{max})^C$ and $(R_{max})^C$ is the semantics of our PRS

We want to determine the set of rewrites that are correct w.r.t. R_{max}. As was already stated, the only internal redexes to be considered are other instantiations of variables. We know the structure of the terms, so all instantiations of the left-hand-side of rule (4) are ruled out by rule (3) except for the instantiations S and K. So the PRS is equivalent to another one where rule (4) is substituted by the rules:

```
! S          → ~ S                                                    (4')
! K          → ~ K                                                    (4'')
```

In the same way we can determine the patterns for which rule (3) is really applicable i.e. not ruled out by rule (1) or rule (2), giving the following rules replacing rule (3):

```
! (Ap S y)                          → ~ (Ap (! S) y)                          (3')
! (Ap K y)                          → ~ (Ap (! K) y)                          (3'')
! (Ap (Ap S a) y)                   → ~ (Ap (! (Ap S a)) y)                   (3''')
! (Ap (Ap (Ap K a) b) y)            → ~ (Ap (! (Ap (Ap K a) b)) y)            (3'''')
! (Ap (Ap (Ap (Ap a b) c) d) y)     → ~ (Ap (! (Ap (Ap (Ap a b) c) d)) y)
                                                                            (3''''')
```

So the set $(R_{max})^c$ is the set determined by this new TRS without priority. This set is sound because evidently all rewrites in it are correct and it is complete because it also contains all rewrites which are correct with respect to it. One easily checks that the resulting rules that determine the semantics of the PRS, are equivalent to the rules of the previous example. So we have proven that our PRS model specifies Curry's example correctly.

Remarks

Note that this proof more or less included the proof for the previous method and that we needed special analysis to determine the semantics of this PRS. In general there is not even always a unique semantics and it is not known whether a semantics always exists (Baeten *et al.* (1986)).

3.3.4 EVALUATION OF THE METHOD

Clearly this method is a lot better than the previous one. There are more rules in the annotation TRS than in the original TRS, but the growth is no longer exponential because the non redex case can now be expressed with one rule (4). Also the funny envelope rules have disappeared.

Still this method has a severe disadvantage. The rules generated still heavily depend on the original TRS so if we want the same strategy for other TRS's we still have to change the rules of each one of them. The proof must be given for every TRS separately as was the case with the previous method. The complexity of the proof is even worse than the complexity of the previous proof because we also have to prove the well-definedness of the PRS. It would be better if we could define a strategy in such a way that the same description is valid for all TRS's.

3.4 DEFINING THE STRATEGY SEPARATELY IN AN ABSTRACT-INTERPRETATION TRS

3.4.1 DESCRIPTION OF THE METHOD

In order to be able to specify strategies uniformly for all TRS's we will consider three different conceptual levels of TRS's:

a) The first TRS is the *user-defined TRS*. This is the original TRS which now remains unchanged. An example of such a TRS is the S-K TRS.

b) The reduction strategy for the user-defined TRS is specified in a separate TRS called the *interpretation TRS*. Functions in the interpretation TRS may have a user-defined TRS or any subterm of such a TRS as argument. For instance one may write:

HeadReduce *S* $\rightarrow S$ (I-a)
HeadReduce *K* $\rightarrow K$ (I-b)

The arguments of HeadReduce are written in *italic* style to indicate that they are part of a TRS on another level. Instead of the rules above one can write a more general rule which is valid for *all* combinators:

HeadReduce *c* : *Combinator* $\rightarrow c$ (I-c)

In this rule *c* is a variable which will be bound to a (sub)term of a TRS. The suffix *Combinator* restricts the number of matching expressions. Such a suffix is called an *abstraction*.

c) These abstractions which are used as patterns in the interpretation TRS's, are defined by a third TRS: the *abstraction TRS*. This abstraction TRS is used to make an abstract syntax of the user-defined TRS available to the interpretation TRS. An example of such an abstraction TRS is:

Combinator $\rightarrow S$ (A-a)
Combinator $\rightarrow K$ (A-b)

Now the extra restriction on *c* imposed by the suffix *Combinator* implies that the actual value of *c* must be a normal form of the term *Combinator* in the abstraction TRS. Hence only HeadReduce *S* and HeadReduce *K* will match rule (I-c). The abstraction level can be seen as a preprocessing level where things are done like syntactical categorizing, type checking, strictness analysis etcetera.

To make the specification of strategies really easy we introduce two primitive functions which can be used in the interpretation TRS:

Match *term rules*

which returns True if the *term* is a redex according to the *rules* and False otherwise.

Rewrite *term rules*

which returns the *term* after one rewrite according to the *rules* if the *term* matches the *rules* and the *term* itself otherwise.

The match function only checks whether or not the given term as a whole matches one of the given rules. The rewrite function will make a non-deterministic choice out of the matching rules. Although it is possible to define these primitive functions precisely in the interpretation TRS, the formal definition is rather tedious.

3.4.2 EXPRESSING HEAD REDUCTION FOR CURRY'S EXAMPLE

First we give the rules of the user-defined TRS:

Ap (Ap (Ap S x) y) z → Ap (Ap x z) (Ap y z) (U-S)
Ap (Ap K x) y → x (U-K)

Furthermore we have to define those abstractions of the TRS that we need at the interpretation level. In this example these abstractions are extremely trivial. The abstraction TRS is:

Combinator → *S* (A-1)
Combinator → *K* (A-2)

Finally we have to define the interpretation TRS. We will start with describing one step of the strategy: we will call that OneStepHead.

When the term is a redex we rewrite it and otherwise we search in the function part of apply for a redex (the propagation rule):

OneStepHead (*Ap f a*) *rules* → Cond (Match (*Ap f a*) *rules*)
(Rewrite (*Ap f a*) *rules*)
(*Ap* (OneStepHead *f rules*) *a*) (I-1)

In this rule Cond has the ordinary meaning. The result of OneStepHead applied to a function Ap with parameters is, if it matches as a whole, the rewrite of it, otherwise the result is the function Ap with as new parameters the result of OneStepHead applied to the function part. This surely terminates because of the next rule (I-2). Note that functions now also must have *rules* as parameter in order to be able to use the matching and rewriting primitives.

Again S and K cannot be rewritten, which we can now express in one rule:

OneStepHead *c:Combinator rules* → *c* (I-2)

The result of OneStepHead applied to a single combinator is that combinator itself.

We will now extend the one-step strategy to a complete strategy by adding another function:

Head *term rules* → Cond (ContainsAHeadRedex *term rules*)
(Head (OneStepHead *term rules*) *rules*)
term (I-3)

When our term contains a head-redex we do one left-most rewrite and continue, in the other case we stop and the result is the term itself.

The definition of ContainsAHeadRedex is very similar to the definition of OneStepHead. When there is a Rewrite in the definition of OneStepHead, we return True and when in OneStepHead we return the parameter as a result, we return False in ContainsAHeadRedex. Of course, recursive calls in OneStepHead are simply converted to recursive calls in ContainsAHeadRedex.

The definition of ContainsAHeadRedex is:

ContainsAHeadRedex *c:Combinator rules*	→ False		(I-4)
ContainsAHeadRedex (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) True (ContainsAHeadRedex *f rules*)	(I-5)

The complete set is now:

OneStepHead (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) (Rewrite (*Ap f a*) *rules*) (*Ap* (OneStepHead *f rules*) *a*)	(I-1)
OneStepHead *c:Combinator rules*	→ *c*		(I-2)
Head *term rules*	→ Cond	(ContainsAHeadRedex *term rules*) (Head (OneStepHead *term rules*) *rules*) *term*	(I-3)
ContainsAHeadRedex *c:Combinator rules*	→ False		(I-4)
ContainsAHeadRedex (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) True (ContainsAHeadRedex *f rules*)	(I-5)

The initial term must be:

Head *Term SKRules*

3.4.3 CORRECTNESS PROOF

In order to prove this specification we first define what the semantics of the abstract-interpretation TRS is. We will proof that the conditions are satisfied after proving some simple lemmas. These lemmas state general facts on the functions that are defined. These facts are easy to find because they cover the intention with which we constructed the functions. It will be rather simple to prove those lemmas.

Semantics of the Abstract-interpretation TRS

We start with settling the semantics of *c:Combinator* in the rules for OneStepHead and ContainsAHeadRedex. Because of the structure of the terms OneStepHead is equivalent to:

OneStepHead (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) (Rewrite (*Ap f a*) *rules*) (*Ap* (OneStepHead *f rules*) *a*)	(I-1)
OneStepHead *S rules*	→ *S*		($I\text{-}2_a$)
OneStepHead *K rules*	→ *K*		($I\text{-}2_b$)

For our semantics this eliminates the abstraction TRS. Furthermore the user-defined and the interpretation TRS can be seen as one and the same TRS with restrictions on the construction of terms, such as: the first argument of OneStepHead is an element of τ_O, the second argument are the

rules, etcetera. During reduction reducts of terms that meet these restrictions also meet these restrictions. So we now have only one TRS with the ordinary meaning. However we do not deal with the full reduction relation but only with those reductions that are allowed by normalizing strategies. Strategies for this TRS are normalizing if they reduce the first argument of Cond and the Cond itself without reducing the other arguments and if they reduce the argument of Head to a term which is an element of τ_O, before applying the Head rule again (this restriction is a consequence of uniting the two levels). So we simply remove those reductions out of the reduction relation. The reducer we get this way is called τ_S again and we will prove the specification following again O'Donnell's definitions.

Proof

Before we proof that the specification satisfies O'Donnell's definitions we want to prove some lemmas about the functions we have defined. The formulation of these lemmas is simple because of the way the functions were constructed.

Lemma The normal form of ContainsAHeadRedex x SKRules with $x \in \tau_O$ is True if x has a head-redex in τ_O, False otherwise.

Proof: Recall the definitions:

Def: x contains a head-redex ⇔ x matches a S or K rule or else
if x = Ap a b, a contains a head-redex.

ContainsAHeadRedex *S rules*	→ False		(I-4$_a$)
ContainsAHeadRedex *K rules*	→ False		(I-4$_b$)
ContainsAHeadRedex (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) True (ContainsAHeadRedex *f rules*)	(I-5)

Suppose $x \in \tau_O$ contains a head-redex then $x \neq S$ and $x \neq K$ (S and K themselves do not match the S or K rule) so x = Ap a b. So ContainsAHeadRedex x SKRules reduces in τ_S to Cond (Match (Ap a b) SKRules) True (ContainsAHeadRedex f SKRules). If x matches a S or K rule then obviously the normal form is True because we restricted the reduction relation to safe strategies. If x does not match a S or K rule then the left part of the application contains a head-redex which by induction and the absence of terms with an infinite spine results in the correct normal form. Proceeding this way it is very simple to prove the other parts of the lemma.

The following lemma is just as simple to proof :

Lemma OneStepHead x SKRules with $x \in \tau_O$ has as its normal form: if x has a head-redex in τ_O, the reduct of x after reducing this head-redex and if x does not have a head-redex in τ_O, its normal form is x.

We can use these lemmas in our proof of the specification. It will make everything very easy. We define E as: $x \in N\tau_O \Rightarrow E(x) = x$; $x \notin N\tau_O \Rightarrow E(x) =$ Head x SKRules. The computational relation is given by reductions of the primitive Rewrite. The definition of d is:

if x is an element of $N\tau_O$ then d(x) = x;

if x can be reduced via non computational reductions to a x' which is an element of $N\tau_O$ then also d(x) = x;

if x is not an element of $N\tau_O$ then we define d a bit special: if x can be reduced to x' via non computational reductions until the only possible reduction is a computational one and x' is of the form : Ap^1 (..... Ap^n ((Rewrite (Ap f a)) .....), perhaps surrounded by Head and SKRules then the decoding of x is defined as x' with applications of Head, SKRules and Rewrite skipped.

In other cases the decoding of x is nil.

Because we are only interested in reducts of elements of E, we restrict τ_S to those reducts.

1a. $d[E] = \tau_O$
Trivially true.

1b. $d^{-1}[N\tau_{O\sigma}] \cap E \subseteq N\tau_S$
Suppose $\alpha \in E$ and $\beta \in N\tau_O$ with $d(\alpha) = \beta$, then clearly $\alpha = \beta$, and since $N\tau_O \subseteq N\tau_S$, α is an element of $N\tau_S$.

2. $\forall \alpha, \beta \in \tau_S \ \alpha \rightarrow_C \beta \Rightarrow d(\alpha) \rightarrow_O d(\beta)$
Suppose α reduces to β via a reduction of Rewrite then if α and β are reducts of elements of E then clearly the decodings also reduce to each other.

3. $\forall \alpha, \beta \in \tau_S \ \alpha (\rightarrow_S\text{-}\rightarrow_C) \beta \Rightarrow d(\alpha) = d(\beta)$
Analogously to 2.

4. $\forall \alpha \in E, \beta \in \tau_O \ d(\alpha) \rightarrow_{O\sigma} \beta \Rightarrow \exists \delta \in E \ \alpha (\rightarrow_S\text{-}\rightarrow_C)^* \rightarrow_C (\rightarrow_S\text{-}\rightarrow_C)^* \delta \ \& \ d(\delta) = \beta$
Suppose $\alpha \in E$, $d(\alpha) \rightarrow_{O\sigma} \beta \Rightarrow \alpha \notin N\tau_O$, so α = Head α' SKRules with $\alpha' \in \tau_O$. Furthermore $d(\alpha) \rightarrow_{O\sigma} \beta$, so α' has a head-redex. Then OneStepHead α' SKRules reduces to δ with $d(\delta) = \beta$ and Head α' SKRules reduces to Head δ SKRules.

5. $\forall \alpha \in N\tau_S \ d(\alpha) \in N\tau_{O\sigma} \cup \{nil\}$
If α is a normal form of an element of E then $d(\alpha) \in N\tau_{O\sigma}$, $d(\alpha)$ is nil otherwise.

6. There is no infinite $(\rightarrow_S\text{-}\rightarrow_C)$ reduction path and moreover there exists a bounded function b: $\tau_O \rightarrow \mathbb{N}$ such that $\forall \alpha, \beta \in E \ \alpha (\rightarrow_S\text{-}\rightarrow_C)^m \rightarrow_C (\rightarrow_S\text{-}\rightarrow_C)^n \beta \Rightarrow m < b(d(\alpha)) \ \& \ n < b(d(\alpha))$.

There is no infinite non computational path because we only use safe strategies. We define b to be: b(x) = 3*(the number of Ap's on the spine of x) + 3. We notate sp(x) for the number of Ap's on the spine of x. If $x \in \tau_O$ then ContainsAHeadRedex x SKRules has a number of non computational reductions which is bound by 3*sp(x). OneStepHead x SKRules is bound by 3*sp(x) + 1 because it has an extra Rewrite. So if $\beta \notin N\tau_S$ then m= 3*sp(α) + 1 and n = 1; if $\beta \in N\tau_S$ then we may need 3*sp(β) + 2 non computational reductions to discover that we have a normal form.

The specification is *effective* because b, $\rightarrow_C$, d and E are all total computable.

Remarks

The proof was easy because the most essential aspects were already covered by the lemmas which were themselves simple and easy to proof. The use of the primitives Match and Rewrite made it possible to reason easily about the TRS system.

3.4.4 TRANSFORMATIONS OF THE SPECIFICATION

The description we have developed is a very nice one but operationally it is not the same as in the other two sections, because a call of ContainsAHeadRedex will induce an extra sweep through the term. If we want our description to mirror exactly the operations that were performed by the other methods, then we must change the rules and make them deliver a composite result consisting of the term and a boolean indicating whether the term was rewritten. The resulting rules are:

OneStepHead (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) (Pair True (Rewrite (*Ap f a*) *rules*)) (Pair (First (OneStepHead *f rules*)) (*Ap* (Second (OneStepHead *f rules*)) *a*))	(I-1')
OneStepHead *c:Combinator rules*	→ Pair False *c*		(I-2')
Head (Pair False *term*) *rules*	→ *term*		(I-3')
Head (Pair True *term*) *rules*	→ Head (OneStepHead *term rules*) *rules*		(I-4')

with as initial term:

Head (OneStepHead *Term SKRules*) *SKRules*.

Probably it will not be very difficult to prove this specification to be equivalent to the one without the booleans. Though this specification can now be considered to be just as efficient as the specifications in the other sections, it does not exactly mirror the same actions. In the other sections annotation functions were used in stead of booleans. Of course it is also possible to give a similar description with the abstract-interpretation method. We will rename Head to * and OneStepHead to !. We will also introduce success and failure constructors +/- and a propagate function called ~. The set of corresponding rules is:

! (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) (+ (Rewrite (*Ap f a*) *rules*)) (~ (*Ap* (! *f rules*) *a*))	(I-1")
! *c:Combinator rules*	→ - *c*		(I-2")
* (- *term*) *rules*	→ *term*		(I-3")
* (+ *term*) *rules*	→ * (! *term rules*) *rules*		(I-4")
~ (*Ap* (+ *x*) *y*) *rules*	→ + (*Ap x y*)		(I-5")
~ (*Ap* (- *x*) *y*) *rules*	→ - (*Ap x y*)		(I-6")

If this specification is compared with that of section 3.2.2 we see that the redex rules and the propagation rule are now all covered by rule (I-1") thanks to the power of the match and rewrite primitives. The non-redex rules are covered by rule (I-2"). The last four rules are the same.

We have shown that it is relatively easy to express a reduction strategy using the abstract-interpretation TRS with several algorithms and that it is also simple to transform one specification to another.

3.4.5 EVALUATION OF THE METHOD

This method enables us to write elegant strategy specifications. The non-redex cases are easily handled using the Match and Rewrite primitives. The different actions for syntactically different terms are conveniently dealt with using the abstraction mechanism. The fact that we write our strategy description on another level helps us in specifying strategies more generally.

The abstract-interpretation TRS system makes it possible to specify strategies formally in such a way that the specification holds for a large class of TRS's. For instance the description given at the interpretation level in this section is a valid head reduction specification for all TRS's using explicit application functions. Only a very trivial adaptation to the abstraction TRS is necessary in order to summarize all combinators. The proof then also holds for this large class of TRS's. Besides that, the proof was less tedious because we were led immediately to some general lemmas which themselves were easy to proof using the properties of the primitives Match and Rewrite. This specification is very short and readable and it appeals to our intuition. It enables us to easily give transformations of one specification to other specifications which have specific properties.

Concluding we can state that we have a promising facility with a great expressive power for describing strategies independently of the TRS.

3.5 CONCLUSIONS AND FURTHER RESEARCH

All of the three methods introduced in this paper are suitable for the specification of reduction strategies. Using an ordinary TRS gives rise to an exponential number of rewrite rules. This drawback disappears when a PRS is used. However the proof of the specification using this PRS was even more difficult than the proof with an ordinary TRS.

The most readable and general specification can be obtained by using an abstract-interpretation TRS extended with special primitives for matching and rewriting. The structure of the specification made it possible to give a simple proof and to construct transformations of the specifications in order to get alternative specifications with special desirable properties. Although this new TRS system is specially designed for the specification of reduction strategies, we think that its descriptive power is suitable for the specification of abstract interpretations (Burn *et al.* (1985)) in general.

In the near future we will search for simplifications of the definition of specification and we will investigate the use of the abstract-interpretation TRS in its full strength for the specification of strategies in graph rewriting systems, for giving correctness proofs of more complex strategies and for the investigation of the descriptive power for other domains.

3.6 ACKNOWLEDGEMENTS

We would like to thank Ronan Sleep, John Glauert and Richard Kennaway of the University of East-Anglia for the many fruitful discussions about reduction strategies and we also thank Jan-

Willem Klop of the Centre for Mathematics and Computer Science in Amsterdam for his explanations. Most of all we are grateful to Henk Barendregt of the University of Nijmegen for several important observations and valuable improvements.

4

TERM GRAPH REWRITING

H.P. Barendregt[1], M.C.J.D. van Eekelen[1], J.R.W. Glauert[2], J.R. Kennaway[2], M.J. Plasmeijer[1] and M.R.Sleep[2].

[1] University of Nijmegen, Nijmegen, The Netherlands.

Partially supported by the Dutch Parallel Reduction Machine Project.

[2] School of Information Systems, University of East Anglia, Norwich, U.K.

Partially supported by the U.K. ALVEY Project.

Abstract

Graph rewriting (also called reduction) as defined in Wadsworth (1971) was introduced in order to be able to give a more efficient implementation of functional programming languages in the form of lambda calculus or term rewrite systems: identical subterms are shared using pointers.

Several other authors, e g. Ehrig (1979), Staples (1980a,b,c), Raoult (1984) and van den Broek & van der Hoeven (1986) have given mathematical descriptions of graph rewriting, usually employing concepts from category theory. These papers prove among other things the correctness of graph rewriting in the form of the Church-Rosser property for "well-behaved" (i e. regular) rewrite systems However, only Staples has formally studied the soundness and completeness of graph rewriting with respect to term rewriting.

In this paper we give a direct operational description of graph rewriting that avoids the category theoretic notions. We show that if a term t is interpreted as a graph g(t) and is reduced in the graph world, then the result represents an actual reduct of the original term t (*soundness*). For weakly regular term rewrite systems, there is also a *completeness* result. every normal form of a term t can be obtained from the graphical implementation. We also show completeness for all term rewrite systems which possess a so called hypernormalising strategy, and in that case the strategy also gives a normalising strategy for the graphical implementation.

Besides having nice theoretical properties, weakly regular systems offer opportunities for parallelism, since redexes at different places can be executed independently or in parallel, without affecting the final result.

4.1 INTRODUCTION AND BACKGROUND

Graph rewriting is a well-known and standard technique for implementing functional languages based on term rewriting (e.g. Turner (1979a)), but the correctness of this method has received little attention, being simply accepted folklore. For both theory and practice, this makes a poor foundation, especially in the presence of parallelism. Staples (1980a,b,c) provides the only published results we are aware of. (A digested summary of these papers is in Kennaway (1984).) Wadsworth (1971) proves similar results for the related subject of pure lambda calculus.

Our principal result is that the notion of graph rewriting provides a sound and complete representation (in a sense precisely defined below) of *weakly regular* TRS's. A counterexample is given to show that for non-weakly regular TRS's completeness may fail: some term rewriting computations cannot be expressed in the corresponding graph rewrite system. A second result

concerns the mapping of evaluation strategies between the term and the graph worlds. A counterexample is exhibited to show that an evaluation strategy which is normalising (i.e. computes normal forms) in the term world may fail to do so when it is transferred to the graph world. We prove that any strategy which satisfies a stronger condition of being *hypernormalising* in the term world is normalising (and indeed hypernormalising) in the graph world. We briefly consider the problem of defining a graph rewriting implementation of non-left linear term rewrite rules.

The general plan of the paper is as follows: Section 4.2 presents basic definitions, and introduces a linear syntax for terms represented as graphs. Section 4.3 introduces a category of term graphs. Section 4.4 defines the notion of graph rewriting, and section 4.5 introduces the notion of *tree rewriting* as a prelude to section 4.6, which develops our theory of how to relate the worlds of term and graph rewriting. Section 4.7 considers the problem of mapping strategies between the two worlds. Finally, section 4.8 gives a summary of the work.

4.2 TERMS AS TREES AND GRAPHS

4.2.1 *Definition.*

(i) Let **F** be a (finite or infinite) set of objects called *function symbols*. A,B,... range over **F**.

(ii) The set T of *terms* over **F** is defined inductively by:
$A \in \mathbf{F}, t_1,...,t_n \in T \Rightarrow A(t_1,...,t_n) \in T \qquad (n \geq 0)$
A() is written as just A. ■

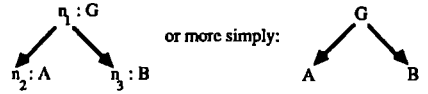
4.2.2 Example. Let $\mathbf{F}=\{0,S\}$. Then $T=\{0,S(0),0(S,S(0,0)), S(S,S,S)...\}$. Note that we do not assume that function symbols have fixed arities. This might appear inconvenient if one wished to represent, for example, the Peano integers, with a constant 0 and a successor operator S, since one also obtains extra "unintended" terms such as some of those listed above. When we define rewrite systems in section 4.4, we will see that this does not cause any problems. ■

4.2.3 Definition. A *labelled graph* (*over* **F**) is a triple (N,lab,succ) involving a (finite or infinite) set N of *nodes*, a function lab: $N \rightarrow \mathbf{F}$, and a function $succ:N \rightarrow N^*$. In this case we say that the $n_1,...,n_k$ are the *successors* of n. The ith component of succ(n) is denoted by $succ(n)_i$. ■

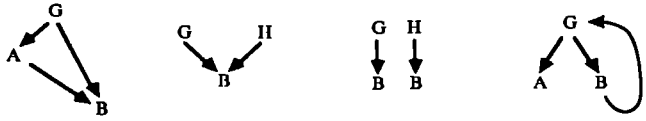
When we draw pictures of graphs, a directed edge will go from each node n to each node in succ(n), with the left-to-right ordering of the sources of the edges corresponding to the ordering of the components of succ(n). The identity of nodes is usually unimportant, and we may omit this information from pictures.

4.2.4 Example. Let $N=\{n_1,n_2,n_3\}$ and define lab and succ on N as follows.
$lab(n_1)=G, \quad lab(n_2)=A, \quad lab(n_3)=B,$
$succ(n_1)=(n_2,n_3), \quad succ(n_2)=(), \quad succ(n_3)=().$

This defines a labelled graph that can be drawn as:



Using this notation, four more examples of graphs are the following.



4.2.5 Definition.

(i) A *path* in a labelled graph (N,lab,succ) is a list $(n_0,i_0,n_1,i_1,\ldots ,n_{m-1},i_{m-1},n_m)$ where $m\geq 0$, $n_0,\ldots ,n_m\in N$, $i_0,\ldots ,i_{m-1}\in \mathbb{N}$ (the natural numbers) and n_{k+1} is the i_k-th successor of n_k. This path is said to be *from* n_0 *to* n_m and m is the *length* of the path.

