1
Introduction

This PhD.Thesis is the result of a four-year research project conducted at the Department of Functional Programming, the Faculty of Mathematics and Informatics, at the University of Nijmegen as 'Assistent-in-Opleiding' under supervision of Professor M.J. Plasmeijer. In this chapter we give a survey of the research project and its theses.

1.1 Functional programming and the Clean project

Functional programming languages are often credited for producing concise programs that are easy to write and understand, are amenable for proving program correctness and for doing program transformations. The major features of functional programming languages that contribute to these desirable properties are *referential transparency*, the *abstraction from a specific order of evaluation*, *higher-order functions*, and *strong typing*.

Referential transparency.

A programming language is said to be referential transparent if every legal expression in the language always has the same meaning regardless of its program context and order of evaluation. By this definition we can distinguish between functional languages that are referential transparent, the *pure* functional languages, and those that are not, the *impure* functional languages.

Abstraction of evaluation order.

A functional program consists of an expression that has to be evaluated according to a given set of function definitions. In general, the expression contains many reducible expressions, or *redexes*. In a pure functional language the order in which redexes are contracted is irrelevant modulo termination. This implies that functional languages are, in theory, well-suited for parallel evaluation. The evaluation strategy of a language is an algorithm that selects redexes. Of the sequential evaluation strategies the lazy and eager strategies are the most frequently used. The lazy evaluation strategy evaluates arguments of a function only if they are needed to contract the function. The eager strategy evaluates all arguments of a function before the function itself is evaluated.

From a theoretical point of view the lazy evaluation scheme is more powerful than the eager scheme because the first is *strongly normalising*, while the second is not. An evaluation strategy is strongly normalising if for every program that has a solution, the *normal form*, then this solution will be found by the strategy. From a programming practical point of view, lazy evaluation increases programming expressiveness (Hughes, 1990).

Again, we can distinguish functional languages that use a lazy evaluation strategy, the *lazy* functional language, and functional languages that use an eager evaluation strategy, the *eager* functional languages.

Higher-order functions.

In functional languages functions are first-class citizens. This means that they can be used as function arguments. However, they can also be used as components of data structures. The latter gives a further increase in programming expressiveness because it enables us to define functions that can apply arbitrary operations on data structures (Hughes, 1990).

Strong type systems.

Programming languages with strong type systems are invaluable as they ensure program correctness with respect to types that can be determined statically. Most functional programming languages allow the programmer to define complex, recursive, data structures easily, using algebraic data types and type constructors for lists, tuples, and functions. Type systems enforce a trade-off between static security and flexibility: not all functional expressions that make sense can be typed in a given type system, and in this PhD.Thesis we will encounter some of these cases which show up because of the complexity of the I/O system.

The functional programming language we consider in this PhD.Thesis is pure and lazy. Due to strong normalisation lazy functional languages are preferable over their eager relatives. However, lazyness complicates their implementation. In general lazy languages are less time and space efficient than eager languages, or imperative languages. A lot of research has been conducted in this area in the last decades, and is still continuing.

One specific approach to investigate and provide solutions to the problems of efficient implementations of functional languages is the (Concurrent) Clean project. *Clean* is a sequential lazy, purely functional programming language (Brus *et al.,* 1987; Groningen *et al.*, 1991; Smetsers *et al*., 1991; Plasmeijer and van Eekelen, 1993). *Concurrent* Clean is the parallel extension to Clean (van Eekelen, 1988; van Eekelen *et al*., 1991; Nöcker *et al.,* 1991).

