7
Conclusions

This chapter concludes the PhD.Thesis. We give a brief summary of the main results of the chapters 2 upto 6, and then discuss a number of interesting aspects of this research. We highlight the role of the type system and the benefits of having worked in a graph rewriting language. We discuss the universality of the world-as-value paradigm with respect to evaluation strategies, the use of interactive processes as modular building blocks, consider some portability issues, and briefly consider efficiency aspects. Finally, we give some leads to future work.

7.1 Summary

Chapter 2 lays the foundations of the research project, and presents the basic concepts and ideas. First of all, we discuss how in a graph rewriting language one can add special basic values, environments, that are encodings of real-world resources such as files, file systems, screens, and so on. Functions are defined that relate these values with each other. Finally, for each environment type, a number of primitive functions are given that represent the operations available to programmers. Local order of evaluation is obtained by data dependency. The uniqueness type system guarantees that updateable environment values uniquely represent unique resources. In this chapter we introduce two classes of I/O, namely file I/O and graphical user interface I/O. File I/O (and teletype I/O) can be incorporated in the language as a special case of the world-as-value paradigm. Graphical user interfaces are programmed as specifications of responses of a program in the context of some state of the program and abstract specification of the event. The corresponding semantics is that of a state transition system. This semantics can be given straightforwardly by a simple set of functions.

We can regard state transition systems as specifications of processes. An important consequence of this point of view is that we can separate the state of a process from its behaviour. The state of a process is just another, complex, data structure. This opens the possibility to store arbitrary processes in, say, lists. However, here we get into conflict with the polymorphic type system which demands that all elements of a list need to have the same type. In Chapter 5 we show how we can define data structures that contain objects of arbitrary type, but still have the same type. This discussion is based on currying and the use of higher-order functions. As one can imagine this solution is inefficient. It is shown that a more efficient solution is obtained by extending the type system of the language with existential types.

Based on the idea that processes are completely represented by data structures that can be assembled in recursive data structures, Chapter 3 shows how we can extend the functional framework to obtain interactive applications that consist of arbitrary many processes. The framework employs a round-robin scheme to compute the state transitions of the processes. We introduce two classes of inter-process communication, namely data sharing and message passing. Data sharing corresponds roughly with processes that communicate by means of some global data structure. The difference is that access to the shared data structure is atomic. We show how we can define both asynchronous as well as synchronous message passing directly in the model. We do not need to extend the model with a concept of channels, but rather can view messages as abstract events. Analogous to graphical user interface elements, messages are dealt with by a new kind of state transition function, collected in receivers. Message passing is polymorphic and type-safe. Receivers can be created and disposed of dynamically by processes.

The framework given in Chapter 3 gives a model of interactive processes that communicate by the file system, data sharing, and message passing. However, it is essentially a sequential system. In Chapter 4 we show how we can obtain, by a local modification of the model, a sequential system that can be evaluated in parallel. In the model of Chapter 3, the global character of the file system is modelled by data sharing. Data sharing introduces data dependency, which in its turn introduces sequentialization. We show that by introducing one special persistent process which manages the file system, and that is globally known to all processes, we can remove this level of data dependency. The new model is ofcourse still sequential. However, because we do not have data dependency, we can in principle compute the state transitions of the processes in parallel. This is an analogous situation as parallelization of the well-known map function.

The final contribution of Chapter 6 is to show that state transition systems can be extended orthogonally and compositionally with local state. Compositions of state transition systems are in fact obtained straightforwardly, but it is shown that in a polymorphic Milner/Mycroft type system we can not construct well-typed general interpretation functions of these structures. We show that one needs to include type constructor classes in the type system of the language to construct well-typed interpretation functions. The scope of this issue is wider than that related to our specific approach of defining interactive programs. The problem is related with the construction of objects in the object-oriented paradigm.

7.2 The role of the type system

Clean is a strongly typed language. Types are invaluable as they provide a means for partial program correctness that can be determined statically. In the type systems of most functional languages complex data structures can be built easily, using algebraic types and the type constructors for lists, tuples, and functions and given suitable names, using synonym types. Types are a concise way of specifying the interface of functions, stating the types of the arguments and the results.

In Clean the type system has also been used to define functions and data types that influence the way they will be evaluated. These are partially strict data types (Nöcker and Smetsers, 1990; 1993) and strictness annotations for function types. This information is used in the strictness analysis phase of the compiler (Nöcker, 1993; 1994) to derive strictness and generate efficient code (Smetsers *et al*., 1991).

The uniqueness type system has in the same style been added to the type system of Clean. In the uniqueness type system one can annotate uniqueness properties to types. The reference count analysis phase of the compiler derives the proper uniqueness information and checks whether the applications are correct with respect to uniqueness.

