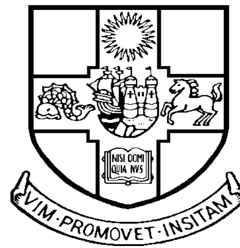
Declarative Programming in Escher

J. W. Lloyd

June 1995 (Revised August 1995) **CSTR-95-013**



University of Bristol

Department of Computer Science

Also issued as ACRC-95:CS-013

Declarative Programming in Escher

J.W. Lloyd

June 1995 CSTR-95-013

Department of Computer Science
University of Bristol
University Walk
Bristol BS8 1TR

© J.W. Lloyd 1995

Preface

Escher[1] is a declarative, general-purpose programming language which integrates the best features of both functional and logic programming languages. It has types and modules, higher-order and meta-programming facilities, and declarative input/output. Escher also has a collection of system modules, providing numerous operations on standard data types such as integers, lists, characters, strings, sets, and programs. The main design aim is to combine in a practical and comprehensive way the best ideas of existing functional and logic languages, such as Gödel, Haskell, and λProlog. Indeed, Escher goes well beyond Gödel in its ability to allow function definitions, its higher-order facilities, its improved handling of sets, and its declarative input/output. Escher also goes well beyond Haskell in its ability to run partly-instantiated predicate calls, a familiar feature of logic programming languages which provides a form of non-determinism, and its more flexible handling of equality. The language also has a clean semantics, its underlying logic being (an extension of) Church's simple theory of types.

This report is divided into two parts. The first part provides a tutorial introduction to Escher. In this part there are many example programs to illustrate the various language features. In particular, these example programs are meant to emphasize the significant practical advantages that come from integrating the best features of existing functional and logic languages. The second part contains a formal definition of the Escher language, including its syntax, semantics, and a description of its system modules. To make the report self-contained, an appendix summarizes the key aspects of the simple theory of types.

In fact, the language definition is not yet complete and its implementation is only at a very early stage. Thus this report describes progress made so far, concentrating largely on the basic computational model and illustrating the main facilities of the language with numerous programs. However, the longer term objective of the research is to design and implement a practical and comprehensive, integrated functional/logic programming language which embodies the best ideas of both fields, including types, modules, higher-order and meta-programming facilities, and non-determinism.

Research on integrating functional and logic programming languages goes back to the 1970's. In 1986, an influential book, edited by Degroot and Lindstrom, appeared which contained some important contributions towards solving this problem. An excellent overview of more recent research is contained in a survey article of Hanus. (See the bibliography.) However, whatever the reasons, there is still no widely-used language which could be regarded as a practical and comprehensive synthesis of the best ideas of functional and logic programming.

What would be the advantages of such a language? To begin with, it would surely have the effect of bringing the fields of functional and logic programming closer together. Currently, there is too much duplication and fragmentation between the two fields. A suitable integrated language cannot fail to better focus the research efforts. For example, it would allow researchers in both

[1] The language is named after the Dutch graphic artist M.C. Escher, a master of endless loops!

fields to co-ordinate their efforts on important research topics such as program analysis, program transformation, parallelization, and programming environments. Furthermore, the existence of an integrated language would greatly facilitate the teaching of programming. It is desirable for students to learn an imperative language and a declarative one. Unfortunately, because of the differences between current functional and logic languages, this usually means learning *two* declarative languages, one a functional language and one a logic language. This seems to me to be undesirable, inefficient, and unnecessarily confusing for students. It would be far preferable to use instead a single exemplar of the family of declarative languages. (These points and some related ones are discussed in greater detail in the first chapter.)

While there are numerous issues of the design of Escher that need to be settled, it is clear that the major unresolved issue is whether Escher be implemented efficiently! It's too early to judge on this, but at least I can see no obvious impediment that would forever prevent an efficient implementation. On the other hand, the design is considerably more ambitious than those of the languages reported in the survey article of Hanus which already have implementation difficulties, so it is clear that an efficient implementation is likely to be a significant challenge. However, having run a fair number of Escher programs, I feel the attractions of this kind of declarative programming language will make worthwhile whatever effort an efficient implementation requires. In any case, I believe the problem of integration is one of the most important and pressing technical challenges facing researchers in declarative programming. Furthermore, a successful integrated declarative language would have enormous practical significance. Thus, I believe, this is a challenge that researchers in both functional and logic programming must take seriously.