At the beginning of this research project, Clean had evolved into the 0.6 version (van Eekelen *et al*., 1990). During this research the Clean language has gone through gradual and considerable changes, resulting in the Clean 1.0 version. All Clean examples in this thesis are presented in the 1.0 syntax. The main features that distinguish Clean from other functional languages are that its semantics is explicitly based on term graph rewriting (Barendregt *et al.,* 1987; Barendsen and Smetsers, 1992; Smetsers, 1993), it has an efficient and powerful strictness analysis (Nöcker, 1993; 1994), it has evaluation annotations for partially eager evaluation (Nöcker and Smetsers, 1993) and parallel evaluation, it has a polymorphic Milner/Mycroft type inference and verification system extended with uniqueness types (Smetsers *et al.,* 1993; Barendsen and Smetsers, 1993*a-b*; see also Barendsen, 1995), record types, existential types, type classes and constructor classes (Plasmeijer and van Eekelen, 1994). Clean is a modular language with abstract types based on the module system.

In a term graph rewriting system functions are represented by graph rewrite rules, and computational expressions are represented by computation graphs. One advantage is that sharing of computations is an explicit and intuitive concept in the system, in contrast with lambda calculus and term rewriting systems. Term graph rewriting obeys the functional semantics. Functional programs written in Clean are therefore term graph rewriting systems in disguise. Usually, programmers need not be aware of this fact, but at some occasions its nature shows up in for instance the way function types are dealt with and the uniqueness type system.

1.2 Research objectives

The Clean project demonstrates that lazy, pure functional programming languages can be compiled efficiently and result in efficient applications. However, lazy evaluation not only complicates the compilation process, but is also an obstacle to the incorporation of input/output (I/O from hereon). Doing I/O in a lazy, pure functional programming language seems to be a contradiction: pure languages do not have the notion of *side-effects* and lazy languages lack a precise *control of the evaluation order*. These are precisely the important aspects for any I/O model. The objectives of this PhD.Thesis were roughly the following:

- Can we reconcile I/O with lazy, pure functional languages?
- How should we program interactive functional programs?
- Can I/O be efficiently implemented in a lazy, pure functional language?

When considering this project there were many functional I/O systems and languages (Henderson, 1982; Redelmeier, 1984; Stoye, 1984; Perry, 1988; Williams and Wimmers, 1988; Dwelly, 1989; Backus *et al*., 1990; Thompson, 1990; Turner, 1990). A survey on the history of functional I/O systems can be found in Gordon (1993), see also Section 2.1 for a brief summary). For a number of reasons we felt that it was justified to initiate a research project of our own on this matter:

In contrast with other functional languages, Clean is a graph rewriting language. We wanted to know to what extent the existing I/O paradigms were transferable to the Clean language, and also we wanted to find out if an I/O system based on a graph rewriting language could provide new kinds of solutions. In this aspect, the ideas behind the file I/O system of Clean 0.6 were different from other functional I/O systems, but we did not know to what extent and if it would turn out to be a fruitful approach when integrating it with graphical user interfaces (dealt with in sections 2.4 upto 2.6 and 7.3).

Graphical user interface applications seemed to require an approach radically different from traditional teletype, or file, I/O because of its dynamic and complex behaviour. We wanted to see if functional languages were capable of expressing this kind of interaction in a declarative, high-level functional style.

It seemed that, when it comes to interacting with the imperative world, all the functional I/O systems we knew at that time had to sacrifice either or both the elegance and purity of functional programming. As we argued in the previous section one of the keys to the success of functional programming is its abstraction from imperativeness. Finding a good solution to this problem has a wide range: by confluence functional languages can be evaluated in parallel. An imperative I/O scheme might ruin this important advantage of functional programming over imperative programming.

Compilers for functional languages, including the Clean compiler, were improving on efficiency compared with imperative languages. To be able to put the Clean language to the test, there was a growing need for 'real-world' applications. Real-world applications require good interactive facilities. Therefore Clean needed to be extended to support interaction.

1.3 Research themes

The research objectives put the research project into two directions: to find a functional I/O scheme that can suitably be incorporated in a graph rewriting language and implemented in Clean, and to find a declarative, high-level style of programming interactive applications to master the complexity and dynamics of such applications. The main research themes that evolved from these two lines are:

Environment passing is a natural paradigm for a graph rewriting language. One of the functional I/O paradigms is the environment passing paradigm. In this paradigm external world resources are defined as ordinary functional values, and operations are defined as ordinary function applications. Although this is very functional, it is not straightforward to incorporate the paradigm in a functional language because one cannot easily maintain the representation of unique resources by non-unique values. We show that environment passing is a natural paradigm for a graph rewriting language, and also that the representation problem can be solved elegantly in such a language.