(ii) A *cycle* is a path of length greater than 0 from a node n to itself. n is called a *cyclic* node.

(iii) A graph is *cyclic* if it contains a cyclic node, otherwise it is *acyclic*. ■

4.2.6 Definition.

(i) A *term graph* (often, within this paper, simply a *graph*) is a quadruple (N,lab,succ,r) where (N,lab,succ) is a labelled graph and r is a member of N. The node r is called the *root* of the graph. (We do not require that every node of a term graph is reachable by a path from the root.) For a graph g, the components are often denoted by N_g, lab_g, $succ_g$, and r_g.

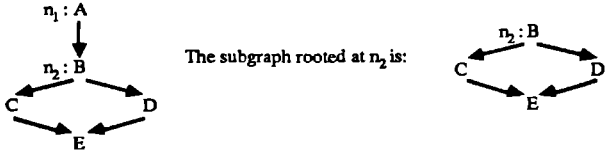
(ii) A path in a graph is *rooted* if it begins with the root of the graph. The graph is *root-cyclic* if there is a cycle containing the root.
When we draw pictures of term graphs, the topmost node is the root. ■

Term graphs are exactly the graphs discussed in the paper Barendregt *et al.* (1987b), which defines a language of generalised graph rewriting of which the rewriting treated in this paper is a special case.

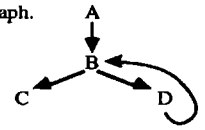
4.2.7 Definition. Let g = (N,lab,succ) be a labelled graph and let $n\in N$. The *subgraph of* g *rooted at* n is the term graph (N',lab',succ',n) where $N'=\{n'\in N \mid$ there is a path from n to n'$\}$ and lab' and succ' are the restrictions of lab and succ to N'. We denote this graph by g|n. The definition also applies when g is a term graph. ■

4.2.8 Examples.

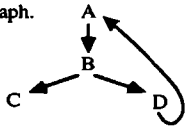
(i)



(ii) A cyclic graph.



(iii) A root-cyclic graph.



A formal description of a graph requires a complete specification of the quadruple (N,lab,succ,r). When writing down examples of finite graphs, it is convenient to adopt a more concise notation, which abstracts away from details such as the precise choice of the elements of N. We will use a notation based on the definition of terms in definition 4.2.1, but with the addition of node-names, which can express the sharing of common subexpressions. The notation is defined by the following context-free grammar, with the restrictions following it.

4.2.9 Definition (linear notation for graphs).

graph ::= node | node + graph

node ::= A(node,...,node) | x | x : A(node,...,node)

A ranges over **F**. x ranges over a set, disjoint from **F**, of *nodeid's* ('node identifiers'). Any nodeid x which occurs in a graph must occur exactly once in the context x : A(node,...,node). Nodeid's are represented by tokens beginning with a lower-case letter. Function symbols will be non-alphabetic, or begin with an upper-case letter. We again abbreviate A() to A. ■

This syntax is, with minor differences, the same as the syntax for graphs in the language LEAN (Barendregt *et al.* (1987b)). The five graphs of the examples 4.2.4 are in this notation: G(A,B), G(A(x:B),x), G(x:B) + H(x), G(B) + H(B) and x:G(A,B(x)). Note that multiple uses of the same nodeid express multiple references to the same node.

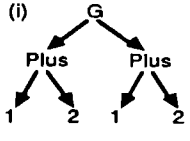
The definition of terms in 4.2.1 corresponds to a sublanguage of our shorthand notation, consisting of those graphs obtained by using only the first production for graph and the first production for node. *So terms have a natural representation as graphs.*

4.2.10 Examples.

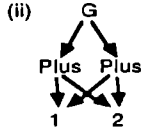
(i) G(Plus(1,2), Plus(1,2)) (ii) G(Plus(n_1: 1, n_2: 2), Plus(n_1, n_2))

(iii) G(n:Plus(1,2), n) (iv) n_1: Cons(3, n_1)

(i)

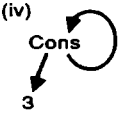


(ii)



(iii)

(iv)



4.2.11 Definition. A *tree* is a graph (N,lab,succ,r) such that there is exactly one path from r to each node in N. ■

Thus example (i) above is a tree, and (ii), (iii), and (iv) are not. Trees are always acyclic. Notice that a graph g is a finite tree iff g can be written by the grammar of 4.2.9 without using any nodeid's.

The natural mapping of terms to graphs represents each term as a finite tree. However, some terms can also be represented as proper graphs, by sharing of repeated subterms. For example, the term G(Plus(1,2),Plus(1,2)) can be represented by any of the graphs pictured in example 4.2.10 (i), (ii), or (iii), as well as by the graphs G(Plus(x:1,2),Plus(x,2)) or G(Plus(1,x:2),Plus(1,x)).

4.3 HOMOMORPHISMS OF GRAPHS AND TREES

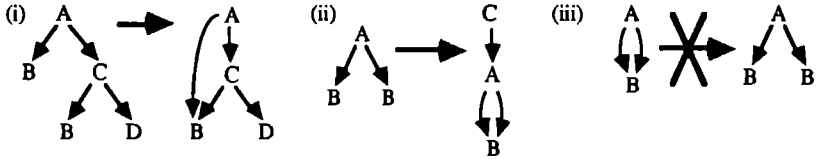
4.3.1 Definition. Given two graphs $g_1 = (N_1, lab_1, succ_1, r_1)$ and $g_2 = (N_2, lab_2, succ_2, r_2)$, a *homomorphism* from g_1 to g_2 is a map $f:N_1 \to N_2$ such that for all $n \in N_1$,

$$lab_2(f(n)) = lab_1(n)$$
$$succ_2(f(n)) = \mathbf{f}(succ_1(n))$$

where $\mathbf{f}$ is defined by $\mathbf{f}(n_1,...,n_k) = (f(n_1),...,f(n_k))$. That is, homomorphisms preserve labels, successors, and their order. ■

4.3.2 Definition. Graph(**F**) is the category whose objects are graphs over **F** and whose morphisms are homomorphisms. Tree(**F**) is the full subcategory of Graph(**F**) whose objects are the trees over **F**. It is easy to verify that these are categories. ■

4.3.3 Examples. We shall write $g_1 \longrightarrow g_2$ when there is a homomorphism from g_1 to g_2. We have the following pictures.



4.3.4 Definition.

(i) A homomorphism $f:g_1 \to g_2$ is *rooted* if $f(r_1)=r_2$.

(ii) An *isomorphism* is a homomorphism which has an inverse. We write $g \sim g'$ when g and g' are isomorphic.

(iii) Two graphs are *equivalent* when they are isomorphic by a rooted isomorphism. We write $g \approx g'$ when g and g' are equivalent. ■

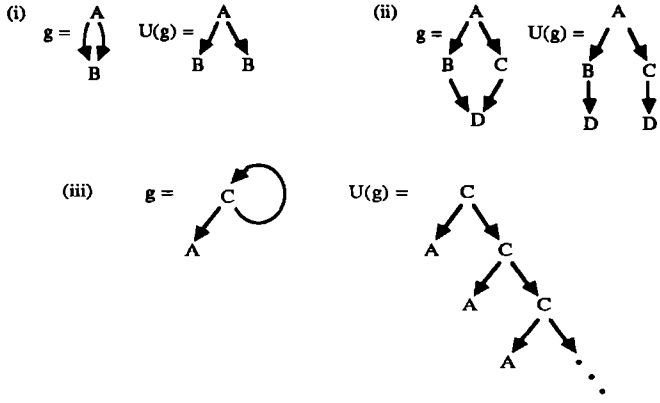
4.3.5 Proposition.

(i) For any graphs g_1 and g_2 we have $g_1 = g_2 \Rightarrow g_1 \sim g_2$.

(ii) Every rooted homomorphism from one tree to another is an isomorphism. ■

4.3.6 Example. These two graphs are isomorphic but not equivalent (recall that in diagrammatic representations the root node is the topmost):

4.3.7 Definition. Given any graph g=(N,lab,succ,r) we can define a tree U(g) which results from "unravelling" g from the root. We start with some examples.



Now we give the formal definition. U(g) has as nodes the rooted paths of g. The root of U(g) is the path (r). For a path $p=(n_0,i_0,\ldots,n_{m-1},i_{m-1},n_m)$, $lab_{U(g)}(p) = lab_g(n_m)$ and $succ_{U(g)}(p) = (p_1,\ldots,p_k)$ where p_i is the result of appending $(i,succ_g(n_m)_i)$ to p. Clearly this is a tree. ■

4.3.8 Proposition. *For every graph g there is a rooted homomorphism* u_g*: U(g)→g defined by:* $u_g(n_0,i_0,\ldots,n_m) = n_m$. ■

4.3.9 Proposition. *A graph g is a tree iff* $g \approx U(g)$. ■

4.3.10 Definition. Two graphs g and g' are *tree-equivalent*, notated $g \approx_t g'$, if $U(g) \approx U(g')$ ■

For example, the graphs of example 4.2.10 (i), (ii) and (iii) are all tree-equivalent. So are these two graphs:

4.4 GRAPH REWRITING

We now turn to rewriting. First we recall the familiar definitions of terms with free variables and term rewriting. We then explain informally how we represent terms with free variables as 'open'

graphs, and define our notion of graph rewriting. Our definition is quite similar to the one in Staples (1980a).

4.4.1 Definition (term rewriting).

(i) Let **V** be a fixed set of function symbols, disjoint from **F**. The members of **V** are called *variables*, and are denoted by lower-case letters. An *open term* over a set of function symbols **F** is a term over $\mathbf{F} \cup \mathbf{V}$ in which every node labelled with a variable has no successors. An open term containing no variables (that is, what we have been calling simply a term) is a *closed* term.

(ii) A *term rewrite rule* is a pair of terms t_L and t_R (written $t_L \rightarrow t_R$) such that every variable occurring in t_R occurs in t_L. t_L and t_R are, respectively, the *left-* and *right-hand* sides of the *term rewrite rule* $t_L \rightarrow t_R$.

(iii) A term rewrite rule is *left-linear* if no variable occurs more than once in its left-hand side. ■

The usual definition of a term rewrite rule requires that t_L be not just a variable. However, our results are not affected by the presence of such rules, so we do not bother to exclude them.

In order to introduce graph rewriting, first we need some preparatory definitions.

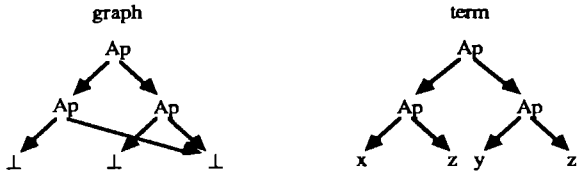
4.4.2 Definition.

(i) An *open labelled graph* is an object (N,lab,succ) like a labelled graph, except that lab and succ are only required to be partial functions on N, with the same domain. A node on which lab and succ are undefined is said to be *empty*. The definition of an *open (term) graph* bears the same relation to that of a (term) graph. When we write open graphs, we will use the symbol $\perp$ to denote empty nodes. As with terms, we talk of closed (labelled or term) graphs and closed trees as being those containing no empty nodes.

(ii) A *homomorphism* from one open graph g_1 to another g_2 is defined as for graphs, except that the "structure preserving" conditions are only required to hold at nonempty nodes of g_1. ■

Open term graphs are intended to represent terms with variables. Instead of using the set **V** of variables, we find it more convenient, for technical reasons, to follow Staples (1980a) by using empty nodes. The precise translation from open graphs to open terms is as follows. Given an open graph over **F**, we first replace each empty node in it by a different variable symbol from **V**, and then unravel the resulting closed graph over $\mathbf{F} \cup \mathbf{V}$, obtaining an open term over **F**. Thus where a graph has multiple edges pointing to the same empty node, the term will have multiple occurrences of the same nodeid.

For example, the graph Ap(Ap(⊥,w:⊥),Ap(⊥,w)) translates to the term Ap(Ap(x,z),Ap(y,z)):



We could obtain any term which only differs from this one by changes of variables. We shall treat such terms as the same.

We now turn to the graph representation of term rewrite rules. We only deal with left-linear rules in this paper. In 4.6.13 we discuss briefly the problems in graphically describing non-left-linear rules.

4.4.3 Definition.

(i) A *graph rewrite rule* is a triple (g,n,n'), where g is an open labelled graph and n and n' are nodes of g, called respectively the *left root* and the *right root* of the rule.

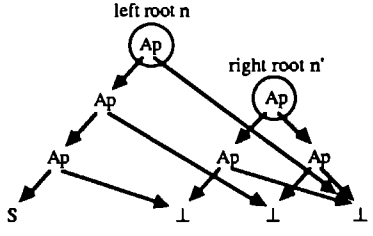
(ii) A *redex* in a graph g_0 is a pair $\Delta = (R,f)$, where R is a graph rewrite rule (g,n,n') and f is a homomorphism from g|n to g_0. The homomorphism f is called an *occurrence* of R. ■

Rather than introduce our formal definition of graph rewriting immediately, we begin with some examples. The formal definition is given in section 4.4.6.

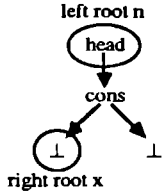
4.4.4 Translation of term rules to graph rules.

Let $t_L \rightarrow t_R$ be a left-linear term rewrite rule. We construct a corresponding graph rewrite rule (g,n,n'), where g is a labelled graph and n and n' are nodes of g. First take the graphs representing t_L and t_R. Form the union of these, sharing those empty nodes which represent the same variables in t_L and t_R. This graph is g. Take n and n' to be the respective roots of t_L and t_R. Here are two examples which should make the correspondence between term and graph rewrite rules clear.

(i) Term rule: Ap(Ap(Ap(S,x),y),z) → Ap(Ap(x,z),Ap(y,z))
Graph rule: (n:Ap(Ap(Ap(S,x:⊥),y:⊥),z:⊥) + n':Ap(Ap(x,z),Ap(y,z)), n, n')

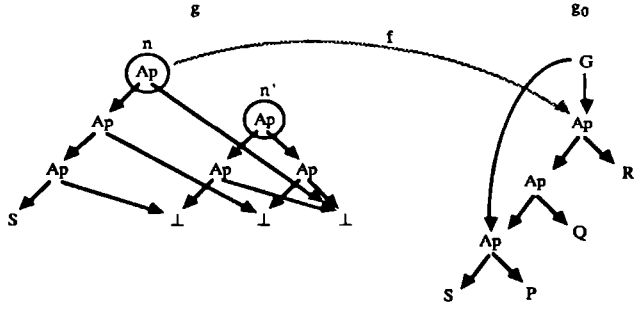


(ii) Term rule: head(cons(x,y)) → x
Graph rule: (n:head(cons(x:⊥,⊥)), n, x)

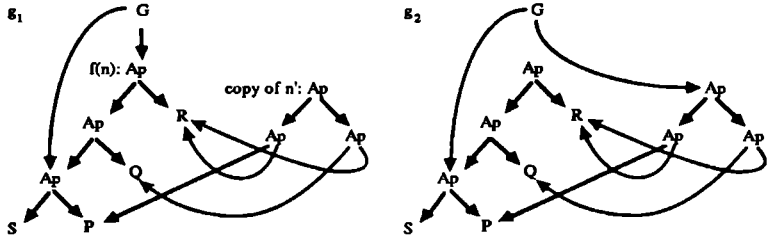


4.4.5 Informal definition of graph rewriting.

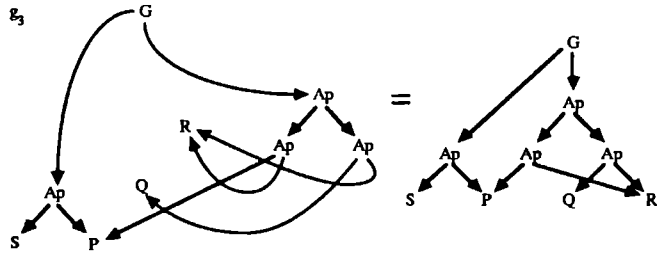
A redex ((g,n,n'), f: g|n → g_0) in a graph g_0 is reduced in three steps. We shall use the following redex as an example: (g,n,n') is the S-rule above, g_0 = G(a,Ap(Ap(a:Ap(S,P),Q),R)) and f operates on n as indicated in the picture (which completely determines how f behaves on the rest of g|n).



First (the *build* phase) an isomorphic copy of that part of g|n' not contained in g|n is added to g_0, with lab, succ, and root defined in the natural way. Call this graph g_1. Then (the *redirection* phase) all edges of g_1 pointing to f(n) are replaced by edges pointing to the copy of n', giving a graph g_2. The root of g_2 is the root of g_1, if that node is not equal to f(n). Otherwise, the root of g_2 is the copy of n'.



Lastly (the *garbage collection* phase), all nodes not accessible from the root of g_2 are removed, giving g_3, which is the result of the rewrite.



Note that the bottommost Ap node of the redex graph and the S node remain after garbage collection, since they are still accessible from the root of the graph. The other two Ap nodes of the redex vanish.

4.4.6 Formal definition of graph rewriting.

We now give a formal definition of the general construction. Let $((g,n,n'), f: g|n \to g_0)$ be a redex in a graph g_0. The graphs g_1 (the build phase), g_2 (the redirection phase) and g_3 (the garbage collection phase) are defined as follows.

(i) The node-set N of g_1 is the disjoint union of N_{g0} and $N_{g|n'} - N_{g|n}$. The root is r_{g0}. The functions lab_{g1} and $succ_{g1}$ are given by:

$$
\begin{array}{lll}
lab_{g1}(m) & = lab_{g0}(m) & (m \in N_{g0}) \\
 & = lab_{g}(m) & (m \in N_{g|n'} - N_{g|n}) \\
succ_{g1}(m)_i & = succ_{g0}(m)_i & (m \in N_{g0}) \\
 & = succ_{g}(m)_i & (m, succ_g(m)_i \in N_{g|n'} - N_{g|n}) \\
 & = f(succ_g(m)_i) & (m \in N_{g|n'} - N_{g|n}, succ_g(m)_i \in N_{g|n})
\end{array}
$$

We write $g_1 = g_0 +_f (g,n,n')$.

(ii) The next step is to replace in g_1 all references to f(n) by references to n'. We can define a substitution operation in general for any term graph h and any two nodes a and b of h.

h[a:=b] is a term graph $(N_h, lab, succ, r)$, where lab, succ, and r are given as follows.

lab(c) = $lab_h(c)$ for each node c of N_h
if $succ_h(c)_i$ = a then $succ(c)_i$ = b, otherwise $succ(c)_i = succ_h(c)_i$
if r_h = a then r = b, otherwise $r = r_h$

With this definition, g_2 is $g_1[f(n):=n']$.

(iii) Finally, we take the part of g_2 which is accessible from its root, by defining $g_3 = g_2|r_{g2}$. We give this operation a name: for any term graph h, we denote $h|r_h$ by GC(h) (*G*arbage *C*ollection).

We denote the result of reducing a redex Δ in a graph g by RED(Δ,g). Collecting the notations we have introduced, we have
RED(((g,n,n'),f),g_0) = GC((g_0 +$_f$ (g,n,n'))[f(n):=n']). ■

Our definition of graph rewriting is a special case of a more general notion, defined in Glauert *et al.* (1987) by a category-theoretic construction. Those familiar with category theory may recognise the build phase of a rewrite as a pushout, and redirection and garbage collection can be given definitions in the same style (though the categories involved are not those defined in this paper). For the purpose of this paper - describing graph rewritings which correspond to conventional term rewritings - the direct "operational" definition we have given is simpler.

4.4.7 Definition.

(i) If g reduces to g' by reduction of a redex Δ, we write $g \rightarrow^{\Delta} g'$, or $g \rightarrow g'$ if we do not wish to indicate the identity of the redex. The reflexive and transitive closure of the relation $\rightarrow$ is $\rightarrow^*$.

(ii) A *graph rewriting system* (GRS) over **F** consists of a pair (G,**R**) where **R** is a set of rewrite rules and G is a set of graphs over **F** closed under rewriting by the members of **R**.

(iii) We write $g \rightarrow_{\mathbf{R}} g'$ if $g \rightarrow g'$ by reduction of a redex using one of the rules in **R**. The reflexive and transitive closure of $\rightarrow_{\mathbf{R}}$ is $\rightarrow^*_{\mathbf{R}}$. If clear from the context, we omit the subscript **R**.

(iv) A graph g such that for no g' does one have $g \rightarrow_{\mathbf{R}} g'$ is said to be an **R**-normal form (or to be in **R**-normal form). If $g \rightarrow^*_{\mathbf{R}} g'$ and g' is in **R**-normal form, we say that g' is an **R**-normal form of g, and that g has an **R**-normal form. Again, we often omit the **R**. ■

Note that a GRS is not required to include all the graphs which can be formed from the given set of function symbols **F**. Any subset closed under rewriting will do. This allows our definition to automatically handle such things as, for example, sorted rewrite systems, where there are constraints over what function symbols can be applied to what arguments, or arities, where each function symbol may only be applied to a specified number of arguments. From our point of view, this amounts to simply restricting the set of graphs to those satisfying these constraints. So long as rewriting always yields allowed graphs from allowed graphs, we do not need to develop any special formalism for handling restricted rewrite systems, nor do we need to prove new versions of our results.

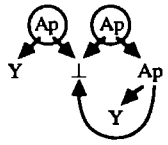
Our definition of a graph rewrite rule allows any conventional term rewrite rule to be interpreted as a graph rewrite rule, provided that the term rewrite rule is left-linear, that is, if no variable occurs twice or more on its left-hand side. As some of the following examples show, however, some new phenomena arise with graph rewrite rules.

4.4.8 Examples.

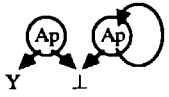
(i) Term rule: A(x) $\rightarrow$ B(x); Graph rule: (n:A(x:$\perp$) + n':B(x), n, n')
Graph: x:A(x); Result of rewriting: x:B(x)

(ii) Term rule: $I(x) \rightarrow x$; Graph rule:(n:I(n':⊥), n, n')
Graph: I(3); Result of rewriting: 3

(iii) The fixed point combinator Y has the term rewrite rule $Ap(Y,x) \rightarrow Ap(x,Ap(Y,x))$. This can be transformed into the graph rewrite rule (n:Ap(Y,x:⊥) + n':Ap(x,Ap(Y,x)),n,n'). However, it can also be given the graph rewrite rule: (n:Ap(Y,x:⊥) + n':Ap(x,n'),n,n'):

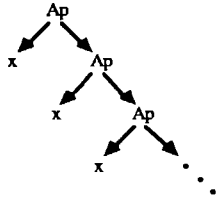


(n:Ap(Y,x:⊥) + n':Ap(x,Ap(Y,x)),n,n')



(n:Ap(Y,x:⊥) + n':Ap(x,n'),n,n')

This captures the fact that the Böhm tree (Barendregt (1984)) of the term Ap(Y,x) is:



The graph rule can do all the 'unravelling' in one step, which in the term rewrite world requires an infinite sequence of rewritings.

(iv) Here is a more subtle example of the same phenomenon illustrated by (iii). Consider the term rewrite rule

$$F(Cons(x,y)) \rightarrow G(Cons(x,y))$$

Our standard representation of this as a graph rewrite rule is:

$$(n:F(Cons(x:\bot,y:\bot)) + n':G(Cons(x,y)), n, n')$$

Note that each application of this rule will create a new node of the form Cons(...,...), which will have the same successors as an existing node Cons (...,...). In a practical implementation, there is no need to do this. One might as well use that existing node, instead of making a copy of it. The following graph rewrite rule does this:

$$(n:F(z:Cons(\bot,\bot)) + n':G(z), n, n')$$

Both the languages Standard ML and Hope, which are languages of term rewriting, allow an enhanced form of term rewrite rules such as (using our syntax):

$$F(z:Cons(x,y)) \rightarrow G(z)$$

with precisely this effect. Of course, given referential transparency (which ML lacks) there is no reason for an implementation not to make this optimisation wherever there is an opportunity, even if the programmer does not. But providing this feature to the programmer may make his programs more readable.

(v) Term rule: $I(x) \rightarrow x$; Graph rule: (n:I(n':⊥), n, n')
Graph: x:I(x); Result of rewriting: x:I(x)

Example 4.4.8(v) is deliberately pathological. Consider the GRS for combinatory logic, whose rules are those for the S, K, and I combinators. The graph can be interpreted as "the least fixed point of I" (cf. the example of the Y combinator above), and in the usual denotational semantics in terms of reflexive domains should have the bottom, "undefined" value. As the graph reduces to itself (and to nothing else), it has no normal form. Thus our operational semantics of graph rewriting agrees with the denotational semantics. This is not true for some other attempts we have seen at formalising graphical term rewriting.

We now study some properties of graph rewrite systems. We establish a version of the theorem of finite developments for term rewriting, and the confluence of weakly regular systems. For reasons of space, the longer proofs are omitted from this paper. They appear in Barendregt *et al.* (1986a).

4.4.9 Proposition. *Garbage collection can be postponed. That is, given* $g \rightarrow^{\Delta 1} g_1 \rightarrow^{\Delta 2} g_2$, $\Delta_i = (R_i f_i)$, $R_i = (g_i, n_i, n'_i)$ $(i = 1,2)$ *and* $g'_1 = (g +_{f1} R_1)[f(n) = n']$, *then* Δ_2 *is also a redex of* g'_1, *and* $g'_1 \rightarrow^{\Delta 2} g_2$. ■

4.4.10 Definition. Two redexes $\Delta_1 = ((g_1,n_1,n'_1),f_1)$ and $\Delta_2 = ((g_2,n_2,n'_2),f_2)$ of a graph g are *disjoint* if:

(i) $f_2(n_2)$ is not equal to $f_1(n)$ for any nonempty node n of $g_1|n_1$, and
(ii) $f_1(n_1)$ is not equal to $f_2(n)$ for any nonempty node n of $g_2|n_2$.