Every new programming language owes a significant debt to those which came before. First, Escher builds on the progress made by Gödel, which was the result of a long and fruitful collaboration with Pat Hill. I also learnt much from a detailed study of the Haskell and λProlog languages. Finally, the design greatly benefitted from interesting discussions with many people including Samson Abramsky, Tony Bowers, Chris Dornan, Kevin Hammond, Michael Hanus, Fergus Henderson, Ian Holyer, Dale Miller, Lee Naish, Simon Peyton-Jones, and Phil Wadler. Further comments and suggestions are welcomed.

Contents

Preface i

I TUTORIAL 1

1 Declarative Programming 3
- 1.1 Declarativeness . . . 3
- 1.2 Teaching . . . 6
- 1.3 Semantics . . . 8
- 1.4 Programmer Productivity . . . 9
- 1.5 Meta-Programming . . . 10
- 1.6 Parallelism . . . 11
- 1.7 Discussion . . . 12

2 Elements of Escher 15
- 2.1 Background . . . 15
- 2.2 Basic Facilities . . . 17
- 2.3 Conditional Function . . . 22
- 2.4 Local Definitions . . . 23
- 2.5 Higher-Order Facilities . . . 25

3 Modules 29
- 3.1 Importing and Exporting . . . 29
- 3.2 Module Declarations and Conditions . . . 31
- 3.3 Programs, Goals, and Answers . . . 32

4 System Types 35
- 4.1 Booleans . . . 35
- 4.2 Integers . . . 37
- 4.3 Lists . . . 39
- 4.4 Characters and Strings . . . 42
- 4.5 Sets . . . 43
- 4.6 Programs . . . 47
- 4.7 Input/Output . . . 47

5 Declarative Debugging 51
- 5.1 Introduction . . . 51
- 5.2 Principles of Debugging . . . 52

5.3 Debugging Example 54
5.4 Practicalities of Declarative Debugging 57

6 Example Programs **59**
6.1 Binary Search Trees 59
6.2 Laziness 64
6.3 Map 65
6.4 Relations 66
6.5 Set Processing 69

II DEFINITION **71**

7 Syntax **73**
7.1 Notation 73
7.2 Programs 73

8 Semantics **75**
8.1 Declarative Semantics 75
8.2 Procedural Semantics 75

9 System Modules **77**
9.1 `Booleans` 77
9.2 `Integers` 79
9.3 `Lists` 81
9.4 `Text` 85
9.5 `Sets` 87
9.6 `IO` 89

A Type Theory **91**
A.1 Monomorphic Type Theory 91
A.2 Polymorphic Type Theory 96

B Local Parts of 3 System Modules **101**
B.1 `Booleans` 101
B.2 `Lists` 105
B.3 `Sets` 110

Bibliography **117**

System Index **119**

General Index **121**

Part I

TUTORIAL

Chapter 1

Declarative Programming

Before getting down to the details of Escher, this chapter sets the stage with a discussion of the nature, practical advantages, problems, and potential of declarative programming.

1.1 Declarativeness

Informally, declarative programming involves stating *what* is to be computed, but not necessarily *how* it is to be computed. Equivalently, in the terminology of Kowalski's equation *algorithm* = *logic* + *control*, it involves stating the *logic* of an algorithm, but not necessarily the *control*. This informal definition does indeed capture the intuitive idea of declarative programming, but a more detailed analysis is needed so that the practical advantages of the declarative approach to programming can be explained.

I begin this analysis with what I consider to be the key idea of declarative programming, which is that

- a program is a theory (in some suitable logic), and
- computation is deduction from the theory.