Interactive applications can be structured as compositions of state transition systems. Graphical user interface applications have a dynamic behaviour and the task of programming such applications is a complex business. We show that by structuring interactive programs as compositions of state transition systems we can master the complexity. Graphical user interface elements such as windows, menus, dialogues, can be integrated fluently in this framework. This approach results in a declarative style of programming interactive applications. It is important to observe that we can handle the semantics completely within a pure functional framework.

State transition systems can be used for concurrent interactive processes. The concept of state transition systems is well suited as an operational definition of interactive processes. We show that in a functional framework we can define the corresponding notions of dynamic process management, and inter-process communication. Furthermore, we show how to arrange the functional semantics in such a way that we obtain concurrent, deterministic, interactive processes.

In the following sections we discuss each of the theses informally and relate them with the contents of this thesis.

1.4 World-as-value

In this section we introduce the basic I/O paradigm developed in this thesis. We illustrate the system by means of a very simple interactive program. This program should first prompt a user for some keyboard input, read the input, echo the input on screen, and then terminate. So the program has two output actions, one input action, and it must terminate. In the imperative programming language C (Kernighan and Ritchie, 1988) this program looks like this:

```
#include <stdio.h>

main()
{
    char* text;

    printf("Type a text:\n");
    text  = gets(text);
    printf("%s\n",text);
}
```

Figure 1.1 A simple interactive program in the imperative language C.

Novice programmers in an imperative language usually write such programs as their first exercise. The order of evaluation is given by the textual order of the commands, separated by the symbol '`;`'. The main feature of imperative languages is their implicit, global state, that can be accessed and updated directly in any part of the program. In the example this is done by the procedures `gets` and `printf`.

In contrast with imperative languages, books on lazy functional programming languages usually have to defer I/O until lazy evaluation has been handled because the most accepted method of incorporating I/O in a lazy functional language is to define an interactive program as a function that transforms an infinite input stream to an infinite output stream.

In this PhD.Thesis we take an alternative route to incorporate I/O in a lazy functional language. This method, the *explicit multiple environment passing* scheme is introduced and described extensively in Chapter 2. Here we introduce the scheme informally and illustrate it by means of the above programming task. For brevity, we coin the term *world-as-value* as a shorthand synonym for the explicit multiple environment passing scheme.

In the world-as-value paradigm the relations between program and global state that are implicit in the imperative style of programming are made explicit. Components of the global state are made explicitly visible as *environments*. An environment is an abstract, specialised data structure that encodes the state of a *specific* part of the world. Operations on that part of the world are represented by functions that take the corresponding environment and yield a new environment in which the change is encoded.

The top environment of the Clean environment hierarchy is the special type `World` in the type system of Clean. It is an abstract type and there is no denotation to obtain a value of this type. The only way a program can obtain a value of type `World` is as an argument of its main function. A program that takes a `World` environment is enforced to yield one as the result, so interactive programs are functions of type `World`→`World`.

Operations on the world environment are collected separately in the I/O libraries of Clean. This is similar to the treatment of basic type operations in Clean: the basic type constructors (integers, reals, booleans, characters, strings) are elements of the type system, while the operations on these values are collected in corresponding modules). It is also similar to the approach of most programming languages to define a standard I/O library (Wilson and Clark, 1993). Figure 1.2 gives the interactive program, but now in the world-as-value paradigm.

```
import StdFile

Start :: World -> World
Start world        = world3
where
    world1         = printf "Type a text:\n" world
    (text,world2)  = readline world1
    world3         = printf text world2
```

Figure 1.2 The interactive program now as a world-as-value program.