Δ_1 and Δ_2 are *weakly disjoint* if either they are disjoint, or the only violation of conditions (i) and (ii) is that $f_1(n_1) = f_2(n_2)$, and the results of reducing Δ_1 or Δ_2 are identical.

A GRS is *regular* (resp. *weakly regular*) if for every graph g of the GRS, every two distinct redexes in g are disjoint (resp. weakly disjoint). ■

4.4.11 Proposition. *Let* $\Delta_1 = ((g_1,n_1,n'_1), f_1)$ *and* $\Delta_2 = ((g_2,n_2,n'_2), f_2)$ *be two disjoint redexes of a graph g. Let* $g \rightarrow^{\Delta 1} g'$. *Then either* $f_2(n_2)$ *is not a node of g', or there is a redex* $((g_2,n_2,n'_2),f)$ *of g' such that* $f(n_2) = f_2(n_2)$. ■

4.4.12 Definition.

(i) With the notations of the preceding proposition, if $f_2(n_2)$ is not a node of g' then Δ_2/Δ_1 is the empty reduction sequence from g' to g'; otherwise, Δ_2/Δ_1 is the one-step reduction sequence consisting of the reduction of $((g_2,n_2,n'_2),f)$. This redex is the *residual* of Δ_2 by Δ_1 and is denoted by $\Delta_2//\Delta_1$. For weakly disjoint Δ_1 and Δ_2, Δ_2/Δ_1 is the empty

reduction sequence from g' to g'. Δ_2/Δ_1 is not defined when Δ_1 and Δ_2 are not weakly disjoint, and $\Delta_2//\Delta_1$ is not defined when Δ_1 and Δ_2 are not disjoint.

(ii) Given a reduction sequence $g \to^{\Delta 1} \to^{\Delta 2} \ldots \to^{\Delta_i} g'$ and a redex Δ of g, the *residual* of Δ by the sequence $\Delta_1 \ldots \Delta_i$, denoted $\Delta//(\Delta_1 \ldots \Delta_i)$ is $(\Delta//(\Delta_1 \ldots \Delta_{i-1}))//\Delta_i$ (provided that $(\Delta//(\Delta_1 \ldots \Delta_{i-1}))$ exists and is disjoint from Δ_i). ■

4.4.13 Proposition. *Let Δ_1 and Δ_2 be weakly disjoint redexes of g, and let $g \to^{\Delta i} g_i$ (i = 1,2). Then there is a graph h such that $g_1 \to^{\Delta 2/\Delta 1} h$ and $g_2 \to^{\Delta 1/\Delta 2} h$. That is, weakly disjoint redexes are subcommutative.* ■

4.4.14 Corollary. *Every weakly regular GRS is confluent. That is, if $g \to^* g_i$ (i = 1,2), then there is an h such that $g_i \to^* h$ (i = 1,2).* ■

4.4.15 Definition. Let g be a graph and F be a set of disjoint redexes of g. A *development* of F is a reduction sequence in which the redex reduced at each step is a residual, by the preceding steps of the sequence, of a member of F. A *complete development* of F is a development of F , at the end of which there remain no residuals of members of F. ■

4.4.16 Proposition. *Every complete development of a finite set of pairwise disjoint redexes F is finite. In fact, its length is bounded by the number of redexes in F.* ■

4.4.17 Proposition. *Let F be a set of redexes of a graph g. Every finite complete development of F ends with the same graph (up to isomorphism). This graph is:*

$$GC((g +_{f1} R_1 +_{f2} \ldots +_{fi} R_i)[f_1(n_1) := n'_1] \ldots [f_i(n_i) := n'_i])$$

where the redexes whose residuals are reduced in the complete development are $\Delta_1 = (f_1,R_1),\ldots,\Delta_i = (f_i,R_i)$. ■

Note that since we allow infinite graphs, a set of redexes F as in the last two propositions may be infinite. Nevertheless, it may have a finite complete development, if rewriting of some members of F causes all but finitely many members of F to be erased.

4.5 TREE REWRITING

In order to study the relationship between term rewriting and graph rewriting, we define the notion of *tree rewriting*. This is a formalisation of conventional term rewriting within the framework of our definitions of graph rewriting.

4.5.1 Definition.

(i) A *tree rewrite rule* is a graph rewrite rule (g,n,n') such that g|n is a tree. For a set of tree rewrite rules **R**, the relation $\to_{tR}$ of *tree rewriting* with respect to **R** is defined by:

$t_1 \to_{tR} t_2 \Leftrightarrow$ for some graph g, $t_1 \to_R g$ and $U(g) = t_2$

(ii) A *tree rewrite system* (TreeRS) over **F** is a pair (T,R) where R is a set of tree rewrite rules and T is a set of trees over **F** closed under $\to_{tR}$. A *term rewrite system* (TRS) is a TreeRS, all of whose trees are finite.

When t_1 reduces to t_2 by tree rewriting of a redex Δ, we write $t_1 \rightarrow_t^{\Delta} t_2$, or $t_1 \rightarrow_t t_2$ when we do not wish to indicate the identity of the redex. ■

Tree rewrite systems differ from conventional term rewrite systems in two ways. Firstly, infinite trees are allowed. We need to handle infinite trees, since they are produced by the unravelling of cyclic graphs. We need to handle cyclic graphs because some implementors of graph rewriting use them, and we do not want to limit the scope of this paper unnecessarily. Secondly, the set of trees of a TreeRS may be any set of trees over the given function symbols which is closed under tree rewriting. This is for the same reason as was explained above for GRS's.

If for each rule (g,n,n') in the rule-set, g is finite and acyclic, the set of all finite trees generated by the function symbols will be closed under tree rewriting. This is true for those rules resulting from term rewrite rules by our standard representation. Thus the conventional notion of a TRS is included in ours.

4.5.2 Definition. Let t, t_1,... t_i be trees, and n_1,...,n_i be distinct nodes of t. We define $t[n_1:=t_1,...,n_i:=t_i]$ to be the tree whose nodes are

(i) all paths of t which do not include any of n_1,...,n_i, and
(ii) every path obtained by taking a path p of t, which ends at n_j ($1 \leq j \leq i$) and contains no other occurrence of $n_1 ... n_i$, and replacing the last node of p by any path of t_j.

For any of these paths p, the label of p is the label of the last node in p, in whichever of t, t_1,... t_i that came from. The successors function is defined similarly. ■

The results concerning disjointness, regularity, and confluence which we proved for graph rewriting all have versions for tree rewriting. Again we omit proofs. We also have the following:

4.5.3 Proposition. *Unravelling can be postponed. That is, if $t_1 \rightarrow_t t_2 \rightarrow_t t_3$, then there are graphs g and g' such that*

(1) $t_2 = U(g)$ *and* $t_1 \rightarrow g$ *(by graph rewriting)*
(2) $g \rightarrow g'$ *and* $t_3 \rightarrow^*_t U(g')$. ■

4.6 RELATIONS BETWEEN TREE AND GRAPH REWRITING

In this section we prove our principal result: for weakly regular rule-systems, graph rewriting is a sound and complete implementation of term rewriting.

4.6.1 Definition. Let (T,R) be a TreeRS.

(i) L(T,R), the *lifting* of this system, is the GRS whose set of graphs is $L(T) = \{g \mid U(g) \in T\}$, and whose rule set is R (but now interpreted as graph rewrite rules). It is trivial to verify that L(T) is closed under $\rightarrow_R$.
(ii) A *graphical term rewrite system* (GTRS) is a GRS of the form L(T,R), where (T,R) is a term rewrite system.
(iii) A GRS (G,R) is *acyclic* if every member of G is acyclic. ■

When (T,**R**) is a term rewrite system, L(T,**R**) represents its graphical implementation. There are two fundamental properties it must have to be a correct implementation, which we now define.

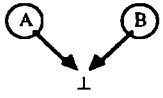
4.6.2 Definition.

(i) A TreeRS (T,**R**) is called *graph-reducible* if for every graph g in L(T), if t is a normal form of U(g) in (T,**R**), then there is a normal form g' of g in L(T,**R**) such that U(g') = t, and if U(g) has no normal form in (T,**R**), then g has no normal form in L(T,**R**).

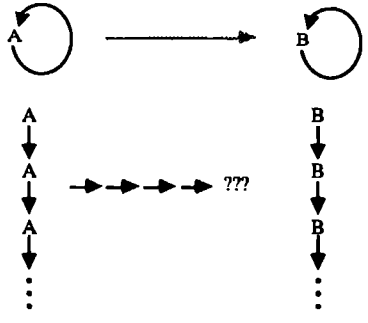
(ii) A GRS (G,**R**) is *tree-reducible* if there is a TreeRS (T,**R**) such that (G,**R**) = L(T,**R**), and such that if g' is a normal form of g in (G,**R**), then U(g') is a normal form of U(g) in (T,R), and if g has no normal form in (G,**R**), then U(g) has no normal form in (T,**R**). ■

L(T,**R**) is the graphical implementation of (T,**R**). Tree-reducibility of L(T,**R**) expresses soundness: every result which is obtainable by graph rewriting in L(T,**R**) is also obtainable by tree rewriting in (T,**R**). Graph-reducibility of (T,**R**) expresses completeness: every result which is obtainable by tree rewriting is also obtainable by graph rewriting. We shall see that every GTRS is tree-reducible, and every weakly regular TRS is graph-reducible. Not all GRS's, even those of the form L(T,**R**), are tree-reducible, nor is every TreeRS graph-reducible, as the following examples show.

4.6.3 Example. Tree reducibility can fail when there are cyclic graphs. Consider the term rewrite rule A(x) → B(x), represented graphically by:



A cyclic graph may contain a single redex with respect to this rule, while its unravelling contains infinitely many:



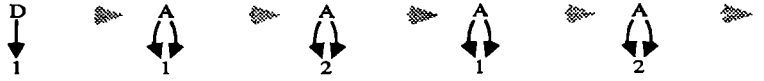
4.6.4 Example. The following TreeRS is not graph-reducible:

T: trees over {A,D,0,1,2}, with the following arities: A is binary, D is unary, and 0, 1, and 2 are nullary.

R: A(1,2) → 0; 1 → 2; 2 → 1; D(x) → A(x,x).

For a counterexample, consider the following tree rewriting sequence:

In the graph rewriting system we have:



In this example, the sharing of (tree) subterms in the graph world has excluded from the graph world certain rewrite sequences of the tree world. Distinct subterms of A(1,1) correspond to the same subgraph of A(x:1,x), forcing synchronized rewriting of siblings, which makes the normal form inaccessible.

4.6.5 Definition.

(i) Redexes $\Delta = ((g,n,n'),f)$ and $\Delta' = ((g',m,m'),f')$ in a graph h are *siblings* if $hlf(n) \approx_t hlf'(m)$.

(ii) For a redex $\Delta = ((U(g),n,n'),f)$ of a tree U(g) we define $u_g(\Delta)$ to be the redex $((g,n,n'),u_g \cdot f)$ of g.

(iii) For a redex Δ of a graph g, the set of redexes Δ' of U(g) such that $u_g(\Delta') = \Delta$ is denoted by $u_g^{-1}(\Delta)$. For a set of redexes F of a graph g, $u_g^{-1}(F)$ denotes $\bigcup\{ u_g^{-1}(\Delta') \mid \Delta' \in F \}$.

(iv) A redex Δ of a graph G is *acyclic* if $u_g^{-1}(\Delta)$ is finite. ■

4.6.6 Proposition. *Let $g \to g'$ by rewriting of an acyclic redex Δ. Then $U(g) \to^*_t U(g')$ by complete development of $u_g^{-1}(\Delta)$. For any redex Δ' of g, weakly disjoint from Δ, $u_g^{-1}(\Delta'//\Delta) = u_g^{-1}(\Delta')//u_g^{-1}(\Delta)$.* ■

4.6.7 Proposition. *Let $g \to^* g'$ by a complete development of a set F of disjoint acyclic redexes of g whose associated rewrite rules are acyclic. Then $U(g) \to^*_t U(g')$ by a complete development of $u_g^{-1}(F)$.* ■

4.6.8 Definition.

(i) In a weakly regular GRS, the relation of *Gross-Knuth* reduction, notation $\to^{GK}$, is defined as follows
$g \to^{GK} g' \Leftrightarrow g \to^* g'$ by complete development of the set of all redexes of g.

(ii) In a weakly regular TreeRS we define *Gross-Knuth* reduction by
$t \to^{GK}_t t' \Leftrightarrow t \to^*_t t'$ by complete development of the set of all redexes of t. ■

4.6.9 Proposition. *Let (T,R) be a weakly regular TRS. Then L(T,R) is weakly regular. Let g and g' be graphs in L(T) such that $g \to^{GK} g'$. Then $U(g) \to^{GK}_t U(g')$.* ■

4.6.10 Proposition. *If every graph in L(T) is acyclic, then L(T,R) is tree-reducible. In particular, a graphical term rewrite system is tree-reducible.* ■

4.6.11 Proposition. *For any TreeRS (T,R) and any graph g in L(T), g is a normal form of L(T,R) iff U(g) is a normal form of (T,R).* ■

Thus in a graphical term rewrite system L(T,**R**), everything which can happen can also happen in the term rewrite system, and all the normal forms are the same. Graph-reducibility may fail, however, since it may be that for some graph g, U(g) has a normal form but g does not.

4.6.12 Theorem. *Every weakly regular TRS is graph-reducible.*

Proof. Let (T,**R**) be a weakly regular TRS. Let g be a graph of L(T,**R**) such that U(g) has a normal form. Proposition 4.6.7 relates the Gross-Knuth reduction sequences for g and U(g) in the following way.

$$g \xrightarrow{GK} g_1 \xrightarrow{GK} g_2 \xrightarrow{GK} \cdots$$

$$U(g) \xrightarrow[t]{GK} U(g_1) \xrightarrow[t]{GK} U(g_2) \xrightarrow[t]{GK} \cdots$$

It is a standard result that for regular TRS's, Gross-Knuth reduction is normalising (Klop (1980)), and the proof carries over immediately to weakly regular TreeRS's. Therefore the bottom line of the diagram terminates with some tree U(g') in normal form such that g reduces to g' in L(T,**R**). Therefore g' is a normal form of g, and (T,**R**) is graph-reducible. ■

4.6.13 Non-left-linearity.

We shall now discuss non-left-linearity, and indicate why we excluded non-left-linear TRS's from consideration. In term rewriting theory, for a term to match a non-linear left-hand side, the subterms corresponding to all the occurrences of a repeated variable must be identical.

Our method of using empty nodes to represent the variables of term rewrite rules suggests a very different semantics for non-left-linear rules. Our representation of a term rule A(x,x) → B would be (n:A(x:⊥,x), n, x). This will only match a subgraph of the form a:A(b: ..., b). That is, the subgraphs matched by the repeated variable must be not merely textually equal, but identical - the very same nodes. If one is implementing graph rewriting as a computational mechanism in its own right, rather than considering it merely as an optimisation of term rewriting, then this form of non-left-linearity may be useful. However, it is not the same as non-left-linearity for term rules.

To introduce a concept more akin to the non-left-linearity of term rules, we could use variables in graphs, just as for terms, instead of empty nodes. A meaning must then be chosen for the matching of a graph A(Var1,Var1) where Var1 is a variable symbol, occurring at two different nodes. Two possibilities naturally suggest themselves. The subgraphs rooted at nodes matched by the same variable may be required to be equivalent, or they may only be required to be tree-equivalent. The latter definition is closer to the term rewriting concept.

When a variable occurs twice or more on the left-hand side of a rule, there is also a problem of deciding which of the subgraphs matched by it is referred to by its occurrences on the right-hand side. One method would be to cause those subgraphs to be first coalesced, replacing the equivalence or tree-equivalence which the matching detected by pointer equality. This technique may be useful in implementing logic programming languages, where non-linearity is much more commonly used than in functional term rewriting. Further investigation of the matter is outside the scope of the present paper.

Lastly, we note that although some term rewriting languages, such as SASL (Turner (1979b)) and Miranda (Turner (1986)), allow non-left-linear rules, they generally interpret the implied equality test neither as textual equality, nor as pointer equality, but as the equality operator of the language (although pointer equality may be used as an optimisation). In these languages, any program containing non-left-linear rules can be transformed to one which does not.

4.7 NORMALISING STRATEGIES

In this section we define the notion of an evaluation strategy in a general setting which includes term and graph rewrite systems. We then study the relationships between strategies for term rewrite systems and for the corresponding graph systems.

4.7.1 Definition.

(i) An *abstract reduction system* (*ARS*) is a pair $(O,\rightarrow)$, where O is a set of *objects* and $\rightarrow$ is a binary relation on O. This notion abstracts from term and graph rewrite systems. The transitive reflexive closure of $\rightarrow$ is denoted by $\rightarrow^*$.

(ii) An element x of an ARS is a *normal form* (nf) if for no y does one have $x\rightarrow y$.

(iii) An element x *has a normal form* if $x\rightarrow^* y$ and y is a normal form.

(iv) A *reduction sequence* of an ARS is a sequence $x_0\rightarrow x_1\rightarrow\ldots\rightarrow x_n$. The *length* of this sequence is n. A sequence of length 0 is *empty*. ■

4.7.2 Definition.

(i) Given an ARS $(O,\rightarrow)$, a *strategy* for this system is a function S which takes each $x\in O$ to a set S(x) of nonempty finite reduction sequences, each beginning with x. Note that S can be empty.

(ii) S is *deterministic* if, for all x, S(x) contains at most one element.

(iii) S is a *one-step strategy* (or *1-strategy*) if for every x in O, every member of S(x) has length 1.

(iv) Write $x\rightarrow_S y$ if S(x) contains a reduction sequence ending with y. By abuse of notation, we may write $x\rightarrow_S y$ to denote some particular but unspecified member of S(x).

(v) An S-*sequence* is a reduction sequence of the form $x_0\rightarrow_S x_1\rightarrow_S x_2\rightarrow_S \ldots\ldots$

(vi) S is *normalising* if for all x having a normal form any sequence $x_0\rightarrow_S x_1\rightarrow_S x_2\rightarrow_S \ldots\ldots$ must eventually terminate with a normal form. ■

4.7.3 Definition.

(i) Let S be a strategy of an ARS $(O,\rightarrow)$. *Quasi*-S is the strategy defined by:

$$\text{quasi-S}(x) = \{x \rightarrow^* x' \rightarrow_S y \mid x' \text{ in } O\}.$$

Thus a quasi-S path is an S-path diluted with arbitrary reduction steps.

(ii) A strategy S is *hypernormalising* if quasi-S is normalising. ■

A 1-strategy for a TreeRS or GRS can be specified as a function which takes the objects of the system to some subset of its redexes. This will be done from now on.

4.7.4 Definition. Let S be a 1-strategy for a TreeRS $(T,\mathbf{R})$. The strategy S_L for the lifted graph rewrite system $L(T,\mathbf{R})$ is defined by $S_L(g) = u_g(S(U(g)))$. ■

For 1-strategies on TreeRS's, this is a natural definition of lifting. For multi-step strategies, it is less clear how to define a lifting, and we do not do so in this paper.

Although a 1-strategy for a TreeRS may be normalising, its lifting may not be. This may be because the lifting of the TreeRS does not preserve normal forms (e.g. as in example 4.6.4), or for more subtle reasons, such as in the following example.

4.7.5 Example. Consider the following TreeRS:
Function symbols: A (binary), B, 1, 2 (nullary).
Rules: $1 \rightarrow 2$, $2 \rightarrow 1$, $A(x,y) \rightarrow B$.

By stipulating that A is binary and B, 1, and 2 are nullary, we mean, as discussed following definitions 4.4.7 and 4.5.1, that trees not conforming to these arities are not included in the system. Define a strategy S as follows (where the redexes chosen by S are boldfaced):

$A(\mathbf{1},1) \rightarrow A(2,1)$ $\quad A(2,\mathbf{2}) \rightarrow A(2,1)$
$A(\mathbf{x},\mathbf{y}) \rightarrow B$, if neither of the preceding cases applies

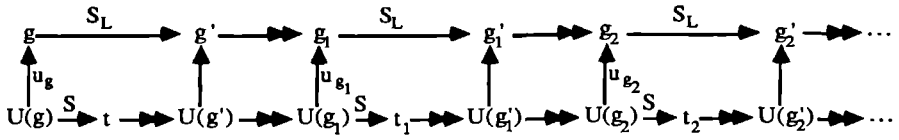
S takes the tree A(1,1) to normal form B in two steps. S_L takes the graph A(x:1,x) to A(x:2,x) and back again in an infinite loop.

The next theorem shows that if a 1-strategy S for a TreeRS is hypernormalising, then S_L is hypernormalising for the corresponding GRS.

4.7.6 Theorem. *Let (T,R) be a TreeRS and let S be a 1-strategy for it. Let (G,R') be the lifting of (T,R). If S is hypernormalising then S_L is hypernormalising.*

Proof. Assume S is hypernormalising. Let g be a graph in G having a normal form, and consider a quasi-S_L reduction sequence starting from g.

By proposition 4.6.7 and the definition of S_L, we can construct the following diagram, where the top line is the quasi-S_L reduction sequence:



Since g has a normal form, so does U(g), so since quasi-S is normalising, the bottom line must stop at some point, with a normal form of U(g). Therefore the top line also stops, and must do so with a graph which unravels to the normal form in the bottom line. ■

4.7.7 Example. The converse does not hold. If S_L is hypernormalising, S need not be. Consider the following TRS.

Function symbols: A (binary), B (nullary)
Rules: $A(x,y) \to B \quad A(x,y) \to A(x,x)$

Every non-normal form of this system has the form $A(\alpha,\beta)$ for some terms α and β. Let S be the strategy:

$$A(\alpha,\beta) \to_S A(\alpha,\alpha) \quad (\text{if } \alpha \neq \beta)$$

$$A(\alpha,\alpha) \to_S B$$

The first S_L-step in any quasi-S_L-sequence will produce either a graph of the form $A(x{:}\alpha,x)$ or the normal form B. In the former case, whatever extra steps are then inserted, the result can only be either another term of the same form or B. In the former case, the next S_L-step will reach B. Therefore S_L is hypernormalising. However, S is not hypernormalising. A counterexample is provided by the term A(A(B,B),B). An infinite quasi-S sequence beginning with this term is:

$$A(A(B,B),B) \to_S A(A(B,B),A(B,B)) \to A(A(B,B),B) \to_S A(A(B,B),A(B,B)) \to \ldots$$

4.7.8 Corollary. *If a TreeRS (T,R) has a hypernormalising 1-strategy then it is graph reducible.*

Proof. By theorem 4.7.5 the lifting (G,R) of the TreeRS has a normalising strategy. Now assume U(g) = t. Suppose g has no nf. Then the S_L path of g is infinite. This gives, by the construction of 4.7.6, an infinite quasi-S-path of t, hence t has no normal form. ■

An application of this result is that strongly sequential TRS's (in the sense of Huet & Lévy (1979)) are graph reducible. This follows from their theorem that the 1-strategy which chooses any needed redex is hypernormalising.

The condition that a strategy be hypernormalising is unnecessarily strong. Inspection of the proofs of the preceding theorem and corollary shows that the following weaker concept suffices.

4.7.9 Definition.

(i) Let S be a 1-strategy of a TreeRS (T,**R**). Then *sib*-S is the strategy defined by:
sib-S(x) = { $x \to_S y \to^* z$ | the sequence $y \to^* z$ consists of siblings of S(x)}.
That is, a sib-S path is an S-path diluted with arbitrary sib-steps from the reduction relation.

(ii) A strategy S is *sib-normalising* if sib-S is normalising. ■

4.7.10 Theorem. *Let (T,**R**) be a TreeRS and let S be a 1-strategy for it. Let (G,**R'**) be the lifting of (T,**R**). If S is sib-normalising then S_L is sib-normalising and (T,**R**) is graph-reducible.*

Proof. Immediate from the proofs of theorem 4.7.6 and corollary 4.7.8. ■

4.8 CONCLUSION

Graph rewriting is an efficient way to perform term rewriting. We have shown:

1. Soundness: for all TRS's, graph rewriting cannot give incorrect results.
2. Completeness: for weakly regular TRS's, graph rewriting gives all results.
3. Many normalising strategies (the hypernormalising, or even the sib-normalising ones) on terms can be lifted to graphs to yield normalising strategies there. In particular, for strongly sequential term rewrite systems, the strategy of contracting needed redexes can be lifted to graphs.

We have also given counterexamples illustrating incompleteness for non-weakly regular TRS's and for liftings of non-sib-normalising strategies.

5

LEAN: AN INTERMEDIATE LANGUAGE BASED ON GRAPH REWRITING

H.P. Barendregt$_2$, M.C.J.D. van Eekelen$_2$, J.R.W. Glauert$_1$,

J.R. Kennaway$_1$, M.J. Plasmeijer$_2$ and M.R. Sleep$_1$.

[1]School of Information Systems, University of East Anglia, Norwich, Norfolk NR4 7TJ, U.K.,
partially supported by the U.K. ALVEY project,

[2]Computing Science Department, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands,
partially supported by the Dutch Parallel Reduction Machine Project.

Abstract.