What logics are suitable? The main requirements are that the logic should have a model theory (that is, a declarative semantics), a computational mechanism (that is, a procedural semantics), and a soundness theorem (that is, computed answers should be correct). Thus most of the better-known logics including first order logic and a variety of higher-order logics qualify. For example, (unsorted) first order logic is the logic of (pure!) Prolog, polymorphic many-sorted first order logic is the logic of Gödel [10], and Church's simple theory of types is the logic for λProlog [15] and Escher (and can be considered as the logic for many functional languages, such as Haskell [5], although this is not the conventional view). In the context of the present discussion, the most crucial of these requirements is that the logic should have a model theory because, as I shall explain below, this is the real source of the "declarativeness" in declarative programming.

Thus the logics (and associated computational mechanisms) of existing declarative languages vary greatly. For example, the computational mechanisms of, say, Gödel and Escher are rather different. For Gödel, this mechanism is theorem proving and, for Escher, it is term rewriting. These two computational mechanisms are closely related but are technically different and certainly look very different to a programmer. With a bit of effort, it is possible to provide a uniform framework (in fact, the one used by Escher will do) to directly compare languages such as Gödel, Haskell and Escher. For Gödel, one considers the completed definitions of predicates as being

equations involving boolean expressions and for Haskell one considers statements in a program as being equations involving λ-expressions. Having done this, programs in all three languages are *equational theories* in a higher-order logic and the various languages can be compared by applying the appropriate computational mechanisms to these equations. Furthermore, depending on the language, other properties of the logic such as completeness or confluence would also be required.

This view of declarative programming shows that the concept is wide-ranging. It includes logic programming and functional programming, and intersects significantly with other research areas such as formal methods, program synthesis, theorem proving, and algebraic specification methods, all of which have a strong declarative component. This view also highlights the fact that the current divide between the fields of functional and logic programming is almost entirely historical and sociological rather than technical. Both fields are trying to solve the same problems with techniques that are really very similar. It's time the gap was bridged.

In fact, the close connection between functional and logic programming is emphasized by the terminology used by Alan Robinson. He calls logic programming, *relational* programming, and he calls the combination of functional and relational programming, *logic* programming. This terminology is very apt, since it emphasizes that the dominant symbols of functional programs are functions and the dominant symbols of relational programs are relations (or predicates). With this terminology, we could effectively identify declarative programming with logic programming. It is a pity that terminology such as relational (or, perhaps, predicative) programming wasn't used from beginning of what is now called logic programming because, with hindsight, it is obviously more appropriate. However, in the following, I conform to the standard terminology to avoid confusion.

I return now to the discussion of the general principles of declarative programming. The starting point for the programming process is the particular problem that the programmer is trying to solve. The problem is then formalized as an interpretation (called the *intended* interpretation) of an alphabet in the logic at hand. The intended interpretation specifies the various domains and the meaning of the symbols of the alphabet in these domains. In practice, the intended interpretation is rarely written down precisely, although in principle this should always be possible.

It is taken for granted here, of course, that it *is* possible to capture the intended application by an interpretation in a suitable logic. Not all applications can be (directly) modelled this way and for such applications other formalisms may have to be employed. However, a very large class of applications can be modelled naturally by means of an interpretation. In fact, this class is larger than is sometimes appreciated. For example, it might be thought that such an approach cannot (directly) model situations where a knowledge base is changing over time. Now it is true that the intended interpretation of the knowledge base is changing. However, the knowledge base should properly be regarded as data to various meta-programs, such as query processors or assimilators. Thus the knowledge base can be accessed and changed by these meta-programs. The meta-programs themselves have fixed intended interpretations which fits well with the setting for declarative programming given above. However, as I explain below, there are limits to the applicability of declarative programming, as defined above, and it is important that these limitations be recognized.

Now, based on the intended interpretation, the *logic component* of a program is then written. This logic component is a particular kind of theory which is usually suitably restricted so as to admit an efficient computational mechanism. Typically, in logic programming languages, the logic component of a program is a theory consisting of suitably restricted first order formulas (often completed definitions, in the sense of Clark [13]) as axioms. In functional programming, the logic component of a program can be understood to be a collection of equations in the simple theory of types. It is crucial that the intended interpretation be a model for the logic component of the

program, that is, each axiom in the theory be true in the intended interpretation. This is because an implementation must guarantee that computed answers be correct in all models of the logic component of the program and hence be correct in the intended interpretation. Ultimately, the programmer is interested in *computing the values of expressions in the intended interpretation*. It should be clear now why the model theory is so important for declarative programming – it is the model theory that is needed to support the fundamental concept of an intended interpretation.