The advantages of representing real-world objects by values in a functional language are that sequences of operations are defined naturally by data dependency, and that computations that are not data dependent can, in principle, be evaluated in parallel. The importance of the latter fact should be stressed: even if one is not interested in parallelism, data independency is important because it also relieves programmers from having to think of the order of evaluation where this is not necessary.

For the latter reason we have given the world in Clean a composite structure letting it consist of a number of sub environments. Amongst others, the world contains a file system environment, of type `Files`. The file system environment itself contains the individual file environments of type `File`. Environments are independent: operations on one environment do not have an effect on another environment. Figure 1.3 sketches this situation in a diagram. Because `World` is an abstract data type it is easy to change the internal structure of the world without the need to change all programs. This makes the approach flexible with respect to future changes.

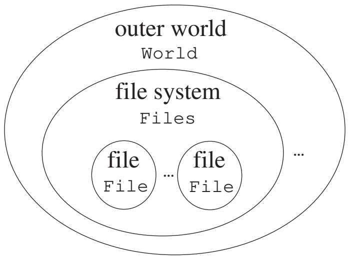

Figure 1.3 The composite structure of the world according to Clean.

In Clean, teletype I/O is based on operations on a special file `stdio` of the file system. Figure 1.4 gives the revised code for our interactive program task. This program code exhibits the typical style of passing values around. Certainly in case of large program texts this may complicate comprehension. However, it is not difficult to adopt another programming style, such as a continuation passing style or a monadic style, on top of this bare style.

```
import StdFile

Start :: World -> World
Start world        = world2
where
    (files,world1) = openfiles world
    (io,files1)    = stdio files
    io1            = fwrites "Type a text:\n" io
    (text,io2)     = freadline io1
    io3            = fwrites text io2
    (_,files2)     = fclose io3 files1
    world2         = closefiles files2 world1
```

Figure 1.4 The simple interactive program in Clean's world-as-value paradigm.

In the world-as-value scheme special, abstract values represent the state of unique real-world resources (such as a file). Resources have unique states. However, values in a functional language do not have this special status. In general, they can be referred to arbitrarily. In Clean terminology, a value that is referred to multiply is a node with a reference count larger than one. Values that uniquely represent resources have reference count one. In Clean, functions that change the encoded state of some resource value can express in their type that they demand a value that has this unique correspondence. This is done by prefixing the argument type with the uniqueness attribute '`*`'. The uniqueness type system of Clean at compile-time determines if the function is actually applied to a unique value (Smetsers *et al.,* 1993; Barendsen and Smetsers, 1993*a-b*; see also Barendsen, 1995).

In Section 2.4 a number of examples are given that illustrate the use of the uniqueness type system. The function in Figure 2.6 does not have a correct type assignment, while the function in Figure 2.7 does. The uniqueness type system also allows function types to be polymorphic with respect to uniqueness attributes. Consider the function `seq`, defined below, which applies a list of functions in head-to-tail order recursively to some value. This function can be applied to both a unique as well as a non-unique second argument, provided the argument functions respect its uniqueness attribute. So the two function types [x→x] x → x, and [*x→*x] *x → *x are both correct types of `seq`. (In Clean, an n-ary function named `f` with arguments of type $\tau_1 \ldots \tau_n$, and result type τ has type $\tau_1\ \tau_2 \ldots \tau_n \to \tau$). Put in other words: the function `seq` is polymorphic with respect to the uniqueness attribute of x. This can be expressed by the single type [u:x→u:x] u:x→u:x in which u is a uniqueness attribute variable. Because in the uniqueness type system type variables are always attributed consistently this can be shorthanded into the type [.x→.x] .x→.x.

```
seq :: [.x -> .x] .x -> .x
seq []     x = x
seq [f:fs] x = seq fs (f x)
```

It should be noted that there are many solutions to preserve the unique correspondence between functional values and resources. One of these solutions was applied in the Clean 0.6 file I/O system. Briefly, in that system, files are represented as abstract values, and file updates succeed only if the file argument at rewriting has reference count one. If not then the program is aborted. The effect is that correct programs have the form of Figure 1.2 and 1.4. It was observed by Sjaak Smetsers that, by a syntactic argument, it could be inferred that programs of this form (in which values are referred to only once) also obey the unique relationship with their resources. This implicit fact could be made explicit in the type of the file operations. The advantage of this approach was that it was possible to implement file I/O without a run-time overhead cost on recording and checking reference counts. This observation led to a fruitful investigation of uniqueness types, and illustrates well how practical considerations and theory can mutually influence each other.