Lean is an experimental language for specifying computations in terms of graph rewriting. It is based on an alternative to Term Rewriting Systems (TRS) in which the terms are replaced by graphs. Such a Graph Rewriting System (GRS) consists of a set of graph rewrite rules which specify how a graph may be rewritten. Besides supporting functional programming, Lean also describes imperative constructs and allows the manipulation of cyclic graphs. Programs may exhibit non-determinism as well as parallelism. In particular, Lean can serve as an intermediate language between declarative languages and machine architectures, both sequential and parallel. This paper is a revised version of Barendregt *et al* (1987b) which was presented at the ESPRIT, PARLE conference in Eindhoven, The Netherlands, June 1987.

5.1 INTRODUCTION

Emerging technologies (VLSI, wafer-scale integration), new machine architectures, new language proposals and new implementation methods (Vegdahl (1984)) have inspired the computer science community to consider new models of computation. Several of these developments have little in common with the familiar Turing machine model. It is our belief that in order to be able to compare these developments, it is necessary to have a novel computational model that integrates graph manipulation, rewriting, and imperative overwriting. In this paper we present Lean, an experimental language based on such a model. In our approach we have extended Term Rewriting Systems (O'Donnell (1985), Klop (1985)) to a model of general graph rewriting. Such a model will make it possible to reason about programs, to prove correctness, and to port programs to different machines.

A Lean computation is specified by an initial graph and a set of rules used to rewrite the graph to its final result. The rules contain graph patterns that may match some part of the graph. If the graph matches a rule it can be rewritten according to the specification in that rule. This specification makes it possible first to construct an additional graph structure and then link it into the existing graph by redirecting arcs.

Lean programs may be non-deterministic. The semantics also allows parallel evaluation where candidate rewrites do not interfere. There are few restrictions on Lean graphs (cycles are allowed

and even disconnected graphs). Lean can easily describe functional graph rewriting in which only the root of the subgraph matching a pattern may be overwritten. Through non-root overwrites and use of global nodeids in disconnected patterns imperative features are also available.

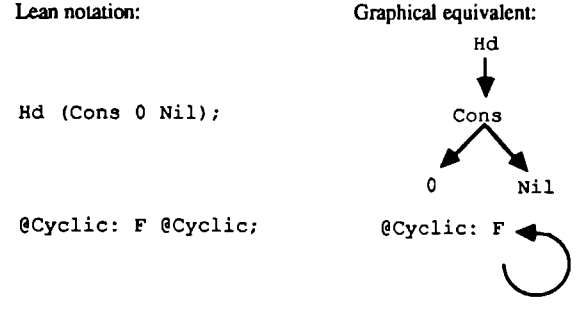
In this paper we first introduce Lean informally. Then we show how a Lean program can be transformed to a program in canonical form with the same meaning. The semantics of Lean is explained using this canonical form. The semantics adopted generalises Staples' model of graph rewriting (Staples (1980a)), allowing, for example, multiple redirections. A formal description of the graph rewriting model used in this paper can be found in Barendregt *et al.* (1987a), as it applies to the special case of purely declarative term rewriting. After explaining the semantics we give some program examples to illustrate the power of Lean. The syntax of Lean and the canonical form is given in the appendix.

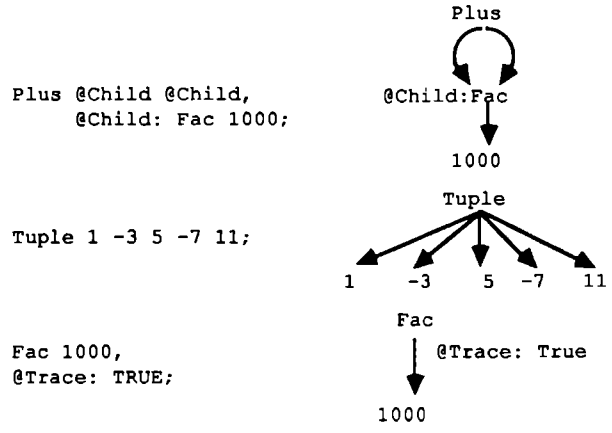
5.2 GENERAL DESCRIPTION OF LEAN

5.2.1 LEAN GRAPHS

The object that is manipulated in Lean is a directed graph called the *data graph*. When there is no confusion, the data graph is simply called *the graph*. Each node in the graph has an unique identifier associated with it (the *node identifier* or *nodeid*). Furthermore a node consists of a *symbol* and a possibly empty sequence of nodeids which define arcs to nodes in the graph. We do not assume that symbols have fixed arities. The data graph is a *closed* graph, that is, it contains no variables. It may be cyclic and may have disjoint components. This class of data graphs is, abstractly, identical to that discussed in Barendregt *et al.* (1987a). We refer to that paper for a formal discussion of the precise connection between graphs and terms.

Programming with pictures is rather inconvenient so we have chosen a linear notation for graphs. In this notation we use brackets to indicate tree structure and repeated nodeids to express sharing, as shown in the examples below. Nodeids are prefixed with the character '@'. Symbols begin with an upper-case character.





5.2.2 LEAN PROGRAMS

A *Lean program* consists of a set of *rewrite rules* including a *start rule*. A rewrite rule specifies a possible transformation of a given graph. The initial graph is not specified in a Lean program (see also section 5.4.2).

The left-hand-side of a rewrite rule consists of a Lean graph which is called a *redex pattern*. The right-hand-side consists of a (possibly empty) Lean graph called the *contractum pattern* and, optionally, a set of *redirections*. The patterns may be disconnected graphs and they are *open*, that is, they may contain *nodeid variables*. These are denoted by identifiers starting with a lower-case letter. Nodeids of the data graph may also occur in the rules. These are called *global nodeids*. When there can be no confusion with the nodeids in the data graph, we sometimes refer to the nodeid variables and the global nodeids in the rules just as nodeids. Here is an example program:

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

Fac 0              →  1                              |
Fac n:INT          →  *I n (Fac (-I n 1))            ;

F (F x)            →  x                              ;

Start              →  Fac (Hd (Cons 1000 Nil))       ;
```

The first symbol in a redex pattern is called the *function symbol*. Rule alternatives starting with the same function symbol are collected together forming a *rule*. The alternatives of a rule are separated by a '|'. Note that function symbols may also occur at other positions than the head of the pattern. A symbol which does not occur at the head of any pattern in the program is called a *constructor symbol*.

5.2.3 REWRITING THE DATA GRAPH

The initial graph of a Lean program is rewritten to a final form by a sequence of applications of individual rewrite rules. A rule can only be applied if its redex pattern matches a subgraph of the data graph. A redex pattern in general consists of variables and symbols. An *instance* of a redex

pattern is a subgraph of the data graph, such that there is a mapping from the pattern to that subgraph which preserves the node structure and is the identity on constants. This mapping is also called a *match*. The subgraph which matches a redex pattern is called a *redex* (*red*ucible *ex*pression) for the rule concerned.

We will use the following rules which have a well-known meaning as a running example to illustrate several concepts of Lean.

```
Add Zero z          →   z                          !   (1)
Add (Succ a) z      →   Succ (Add a z)             ;   (2)
```

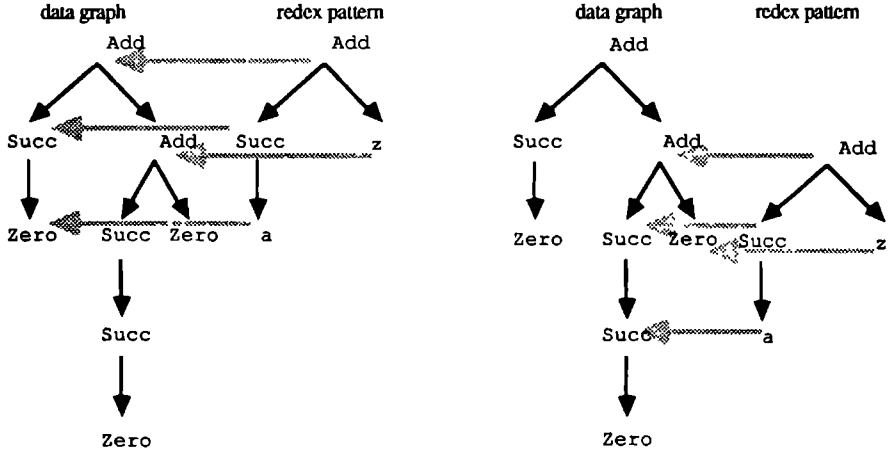
Now assume that we have the following data graph:

```
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)      ;
```

There are two redexes:

```
                a                  z
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)   redex matching rule 2

                          a              z
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)   redex matching rule 2
```

In graphical form this is:



Note that there may be several rules for which there are redexes in the graph. A rule may match several redexes and a redex can match several rules. For instance, in the example above there is only one rule which matches any part of the data graph, but it matches two redexes. In general, therefore, there are many rewriting sequences for a given graph.

Evaluation of a Lean program is controlled by a *rewriting strategy*. In its most general form:

1. It decides which rewritings to perform.
2. It decides when to perform no further rewritings. The graph at this point is said to be in strategy normal form, or briefly, in normal form.
3. It specifies what part of the resulting graph is the outcome of the computation.

For the purposes of graphical implementations of functional languages, strategies need only consider the subgraph of nodes accessible from the data root, for the purposes of identifying both redexes and terminal states. However, more general applications of Lean may not wish to be constrained in this way: for example, graphical rewrite rules may be used to represent non-terminating behaviours of practical interest such as operating systems.

The choices made by a rewriting strategy may affect the efficiency of rewriting, as well as its termination properties. We have not imposed an a priori restriction on the reduction strategy with which a Lean program should be evaluated, e.g. the rules are ordered but the strategy may or may not make use of this ordering. In the future we aim to incorporate facilities into Lean to permit programmer control of strategy where necessary. this would enable the user to guide the evaluation.

Once the strategy has chosen a particular redex and rule, rewriting is performed. The first step is to create an instantiation of the graph pattern specified on the right-hand-side of the chosen rule. This instantiation is called the *contractum*. In general this contractum has links to the original graph since references to nodeid variables from the left-hand-side are linked to the corresponding nodes identified during matching. A new data graph is finally constructed by redirecting some arcs from the original graph to the contractum. In most cases all arcs to the root node of the redex are redirected to the root node of the contractum as in Staples' model (Staples (1980a)). This has an effect similar to "overwriting" the root of the redex with the root of the contractum. This is what happens when no redirections are given explicitly in the rule. Explicit redirection of arbitrary nodes is also possible.

The process of performing one rewrite step is often called a *reduction*. The graph after one reduction is called the *result* of the reduction. Initially, the data graph contains a node with the symbol `Start`. Hence, the rewriting process can begin with matching the start rule and hereafter rewriting is performed repeatedly until the strategy has transformed the graph to one which it deems to be in normal form.

Barendregt *et al.* (1987a) gives a formal discussion of how graph rewrite rules with root-only redirection model term rewriting, and proves certain soundness and completeness results. The definition of rewriting given in that paper only covers rules of this form, but the extension of the formal description to the general cases of multiple and/or non-root redirection is straightforward.

The data graph of the previous example can be rewritten in the following way:

```
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)                    → (2)
Succ (Add Zero (Add (Succ (Succ Zero)) Zero))                    → (1)
Succ (Add (Succ (Succ Zero)) Zero)                               → (2)
Succ (Succ (Add (Succ Zero) Zero))                               → (2)
Succ (Succ (Succ (Add Zero Zero)))                               → (1)
Succ (Succ (Succ Zero))
```

Note that in this example the graph was actually a tree, and remained a tree throughout. There was no difference with a Term Rewriting System. In the following example there is a data graph

in which parts are shared. Rewriting the shared part will reduce the number of rewriting steps compared to an equivalent Term Rewriting System.

```
Add @X @X,  @X: Add (Succ Zero) Zero                          → (2)
Add @X @X,  @X: Succ (Add Zero Zero)                          → (1)
Add @X @X,  @X: Succ Zero                                     → (2)
Succ (Add @Z @X),  @X: Succ @Z,  @Z: Zero                     → (1)
Succ (Succ Zero)
```

5.2.4 PREDEFINED DELTA RULES

For practical reasons it is convenient that rules for performing arithmetic on primitive types (numbers, characters etc.) are predefined and efficiently implemented. In Lean a number of basic constructors for primitive types such as `INT`, `REAL` and `CHAR` are predefined. Representatives of these types can be denoted: for instance `5` (an integer), `5.0` (a real), `'5'` (a character). Basic functions, called *delta rules*, are predefined on these basic types.

The actual implementation of a representative of a basic type is hidden for the Lean programmer. It is possible to denote a representative, pass a representative to a function or delta-rule and check whether or not an argument is of a certain type in the redex pattern.

```
Nfib 0          → 1                                           |
Nfib 1          → 1                                           |
Nfib n:INT      → ++I (+I (Nfib (-I n 1)) (Nfib (-I n 2)))    ;
```

In this example '`0`' is an abbreviation of `INT ...` which is a denotation for some hidden representation of the number 0 (analogue for '`1`' and '`2`'), '`+I`', '`-I`' and '`++I`' are function symbols for predefined delta rules defined on these representations. Hence, an integer consists of the unary constructor `INT` and an unknown representation. Note that in general one is allowed to specify just the constructor in the redex pattern of a rule. The value can be passed to a function by passing the corresponding nodeid (`n` in the example).

These predefined rules are however not strictly necessary. For instance, one could define numbers as: `INT Zero` to denote `0`, `INT (Succ Zero)` to denote `1`, `INT (Succ (Succ Zero))` to denote `2` etc., and define a function for doing addition

```
PlusI (INT x) (INT y)  → INT (Add x y)
```

where `Add` is our running example. This kind of definition makes it possible to do arithmetic in a convenient way. However, for an efficient implementation one would probably not choose such a Peano-like representation of numbers, but prefer to use the integer and real representation and the arithmetic available on the computer.

5.3 TRANSLATING TO CANONICAL FORM

Lean contains syntactic sugar intended to make programs easier to read and write. Explaining the semantics of Lean will be done with a form with all syntactic sugar removed known as *Canonical Lean*. In this section we show how a Lean program can be transformed to its canonical form. Canonical Lean programs are valid Lean programs and are unaffected by this translation

procedure. Every Lean program can be seen as a shorthand for its canonical form. Note that this section is all about syntax. The semantics of the canonical form are explained in section 5.4.

In the canonical form every node has a definition and definitions are not nested. Every redirection, including any redirection of the root, is done explicitly and in patterns all arguments of constructors are specified. In this canonical form a rewrite rule has the following syntax:

```
Graph    →    [ Graph, ] Redirections
```

The first `Graph` is the redex pattern. The second is the optional contractum pattern. Each pattern is represented as a list of node definitions of the form:

```
Nodeid:  Symbol { Nodeid }
```

Braces mean zero or more occurrences. The initial *Nodeid* identifies the node, *Symbol* is some function or constructor symbol and the sequence of nodeids identifies zero or more child nodes. Occurrences of nodeids before a colon are *defining* occurrences. Every nodeid must have at most one defining occurrence within a rule. Defining occurrences of global nodeids are allowed on the left-hand-side only. Within a rule a nodeid which appears on the right-hand-side must either have a definition on the right-hand-side or it must also appear on the left-hand-side.

5.3.1 ADD EXPLICIT NODEIDS AND FLATTEN

In the canonical form all nodes have explicit nodeids and there are no nested node definitions. Hence in each rule we have to introduce a new unique nodeid variable for every node that does not yet have one. Every nested node definition in the rule is then replaced by an application of the corresponding nodeid variable, and the definitions are moved to the outer level. Applying this transformation to our running example gives:

```
Add      y   z,
y: Zero              →   z                          |
Add      y   z,
y: Succ  a           →   m: Succ n,
                         n: Add  a z                ;
```

All arguments of symbols (such as `Add` and `ucc`) have now become nodeids and brackets are no longer needed.

5.3.2 SPECIFY THE ARGUMENTS OF CONSTRUCTORS

In Lean one may write the following function which checks to see if a list is empty:

```
IsNil n,
n: Nil               →   t: TRUE                    |
IsNil n,
n: Cons              →   t: FALSE                   ;
```

`Cons` is a binary constructor symbol, but in Lean one may omit the specification of the arguments if they are not used elsewhere in the rule. This is not allowed in the canonical form hence the arguments are made explicit by introducing two new nodeid variables. Transformation of the example above will give:

```
IsNil n,
n: Nil              →   t: TRUE                    |
IsNil n,
n: Cons y z         →   t: FALSE                   ;
```

5.3.3 MAKE ROOT REDIRECTIONS EXPLICIT

The meaning of both rules in the running example is that the root of the pattern is redirected to the root of the contractum. Redirections are always made explicit in the canonical form. If no redirections are specified explicitly, a redirection is introduced to redirect the redex root to the contractum root. Note that if the right-hand-side of a rule consists only of a nodeid, the root of the redex is redirected to this nodeid. The running example with explicit redirections now becomes:

```
x: Add  y   z,
y: Zero             →   x := z                     |
x: Add  y   z,
y: Succ     a       →   m: Succ n,
                        n: Add  a z,
                        x := m                     ;
```

5.4 SEMANTICS OF LEAN

5.4.1 GRAPH TERMINOLOGY

- Let F be a set of symbols and N be a set of nodes.
- Further, let C be a function (the *contents function*) from N to $F \times N^*$.
- Then C specifies a *Lean Graph* over F and N.
- If node n has contents F n_1 n_2 ... n_k we say the node contains *symbol* F and *arguments* $n_1, n_2, \ldots, n_k$.
- There is a distinguished node in the graph which is the *root* of the graph.

In standard graph theory, a Lean graph is a form of directed graph in which each node is labelled with a symbol, and its set of out-arcs is given an ordering. In Lean nodes are denoted by their names, i.e. their nodeids. The canonical form defined in section 5.3 can be regarded as a tabulation of the contents function. We will explain the semantics of Lean using this canonical form.

5.4.2 THE INITIAL GRAPH

The initial graph is not specified in a program. It always takes the following form:

```
@DataRoot:    Graph @StartNode @GlobId1 @GlobId2 ... @GlobIdm,
@StartNode:   Start,
@GlobId1:     Initial,
@GlobId2:     Initial,
...
@GlobIdm:     Initial;
```

The root of the initial graph contains the nodeid of the start node which initially contains the symbol `Start`. The root node will always contain the root of the graph to be rewritten.

Furthermore the root node contains all global nodeids addressed in the Lean rules. The corresponding nodes are initialised with the symbol `Initial`.

5.4.3 OPERATIONAL SEMANTICS FOR REWRITING

Let G be a Lean graph, and R the ordered set of rewrite rules. A reduction option, or *redop*, of G is a triple T which consists of a redex g, a rule r and a match μ. The match μ is a mapping from the nodeids of the redex pattern p to the nodeids of the graph G such that for every nodeid x of p, if $C_p(x) = s\ x_1\ x_2\ ...\ x_n$ then $C_g(\mu(x)) = s\ \mu(x_1)\ \mu(x_2)\ ...\ \mu(x_n)$. That is, μ preserves node structure. Note that μ maps multiple occurrences of nodeids in a redex pattern to one and the same node in the graph. A redop introduces an available choice for rewriting the graph. A redop that is chosen is called a *rewrite* of the graph. The process of performing a rewrite is also called *rewriting*.

The contractum pattern may contain nodeid variables which are not present in the redex pattern. These correspond to the identifiers of new nodes to be introduced during rewriting. The mapping μ' is introduced taking as its domain the set of nodeid variables which only appear in the contractum pattern. Each of these is mapped to a distinct, new, nodeid which does not appear in G or R.

The domains of μ and μ' are distinct, but every nodeid variable in the contractum pattern is in the domain of one or the other. In order to compute the result of a rewrite one applies the mapping μ'' formed by combining μ and μ', to the contractum pattern resulting in the *contractum*.

Finally the new graph is constructed by taking the union of the old graph and the contractum, replacing nodeids in this union (and in the case that global nodeids are mentioned also in the rules) as specified by the redirections in the rewrite rule of the chosen redop.

Hence rewriting involves a number of steps:

1. A redop is chosen by the rewriting strategy. This gives us a redex in the graph G, a rule which specifies how to rewrite the redex and a mapping μ.

2. The contractum is constructed in the following way.
   - invent new nodeids (not present in G or R) for each variable found only in the contractum pattern. This mapping is called μ'.
   - apply μ'', the combination of μ and μ', to the contractum pattern of the rule yielding the contractum graph C. Note that the contractum pattern, and hence C, may be empty.

3. The new graph G' is constructed by taking the union of G and C.

4. Each redirection in a rule takes the form O := N. In terms of the syntactic representation, this is performed by substituting N for every applied occurrence of O in the graph G' and in the rules R. The definition of O still remains. The nodeids O and N are determined by applying μ'' to the left-hand-side and the right-hand-side of the redirection. All redirections specified in the rule are done in parallel. This results in the new graph G".

The strategy will start with a rewrite rule which matches the symbol `Start` in the initial graph. When a computation terminates, its *outcome* is that part of the final graph which is accessible from the root. Thus a "garbage collection" is assumed to be performed at the end of the computation only. A real implementation may optimise this by collecting nodes earlier, if it can predict that so doing will not affect the final outcome. Which nodes can be collected earlier will in general depend on the rule-set of the program and the computation strategy being used. Note that before the computation has terminated, nodes which are inaccessible from the root may yet have an effect on the final outcome, so they cannot necessarily be considered garbage. For certain strategies and rule-sets they will be, but inaccessibility is not in itself the definition of garbage.

Redirection of global nodeids has as a consequence that all references to the original global nodeid have to be changed. An efficient implementation of redirection can be obtained by overwriting nodes and/or using indirection nodes. Also references in the rewrite rules to global nodeids have to be redirected. Hence global nodeids 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). If global nodeids are redirected, also references to them in the rewrite rules change accordingly.

5.4.4 A SMALL EXAMPLE

We return to our running example with a small initial graph and see how rewriting proceeds. The rewriting strategy we choose will rewrite until the data graph contains no redexes only examining nodes accessible from the `@Dataroot`.

```
x: Add  y   z,
y: Zero                 →   x := z                          |   (1)
x: Add  y   z,
y: Succ a               →   m: Succ n,
                            n: Add  a z,
                            x := m                          ;   (2)
x: Start                →   m: Add  n o,
                            n: Succ o,
                            o: Zero,
                            x := m                          ;   (3)
```

Initially we have the following graph G:

```
@DataRoot : Graph @StartNode,
@StartNode: Start;
```

We now follow the rewrite steps.

1. The start node is the only redex matching rule (3). The mapping is trivial: $\mu(x)$ = `@StartNode` and the redex in the graph is:

   ```
   @StartNode: Start;
   ```

2. The variables found only in the contractum pattern are m, n, and o. We invent a new nodeid for each of these, defining a mapping $\mu'(m)$ = `@A`, $\mu'(n)$ = `@B`, $\mu'(o)$ = `@C`. Applying μ'', the combination of μ and μ', to the contractum pattern gives the contractum C:

```
@A: Add  @B @C,
@B: Succ @C,
@C: Zero;
```

In fact, for this example, μ is not used in making the contractum, as the contractum pattern does not refer to x.

3. The union of C and G is G':

```
@DataRoot : Graph @StartNode,
@StartNode: Start,
@A: Add  @B @C,
@B: Succ @C,
@C: Zero;
```

4. We have to redirect μ"(x) = `@StartNode` to μ"(m) = `@A`. All applied occurrences of `@StartNode` will be replaced by occurrences of `@A`. The graph G" after redirecting is:

```
@DataRoot : Graph @A,
@StartNode: Start,
@A: Add  @B @C,
@B: Succ @C,
@C: Zero;
```

This completes one rewrite. The start node will not be examined by the strategy anymore, as it is inaccessible from `@DataRoot`. Therefore it can be considered as garbage and it will be thrown away. The strategy will not stop yet because the graph still contains a redex accessible from the `@DataRoot`.

1. The strategy will choose the only redop. It matches rule 2: μ(x) = `@A`, μ(y) = `@B`, μ(z) = `@C`, μ(a) = `@C`;

2. Invent new nodeids and map the variables as follows: μ'(m) = `@D`, μ'(n) = `@E`. The contractum is:

```
@D: Succ @E,
@E: Add  @C @C;
```

3. The union of the graph and the contractum is:

```
@DataRoot: Graph @A,
@A: Add  @B @C,
@B: Succ @C,
@C: Zero,
@D: Succ @E,
@E: Add  @C @C;
```

4. We have to redirect μ"(x) = `@A` to μ"(m) = `@D`. Then after removing garbage the graph is:

```
@DataRoot: Graph @D,
@C: Zero,
@D: Succ @E,
@E: Add  @C @C;
```

It is now clear how this process may continue: `@E` is a redex and it matches rule 1: μ(x) = `@E`, μ(y) = `@C`, μ(z) = `@C`. The strategy chooses this redop, there is no new contractum graph but just a single redirection which takes μ"(x) = `@E` to μ"(z) = `@C` yielding the expected normal form:

```
@DataRoot: Graph @D,
@C: Zero,
@D: Succ @C;
```

5.5 SOME LEAN PROGRAMS

5.5.1 MERGING LISTS

The following Lean rules can merge two ordered lists of integers (without duplicated elements) into a single ordered list (without duplicated elements).

```
Merge   Nil     Nil     →   Nil                                    |
Merge   f:Cons Nil      →   f                                      |
Merge   Nil     s:Cons  →   s                                      |
Merge   f:(Cons a b)
        s:(Cons c d)    →   IF   (<I a c)
                                 (Cons a (Merge b s))
                                 (IF  (=I a c)
                                      (Merge f d)
                                      (Cons c (Merge f d)))        ;
```

=I and IF are predefined delta rules with the obvious semantics. Note that the right-hand-side of the last rule uses an application of the argument as a whole as well as its decomposition.

5.5.2 HIGHER ORDER FUNCTIONS, CURRYING

In this example we show how higher-order functions are treated in Lean, by giving the familiar definition of the function `Map`.