Working in a strongly typed system has the advantage that programs that are accepted by the compiler 'can't go wrong'. However, in this thesis we have encountered many examples of programs that did not belong to the class of typeable programs in the Milner/Mycroft type system, but also don't go wrong (see chapters 3, 5, and 6). It is interesting to observe that these typing problems occur in cases where programming flexibility is required, leading to polymorphic data structures that are interpreted by predefined general functions.

7.3 The role of graph rewriting

The research project has been based on a functional language with a term graph rewriting semantics. This has been profitable in many aspects. One obvious benefit was that the uniqueness type system, as a concept, is intuitive and clear when thinking of functions as graph rewrite rules and computational expressions as graphs. Sharing is an explicit notion in graph rewriting and is one of the reasons why the implementation of the I/O system can efficiently avoid copying large graph structures, such as the `IOState` environment or program states, and instead rely on copying references.

Another benefit shows up when considering message passing between interactive processes (Section 3.5). Message passing is polymorphic, so messages can be values of arbitrary type. In a lazy language they can in particular be function applications or infinite data structures. The two forms of message passing that have been introduced, synchronous and asynchronous message passing, do not evaluate the message argument which is sent unevaluated to the receiving process. So the act of passing a message is eager, but the messages themselves are sent lazily. In Clean the notion of *lazy copying* has been introduced to describe communication without duplication of work (van Eekelen *et al.*, 1991; Barendsen and Smetsers, 1992; see also Smetsers, 1993). This notion is meaningful in both a sequential as a parallel context. Therefore it is possible to give a clear semantics to message passing of lazy messages in Clean that holds for both the interleaved system of Chapter 3 and the concurrent system of Chapter 4. It is interesting to contrast this with the discussion in Turner (1990), – pp. 210-212 – were a number of arguments are given to introduce only synchronous, hyper-strict message passing in the KAOS system.

7.4 The world-as-value paradigm

We have seen how values in the world-as-value paradigm represent states of independent real-world resources, and functions represent actions on these resources. These functions are hyper-strict in environment arguments. The uniqueness type system has been applied to specify explicitly which values uniquely represent the unique state of the encoded resource. It should be observed that, although we have focussed in this thesis on lazy languages, the world-as-value paradigm can be applied universally to functional languages with arbitrary reduction strategies. The only proviso is that the reduction strategy obeys the hyper-strict evaluation of environment arguments.

The world-as-value paradigm incorporates I/O in a lazy language. Because of the unique relationship between values and real-world resources an efficient side-effecting implementation can be made. This seems to contradict an assertion made by Hughes (1990, pp. 26) that lazy evaluation and side-effects can not coexist. He argues that because lazy languages do not give the programmer control over the order of evaluation, reasoning about the meaning of a program that relies on side-effects is effectively too complex. He concludes that relying on side-effects and lazy evaluation destroys the modularity of lazy programming techniques. The world-as-value paradigm does not contradict this assertion because it is only the implementation that is side-effecting, triggered by the lazy evaluation of operations on abstract values that represent real-world resources. The only order of evaluation a programmer can sensibly indicate is given by data dependency.

7.5 Processes as building blocks

In this thesis we have considered state transition systems as operational specifications of interactive processes. This point of view is useful because it allows a programmer to split a large programming task into interactive chunks of program code that can lateron be glued together to form the complete program. If this is bound to be succesful then one must be able to reason about individual processes. Their properties should be retained after composition, and furthermore the properties of compositions of processes should be derivable from the processes the composition is constructed.

Single processes are state transition systems. The process state is a composite structure consisting of a local, private state, a public, shared state, and the `IOState` environment needed for graphical user I/O and file I/O (so we discuss processes of the concurrent event I/O system, but file system operations in the interleaved event I/O system follows the discussion on data sharing). It should be observed that access to the private state and graphical user interface elements is always local to interactive processes: it is not possible for processes to directly access the local state or the graphical user interface elements of another process. For these components we can safely assume sequential, atomic behaviour of the process.

In general, when defining an interactive process (program) that uses resources that are available to other programs as well, such as the file system or files, one has to be aware of interference of other processes. In the concurrent event I/O system we can include these processes simply as part of the world. This implies that we can reason about the meaning of a program with respect to its context. The semantics of other solutions for I/O in functional languages rely on an external partner (usually the Operating System) whose behaviour we need to understand to reason about the meaning of a program. In our framework we can cover these cases within the framework itself.

7.6 Portable interactive programs

Portability is an important feature of a programming language that contributes to its acceptance in the programming community. A programming language is portable if the meaning of a program is independent of the actual platform it is running on. For a language with a standard library this means that the libraries have to be present on all systems for which a port is provided while retaining the semantics of the language. When dealing with graphical user interfaces one specific aspect of portability shows up, namely that a program should not only retain its semantics, it should run as if it were a native application on that platform with respect to look-and-feel. So, an interactive program when compiled on a Macintosh platform, should behave as a Macintosh application, while when compiled on a X Window system, it should obey the rules of the local window manager.