1.5 Structured programming with state transition systems

As stated in Section 1.2, one of our research objectives is to find a declarative, high-level, functional style for programming interactive applications. The major part of this PhD.Thesis is concerned with showing that state transition systems can be used to structure interactive programs in a declarative style in which graphical user interfaces can be integrated smoothly.

The main motivation for this point of view is given by the observation that one can define the behaviour of an interactive program in terms of responses to user activities. The response of a program depends on an internal state. In general, a response changes the state of the program and triggers output actions. This suggests that interactive applications can be defined by specifying the initial state of a program as a value, and the responses as a set of functions that change the state and do output.

Graphical user interfaces integrate smoothly in this scheme because essentially its elements (such as menu items, control buttons, slider bars) determine a context *when* a certain state transition function should be evaluated to act as the response of the program. This means that, if we represent graphical user interface elements as values, then we can associate with them the corresponding state transition functions as functional arguments. We can see these components combined in Figure 1.5, which gives a small example of a simple counter. In this example there are two main graphical user interface elements, namely a menu and a window. The menu, named “File”, consists of one menu item, named “quit”. With this menu item the function named `off` is associated. The intended meaning is that ‘whenever the menu item named “quit” is selected, the program should respond as the function off’. The window, named “Counter”, consists of four controls. Only the button controls, labelled “-”, “+”, and “off”, are associated with a function. Analogously, the intended meaning is that ‘whenever the button is selected, the program should respond as the associated function’.

The program state is a composite value, consisting of a local state and a public state. The initial local state of the program is the integer value `0` that holds the value shown in the text control of the window. The actual value of the public state is immaterial because this program is polymorphic in the type of its public state as we can see by the types of the functions `set` and `off`. Therefore the expression `init` may denote a value of arbitrary type.

The state transitions are computed by the library function `OpenIO`. Given the initial program state it first evaluates the initial actions of the program given as its first argument. In this case the initial actions of the program open the window and the menu. After the initial actions have been evaluated, `OpenIO` polls for events, and in case an event coincides with one of the currently available functions of the graphical user interface elements, it evaluates that function. The function is applied to the current program state and yields a new program state with which `OpenIO` continues. The termination button “off” applies the library function `QuitIO` to its program state causing `OpenIO`, and therefore the program, to terminate.

```
import StdEnv, StdEventIO, StdWindow, StdControl, StdMenu

Start :: *World -> *World
Start world
=   OpenIO [seqPIO [OpenWindow counter,OpenMenu 0 menu]] (0,init) world
where
    menu    = Menu "File" [MenuItem "quit" [MenuFunction off]] []
    counter = DialogWindow "Counter"
                [TextControl   "0  " [ControlId       dispid],
                 ButtonControl "-"   [ControlFunction (set (-1)),
                                      ControlPos      left],
                 ButtonControl "+"   [ControlFunction (set 1)],
                 ButtonControl "off" [ControlFunction off,
                                      ControlPos      left]]
                [WindowId winid]
    left              = (Left,(0,0))
    [winid,dispid:_] = [0..]

    set :: Int (PState Int .p) -> PState Int .p
    set dx ps = seqPIO [SetWindow      winid
                       [SetTextControl dispid (toString new)]]
                       {ps & pLocal=new}
    where
        new = ps.pLocal+dx

    off :: (PState .l .p) -> PState .l .p
    off ps = seqPIO [QuitIO] ps
```

Figure 1.5 A simple Clean graphical user interface program. See Appendix A.2.1 for the functions OpenIO, QuitIO, and seqPIO.