```
Map f    Nil            →   Nil                                  |    (1)
Map f    (Cons a b)     →   Cons (Ap f a) (Map f b)              ;    (2)
Ap  (*I a)    b         →   *I a b                               ;    (3)
Start                   →   Map (*I 2) (Cons 3 (Cons 4 Nil))     ;    (4)
```

This can be rewritten, for example, in the following way:

```
Start                                                          → (4)
Map (*I 2) (Cons 3 (Cons 4 Nil))                               → (2)
Cons (Ap @L 3) (Map @L (Cons 4 Nil)),      @L:*I 2             → (3)
Cons (*I 2 3) (Map @L (Cons 4 Nil)),       @L:*I 2             → (*I)
Cons 6 (Map @L (Cons 4 Nil)),              @L:*I 2             → (2)
Cons 6 (Cons (Ap @L 4) (Map @L Nil)),      @L:*I 2             → (3)
Cons 6 (Cons (*I 2 4) (Map @L Nil)),       @L:*I 2             → (*I)
Cons 6 (Cons 8 (Map @L Nil)),              @L:*I 2             → (1)
Cons 6 (Cons 8 Nil)
```

Rule `(3)` of this example will rewrite `(Ap (*I 2) 3)` to its uncurried form `(*I 2 3)` which makes multiplication possible. One will need such an "uncurry" rule for every function which is used in a curried manner. Note that during rewriting the node `@L: (*I 2)` is shared. In this case sharing only saves space, but not computation.

5.5.3 GRAPHS WITH CYCLES

The following example is a solution for the Hamming problem: it computes an ordered list of all numbers of the form 2^n3^m, with n, m $\geq$ 0. We use the map and merge functions of the previous examples.

```
Ham → Cons 1 (Merge (Map (*I 2) Ham) (Map (*I 3) Ham))          ;
```

A more efficient solution to this problem can be obtained by means of creating cyclic sharing in the contractum making heavy use of computation already done. This cyclic solution has a polynomial complexity where the previous one has an exponential complexity. The new definition is:

```
x: Ham   → Cons 1 (Merge (Map (*I 2) x) (Map (*I 3) x))      ;
```

5.5.4 COPYING A TREE STRUCTURE

This example is very straightforward if the structure of tree nodes is known. Here is a program which copies a binary tree structure.

```
Copy     (Bin left right)    → Bin (Copy left) (Copy right)      |
Copy     Leaf                → Leaf                              ;
```

In the present version of Lean it is not possible to copy an arbitrary unknown data structure. We hope to support more general solutions in a future version of Lean.

5.5.5 COUNTING SPECIFIC REWRITES VIA GLOBAL ASSIGNMENT

```
r: Hd (Cons a b),
@HdCount: Total n:INT  →   newvalue: Total (++I n),
                           r := a,
                           @HdCount := newvalue                ;

r: Start               →   nr: Hd (Cons 1 (Cons 2 Nil)),
                           initvalue: Total 0,
                           r := nr,
                           @HdCount := initvalue               ;
```

We are dealing with disconnected graphs and patterns in this example. The global nodeid `@HdCount` in the graph is addressed in a rewrite rule. The integer value in `@HdCount` will be increased each time a head of a list is taken. Global nodeids and arbitrary redirections in rewrite rules make other styles of programming possible involving globals and side effects. Here, the retention of the canonical notation forces the user to make his text inelegant. Perhaps a useful danger signal, both to reader and writer?

5.5.6 UNIFICATION USING REDIRECTION

This program implements a simple unification algorithm. It operates on representations of two types, returning "cannot unify" in case of failure. The types are contructed from three basic types `I`, `B` and `Var` and a composing constructor `Com`. Different type variables are represented by distinct nodes. Repeated type variables are represented by shared nodes. References to such a shared node are taken to be references to the same type variable.

```
r: Start                                  →   Unify t1 t2 r,
                                              t1: Com i t1,
                                              t2: Com i (Com i t2),
                                              i: I                                    ;

   Unify x       x       r                  →   x                                       |
o: Unify t1:(Com x y)  t2:(Com p q)          r
                                          →   n: Com (Unify x p r) (Unify y q r),
                                              o := n, t1 := n, t2 := n                |
o: Unify t1:Var t2            r           →   o := t2, t1 := t2                       |
o: Unify t1     t2:Var        r           →   o := t1, t2 := t1                       |
   Unify t1:Com t2:I          r           →   n: "cannot unify", r := n               |
   Unify t1:Com t2:B          r           →   n: "cannot unify", r := n               |
   Unify t1:I   t2:Com        r           →   n: "cannot unify", r := n               |
   Unify t1:B   t2:Com        r           →   n: "cannot unify", r := n               |
   Unify t1:I   t2:B          r           →   n: "cannot unify", r := n               |
   Unify t1:B   t2:I          r           →   n: "cannot unify", r := n               ;
```

Of course this does not solve the general unification problem, but it gives an idea of the power of redirection and how it might be used to solve this kind of problems.

5.5.7 COMBINATORY LOGIC

Here we show the Lean equivalent of a well-known TRS using explicit application: combinatory logic.

```
Ap (Ap (Ap S a) b) c)  →   Ap (Ap a c) (Ap b c)                    |
Ap (Ap K a) b)         →   a                                       ;

Start    →    Ap (Ap (Ap S (Ap K K)) (Ap S K)) (Ap (Ap K K) K))    ;
```

5.6 FUTURE WORK

Lean is the result of collaboration between two research groups: the Dutch Parallel Reduction Machine (DPRM) group at Nijmegen and the Declarative Alvey Compiler Target Language (DACTL) group at UEA. Recognising the current instability of emerging languages and architectures, both groups wish to identify a computational model appropriate to a new generation rewriting model of computing. The DPRM group has developed a subset of Lean, called Clean (Brus *et al.* (1987)), for the support of purely functional languages. Dactl0 (Glauert *et al.* (1987c)) predates Lean, and includes some concepts not present in Lean. In the future, our groups plan to continue to collaborate on further developing and refining the computational model and the Lean language based on it. It is intended that later versions of Lean and Dactl will converge.

Because rewriting strategies have a critical influence on efficiency and outcome, future versions of Lean aim to offer the programmer explicit control. Strategies should be based mainly on local information so that concurrent evaluation is not constrained. One approach is to employ fine grain control annotations so that a rule may nominate which of the nodes it creates should be considered as roots for future redexes. Dactl0 adopts this approach. Its main advantage is that a simple execution model is obtained. Another approach is to have a high level specification of strategies and a formalism for combining strategies during evaluation. This approach holds out promise for global reasoning (van Eekelen & Plasmeijer (1986)). We believe that the way

forward should involve a careful combination of these approaches. At the high level formally specified strategy information should be used, allowing analysis and transformation of programs using abstract interpretation techniques. Correctness preserving translation tools would then convert such a program into a form using a small set of well-designed control primitives suitable for efficient parallel implementation.

Besides strategies, there are several other concepts that may be incorporated in Lean in the near future. These include: more general typing; annotations to allow compiler optimisations; interfacing with the outside world; modules and separate compilation facilities; support for unification.

5.7 CONCLUSIONS

Lean is an experimental language for specifying computations in terms of graph rewriting. It is very powerful since there are few restrictions on the graph that is transformed and the transformations that can be performed.

The graph rewriting model underlying Lean is of independent interest as a general model of computation for parallel architectures. It includes as special cases, more restricted systems, such as Graph Rewriting Systems which model Term Rewriting Systems. For these GRS's certain soundness and completeness results are shown in Barendregt *et al.* (1987a).

Lean is designed to be a useful intermediate language for those language implementations which rely on graph rewriting. Compilers targeted to Lean are being implemented for functional languages. An interpreter for Lean is available (Jansen (1987)) which allows mixing of several reduction strategies. A compiler for a restricted subset of Lean (Clean) is running on a Vax750 (Unix) (Brus *et al.* (1987)). The performance is encouraging.

The design of Lean has heavily influenced the design of Dactl1 (Glauert *et al.* (1987d), Glauert *et al.* (1987a)), which the UK Flagship machine (Watson & Watson (1987)) supports. Apart from some surface syntax differences which reflect local prejudices, Dactl1 is essentially Lean PLUS fine grain control markings MINUS global terms. The reduction relation is identical: all that Dactl1 control markings do is to prohibit certain reduction sequences.

5.8 ACKNOWLEDGEMENTS

We would like to thank Jan-Willem Klop of the Centre for Mathematics and Computer Science in Amsterdam for his explanations and Nic Holt of ICL for his valuable comments.

5.A APPENDIX: SYNTAX

```
LeanProgram      = {Rule}.
Rule             = RuleAlt {'|' RuleAlt} ';'.
RuleAlt          = Graph '->' Graph [',' Redirections]
                 | Graph '->' Redirections.
Graph            = [Nodeid ':'] Node {',' NodeDefinition}.
NodeDefinition   = Nodeid ':' Node .
Node             = Symbol {Term}.
Term             = Nodeid
                 | [Nodeid ':'] Symbol
                 | [Nodeid ':'] '(' Node ')'.
Redirections     = Redirection {',' Redirection}
                 | Nodeid {',' Redirection}.
Redirection      = Nodeid ':=' Nodeid.
```

For the canonical form of Lean replace the following rules in the syntax above:

```
RuleAlt          = Graph '->' [Graph ','] Redirections.
Graph            = NodeDefinition {',' NodeDefinition}.
Term             = Nodeid.
Redirections     = Redirection {',' Redirection}.
```

6

CLEAN — A LANGUAGE FOR FUNCTIONAL GRAPH REWRITING

T.H. Brus, M.C.J.D. van Eekelen, M.O. van Leer, M.J. Plasmeijer.

Computing Science Department, University of Nijmegen,
Toernooiveld 1, NL-6525 ED Nijmegen, The Netherlands.
E-mail: . !mcvax!hobbit!{tom,marko,maarten,rinus}

Partially supported by the Dutch Parallel Reduction Machine Project,
sponsored by the Dutch ministry of Science and Education.

Abstract.

Clean is an experimental language for specifying functional computations in terms of graph rewriting. It is based on an extension of Term Rewriting Systems (TRS) in which the terms are replaced by graphs. Such a Graph Rewriting System (GRS) consists of a, possibly cyclic, directed graph, called the data graph and graph rewrite rules which specify how this data graph may be rewritten. Clean is designed to provide a firm base for functional programming. In particular, Clean is suitable as an intermediate language between functional languages and (parallel) target machine architectures. A sequential implementation of Clean on a conventional machine is described and its performance is compared with other systems. The results show that Clean can be efficiently implemented.

6.1 INTRODUCTION

In order to be able to reason about (future) functional languages and their implementations as well as for the comparison of new machine architectures (reduction machines), it is necessary to choose a computational model. Functional languages and their implementations have very little in common with the familiar Turing machine model of computation. The λ-calculus is often seen as the computational model for these languages (Peyton Jones (1987a)). However, most implementations are not really based on λ-calculus but on combinatory logic (Turner (1979a), Johnson (1984), Cousineau *et al.* (1985)). Furthermore graphs are used for the representation of functional programs in which redundant computations are prevented via sharing of subgraphs. The presence of patterns in functional languages is very essential. Though it is possible to translate them to ordinary tests it appears to be worth-while to incorporate patterns in the computational model. Consequently, if one wants to have a computational model for functional languages which is also close to their implementations, pure λ-calculus is not the obvious choice anymore.

Another reason for reconsidering the computational model is that functional languages are still being further developed. Several researchers investigate how to incorporate concepts such as parallelism and unification (Hudak & Smith (1986), de Groot & Lindstrom (1986)). These

appreciated concepts in some declarative languages are not straightforward to incorporate in functional languages nor in the underlying computational model of the λ-calculus.

Hence, we have developed an alternative computational model by extending Term Rewriting Systems (O'Donnell (1985), Klop (1985)) to a model of general graph rewriting. Via this general model it must be possible to reason about differences between languages, to prove correctness, to port declarative programs to different (parallel) machines. Lean (the Language of East-Anglia and Nijmegen) (Barendregt *et al.* (1987b)) is a first proposal for a language based on such a model. It is the result of collaboration between two research groups: the Declarative Alvey Compiler Target Language group at the University of East-Anglia (Glauert *et al.* (1985)) and the Dutch Parallel Reduction Machine group at Nijmegen.

The language Clean presented in this paper is roughly the subset of Lean intended for functional languages only. In Clean, graph representations of terms are used to perform term rewriting more efficiently. The design of Clean, done in parallel with the Lean language, was triggered by the need for an intermediate language and corresponding computational model in the Dutch Parallel Reduction Machine Project. This project, a collaboration between the Dutch Universities of Amsterdam, Utrecht and Nijmegen, has as its goal the development of a parallel reduction machine. An overview of the results of the project is given in Barendregt *et al.* (1987c).

The basis of Clean is that a computation is represented by an initial data graph and a set of rules used to rewrite this graph to its result. The rules contain graph patterns that may match some part of the graph. If the data graph matches a rule it can be rewritten according to the specification in that rule. This specification makes it possible to first construct an additional graph structure and then link it into the data graph by redirecting arcs from the original graph. Clean describes functional graph rewriting in which only the root of the subgraph matching a pattern may be overwritten. The semantics allow parallel rewriting where candidate rewrites do not interfere. The rewriting process stops if none of the patterns in the rules match any part of the graph which means that the graph is in normal form.

In this paper we first informally introduce the language Clean giving some examples how graph rewriting is performed. The general semantics of the graph rewriting process is explained in Barendregt *et al.* (1987b). A formal description of the basis and theoretical properties of the graph rewriting model followed in this paper can be found in Barendregt *et al.* (1987a). After the introduction to the language some examples are given to show its expressive power. Hereafter an implementation of Clean on a conventional machine is discussed. Its speed will be compared to other implementations of functional languages.

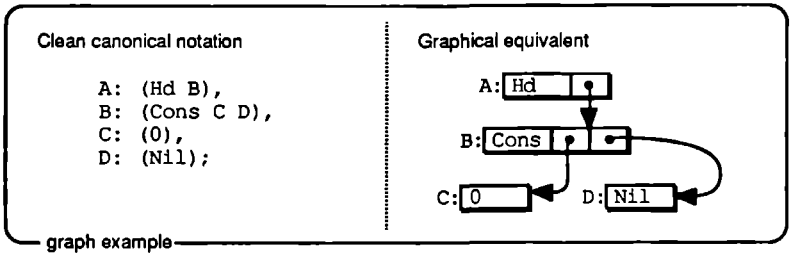
6.2 GENERAL IDEA OF THE LANGUAGE

6.2.1 CLEAN GRAPHS

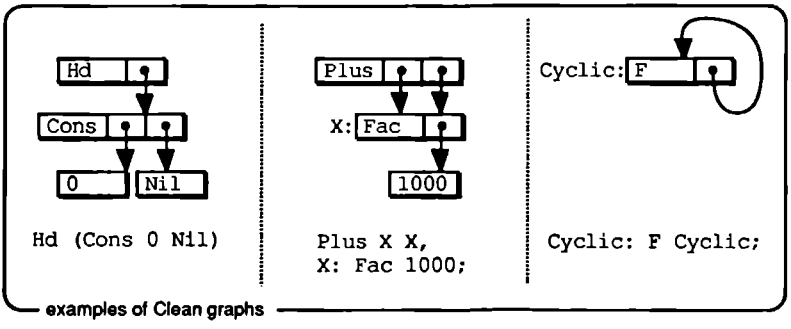
The object that is manipulated in Clean is a connected, possibly cyclic, directed graph called the *data graph*. When there is no confusion, the *data graph* is simply called *the graph*. Each node in the graph has an unique identifier associated with it (the *node identifier* or *nodeid*). Each node

contains a *symbol* and a possibly empty sequence of nodeid's (the *arguments* of the symbol) which define directed arcs to nodes in the graph. Symbols have fixed arities. The data graph is a *closed graph* i.e. contains no variables, this in contrast with the Clean graphs specified in rules.

Programming with pictures is rather inconvenient so we have chosen for a linear notation for graphs. In the most extensive form of this notation (the canonical form) graphs are represented by giving the list of the nodes out of which the graph is built.



In order to get a more readable form we may substitute the contents of a node for a nodeid mentioned in a node and furthermore we only explicitly have to notate nodeid's if we need them to express sharing. Brackets are left out if they are redundant. This way of representing graphs has the advantage that it is very comprehensive. Note that each Clean graph described in this way can be transformed to an equivalent graph notated in Clean's canonical form. The syntax of Clean is given in appendix A.



6.2.2 CLEAN PROGRAMS

Although for the understanding of the rewriting process it is important to know what a data graph looks like, the data graph itself is never specified in a Clean program. The initial data graph is a given object generated by the operating system as we will explain later. Consequently a *Clean program* only consists of a set of *rewrite rules*. Each rewrite rule specifies a possible transformation of the data graph.

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

Add Zero n          -> n                                        |
Add (Succ m) n      -> Succ (Add m n)                           ;

Fac 0               -> 1                                        |
Fac n               -> *I n (Fac (-I n 1))                      ;

F (F x)             -> x                                        ;

Start stdin         -> Add (Succ Zero) (Succ (Succ Zero))       ;
```

The left-hand-side of a rewrite rule consists of a Clean graph which is called a *redex pattern*. The right-hand-side either consists of a Clean graph called *contractum pattern* or the right-hand-side contains only a *redirection*. The patterns are said to be *open* since they contain variable nodeid's expressed by the identifiers starting with a lower-case letter. A redirection is not a graph but just consists of a single nodeid variable. The first symbol in a redex pattern is called the *function symbol*. Rules starting with the same function symbol are collected together forming a *rule-group*. The members of a rule-group are separated by a '|'. Symbols other than function symbols are called *constructors* because they are usually used to construct data structures or data types. Note that function symbols may also occur at other positions than the head of the pattern. At such occurrences function symbols are also called *constructors*. The use of the start rule and its special argument is explained in the section on input/output.

6.2.3 REWRITING THE DATA GRAPH

The initial graph of a Clean program is rewritten to a final form by a sequence of applications of individual rewrite rules. For a rule to be included in the sequence, there must be a correspondence between a redex pattern of the rule and some subgraph of the data graph.

An *instance* of a redex pattern is a subgraph of the data graph for which there exists a mapping from the pattern to that subgraph in such a way that the mapping preserves the node structure (corresponding nodes must have the same arity) and that it is the identity on constants. This mapping is also called a *match*. The subgraph which matches a redex pattern is called a *redex* (*red*ucable *ex*pression) for the rule concerned.

Assume that we have the following Clean rules:

```
Add Zero n          -> n                                        |    (1)
Add (Succ m) n      -> Succ (Add m n)                           ;    (2)
```

and assume that we have the following data graph

```
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)
```

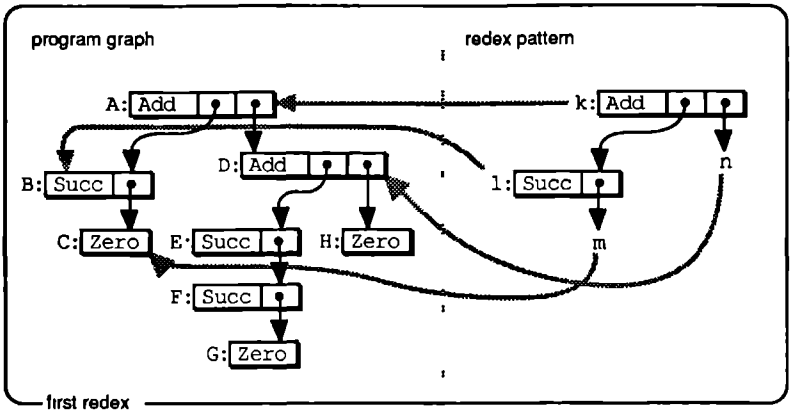
There are two redexes, both matching rule 2:

```
Add (Succ  m )  ____________n_____________
Add (Succ Zero) (Add (Succ (Succ Zero)) Zero)
```

and:

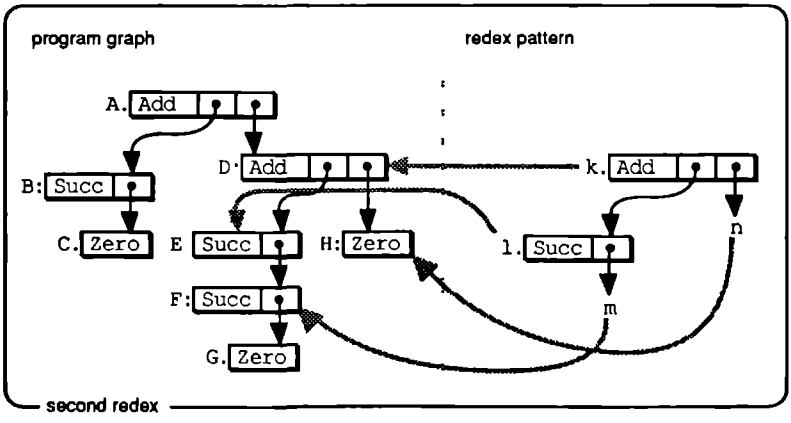
```
                    Add (Succ ____m_____) __n__
Add (Succ Zero) ( Add (Succ (Succ Zero)) Zero)
```

In graphical form the first redex can be found by performing the following mapping:



We see that the redex pattern of rule 2 matches the indicated subgraph of the data graph if we substitute the following nodeid's of the graph for the variable nodeid's in the redex pattern `k` := `A`, `l` := `B`, `m` := `C` and `n` = `D` Note that in order to perform this mapping we have to use the canonical form of the graphs This means for nodeid's not explicitly mentioned in the patterns new unique variable nodeid's (in the example `k` and `l`) have to be invented

The redex pattern of rule 2 can also be mapped on another part of the data graph if we substitute `k` = `D`, `l` := `E`, `m` = `F` and `n` = `H`, as shown in the next picture.



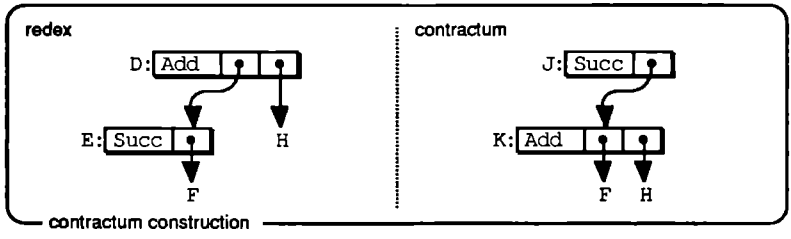
If a particular rule is applied to a matching redex, the graph is rewritten according to the right-hand-side of that rule If this right-hand-side consists of a *contractum* pattern, the first step is to create an instantiation of this pattern which is called the contractum The contractum is a new

Clean graph as specified in the right-hand-side in which the nodeid variables defined on the left-hand-side are replaced by the corresponding matching nodeid's in the redex. New nodeid constants are created for those nodeid variables in the right-hand-side which are not defined in the left-hand-side.

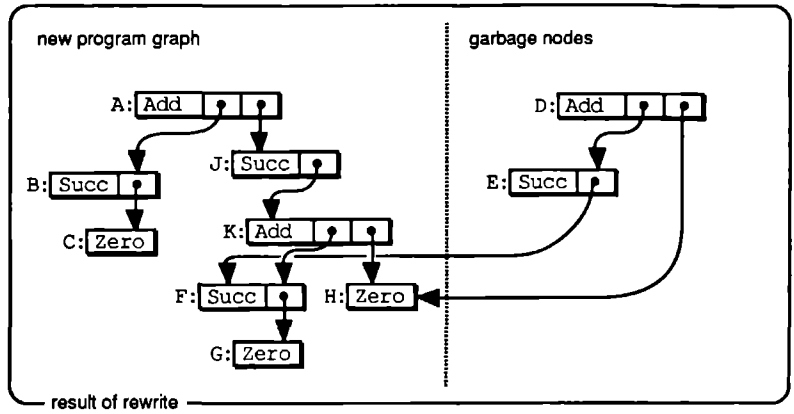
The new data graph is finally constructed by taking all arcs pointing to the root node of the redex and redirecting them to the root node of the contractum. This has the effect of "overwriting" the root of the redex with the root of the contractum. If the right-hand-side is a redirection no contractum has to be built. All arcs pointing to the root node of the redex are now redirected to the single nodeid that matches that nodeid variable. This "overwrites" the root of the redex with the root of a subgraph of the data graph. This concept of redirecting has the advantage over the usual "overwriting of node's" semantics that we do not have to deal with indirection nodes on the semantic level.

After the rewriting, nodes which are no longer reachable from the root of the data graph are considered to be garbage and may be collected by a garbage collector.

We see that in the example above the second redex matches the data graph if we take the following mapping from the nodeid variables to the nodeid of the data graph: `k` := `D`, `l` := `E`, `m` := `F` and `n` := `H`. The right-hand-side of the second rule specifies that in this case the contractum can be constructed as follows:



For the variables `m` and `n` in the right-hand-side we have to take the same mapping (`m` := `F`, `n` := `H`). For the other variables (say `o` and `p`, they are not specified explicitly) we invent new unique nodeid's (say `J` and `K`). Now the contractum is glued to the data graph by redirecting all nodes pointing to the root of the redex (`D`) to the root of the contractum (`J`). All nodes not reachable from the root of the data graph are considered to be garbage. If we remove these nodes (`D` and `E`) we finally have the new data graph and can start another rewriting.



The graph after rewriting is called the *result*. The process of performing a rewriting is often called a *reduction* step. A data graph containing no redexes is said to be in *normal form*. The rewriting process will start with the start rule and rewriting is performed repeatedly until the strategy has transformed the data graph to normal form.

