Summary

The development of correct and reliable software is a complex bussiness. In a society that increasingly relies on an increasing amount of software it is clear that it is urgent that programs are correct. One way to reduce this complexity is to use declarative programming languages. In this family of programming languages programs are constructed by defining *what* should be solved instead of *how* it should be solved. One class of declarative programming languages is formed by functional programming languages. These are often credited for producing concise programs that are easy to write and understand, are amenable for proving program correctness and program transformations. One of the reasons that functional programming languages are declarative is that they have a computational model that is an abstraction from the notion of computation done by computing machinery. Functional programmers are not concerned with memory management and computational control flow.

Society increasingly uses computers. The reason that computers are so useful is the fact that they not only compute, but also are *interactive*, in the true sense of the word[1]. Computers can interact with users, other computers, and conventional machinery such as robots, plains, trains and automobiles. Programs define how this interaction should occur. However, the development of correct and reliable *interactive* software is complex. One way to decrease this complexity is to find a declarative method. This is the major theme of this PhD.Thesis.

In this thesis we claim that functional computation blends smoothly with interaction. The language we have used as our research tool is the functional programming language Clean. The approach taken in this PhD.Thesis consists of two main steps. The first, and fundamental step is to arrange interaction in such a way that it is a form of functional computation. So *interaction is computation*, and computation is what functional languages are good at. This step has been made possible by applying the special uniqueness type system of Clean. This type system is based on the term graph rewriting semantics of Clean. The second step of our approach is to consider how to fit in the various kinds of interaction in the functional scheme.

The kinds of interaction that we have studied are file I/O and graphical user interface I/O. It turns out that, in terms of programming style file I/O can be handled as a special case of computation. Graphical user interface I/O is distinguished from file I/O by being dynamic. Therefore programming graphical user interfaces requires a different approach. In this approach we have applied the flexibility of algebraic data types of functional languages to define a specification language in which graphical user interface elements are described. These consist of standard elements such as windows, menus, and controls, but also timers and receivers. In these definitions one includes functions, called abstract event handlers, that define the response of the program in case a certain situation, or abstract event, arises. Each of these functions can be applied to the current state of the program and yields a new state. Because Clean is a strongly typed language, the type system verifies that all specifications are type-correct. Given the specification of an interactive program, the system will do all low-level event handling and evaluation of the proper abstract event handlers. The corresponding semantics of the interactive program is that of a state transition system. This semantics can be given straightforwardly by a simple set of functions.

Modular programming is a language independent key factor in reducing the complexity of software development. It allows programmers to split a programming task into smaller, inde-

[1] **interactive** …(1832) **1:** mutually or reciprocally active **2:** of, relating to, or being a two-way electronic communication system (as a telephone, cable television, or a computer) that involves a user's orders (as for information or merchandise) or responses (as to a poll)… (Webster's, 1985)

pendent programming tasks. In this thesis we show how to construct interactive programs in a modular fassion based on interactive processes. The concept of interactive process is more general than the concept of interactive program. An interactive process is in essence a state transition system, and is specified in exactly the same way as interactive programs earlier. Interactive programs consist of an arbitrary number of interactive processes. Each interactive process can create new interactive processes, and each interactive process can do file I/O and graphical user interface I/O. It is shown that interactive processes can be evaluated purely sequentially, in an interleaved fassion, but also in parallel.

In this thesis we introduce two forms of inter-process communication: data sharing and message passing. Both forms of communication integrate smoothly in the system. Data sharing is a form of communication by a common piece of global data. Access to this data is restricted to one process at a time. Message passing is a flexible means of many-to-one communication. Provided that interactive processes have the identification of a given interactive process they can send messages to that interactive process.

State transition systems prove to be a useful and generally applicable concept. We have applied them as a basis for interactive processes and data sharing. Taking a slightly different point of view one can regard state transition systems also as objects of the object-oriented paradigm. In this view one can construct objects from objects in an orthogonal way, and assign a functional, operational semantics to object structures. As an application, we demonstrate how we can extend graphical user interface elements with local state, and obtain interactive processes and data sharing as a special case of structuring objects.

All systems presented in this thesis are polymorphic and type-safe. Polymorphism allows free choice of the type of states, messages, and object data. We have applied the strong type system of Clean to obtain a type-safe system. During this research it has become apparant that the conventional polymorphic Milner/Mycroft type system, on which the type systems of many modern functional languages are based, is not sufficiently powerful to handle these complex systems. We show that although multiple implementation abstract data types can be handled in a polymorphic Milner/Mycroft type system, a more elegant and more general solution is to extend the type system with existential types. The message passing system is type-safe, but cannot be typed statically in the type system. To construct well-typed interpretation functions of object structures we need to extend the type system with type and constructor classes.

In this thesis we have constructed a functional, declarative system in which one can develop interactive functional programs on a high level of abstraction. We think that it is one step towards reducing the complexity of developing correct and reliable software.