The graphical user interface elements are defined as values of algebraic data types. In Clean, algebraic data types are defined by an optionally parameterized type constructor. The arguments of an algebraic type constructor must be type variables. An algebraic data type has atleast one alternative constructor. Alternative constructors are separated by ‘|’. Alternative constructors can be parameterized either with the type variables of their type constructor, or with other types. Type constructors and data constructors are character sequences starting in uppercase, type variables are character sequences starting in lowercase. Throughout this thesis we adopt the notational convention to present alternative constructors of algebraic data types in **boldface**. An algebraic list type is for instance defined as List x = **Cons** x (List x) **|** **Nil**. The program state is a record structure. In Clean, the expression ‘r.x’ denotes the selection of the field ‘x’ of a record value r. The expression ‘{r & x=y}’ denotes a new record equal to r except that the field ‘x’ has value y.

A list with elements $x_1,\ldots,x_n$ and with tail list xs is denoted by '$[x_1,\ldots,x_n:xs]$'. The '_' symbol denotes an anonymous identifier. The where clause of a function definition contains the local definitions of a function.

It should be stressed that what we are defining is in fact an abstract graphical user interface: in the semantics of state transition systems there is no mention of event handling, resource management, and so on. The meaning of the program must be derivable from the program only, and should not depend on whatever platform it is running.

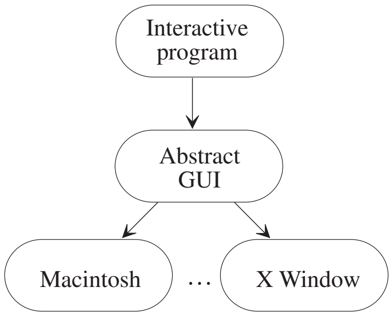

Figure 1.6 The use of an abstract graphical user interface to obtain portable interactive programs.

If the abstraction level of the system is sufficiently high then this can be helpful in achieving portable interactive applications. This is illustrated in Figure 1.6. graphical user interface tools of different platforms are incomparable and incompatible (compare for instance mainstream systems such as Macintosh, OS/2, MS-Windows, and the X Window system). It is important that the design of the I/O system is not biased towards one concrete system, but it must also be sufficiently concrete to do some sensible programming. We have studied the abstractions of the design of the system by defining ports to the Macintosh, OS/2, and X Windows using the Open Look tool kit (Pillich, 1992). Figure 1.7 shows how the example counter of Figure 1.5, when recompiled but not rewritten, looks like on these systems.

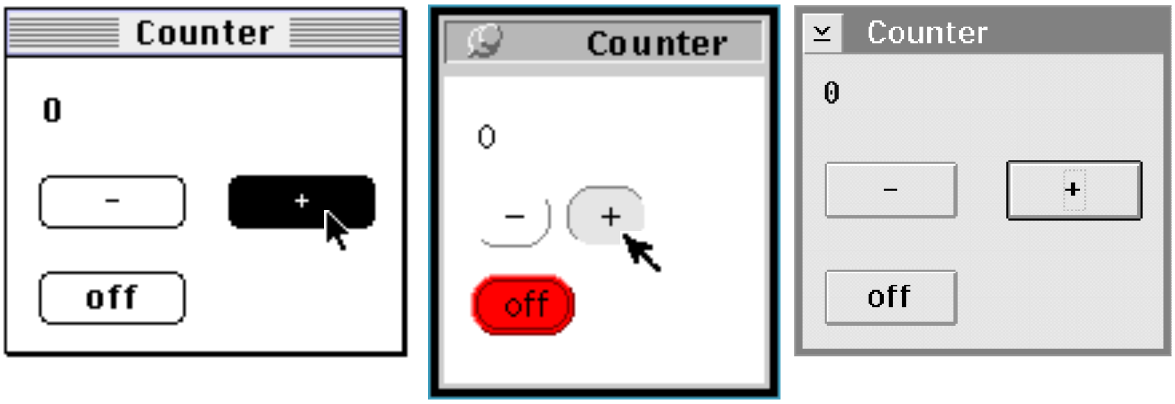

Figure 1.7 The example counter running on a Macintosh, X Windows using the Open Look tool kit, and OS/2 system.