6.2.4 REDUCTION STRATEGIES

In general there will be several possible redexes in the graph. It may even be the case that one and the same redex can be reduced according to more than one rule; a typical situation which is called an ambiguity in the literature. An algorithm which repeatedly rewrites the graph making choices out of the available redexes and out of all the possible matches of those redexes is called a *rewriting strategy* or a *reduction strategy*. Note that this definition of strategy is somewhat more liberal than some definitions circulating in the literature. It allows the strategy to choose out of several possible matches of one and the same redex. Furthermore, it is also not necessary for a strategy to rewrite the graph until a normal form is reached, which e.g. allows strategies that reduce to head normal form only.

Given a set of rules (including a start rule), an initial graph and a rewriting strategy we have a system with a dynamic behavior, a *rewriter*. Although it is sometimes only implicitly defined, every implementation of a rewriting system must rewrite according to a given strategy. If the strategy is deterministic, every program (including a so-called ambiguous one) will always have exactly the same result.

Every Clean program is reduced with one and the same strategy. This strategy is called the *functional* strategy, because it resembles very much the way in which normally reducing is performed in lazy functional languages. Below we will give an operational definition of the functional strategy. A formal description can be found in Goos & van Latum (1987) using a formal method described in van Eekelen & Plasmeijer (1986).

The functional strategy proceeds as follows: the strategy considers one or more candidates for rewriting. When a match is found rewriting is performed as described in the previous section.

The functional strategy starts with reducing the root node of the graph to head normal form (RtoHNF). The result will be a graph with the property that its root is not part of any redex. Thereafter this reduction to head normal form is recursively called on the arguments of the obtained result (from left to right).

The RtoHNF starts with the examination of the graph it is applied to: if the symbol in the root node of that graph is a constructor the reduction is finished. If the symbol is a function symbol the corresponding rewrite rules for that function are examined in order to see if the given graph is a redex or can become a redex. In *textual order* the corresponding rules are examined to see if one of the redex patterns matches or can be made to match. The graph is rewritten according to the first rule that matches and hereafter the RtoHNF is recursively applied to the subgraph with the redirected nodeid as root. If no rule can be made to match the reduction is finished.

In order to examine the matching of redex pattern and graph the redex pattern is traversed in preorder and, possibly after forcing evaluation of corresponding parts in the graph, redex pattern and graph are compared. If there is a variable in the pattern, the traversal is continued. If a function symbol is encountered in the graph where there is a symbol in the pattern, the RtoHNF recursively calls itself to force evaluation of this function. This aspect of the functional strategy is remarkable because evaluation is forced during a matching attempt. The resulting graph will be in head normal form. Hereafter a symbol encountered in the pattern must be the the same symbol as in the graph. If they are different a match is impossible and the next rule is tried. If they are the same the traversal is continued. If the traversal reaches the end of the pattern a match is found. The result of this lazy evaluation scheme is that after the traversal we might end up with a redex after all and the rule can be applied.

In the following example a data graph is constructed in which parts are shared. Note that when the data graph is actually a tree there is no difference with a term rewriting system.

```
Start stdin        -> Double (Add (Succ Zero) Zero)        ;    (A)
Double a           -> Add a a                              ;    (B)
Add Zero n         -> n                                    |    (1)
Add (Succ m) n     -> Succ (Add m n)                       ;    (2)
```

Rewriting a shared part will reduce the number of rewriting steps compared to an equivalent term rewriting system. The rewriting will take place as specified below. Note that when a nodeid variable appears more than once at a right-hand-side, the rewriting process will generate a contractum in which the corresponding matching node is shared.

```
Start Nil                                                  →    (A)
Double (Add (Succ Zero) Zero)                              →    (B)
Add X X, X:Add (Succ Zero) Zero                            →    (2)
Add X X, X:Succ (Add Zero Zero)                            →    (2)
Succ (Add M X), X:Succ M, M:Add Zero Zero                  →    (1)
Succ (Add Z X), X:Succ Z, Z:Zero                           →    (1)
Succ (Succ Zero)
```

Although this functional strategy will look very familiar for people acquainted with functional languages, it really is a very peculiar strategy in the TRS and GRS world. To have a priority in the rewrite rules leads in general to a rewrite system without proper semantics (Klop (1985)). In this case the system is sound due to the forced evaluation of the arguments of a function as

described above. Although we theoretically prefer a TRS without such a priority in rules, we have adopted the functional strategy because it is used so often in practice.

6.2.5 DATA TYPES

Constructors are not only handy to create datastructures in the form of directed, possibly cyclic graphs, such as list and tuples, but they can also be used to represent any other object or to indicate the type of an object. For instance, one can define numbers as:

```
0   -> Num Zero                                    ;
1   -> Num (Succ Zero)                             ;
2   -> Num (Succ (Succ Zero)                       ;
...
```

Here the constructor Num (also called a *type constructor*) indicates the type of the number objects while the constructors Succ and Zero (also called *data constructors*) are used to represent numerical values. A function for doing addition that yields a result of type Num could look like:

```
Add     (Num x)     (Num y)   -> Num (Add2 x y)      ;
Add2    Zero        y         -> y                   |
Add2    (Succ x)    y         -> Succ (Add2 x y)     ;
```

In Clean one is not obliged to specify the arguments of a constructor in a redex pattern if they are not used elsewhere in the rule. This is in particular a handy notation when one wants to write rules for objects of a certain type. For example instead of:

```
Fac 0               -> 1                           |
Fac n:(Num x)       -> Times n (Fac (Minus n 1))   ;
```

one may write:

```
Fac 0               -> 1                           |
Fac n:Num           -> Times n (Fac (Minus n 1))   ;
```

The value can be passed to a function by passing the corresponding nodeid (n in the example). Note that in this example the type of the argument is checked at run-time in the matching phase. Of course this check can be prevented by not using the Num constructor in the pattern or the objects.

6.2.6 BASIC TYPES AND PREDEFINED DELTA RULES

For practical reasons it is convenient that rules for performing arithmetic on primitive types (numbers, characters etc.) are predefined such that they can be implemented efficiently, preferably by using the integer and real representation and corresponding arithmetic available on the computer.

In Clean for primitive types a number of constructors such as INT, REAL and CHAR are predefined with hidden arity. Objects of these primitive types can be denotated: for instance 5 (an integer), 5.0 (a real), '5' (a character). The standard basic functions for arithmetic defined on these basic types are also predefined. These predefined rules are called *delta rules* .

The possibility in Clean to leave out the specification of the arguments of a constructor in a redex pattern is mandatory for primitive type constructors. As a consequence how an object of a certain primitive type is represented will be hidden for the Clean programmer. Besides this special restriction, added only for software engineering reasons, primitive type constructors act as ordinary constructors.

6.2.7 INPUT AND OUTPUT

Input and output is always somewhat problematic in functional languages. We have chosen for a solution in which the operating system builds the initial graph. The initial graph contains the standard input as shown below.

```
Root:   Start Stdin,
Stdin:  Cons "line1\n" (Cons "line2\n" (Cons .....));
```

The input can be accessed in the Clean program via the argument of the `Start` rule. The output generated by a Clean program is in principle a depth-first representation of the normal form to which the initial data graph is reduced. As soon as the initial graph is in head normal form the head symbol is printed and hereafter the printing process is recursively applied to the arguments of that symbol. In the near future it will be possible to associate printing actions with predefined constructors like in Miranda (Turner (1985)).

6.2.8 ANNOTATIONS

In Clean to every node an attribute can be assigned via an annotation. Annotations have in general the form of a list of strings between curly braces. Annotations are to be considered as compiler and run-time directives (pragmats). The number and type of annotations are left open and will depend on the actual implementation. Although annotations may influence the efficiency and strategy of the rewriting process, they are of course not allowed to influence the outcome of a computation. It is all right for a Clean compiler to ignore annotations.

At this moment in our compiler only one annotation is implemented indicating that the annotated argument is needed for the computation ("`!`" or "`{strict}`"). Future annotations are planned for work to be done in parallel, for load distribution, etc.

6.3 EXAMPLES OF CLEAN PROGRAMS

6.3.1 MERGING LISTS

The following Clean rules are capable of merging two ordered lists of integers (without duplicate elements) into a single ordered list (again without duplicate elements)* :

```
Merge Nil          Nil          -> Nil                          |
Merge f:Cons       Nil          -> f                            |
Merge Nil          s:Cons       -> s                            |
Merge f:(Cons a b) s:(Cons c d) -> IF (<I a c)
                                      (Cons a (Merge b s))
```

* `<I` and `=I` are delta rules for integer comparison, `IF` is a delta rule for the conditional.

```
                                        (IF (=I a c)
                                            (Merge f d)
                                            (Cons c (Merge f d)))   ;
```

Note that in the last rule the arguments as a whole as well as their decomposition is used.

6.3.2 HIGHER ORDER FUNCTIONS, CURRYING

In this example we show how higher-order functions are treated in Clean, by giving the familiar definition of the function map.

```
Map f        Nil         -> Nil                                  |    (1)
Map f        (Cons a b)  -> Cons (Ap f a) (Map f b)              ;    (2)
Ap  (*IC a) b            -> *I a b                               ;    (3)
Start stdin              -> Map (*IC 2) (Cons 3 (Cons 4 Nil))    ;    (4)
```

This will be rewritten in the following way:

```
Start Nil Nil Nil                                                →    (4)
Map (*IC 2) (Cons 3 (Cons 4 Nil))                                →    (2)
Cons (Ap L 3) (Map L (Cons 4 Nil)), L:(*IC 2)                    →    (3)
Cons (*I 2 3) (Map L (Cons 4 Nil)), L:(*IC 2)                    →    *I
Cons 6 (Map L (Cons 4 Nil)), L:(*IC 2)                           →    (2)
Cons 6 (Cons (Ap L 4) (Map L Nil)), L:(*IC 2)                    →    (3)
Cons 6 (Cons (*I 2 4) (Map L Nil)), L:(*IC 2)                    →    *I
Cons 6 (Cons 8 (Map L Nil)), L:(*IC 2)                           →    (1)
Cons 6 (Cons 8 Nil)
```

`*I` is a predefined delta rule which multiplies two integers. Rule 3 of this example will rewrite `(Ap (*IC 2) 3)` using the constructor `*IC` which is the curried version of `*I`, to its uncurried form `(*I 2 3)` making the multiplication possible. One will need such an "uncurry" rule for every function which is used on a curried manner. Note that during rewriting the node `L:(*IC 2)` is shared. In this case sharing only saves space, but not computation.

6.3.3 GRAPHS WITH CYCLES

The following example is is a solution for the Hamming problem: it computes an ordered list of all numbers of the form 2^n3^m, with n,m ≥ 0. We use the map and merge functions of the previous examples.

```
Ham -> Cons 1 (Merge (Map (*IC 2) Ham) (Map (*IC 3) Ham))        ;
```

A more efficient solution to this problem can be obtained by creating a cycle in the contractum. With these cycles we make heavy use of computations already performed. The new definition is:

```
Ham -> x: Cons 1 (Merge (Map (*IC 2) x) (Map (*IC 3) x))         ;
```

6.3.4 COMBINATORY LOGIC

Finally we show the Clean equivalent of a well-known TRS.

```
Ap (Ap (Ap S a) b) c   -> Ap (Ap a c) (Ap b c)                   |
Ap (Ap K a) b          -> a                                      ;
```

6.4 THE IMPLEMENTATION OF CLEAN

This section will describe the current implementation of Clean. This implementation was developed as a testbed for the definition of Clean. It was partly constructed concurrently with the language itself. The advantage was that the definition of Clean could often be corrected or adjusted when an inconsistency was overlooked and became apparent in the implementation.

The Clean compiler was developed on a VAX/750 running UNIX BSD 4.2. UNIX and VAX specific aspects will now and then surface in the implementation and in the following sections. We have tried to minimize this.

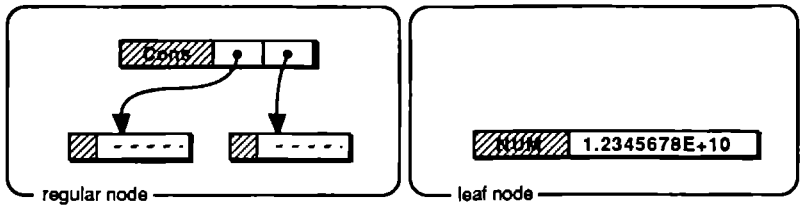
6.4.1 CLEAN RUN-TIME PHILOSOPHY

Clean is a graph rewriting language, therefore in principle we need a heap to build graphs in. The initial graph is built by the run-time system. Under control of the reduction strategy this graph will be transformed to normal form. These transformations are performed by the compiled code, using a heap and 2 stacks (a system stack and an argument stack). The functional strategy is compiled into this code. This means that for the implementation of a new strategy it is necessary to change the compiler or at least its code generator.

The basic implementation algorithm looks for a matching redex according to the functional strategy. It will overwrite the matching redex with the corresponding right-hand-side, thereby realizing redirection. This continues until there is no redex left. The main work that is being done this way is building graphs. Hence the code will not be fast, because the system is continuously allocating nodes in the heap. As a stack mechanism is inherently faster then a heap mechanism, at least in Von Neumann like machine architectures, we have tried to put the graph on a stack instead of in a heap whenever possible. The main issue in this respect is the LIFO access characteristic of a stack opposed to the random access in a heap. We had to find LIFO behaving mechanisms in our language, or its implementation. Lazy evaluation does not behave LIFO, eager evaluation does. This is the reason we need a strictness analyzer, which could free us from a lot of laziness, and give us eagerness instead. In 4.5 we discuss how we used strictness.

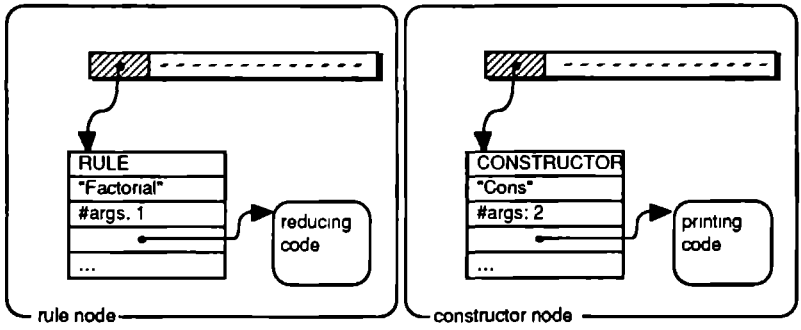
6.4.2 GRAPHS

In a Clean graph we can distinguish regular nodes and leaf nodes. Every node has a symbol field which indicates the kind of symbol stored in the node. It is implemented as a pointer to a record containing all the necessary symbol information. If the symbol field labels a node as a regular node it can only be filled with references to nodes. If the symbol field indicates a leaf node then the rest of the node has no node reference at all. The other bits of the node will contain information like a number or a character code.



This strict division is made to enable the garbage collector to easily and quickly follow all the necessary links in the heap.

Regular nodes are either rule or constructor instantiations. Rules have code associated with them which needs to be called for reduction to head normal form. Constructors have code, that will be called when the constructor needs to be printed. This includes code to evaluate arguments.



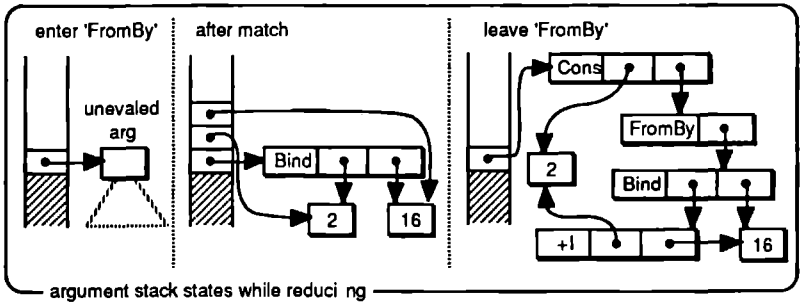
Graphs are built from right to left, from bottom to top using the argument stack. This works fine if we have no sharing and cycles in the right-hand-side. If a certain subtree is shared, we will save a reference to this subtree as soon as it is built. If the subtree is needed again, the saved reference can be taken. Cycles can be solved by inserting a place holder on the argument stack whenever we find a link back to a former node. We save a reference to the node with the place holder in it. As soon as the node to which the link back referred has been built, the place holder is replaced by the actual reference.

6.4.3 REDUCING GRAPHS

Reducing first involves finding a redex, using the functional strategy, by matching the formal and actual arguments of a rule. Every formal argument is a graph of node patterns. A node pattern can either be a variable or a pattern. In case of a variable the reference to the actual argument is copied to the argument stack. In case of a pattern, the actual argument is first reduced to head normal form. If the result matches the pattern a pointer to the actual argument is copied to the argument stack.

After a match has been found the rule must be rewritten. Due to the match the argument stack contains references to all the left-hand-side variables. The rewrite code of the rule will use these during its rewrite. Having rewritten the rule all the references are popped of the stack and the result is pushed on top of it. The following picture will illustrate this* :

```
FromBy (Bind f b) -> Cons f (FromBy (Bind (+I f b) b));
```



The above scheme works fine for eager evaluation. We actually have the top node available on the stack at all times. Using lazy evaluation we sometimes have to rewrite a node which has already been built, therefore we have to adapt this scheme. First the contents of the node in the graph is copied to the stack, then it is rewritten. This will return a new node, in head normal form, on the stack. But the real top node is still untouched in the heap. The redirection is implemented by making the old node an indirection node pointing to the new node. Overwriting the old node is in general impossible because the new node could be bigger than the old one.

6.4.4 HEAP MANAGEMENT

The heap delivers variable sized nodes. Once created, a nodes size can not be changed. Heap management routines take care of garbage collection in the heap. The garbage collector is based on a simple mark/scan algorithm.

The memory management used is an ad hoc solution, which happens to perform satisfactory. It could be streamlined significantly, or even be replaced altogether, to get a better performance. A fast memory management is essential.

Here it becomes clear why we can not merge the argument stack with the system stack, why we need a separate stack with node references. Our compilation scheme does not guarantee that all non-garbage nodes can be found from the root of the data graph. Therefore the garbage collector will have to look in the stack for references to find all non-garbage nodes. Because it is impossible for the garbage collector to identify items on a stack as node references or other values, such as reals or integers, we save references to nodes on a special stack.

---

* `+I` is the delta rule for integer addition.

6.4.5 OPTIMISATIONS USING STRICTNESS

As we have seen, we want to make use of the stack, and ban the use of the heap, as much as possible. Lazy evaluation prohibits this, eager evaluation enables this. This led us to methods of trading laziness for eagerness where ever possible, without endangering the termination of the reduction process.

The functional strategy enables us to compile the right-hand-side of rules in an efficient way. To illustrate this we will first introduce two types of contexts which can be identified in the right-hand-side. Then we will see how to use them.

- *Immediate context*: upon entering the rule, nodes in an immediate context may be evaluated to head normal form immediately.
- *Postponed context*: upon entering the rule, nodes in a postponed context may not be evaluated, and must be built as graphs, which can be passed as arguments to other rules, or given as a result.

We will call nodes immediate or postponed according to their context. The top node of a right-hand-side is an immediate node. All subnodes of a postponed node are postponed. The symbol of an immediate node determines the context of its argument nodes. For a node with a rule symbol all strict arguments are immediate, all other arguments are postponed. For nodes with a constructor symbol all arguments are postponed. Strictness for user-rules is given by annotations, for delta-rules it is known by the compiler.

Consider the following rules, in which rule 's' has one strict argument and rule 'NS' has one non-strict argument (the boxes are postponed contexts):

```
F1 x -> Cons [(S [...])] [Nil] ;

F2 x -> +I (*I x 10 ) 20 ;

F3 x -> +I (S x ) (NS [...]) ;

F4 x -> IF (=I (S x ) (NS [...])) (S x ) (NS [...]) ;
```

postponed contexts

In principle we have to build the right-hand-side graphs, as they are. However, if we discover an immediate node, while building the right-hand-side, we will not allocate it in the heap, but try to reduce it first and use the result. For user-rules this means calling the reduction code, for delta-rules the appropriate instructions are executed. If the top node of a right-hand-side contains a function symbol, the user rule will always be called (the top node is always immediate!). In the code we change this to a jump to the rule. This way we automatically remove tail recursion. For example:

```
F x -> F (...argument...);
```
tail recursion

F will actually be a loop in the generated code.

Things are less straightforward when we introduce sharing and cycles. We will not discuss the solutions here. We were able to devise a compilation scheme to cover all possible combinations of sharing and cycles in right hand sides, with the above principles.

6.4.6 SMALL STRICTNESS ANALYSIS

Although we consider strictness annotations to be generated by the compiler generating Clean, we incorporated a very simple strictness analyzer in our compiler. This analysis is based on certain aspects of the functional strategy.

Consider a rule with a pattern at the left-hand-side. Upon entering the rule we will always evaluate the actual argument for the first pattern. At compile time it is undecidable whether we have to match any of the other patterns in this rule, because the first match may fail. Therefore it is only the first argument with a pattern in a rule that can be marked as strict.

For example (the strict arguments are surrounded by boxes):

```
F1 x [y:(Cons a b)] z -> .....;

F2 [Nil]              -> .....|
F2 [x]                -> .....;

F3  x [10]            -> .....|
F3 10 [y]             -> .....;

F4 [Nil] Nil Nil      -> .....;
```
strict arguments

6.4.7 EFFICIENCY OF THE GENERATED CODE

The compiler we constructed is slow, due to the fact that flexibility of the compiler was more important than compilation speed. The speed of the generated code, on the other hand, was of primary importance. The optimisations we devised are very suitable for VAX-like machines (PDP, MC68000). For other machines they may not always be the best. To get an impression of the speed of the code generated by the current implementation one can look in appendix B where some benchmark results are shown. Although these results show that lml is an order of magnitude faster then Clean, we may conclude that we are on the right track. Specially when we bear in mind that not yet all of the possible optimisations are included in the current Clean

implementation. For example, leaf nodes are always built in the heap while the values could often be maintained on a stack.

6.5 CONCLUSIONS AND FUTURE RESEARCH

Clean is an experimental language with many facets. First of all it is a language for specifying computations in terms of graph rewriting. As such it is a convenient and elegant language.

Clean also has a very interesting underlying model of computation: a Graph Rewriting System which can be seen as an extension of a Term Rewriting System (Klop (1985)). This has the advantage that a lot of theoretical properties from the TRS world are inherited and provide a sound foundation for a GRS theory. For instance, in Barendregt *et al.* (1987a) it is proven that all hyper-normalizing strategies in the TRS world, a class to which all well-known normalizing strategies belong, are also normalizing in the GRS world.

Clean can be used as intermediate language between functional languages and (parallel) machine architectures. In (Koopman & Nöcker (1988)) it is shown that functional languages like SASL (Turner (1979a)), Miranda (Turner (1985)), OBJ2 (Futatsugi (1985)) and Tale (Barendregt & van Leeuwen (1986)) can easily be compiled to Clean code. Compilers (one written in Modula2, one written in Miranda) are being implemented targeted to Clean. With the current Clean implementation they run 30 to 50 times faster than the current Miranda system. The Clean implementation described in this paper runs reasonably fast considering the fact that we did not want to spend much time on trivial, but time-consuming, ad hoc optimisations (see appendix B).

Our plans are to improve Clean in the near future. We will do this in the more general Lean framework (Barendregt *et al.* (1987b)) in which Clean will be one of several possible subsets with certain desired properties (in this case geared to functional languages and suited for parallel architectures). Our intentions are to include separate compilation, modularization, general type system, unification, general IO etc. All this must be accomplished without loosing the basic elegance, the practical usability and the theoretical framework of the model. This will take some time.

Because strategies have a critical influence over efficiency future versions of Clean aim to give the programmer explicit control over rewrite order, for instance via high level specification of (parallel) reduction strategies and a formalism for mixing several strategy schemes during evaluation (van Eekelen & Plasmeijer (1986)).

We will improve the efficiency of the compiler and the code generated by the compiler. Implementations of Clean are planned for Motorola based architectures and parallel architectures like the Experimental Parallel Reduction Machine (Hartel & Vree (1986)) and the Distributed Object Oriented Machine (Odijk (1985)) being developed in the Philips Laboratories, the Netherlands. Requests for the current implementation can be sent to one of the authors or E-mailed to: *...!mcvax!hobbit!cleanrequest.*

6.6 ACKNOWLEDGEMENTS

We are grateful to Henk Barendregt and Pieter Koopman of the University of Nijmegen for several suggestions and inspiring discussions. We also thank Ronan Sleep, John Glauert and Richard Kennaway of the University of East-Anglia very much for the fruitful collaboration on the Lean work, which heavily influenced Clean.

6.A APPENDIX A: CLEAN SYNTAX

Clean syntax:

```
CleanProgram     = { RuleGroup }
RuleGroup        = [ `STRATEGY' StrategyName `;'] Rule { `|' Rule } `;'
Rule             = Graph `->' Graph
                 | Graph `->' Redirection
Graph            = [ Annotation ] [ Nodeid `:' ] Node { `,' NodeDefinition}
Redirection      = [ Annotation ] Nodeid
NodeDefinition   = [ Annotation ] Nodeid `:' Node
Node             = Symbol { [Annotation] Term }
Annotation       = `{' AnnotationName { `,' AnnotationName } `}'
                 | ShorthandAnnotation
Term             = Nodeid
                 | [ Nodeid `:' ] Symbol
                 | [ Nodeid `:' ] `(' Node `)'
```

Clean name conventions:

```
Symbol               = FunctionSymbol
                     | ConstructorSymbol
                     | DeltaRuleSymbol
                     | TypeConstructor
                     | TypeDenotation
Nodeid               = (* Character sequence starting with a lower-case character  *)
FunctionSymbol       = (* Character sequence starting with a upper-case character  *)
ConstructorSymbol    = (* Character sequence starting with a upper-case character  *)
DeltaRuleSymbol      = (* A predefined delta rule name                             *)
AnnotationName       = (* Implementation dependent                                 *)
ShorthandAnnotation  = (* Implementation dependent                                 *)
StrategyName         = `Functional'
TypeConstructor      = `INT' | `REAL' | `CHAR' | `STRING' | `BOOL'
TypeDenotation       = 5,4.6e-3,'a',"a string\007",TRUE (* Examples *)
```

Some context sensitive restraints:

- Graphs are connected.
- Sharing of labels is not allowed in left hand sides of rules.
- Symbols have a fixed arity.
- Every function is defined once.
- Every label is defined once in a rule.
- Delta rules can not be re-defined.