In this thesis we have tried to design the graphical user interface library in such a way that it is not biased towards any windowing platform. This is reflected in the fact that graphical user interface elements are programmed as abstract categories of concrete graphical user interface elements such as menus, windows, dialogues, and so on. Other functional approaches for graphical user interface applications, such as FUDGETS (Carlsson and Hallgren, 1993), gadgets (Noble and Runciman, 1995), Haggis (Finne and Peyton Jones, 1995), can be characterized as *widget level* approaches. The latter systems have all been designed for X Window systems, and the question is open if they can be ported to other mainstream graphical user interface systems such as Macintosh, OS/2, and MS-Windows retaining look-and-feel.

7.7 Efficiency

In Section 1.8 we stated that although efficiency was an important design issue when implementing the Clean event I/O system, we have not investigated its efficiency in a structured way. In this section we derive the overhead cost of the Clean event I/O system. We first determine a general, absolute, optimal performance figure, and compare this with the pure overhead cost of the implementation. The two programs that we use for this purpose measure the Macintosh implementation. They have been compiled on a Macintosh Quadra 950, with the following options: stack and index checks on, 1Mb heap, 100Kb stack size and 100Kb extra memory. Garbage collection times have been subtracted from the overall execution time.

We first determine a general, absolute, optimal performance figure. The two basic tasks of the Clean event I/O system are (**a**) to retrieve low-level events from the event stream environment, and (**b**) for each event interpret all interactive processes and evaluate candidate abstract event handlers. The optimal performance can be reached if one assumes that task (**b**) takes zero time. So the optimal performance figure can be determined by measuring a program that only retrieves low-level events. This is the benchmark `minimumcost` (Figure 7.1). Low-level events are retrieved by means of the Macintosh system call `EventsWaitEvent`. The function `loop` repeatedly applies this system call until its counter value reaches some constant value `Bound` (10,000 in this program). The measured performance of `minimumcost` is approximately 1400 system calls per second (tests resulted in an average of 1366). This figure is absolute and optimal in the sense that it gives an upper bound on the number of abstract event handler calls per second.

```
module minimumcost

import StdInt, StdClass, event

Bound :== 10000

Start :: Int
Start = loop 0 EmptyEvents NewToolbox

loop  :: !Int !*EVENTS !Toolbox -> Int
loop  count es tb
|     count>=Bound = count
|     otherwise    = loop (count+1) es1 tb1
where (_,es1,tb1)  = EventsWaitEvent 0 0 es tb
```

Figure 7.1 The optimal benchmark `minimumcost`.

We now determine the pure overhead cost of the Clean event I/O implementation. As stated above, the system is a recursive loop handling tasks (**a**) and (**b**). The pure overhead cost is determined by measuring how many iterations of (**a**) and (**b**) per second the system can handle given *one* interactive process, without evaluating abstract event handlers. In other words, the pure overhead cost assumes that evaluation of abstract event handlers takes zero time. This is achieved by a small change in the central event loop code of the library function `OpenIO`. We replace its termination condition by letting it evaluate `Bound` times (again 10,000) the event loop. Although the only interactive process is empty, the implementation checks the event for all abstract devices. So the pure overhead cost figure is representative in case of non-empty interactive processes. The measured pure overhead cost is approximately 600 abstract event handler calls per second (test average was 596).

The Clean event I/O system is an interpreter that provides programmers with high-level interactive processes as described in Chapter 3. Its pure overhead cost is 43% of the optimal performance figure. However, it is our observation that in general interactive processes are either foreground processes, consisting of windows, menus and occasionally timers, or background processes, consisting of timers and receivers only. Most interactive programs consist of atleast one foreground process. In any case, programmers need to include low-level event handling code for atleast window and menu handling in their programs. So the performance of specialized programs is neither optimal. The burden of writing error prone low-level event handling code is already taken care of correctly in the Clean event I/O system.

7.8 Future work

The work presented in Chapter 2 strongly corresponds with the 0.8 distribution version of the Clean I/O libraries. The other chapters are part of later experimental versions. At this moment, most of the material of Chapter 3 and 5 has been incorporated in the current 1.0 Clean event I/O version. We are considering to merge synchronous message passing (Chapter 3) with remote procedure calls (Chapter 4). Compositional object structures (Chapter 6) need to be incorporated. In parallel to these projects, we have been working on a more orthogonal design of the I/O system. As has been mentioned in Chapter 3 and 4, the functional framework manages concurrent processes, has advanced inter-process communication primitives, and is integrated with file services. This sums up to a small functional operating system. One interesting future direction building on the work presented in this thesis is to study the definition of a functional framework of a concrete distributed operating system.