1.6 Interactive functional processes

One of the major design criteria during the research project was to define an I/O system that in principle could be evaluated concurrently. The world-as-value paradigm was introduced with this aim. State transition systems as introduced in the previous section are in essence sequential, with only local possibilities for concurrent evaluation. Our final research effort is to demonstrate that programs can be constructed dynamically of state transition systems, each of which can be evaluated concurrently. In fact what we show is that state transition systems are operational definitions of processes. Therefore we call these processes interactive processes.

In Chapter 3 we identify the primitives to create interactive processes dynamically and inter-process communication primitives. One form of inter-process communication, data sharing, is a logical consequence of using state transition systems. Processes that communicate by data sharing have atomic access to a specific data structure. The other form of inter-process communication is synchronous and asynchronous message passing. The reason to introduce message passing in a concurrent language, which already has a means to communicate, is that inter-process communication by message passing is a data-driven kind of communication. Concurrent implementations of lazy functional languages usually implement process communication by demand-driven evaluation of lazy streams. In demand-driven communication the initiative to communicate lies with the receiver. Because in general the sender may not have a data item to be sent this will block evaluation of the receiver until the data item is available. In data-driven communication the initiative to communicate lies with the sender. If the communication is synchronous, the sender will block until the receiver is ready to accept the message, if the communication is asynchronous, the sender can simply continue. So inter-process communication by message passing contributes to the expressiveness of the system.

Our goal to construct a framework of dynamic processes with inter-process communication in a strongly typed language runs into type assignment problems. In this case these are:

- The functional interpretation of processes that have different types of state requires recursive, polymorphic data structures (such as lists) that can contain elements of different and arbitrary type (such as the list [5, True, 3.14]). We show in Chapter 5 that we can obtain such data structures in a functional language with a polymorphic Milner/Mycroft type system using currying and higher-order functions. We demonstrate that such a solution is less efficient and elegant than by extending the type system with existential types (Mitchell and Plotkin, 1985; 1988).
- The message passing mechanism as offered to the programmer is polymorphic and type-safe, so messages can be values of arbitrary type, and processes can not send or receive messages of the wrong type. However, the functions in the framework that define message passing cannot be typed in a polymorphic Milner/Mycroft type system, even if it has been extended with support for existential types (see Section 3.5). Here the problem what type system can assign a type to the functions in question is still open.

The final step to obtain truly concurrent interactive processes is taken in Chapter 4. We show that interactive processes can share the file system also as a globally known process. The importance of this step is that in the functional, sequential model one level of data dependency is removed. This means that in principle interactive processes can be evaluated concurrently.

1.7 Outline of the thesis

The chapters in this thesis are papers that have either been published or submitted for publication. Further changes of the text are mainly concerned with spelling and grammar, but also some small additions to the original text. To obtain a uniform presentation throughout the document, all examples are in Clean 1.0 syntax, and the abstract device definitions all comply with Appendix A. The chapters in this thesis can be divided into two parts. Part *one* starts to explain in Chapter 2 how interactive programs are constructed, and then extends the model with interleaved interactive processes in Chapter 3 and with concurrent interactive processes in Chapter 4. Part *two* focusses on type system related topics. These concern data abstraction when combined with uniqueness typing in Chapter 5 and the well-typed interpretation of hierarchic data structures with local state, also called object structures, in Chapter 6.

(2) *Single Interactive Functional Programs*. This chapter appeared as Achten and Plasmeijer (1995*a*). Earlier versions of this paper appeared as Achten *et al.* (1993), Achten and Plasmeijer (1993). This chapter gives an extensive discussion of the world-as-value paradigm and the event I/O system. The event I/O system collects the basic ideas of structuring programs and graphical user interfaces by means of state transition systems.