6.B APPENDIX B: PERFORMANCE MEASUREMENTS

The results of two benchmarks are reproduced here to give an idea about the speed of the compiled code. Benchmark 1 involves the reversion of a list, benchmark 2 is the all time favorite nfib number. The reversion benchmark reverses a list of n elements n times, this means doing n^2 reversion steps. In our tests n ranged from 1 to 1000. The nfib benchmark gives the number of function calls it did as output. We will only give the Clean programs here, it is straightforward to translate them to other languages* .

```
Reverse n              -> Walk (Rev_n n (FromTo 1 n));

Walk (Cons x Nil)      -> x      |
Walk (Cons x r)        -> Walk r;

Rev_n 1 list           -> Rev list Nil                  |
Rev_n n list           -> Rev_n (--I n) (Rev list Nil);

Rev (Cons x r) list -> Rev r (Cons x list) |
Rev Nil list           -> list                ;
```

benchmark 1, Clean version.

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

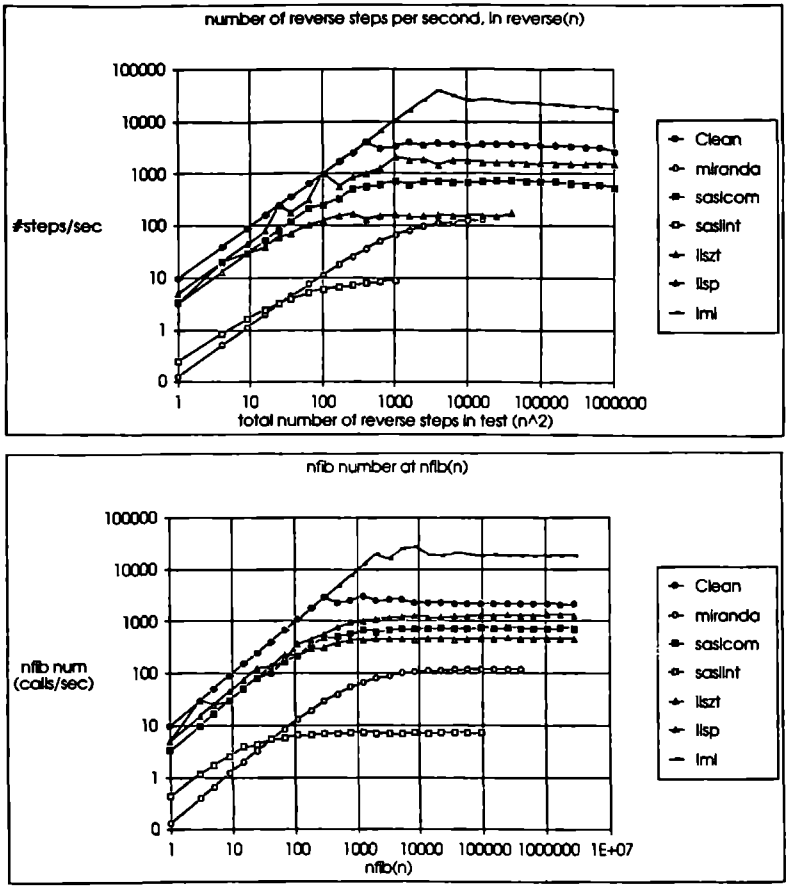
benchmark 2, Clean version.

The following programming systems were tested:

Clean	Clean Compiler, version 4.0, University of Nijmegen, Netherlands. Authors: Tom Brus, Maarten van Leer.
lisp	Franz Lisp interpreter, Opus 38.79, Unix 4.2 BSD distribution. Author: Keith Skowler.
liszt	lisp compiler, VAX version 8.36 [.79], Unix 4.2 BSD distribution. Author: John Foderaro.
lml	lml compiler, preliminary version, Chalmers, Sweden. Author: Lennart Augustsson, Thomas Johnsson.
miranda	miranda interpreter, version 0.292, Research Software Ltd., England. Author: David Turner.
saslcom	sasl compiler, version 1.1, University of Nijmegen, Netherlands. Author: Riet Oolman.
saslint	sasl interpreter, version 1.1, University of Nijmegen, Netherlands. Author: Riet Oolman.

All tests were done on a VAX11/750 under UNIX BSD 4.2, partly during working hours. All times mentioned are user times returned by the time(1) command. We measured the number of reverse steps per second (for reverse), and the number of function calls per second (for nfib):

* `++I` and `--I` are delta rules for integer increment and decrement, `-I` is for integer subtraction.



We see that these numbers stabilize to what we call the reverse number and the nfib number of the implementation. Below, these numbers are tabled separately.

language	rev number	nfib number
saslint	8	7
mira	123	120
lisp	151	467
sasicom	677	728
liszt	1669	1258
clean	3521	2322
lml	23436	19635

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!

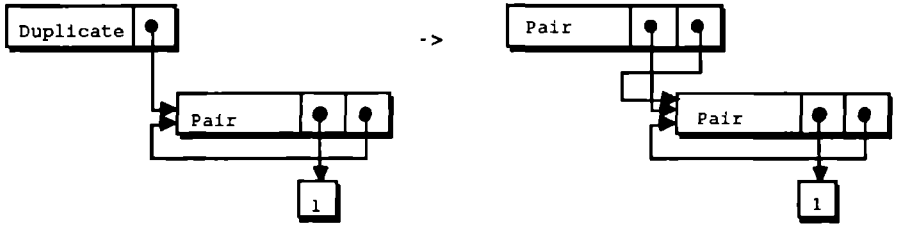
An ordinary rewrite rule:

```
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:

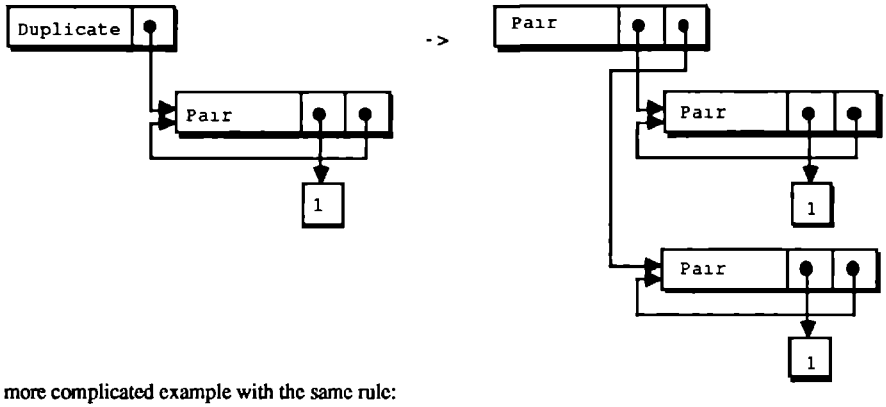
```
Duplicate x -> Pair x x_c                                    ;
```

the left graph such

reduces to the right graph. The new nodeid's are chosen in 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:



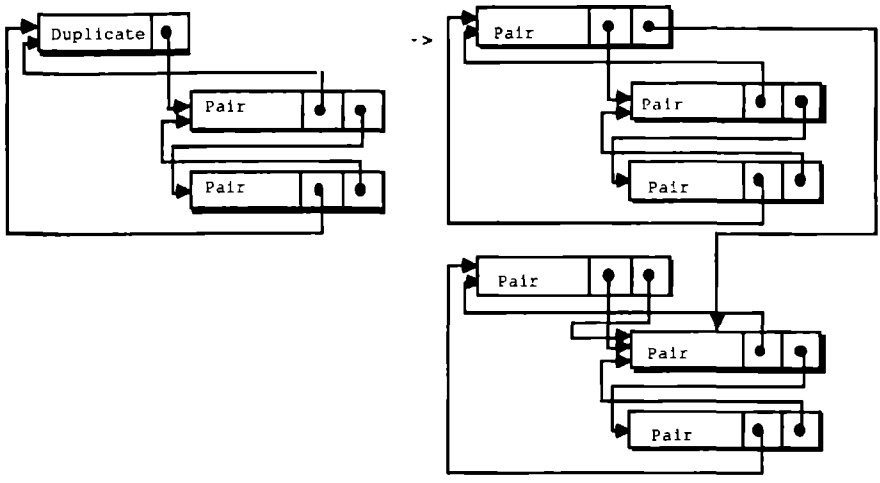
A more complicated example with the same rule:

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:

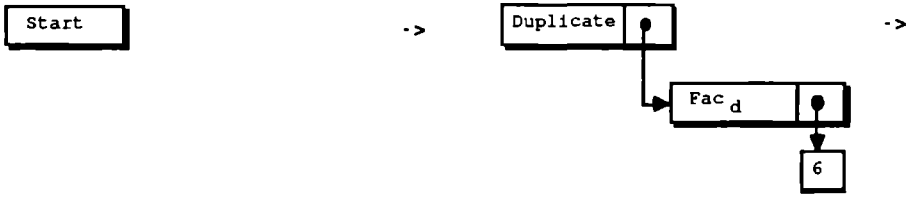
```
Start          -> Duplicate (Fac_d 6)                    ;

Duplicate x    -> Pair x x_c                             ;

Fac 0          -> 1                                      |
Fac n          -> *I n (Fac (--I n))                     ;
```

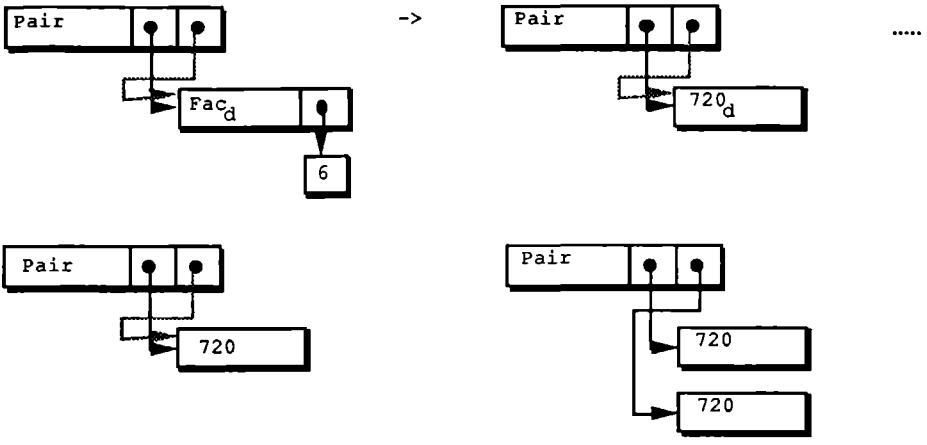
the following rewrites occur:

```
@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 @5_c1 @5_c2,              @4:  Triple @5 @15 @5_c2,
@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:

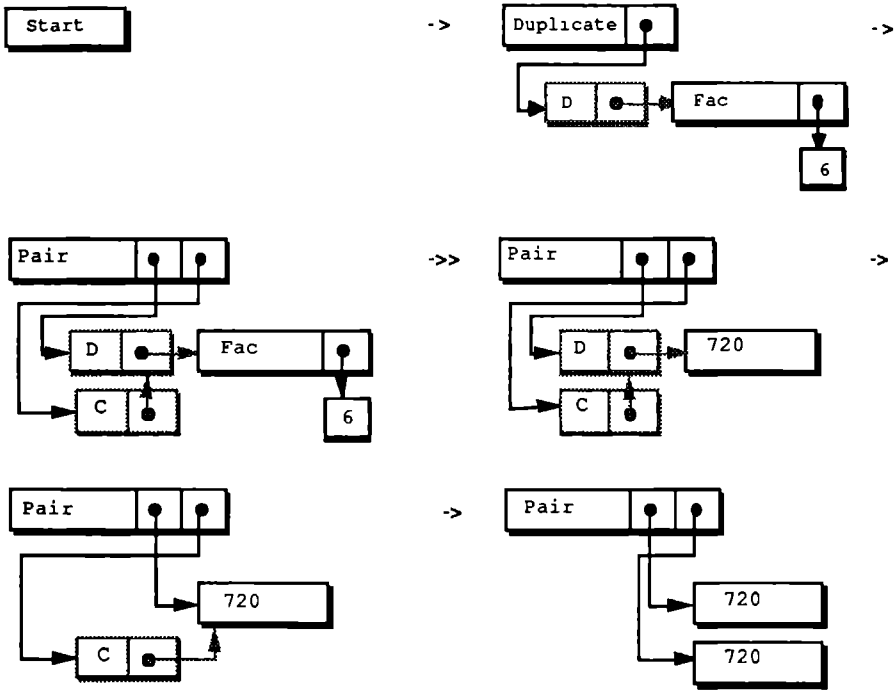
```
D (!) x      -> x                    ;
C (!) x      -> x_c                  ;
```

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;`	ll a cycle.
`Start -> x: Pair`$_d$ `1 x ;`	`@1: Pair 1 @1;`	ll a cycle.
`Start -> x: Pair 1 x`$_c$`;`	`@1: Pair 1 @11,`	ll a cycle with a copy of it:
	`@11:Pair 1 @11;`	ll a once unravelled cycle.
`Start -> x: Pair`$_d$ `1 x`$_c$`;`	`@1: Pair 1 @11,` `@11:Pair 1 @11;`	ll again: ll a once unravelled cycle.
`Start -> x: Triple`$_d$ `1 x`$_c$ `x`$_c$`;`	`@1: Triple 1 @11 @1`$_{c1}$`,` `@11:Triple 1 @11 @1`$_{c2}$`;`	ll every copy that is done, ll leads to more ll unravelling: yielding ll 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 situtation, 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.

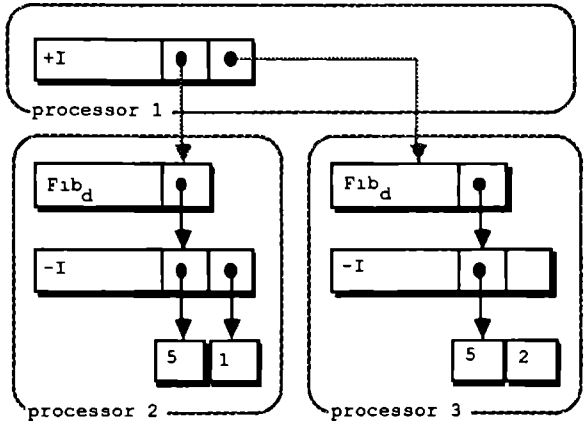
A self-contained subgraph will be regarded as a *virtual processor* because it has the property that it may be reduced on another processor. A reducer is also called a *process.*

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:
$n : \{e\}\ Sym\ a_1\ ..\ a_n$	$n : I\ x_c,$ $x : \{!!\}\ I_d\ y_c,$ $y : Sym\ a_1\ ..\ a_n$

A reducer is created by the {!!} annotation, it will reduce a node which contains the identity function of a lazy copy of the annotated node $Sym\ a_1\ ..\ a_n$. 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.

$n : \{i\}\ Sym\ a_1\ ..\ a_n$	$n : \{!!\}\ I_d\ x,$ $x : Sym\ a_1\ ..\ a_n$

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 {1}, 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).

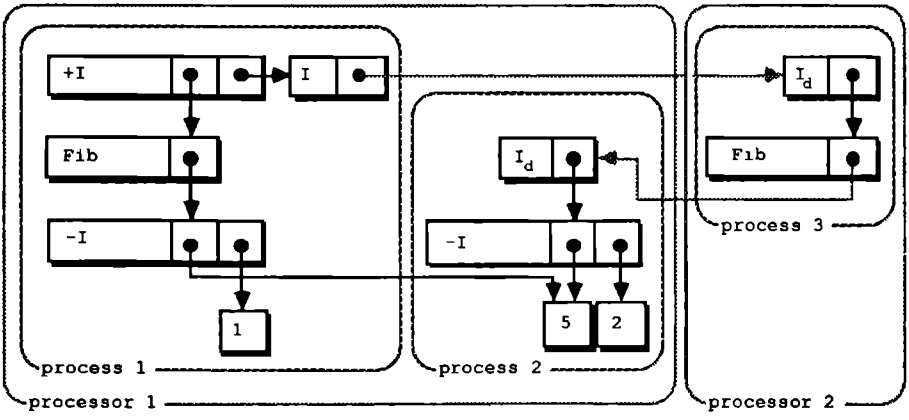
The `Fib` example using a non-hierarchical process structure (which is very unconventional for `Fib`): the second call of `Fib` will be executed on another virtual processor but the argument of that call is reduced internally on the virtual processor that also does the first call of `Fib`

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

which is equivalent to:

```
Fib 0    ->   1                                          |
Fib 1    ->   1                                          |
Fib n    ->   +I (Fib (-I n 1)) m,
              m : I xc,
              x : {''} Id yc,
              y : Fib o,
              o : {''} Id 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.

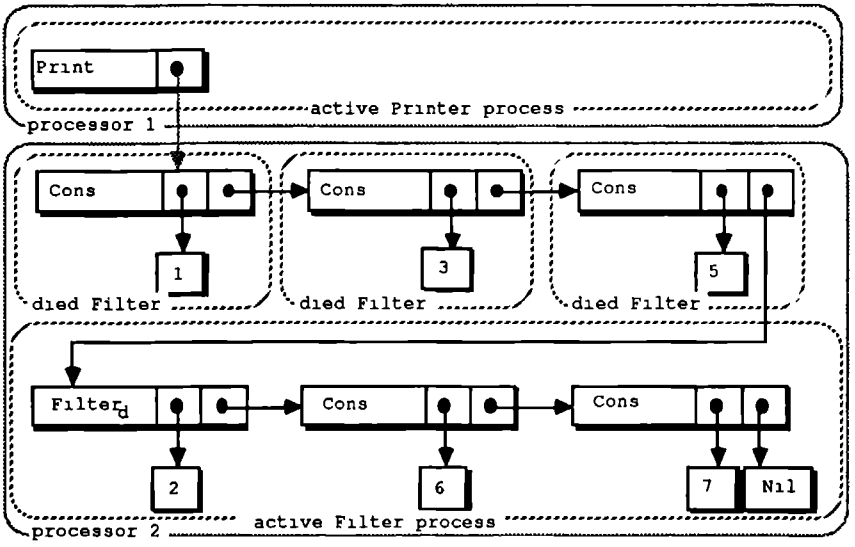
A virtual processor may contain several such families each producing a stream via a chain of channels. In the case of the following filter example the virtual processor contains exactly one such process: `Filter`. It sends a stream via the channel to the process `Print`.

The following example describes an asynchronous communication behaviour with streams:

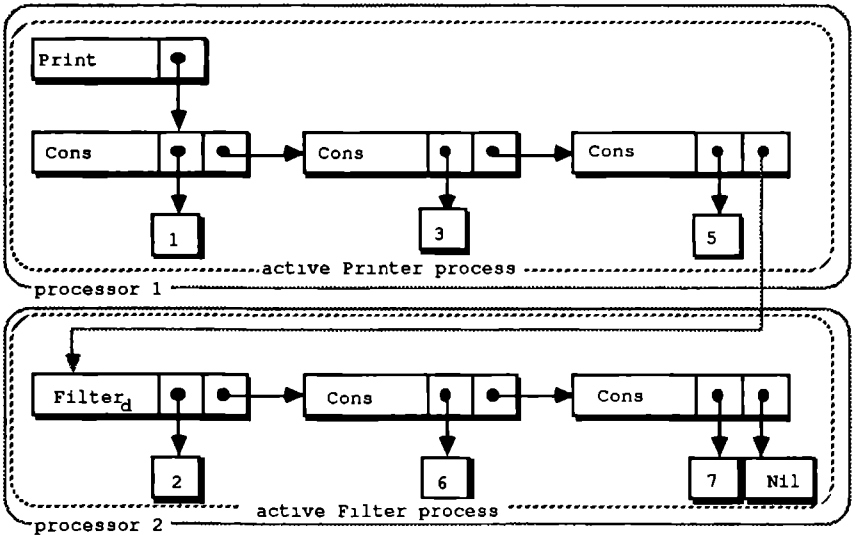
```
Start list                  ->  Print s,
                                s: {e} Filter list 2    ;


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.



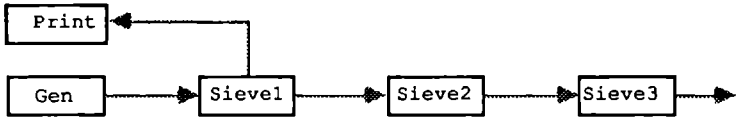
The sieve of Eratosthenes

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

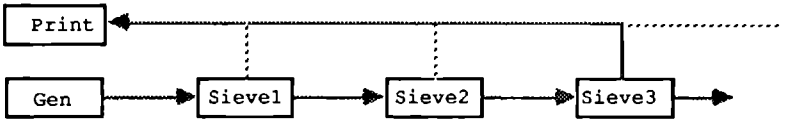
foregoing primes in the pipeline. If an incoming integer is not divisible by the local prime as well, it is send to the next `Sieve`. A newly created `Sieve` accepts the first incoming integer as its own prime and outputs this prime and the channel of the next `Sieve` to a printing processor. After that 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 : {el} Sym a1 .. an`	`n : I xc,`
	`x : LId yc,`
	`y : Sym a1 .. an`

`n : {il} Sym a1 .. an`	`n : LId x,` `x : Sym a1 .. an`

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: {el} 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: {il} 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 ($i \neq j$ & $i \neq k$). 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 ($0 \leq i$ & $1 \leq j,k$ & $i,j,k \leq N$).
So

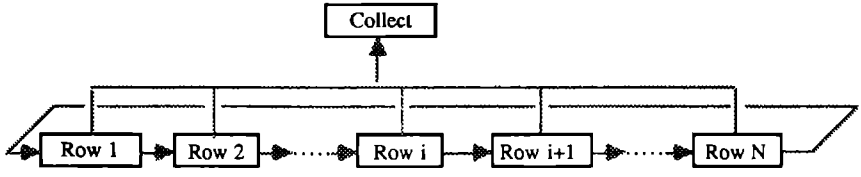
SP (j,k,0) = 0 if j=k
= d if there is an edge from j to k with distance d
= ∞ otherwise
SP (j,k,i) = minimum (SP (j,k,i-1), SP (j,i,i-1) + SP (i,k,i-1))

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 `Row`s 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:			the normal form can be: (doing I before F)	but also it can be: (doing F before I)
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;		

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.

8

REFERENCES

America, P. (1988). Definition of POOL2, a parallel object-oriented language. Philips Research Laboratories, Eindhoven, The Netherlands. ESPRIT project 415 A Doc. 0364.

Arvind, Nikhil, Rishiyur S. (1987). Executing a Program on the MIT Tagged Token Dataflow Architecture. Proceedings of Parallel Architectures and Languages Europe (PARLE), part I, Eindhoven, The Netherlands. *Springer Lec. Notes Comp. Sci.* **258**, 1-29.

Augusteijn L. (1985a). The Warshall shortest path algorithm in POOL-T. Philips Research Laboratories, Eindhoven, The Netherlands. Esprit project 415 A Doc. 0105.

Augusteijn L. (1985b). POOL_T User Manual. Philips Research Laboratories, Eindhoven, The Netherlands. Esprit project 415 A Doc. 0104.

Backus, J. (1978). Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, Vol **21**, 613-641.

Baeten, J.C.M. (1986). *Procesalgebra.* Kluwer.

Baeten, J.C.M., Bergstra, J.A., Klop, J.W. (1986). Term Rewriting Systems with Priorities. University of Amsterdam, Report FVI 86-03.

Baeten, J.C.M., Bergstra, J.A., Klop, J.W. (1987). Term Rewriting Systems with Priorities. Proceedings of the conference on Rewriting Techniques and Applications, Bordeaux, *Springer Lec. Notes Comp. Sci.* **256**, 83-94.

Barendregt, H.P. (1984). *The Lambda Calculus, Its Syntax and Semantics* (revised edition). Studies in Logic and the Foundations of Mathematics 103, North-Holland.

Barendregt, H.P., Leeuwen, M. van (1986). Functional Programming and the Language Tale. *Springer Lec. Notes Comp. Sci.* **224**, 122-207.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R. (1986a). Term graph rewriting. Internal Report 87, Department of Computer Science, University of Nijmegen, and also as Report SYS-C87-01, School of Information Systems, University of East Anglia.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R. (1986b). Towards an Intermediate Language based on Graph Rewriting. University of East Anglia and University of Nijmegen, Nijmegen internal report 88.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R. (1987a). Term Graph Rewriting. Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, The Netherlands. *Springer Lec. Notes Comp. Sci.* **259**, 141-158.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R. (1987b). Towards an Intermediate Language based on Graph Rewriting. Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, The Netherlands. *Springer Lec. Notes Comp. Sci.* **259**, 159-175.

Barendregt, H.P., Eekelen, M.C.J.D. van, Plasmeijer, M.J., Hartel, P.H., Hertzberger, L.O., Vree, W.G. (1987c). The Dutch Parallel Reduction Machine Project. Proceedings of the International Conference on Frontiers in Computing, Amsterdam, to appear in *Future Generations Computer Systems*.

