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.