(3) *Interleaved Interactive Functional Processes*. This chapter appeared as Achten and Plasmeijer (1994*b*). An earlier version of this paper appeared as Achten and Plasmeijer (1994*a*). In this chapter we introduce the interleaved event I/O system. This system is a generalisation of the event I/O system presented in Chapter 2. We show how a program can create interactive processes dynamically and study a number of inter-process communication primitives. The methods presented in Chapter 5 are applied to define a well-typed interpretation of interactive processes. Message passing in this system is type-safe and polymorphic. The functional framework that defines message passing however can not be typed within the framework. This problem is still an open problem.

(4) *Concurrent Interactive Functional Processes*. This chapter appeared as Achten and Plasmeijer (1995*b*). In this chapter we define the concurrent event I/O system. We show how to obtain from the interleaved event I/O system of the previous chapter a functional interpretation that can be evaluated concurrently.

(5) *Data Abstraction*. (revised internal report). In this chapter we consider two known data abstraction techniques and combine them with the uniqueness type system. One technique extends the type system with existential types and can be used straightforwardly when uniqueness types are involved. The other technique, relying on a polymorphic Milner/Mycroft type system, requires care when uniqueness types are involved. Finally, we compare the efficiency of the two methods.

(6) *Object Structures*. (Achten and Plasmeijer, 1995*c*). In this chapter we show two forms of compositions of state transition systems that allow the incorporation of polymorphic local state, and their corresponding functional interpretations. Being able to have local state is an important strengthening of the system as it increases the modularity and locality of the system. It is demonstrated that in a polymorphic Milner/Mycroft type system we cannot construct well-typed interpretation functions. The necessary extensions to the system are type classes and constructor classes which have been included in Clean's type system.

(7) *Conclusions*. We give a summary and evaluate the research project and its results.

Appendix A contains the definitions of the I/O system (version 1.0), for reference and to check with the examples. Appendix B contains the semantic definition related with Chapter 4.

1.8 Topics outside the scope of the thesis

Our main research objective was to provide the Clean language with an I/O system that is able to handle files and graphical user interfaces. So we were not only interested in describing and defining the system, but also in an efficient implementation of the system. A major part of the work done during this research project consists of implementation work. The implementation of the I/O system has been done for a major part in Clean itself. Having a good implementation enabled us to test ideas but also could be used to construct real world interactive applications. In 1993 a first distribution release appeared of the I/O system, and the interested reader can find an overview of it in Plasmeijer and van Eekelen (1993). From this version we learned how to improve the design and we have continued working on the system. The reader can get a flavour of the current state-of-the-art by looking at the definitions given in Appendix A.

This PhD.Thesis lacks a full semantic definition of the I/O system. A first attempt in this direction appeared as Achten (1992). In this report an operational semantics was given on a small and simple set of an early version of the Clean event I/O system. Because the operational semantics should be used by programmers it had been worked out on the level of abstract device definitions. This is the conceptual level of programming with the library. The semantics was defined in Clean. In this way the semantics could be type-checked, and was also an executable specification. We have not proceeded maintaining an operational semantics report because we wanted to focus more on pragmatic issues for the purpose of this thesis.

We have written numerous programs, small and big, to test the I/O system, our ideas, and the Clean language itself. The reader might expect to find a chapter on our experience with programming in this system, and some structured way of measuring efficiency and performance of interactive programs. Although efficiency considerations play a role in the implementation of the system, we do not give an expose of it (we briefly discuss this subject in Section 7.7). Furthermore we have not considered a systematic comparison of the efficiency of interactive Clean programs with other interactive systems. However, it has been shown that complex applications can be written based on the I/O system. So far it has been used for writing a multi-window text editor, relational database, spreadsheet, and many games. Experience with programming real-world applications in the Clean system have been reported in de Hoon *et al.* (1995), and Noble and Runciman (1994). The performance of the applications was surprisingly good. As an example, in one project we have also written the programming environment we use on the Macintosh system to develop Clean programs in Clean itself using the I/O system. The original programming environment was written in C. The Clean environment runs a lot faster and particularly smoother than the original C version.