Barendregt, H.P. (1988). Functional Programming and Lambda calculus, to appear in the Handbook of Theoretical Computer Science, North-Holland.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R. (1988). Towards an Intermediate Language based on Graph Rewriting. Revised version. to appear in the special issue of the *Journal of Parallel Computing* with selected papers of the conference on Parallel Architectures and Languages Europe (PARLE), Eindhoven, The Netherlands.

Boute, R.T. (1986). System Semantics and Formal Circuit Description. *IEEE Transactions on circuits and Systems*, Vol. CAS-**33**, No **12**, 1219-1231.

Broek, P.M. van den, Hoeven G.F. van der (1986). Combinatorgraph Reduction and the Church-Rosser Property. Department of Informatics, Twente University of Technology. Internal Report INF-86-15.

Brus, T., Eekelen, M.C.J.D. van, Leer, M.O. van, Plasmeijer, M.J. (1987). Clean - A Language for Functional Graph Rewriting. Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, USA. *Springer Lec. Notes Comp. Sci.* **274**, 364-384.

Burks, A.W., Goldstine, H.H., Neumann, J. von, (1946). Preliminary discussion of the logical design of an electronic computing instrument, in John von Neumann, Collected Works, Vol. **5**, Oxford, 35-79.

Burn, G.L., Hankin, C.L., Abramsky, S. (1985). The Theory and Practice of Strictness Analysis for Higher Order Functions. Research Report DoC 85/6, Imperial College London.

Burton, F.W. (1987). Functional Programming for Concurrent and Distributed Computing. *The Computer Journal* **30-5**, 437-450.

Church, A. (1932/1933). A Set of Postulates for the Foundation of Logic. *Annals of Math.* (2) **33**, 346-366 and **34**, 839-864.

Church, A., Rosser, J.B. (1936). Some Properties of Conversion. *Trans. Amer. Math. Soc.* **39**, 472-482.

Clocksin, W.F., Mellish, C.S. (1984). *Programming in Prolog*. Springer-Verlag.

Cousineau, G., Curien, P.L., Mauny, M. (1985). The Categorical Abstract Machine. Proceedings of the 2nd International Conference on Functional Programming Languages and Computer Architecture, Nancy, 50-64.

Curien, P.-L. (1986). *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Pitman.

Curry, H. B. (1930). Grundlagen der Kombinatorischen Logik. *Amer. J. Math.* **52**, 509-536, 789-834.

Eekelen, M.C.J.D. van, Plasmeijer, M.J. (1986). Specification of rewriting strategies in Term Rewriting Systems. Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico. *Springer Lec. Notes Comp. Sci.* **279**, 215-239.

Eekelen, M.C.J.D. van, Plasmeijer, M.J., Smetsers, J.E.W. (1988). Parallel Graph Rewriting on Loosely Coupled Machine Architectures. University of Nijmegen, Internal Report 88-9.

Ehrig, H. (1979). Introduction to the algebraic theory of graph grammars, in: Graph grammars and their Applications in Computer Science and Biology. *Springer Lec. Notes Comp. Sci.* **73**, 1-69.

Field, A.J., Harrison, P.G. (1988). *Functional Programming*. Addison-Wesley Publishers Ltd.

Futatsugi, K., Goguen, J., Jouannaud, J.P., Meseguer, J. (1985). Principles of OBJ2. *12th ACM Symp. on Principles of Programming Languages*, 52-66.

Glauert, J.R.W. (1978). A Single Assignment Language for Data Flow Computing. M.Sc. Thesis, Victoria University of Manchester.

Glauert, J.R.W., Holt, N.P., Kennaway, J.R., Reeve, M.J., Sleep, M.R., Watson, I. (1985). DACTL0: A Computational Model and an associated Compiler Target Language. University of East Anglia, internal report.

Glauert, J.R.W., Hammond, K., Kennaway, J.R., Sleep, M.R., Somner, G.W., Holt, N., Reeve, M., Watson, I. (1987a). Extensions to Core Dactl1. University of East Anglia.

Glauert, J.R.W., Kennaway, J.R., Sleep, M.R. (1987b). Category theoretic concepts of graph rewriting and garbage collection, in preparation, School of Information Systems, University of East Anglia.

Glauert, J.R.W., Kennaway, J.R., Sleep, M.R. (1987c). DACTL: A Computational Model and Compiler Target Language Based on Graph Reduction. *ICL Technical Journal* **5**, 509-537.

Glauert, J.R.W., Kennaway, J.R., Sleep, M.R. (1987d). Specification of Core Dactl1. University of East Anglia, report SYS-C87-09.

Glauert, J.R.W., Plasmeijer, M.J., Reeve, M.J. (198-). *Programming and Implementing Parallel Systems using Graph Rewriting*. To appear, MIT-Press.

Goguen, J., Kirchner, C., Meseguer, J. (1986). Concurrent term rewriting as a model of computation. Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico. *Springer Lec. Notes Comp. Sci.* **279**, 53-94.

Goos, J., Van Latum, F. (1987). Complete specification of practical rewriting strategies. M.Sc. Thesis, University of Nijmegen.

Groot, D. de & Lindstrom G. (eds) (1986). *Logic Programming: Functions, Relations and Equations*. Prentice-Hall.

Gurd, J.R., Kirkham, C.C., Watson, I. (1985). The Manchester Prototype Dataflow Computer. *Communications of the ACM*. **28-1**, 34-52.

Hartel, P., Vree, W. (1986). A Load Distribution Network for a Multi Processor Reduction Machine. Internal Report D-6, Dutch Parallel Reduction Machine project, University of Amsterdam.

Hintum, M. van, Schelven, R. van. (1988). MCC V3.0 Implementation Manual - a Miranda to Clean Compiler. University of Nijmegen. Internal Report 88-2.

Hoare, C.A.R. (1985). *Communicating sequential processes*, Prentice-Hall.

Hudak, P., Smith, L. (1986). Para-functional Programming: A Paradigm for Programming Multiprocessor Systems. *12th ACM Symp. on Principles of Programming Languages*, 243-254.

Huet, G. and Lévy, J.J. (1979). Call-by-need computations in non-ambiguous term rewriting systems. IRIA-Laboria, B.P. 105, 78150 Le Chesney, France, Report 359.

Jansen, T. (1987). Interpreting Lean. M. Sc. thesis, University of Nijmegen.

Johnson Th. (1984). Efficient compilation of lazy evaluation. Proceedings of the ACM SIGPLAN '84, Symposium on Compiler Construction. *SIGPLAN Notices* **19/6**.

Kennaway, J.R. (1984). An outline of some results of Staples on optimal reduction orders in replacement systems, Report CSA/19/1984, School of Information Systems, University of East Anglia, Norwich, England.

Kennaway, J.R. (1988a). Correctness proof for Functional Dactl1. University of East Anglia. Internal Report: in preparation.

Kennaway, J.R. (1988b). Implementing Term Rewrite Languages in Dactl. Proceedings of the 13th Colloquium on Trees in Algebra and Programming (CAAP'88), Nancy. *Springer Lec. Notes Comp. Sci.* **299**, 102-116.

Kennaway, J.R., Sleep, M.R. (1988). Director Strings as Combinators. to appear in *Transactions on Programming Languages and Systems.*

Klop, J.W. (1980). *Combinatory Reduction Systems*, Mathematical Centre Tracts n.127, Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam.

Klop, J.W. (1985). Term rewriting systems. Notes for the Seminar on Reduction Machines, Ustica, to appear.

Klop, J.W. (1987). Term rewriting systems: a tutorial. Center for Mathematics and Computer Science, CWI Amsterdam. Note CS-N8701.

Kluge, W.E. (1983). Cooperating reduction machines. *IEEE Transactions on computers* **C-32/11**, 1002-1012.

Koopman, P.W.M., Nöcker, E.G.J.M.H. (1988). Compiling Functional Languages to Term Graph Rewrite Systems. University of Nijmegen. Internal Report 88-1.

Koster, C.H.A. (1971). Affix Grammars. in: Algol68 Implementation, 95-109, North-Holland.

Lewis, H.R., Papadimitriou, C.H. (1981). *Elements of the theory of Computation.* Prentice-Hall.

Magó, G.A. (1980). A Cellular Computer for Functional Programming. digest of Papers, IEEE Comp. Soc. COMPCON, 179-187.

McBurney D.L., Sleep, M.R. (1987). Transputer-based experiments with the ZAPP architecture. Proceedings of Parallel Architectures and Languages Europe (PARLE), part I, Eindhoven, The Netherlands. *Springer Lec. Notes Comp. Sci.* **258**, 242-259.

Meijer, H. (1986). *Programmar.* Ph.D. Thesis, University of Nijmegen, The Netherlands.

Milner, R. (1978). A Theory of Type Polymorphism in Programming, *Journal of Computer and System Sciences*, Vol. **17**, no. **3.**, 348-375, Academic Press.

Mycroft, A. (1984). Polymorphic type checking and recursive definitions, Proceedings 6th International Conference on Programming, Toulouse, *Springer Lec. Notes Comp. Sci.* **167.**

Nagl, M. (1979). *Graph Grammars - Theory, Applications, Implementation* (in German), Vieweg Verlag.

Nöcker E. (1988). Strictness analysis and pattern matching. University of Nijmegen. Internal Report: in preparation.

O'Donnell, M.J. (1985). *Equational Logic as a Programming Language*. Foundations of Computing Series, MIT Press.

Odijk, E.A.M. (1985). DOOM: a Decentralized Object-Oriented Machine. Philips, Eindhoven, Esprit 415 A internal report, Doc. 0125.

Odijk, E.A.M. (1987). The DOOM system and its applications: a survey of Esprit 415 subproject A. Proceedings of Parallel Architectures and Languages Europe (PARLE), part I, Eindhoven, The Netherlands. *Springer Lec. Notes Comp. Sci.* **258**, 461-479.

Peyton Jones, S. L. (1987a). FLIC - a Functional Language Intermediate Code. Dept. of Comp. Sc., University College London, internal working paper.

Peyton Jones, S.L. (1987b). *The Implementation of Functional Programming Languages*. Prentice-Hall.

Plasmeijer, M.J., Eekelen, M.C.J.D. van (198-). *Functional Programming and Graph Rewriting Systems*. To appear as textbook, end of 1989.

Raoult, J.C. (1984). On graph rewritings. *Theor. Comput. Sci.* **32**, 1-24, North-Holland.

Robinson, J.A. (1965). A machine-oriented logic based on the resolution principle, *Journal of the A.C.M.* **12**, 1.

Schönfinkel, M. (1924). Über die Bausteine der mathematischen Logik. *Math. Annalen* **92**, 305-316.

Smetsers, J.E.W., Eekelen, M.C.J.D. van, Plasmeijer, M.J. (1988). Operational semantics of Concurrent Clean. University of Nijmegen. Internal report: in preparation.

Staples, J. (1980a). Computation on graph-like expressions, *Theor. Comput. Sci.* **10**, 171-185, North-Holland.

Staples, J. (1980b). Optimal evaluations of graph-like expressions, *Theor. Comput. Sci.* **10**, 297-316, North-Holland.

Staples, J. (1980c). Speeding up subtree replacement systems, *Theor. Comput. Sci.* **11**, 39-47, North-Holland.

Turing, A.M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings, London Mathematical Society*, **2**, no. **42** (1936), 230-265, and no. **43**, 544-546.

Turing, A.M., (1937). Computability and λ-definability. *J. Symbolic Logic* **2**, 153-163.

Turner, D.A. (1979a). A new Implementation Technique for Applicative Languages. *Softw. Pract. and Experience*, Vol **9** (**1**), 31-49.

Turner, D.A. (1979b). SASL Language Manual, "combinators" version, University of St. Andrews, U.K.

Turner, D.A. (1985). Miranda: A non-strict functional language with polymorphic types. Proceedings of the 2nd International Conference on Functional Programming Languages and Computer Architecture, Nancy, *Springer Lec. Notes Comp. Sci.* **201**, 1-16.

Turner, D.A. (1986). Miranda System Manual. Research Software Ltd.

Veen, A.H. (1985). *The Misconstrued Semicolon*. Ph.D. Thesis, Technical University Eindhoven, The Netherlands.

Vegdahl, Steven R. (1984). A Survey of Proposed Architectures for the Execution of Functional Languages. *IEEE Transactions on Computers*, vol. c-**33**, no. **12**.

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

Wadsworth, C.P. (1971). *Semantics and Pragmatics of the Lambda-Calculus*. Ph.D. Thesis, Oxford University.

Watson P., Watson I. (1987). Evaluating Functional Programs on the Flagship Machine. Proceedings of the 3rd International Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, USA. *Springer Lec. Notes Comp. Sci.* **274**, 88-97.

Wirth, N. (1982). *Programming in Modula-2*. Springer-Verlag.

SUMMARY

A *model of computation* (or a *computational model*) of a programming language is a formal model as close as possible to both semantics and implementation modelling only the essential aspects of them by making some abstractions. Via a model of computation it is much easier to reason about the language, its programs, its compilers and its dedicated machines.

It is of course very difficult to find an ideal model which models every essential aspect of a programming language. In fact the classical (sequential and imperative) programming languages all have the same model of computation: the Turing machine. Clearly this model can only describe the very basic concepts of those languages. In this thesis a computational model is investigated for a general paradigm of programming (the declarative paradigm) and for one style of programming in particular (functional programming).

In the *declarative* paradigm a desired computation is expressed in a static fashion as a list of declarations and an expression to be evaluated. A program is considered to be an executable specification. The most important property in the declarative paradigm is that an expression always has the same meaning independent of the history of the computation (*referential transparency*). Functional programming is very attractive because mainly through the use of higher order functions the expressive power of functional programming languages is higher than of conventional languages. Functional programming is more like mathematically specifying the algorithm. Functional programs are therefore generally shorter than their conventional counterparts and thus easier to enhance and maintain.

Originally, the λ-calculus was commonly used as the computational model for functional languages. Combinatory logic however, has a much simpler substitution mechanism than the λ-calculus but it lacks pattern matching. Term rewriting systems do contain pattern matching but it is impossible to express sharing directly in term rewriting systems. Graph rewriting systems combine all these aspects, so they are investigated in this thesis in a general context and more specifically as the model of computation for functional languages and their implementations.

In chapter 3 we have introduced multi-level rewriting systems which in practise are proven to be very useful for high level specifications varying from specifying reduction strategies to specifying the operational semantics of a programming language.

With term graph rewriting which is introduced in chapter 4, some fundamental theoretical results are obtained on modelling term rewriting with graph rewriting. *Term graph rewriting* means that a term rewriting system (TRS) is interpreted (lifted) as a graph rewriting system (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. Term graph rewriting is a very promising topic for further research.

In chapter 5 generalized graph rewriting is defined which is very powerful and of independent interest as a general model of computation for parallel architectures. Lean is an experimental language for specifying computations in terms of graph rewriting. It is very powerful since there are few restrictions on the graph that is transformed and the transformations that can be performed. It is worthwhile to further investigate generalized graph rewriting yielding as a spin-off programming languages based on subclasses with specific properties and advantages

Clean which is described in chapter 6, is an example of such a spin-off language. The language is based on restricted graph rewriting (functional graph rewriting). Clean is in practise proven to be very suited as an intermediate language for functional languages and sequential machine architectures. An efficient sequential implementation of a high level functional language has been constructed by using Clean as an intermediate language.

In chapter 7 graph rewriting is extended with lazy copying and explicit parallelism yielding a very promising model for loosely coupled parallel evaluation of functional programs. Its expressive power and its properties will make it possible in the near future to exploit the potential parallelism in functional programs successfully in a general way. Actual parallel implementations are started.

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 are necessary to achieve more experience and to identify key issues.

Parallel Graphherschrijven

—

Enkele bijdragen aan de theorie, de implementatie en de toepassing ervan.

SAMENVATTING

Een *berekeningsmodel* van een programmeertaal is een formeel model zo dicht mogelijk bij zowel de semantiek als de implementatie waarbij alleen de essentiële aspecten gemodelleerd worden door enkele abstracties te maken. Door zo'n berekeningsmodel is het veel gemakkelijker om te redeneren over de taal en de bijbehorende programma's, vertalers en machines.

Het is natuurlijk heel moeilijk om een ideaal model te vinden dat elk essentieel aspect van een programmeertaal modelleert. Alle klassieke (sequentiële en imperatieve) programmeertalen hebben in feite het zelfde berekeningsmodel: de Turing machine. Het is duidelijk dat dit model alleen de meest fundamentele aspecten van die talen kan modelleren. Dit proefschrift beschrijft onderzoek naar een berekeningsmodel voor een algemeen paradigma van het programmeren (het declaratieve paradigma) en voor één stijl van programmeren in het bijzonder (functioneel programmeren).

In het *declaratieve* paradigma wordt een gewenste berekening uitgedrukt op een statische manier als een lijst van decalaraties met een expressie die geëvalueerd dient te worden. Een programma wordt beschouwd als een executeerbare specificatie. De belangrijkste eigenschap in het declaratieve paradigma is het feit dat een expressie altijd dezelfde betekenis heeft onafhankelijk van de geschiedenis van de berekening (de taal is *referentieel transparant*). Functioneel programmeren is heel aantrekkelijk omdat voornamelijk door het gebruik van hogere orde functies de uitdrukkingskracht van functionele programmeertalen groter is dan die van conventionele talen. Functioneel programmeren lijkt meer op het mathematisch specificeren van het algorithme. Functionele programma's zijn daarom over het algemeen korter dan hun conventionele tegenhangers en dus zijn ze gemakkelijker te verbeteren en te onderhouden.

Oorspronkelijk werd λ-calculus algemeen gebruikt als het berekeningsmodel voor functionele talen. Combinatorische logica heeft echter een eenvoudiger substitutiemechanisme dan de λ-calculus maar het heeft geen pattern matching. Termherschrijfsystemen bevatten wel pattern matching maar termherschrijfsystemen hebben niet de mogelijkheid om sharing direct uit te drukken. Graphherschrijfsystemen combineren al deze aspecten. Daarom zijn graphherschrijfsystemen het onderwerp van onderzoek in dit proefschrift. Graphherschrijfsystemen worden onderzocht in algemene zin en als berekeningsmodel voor functionele talen en hun implementaties.

In hoofdstuk 3 hebben we meer-niveau herschrijfsystemen geïntroduceerd. Van deze systemen is het in de praktijk bewezen dat ze bijzonder nuttig zijn voor hoog niveau specificaties varierend van het specificeren van reductiestrategieën tot het specificeren van de operationele semantiek van een programmeertaal.

Met behulp van termgraphherschrijven (ingevoerd in hoofdstuk 4) zijn enkele fundamentele resultaten behaald op het gebied van het modelleren van termherschrijven met graphherschrijven. *Termgraphherschrijven* betekent dat een TermHerschrijfSysteem (THS) wordt geïnterpreteerd als (getild naar) een GraphHerschrijfSysteem (GHS). De normaalvormen van het GHS, dat zijn graphen, worden ontrafeld tot termen in de THS-wereld. Via termgraphherschrijven wordt bewezen dat het 'sharen' van termen gezond is. Bovendien worden restricties gegeven die de compleetheid garanderen van implementaties die sharing gebruiken. Termgraphherschrijven is een veelbelovend onderwerp voor verder onderzoek.

In hoofdstuk 5 wordt gegeneraliseerd graphherschrijven gedefinieerd. Dit berekeningsmodel is bijzonder krachtig en het is van onafhankelijk belang als algemeen berekeningsmodel voor parallelle architecturen. Lean is een experimentele taal voor het specificeren van berekeningen in termen van graphherschrijvingen. Deze taal is bijzonder krachtig aangezien er slechts weinig restricties zijn op de graph die getransformeerd wordt, en eveneens zijn er weinig restricties op de transformaties die uitgevoerd kunnen worden. Het is de moeite waard gegeneraliseerd graphherschrijven verder te onderzoeken, wat als 'spin-off' programmeertalen oplevert die gebaseerd zijn op deelklassen met specifieke eigenschappen en voordelen.

Clean (beschreven in hoofdstuk 6) is een voorbeeld van zo'n spin-off taal. De taal is gebaseerd op een beperkt soort graphherschrijven (functioneel graphherschrijven). De praktijk heeft uitgewezen dat Clean zeer geschikt is als tussentaal voor functionele talen en sequentiële machine-architecturen. Een efficiënte sequentiële implementatie van een hogere functionele programmeertaal is met behulp van Clean als tussentaal tot stand gekomen.

In hoofdstuk 7 wordt graphherschrijven uitgebreid met lui copiëren en expliciet parallellisme, hetgeen een veelbelovend model oplevert voor los-gekoppelde parallelle evaluatie van functionele programma's. De uitdrukkingskracht en de eigenschappen van die uitbreidingen zullen het in de nabije toekomst mogelijk maken om het potentiële parallellisme in functionele programma's op een algemene manier succesvol te benutten. Met daadwerkelijk parallelle implementaties is een begin gemaakt.

Er dient nog veel onderzoek op het gebied van graphherschrijfsystemen gedaan te worden (taxonomie, typering, strategieën, strictheidsanalyse, implementatietechnieken, parallelle evaluatie, 'garbage' collectie, etcetera). Maar de tot nu toe geboekte resultaten zijn veelbelovend hetgeen verder onderzoek rechtvaardigt op het gebied van de graphherschrijftheorie en naar het indentificeren van speciale klassen van graphherschrijfsystemen. Bovendien zijn daadwerkelijke experimenten met sequentiële en parallelle implementaties noodzakelijk om meer ervaring te verkrijgen en om de kernproblemen te identificeren.

CURRICULUM VITAE

De schrijver van dit proefschrift is geboren op 5 December 1956 in Bergen op Zoom (N-B). Na aldaar aan de St. Jozefschool het lager onderwijs genoten te hebben, ging hij op tienjarige leeftijd naar het Mollerlyceum in dezelfde plaats waar hij het gymnasium-β diploma in 1973 behaalde. In hetzelfde jaar begon hij aan de wiskunde studie op de Katholieke Universiteit Nijmegen waar hij in 1981 het doctoraalexamen wiskunde behaalde inclusief onderwijsbevoegdheid.

Sindsdien is hij als wetenschappelijk medewerker/universitair docent werkzaam bij de sectie/discipline Informatica van de Faculteit der Wiskunde en Natuurwetenschappen aan de Katholieke Universiteit Nijmegen. In die hoedanigheid heeft hij achtereenvolgens gewerkt bij de afdeling Informatica I (Programmeertalen en hun Vertalers) bij Prof. C.H.A. Koster, de afdeling Informatica II (Machinearchitectuur en Bedrijfssystemen) bij Prof. dr. ir. R.T. Boute en de vakgroep Theoretische Informatica en Berekeningsmodellen bij Prof. dr. H.P. Barendregt. Voorts heeft hij onder leiding van Dr. ir. M.J. Plasmeijer van 1984 tot en met 1987 gewerkt bij de Nijmeegse tak van het Nederlandse Parallelle-Reductie-Machine project (een samenwerkingsproject van drie Nederlandse Universiteiten).

Stellingen behorend bij het proefschrift

Parallel Graph Rewriting

—

Some Contributions to its Theory, its Implementation and its Application

Marko van Eekelen 2 December 1988

1 Het is mogelijk met behulp van graphherschrijfsystemen functionele programmeertalen efficiënt te implementeren op parallelle machine-architecturen die bestaan uit los-gekoppelde traditionele sequentiële processoren.

2 Het overvloedig gebruik van functiecompositie als programmeerstijl komt weliswaar vaak de correctheid en de bewijsbaarheid ten goede maar de leesbaarheid wordt er veelal door geschaad.

3 Het afleiden van types in de functionele programmeertaal Miranda dient beschouwd te worden als een faciliteit van de programmeeromgeving en niet als een inherente eigenschap van die programmeertaal.

4 Het verdient aanbeveling bij het inleidend universitair programmeeronderwijs de beginselen van functioneel programmeren te onderwijzen voordat de beginselen van imperatief programmeren aan de orde komen.

5 Samenwerking tussen onderzoekers op internationaal en op nationaal niveau is vruchtbaarder naarmate de motivatie voor de samenwerking meer op inhoudelijk dan op financieel gebied ligt.

6 De totale tijd die door systeembeheerders en gebruikers besteed wordt aan het functioneren van electronic mail, is veel groter dan de resulterende tijdswinst bij het overbrengen van boodschappen.

7 Bij elk nieuw boek dient de verantwoordelijke uitgever de inhoud via electronische middelen aan de blindenbibliotheek ter beschikking te stellen opdat het boek ook in braille snel en goedkoop beschikbaar kan komen.

8 De bepaling in de promotie-reglementen dat in een proefschrift de promotor en de co-referent niet bedankt mogen worden, doet onrecht aan hun voortreffelijke ondersteuning bij het tot stand komen van het proefschrift.

9 Wanneer men de vergrijzing van Nederland effectief wil bestrijden dan moet men er voor zorgen dat er bij elke werkplek in Nederland op geringe afstand goede kinderopvang aanwezig is.

10 Voor het welslagen van een volksdansdemonstratie is het noodzakelijk dat de demonstratiegroep zich in de betreffende volksaard inleeft zonder dat spontaniteit en enthousiasme verloren gaan. Wellicht is dit de reden dat veelal het meest treffende resultaat bereikt wordt, wanneer dans en volksaard natuurlijkerwijs overeenkomen.

11 Hoewel de uitspraak "D'r ga niks bove Berrege" geografisch en taalkundig evident onjuist is, verkondigt hij voor diegenen bij wie de Bergse Vastenavend met de paplepel is ingegoten, gevoelsmatig een eeuwigdurende waarheid.

12 Een kwaliteitskrant zou niet alleen gekenmerkt moeten worden door het feit dat de lezer zelden afgeeft op de krant, maar ook door het feit dat de krant zelden afgeeft op de lezer.