Perhaps it is worth remarking that the term "model theory" is not one which is commonly used when functional programmers describe the declarative semantics of functional programs. The traditional functional programming account starts by mapping the constructs of the functional language back into constructs of the λ-calculus. Having done this, the declarative semantics of a program is given by its denotational semantics, that is, a suitable domain is described and the meaning of expressions is given by assigning them meanings in this domain. (An account of this is given in [6], for example.)

What is common to the functional programming approach of denotational semantics and the model-theoretic approach adopted here is that both assign values to expressions. But there are substantial differences. For example, the (standard) denotational approach ignores types, while I much prefer the view that programs should be regarded as *typed* theories from the beginning. Furthermore, the denotational approach is greatly concerned with expressions whose value is undefined, while the model-theoretic approach assumes functions are everywhere defined. In the conventional logic programming approach where almost everything is modelled by predicates, the undefinedness problem can be finessed by making predicates false on the appropriate arguments. However, in an integrated language, such as Escher, where function (in contrast to predicate) definitions pervade, this undefinedness problem cannot easily be avoided. In any case, a detailed study of the connections between denotational semantics and the logical model-theoretic approach to the declarative semantics would not only provide a link between the functional programming and logic programming views of semantics, but would also seem to be essential in providing a completely satisfactory semantics for an integrated language. In the meantime, for Escher, I have adopted the standard logical view of declarative semantics. A consequence of this is that programmers are expected to use Henkin interpretations (usually called general models or Henkin models) to formalize intended interpretations. Henkin interpretations have the nice property that they generalize first order interpretations in a natural way. Thus I expect that they should be understandable by ordinary programmers, although I have no real evidence to support this!

The programmer, having written a correct logic component of a program, may now need to give the *control component* of the program, which is concerned with control of the computational mechanism for the logic. Whether the control component is needed, and to what extent, depends on the particular situation. For example, querying a deductive database with first order logic, which is a simple form of declarative programming, normally wouldn't require the user to give control information since deductive database systems typically have sophisticated query processors which are able to do find efficient ways of answering queries without user intervention. On the other hand, the writing of a large-scale Prolog program will usually require substantial (implicit and explicit) control information to be specified by the programmer. Generally speaking, apart from a few simple tasks, limitations of current software systems require programmers to specify at least part of the control.

Now declarative programming can be understood in two main senses. In the weak sense, declarative programming means that programs are theories, but that a programmer may have to supply control information to produce an efficient program. Declarative programming, in the strong sense, means that programs are theories and all control information is supplied automatically by the sys-

tem. In other words, for declarative programming in the strong sense, the programmer only has to provide a theory (or, perhaps, an intended interpretation from which the theory can be obtained by the system). This is to some extent an ideal which is probably not attainable (nor, in some cases, totally desirable), but it does at least provide a challenging and appropriate target. Typical modern functional languages, such as Haskell, are rather close to providing strong declarative programming, since programmers rarely have to be aware of control issues. On the other hand, typical modern logic programming languages, such as Gödel, provide declarative programming only in the weak sense. The difference here is almost entirely due to the complications caused by the (explicit) non-determinism provided by logic languages, but not by functional languages.

The issue of how much of the control component of a program a programmer needs to provide is, of course, crucially important in practice. However, systems which can be used both as weak and, on occasions, strong declarative programming systems are feasible now and, indeed, may very well survive long into the future even after the problems of providing strong declarative programming in general have been overcome. The point here is that, for many applications, it really doesn't matter if the program is not as efficient as it could be and so strong declarative programming, even in its currently very restricted form, is sufficient. However, for other applications, the programmer may supply control information, either because the system is not clever enough to work it out for itself or because the programmer has a particular algorithm in mind and wants to force the system to employ this algorithm. Generally speaking, even strong declarative programming systems should provide a way for programmers to ensure that desired upper bounds on the space and/or time complexity of programs are not exceeded. There is no real conflict here – the system can either be left to work out the control for itself or else control facilities (and, very likely, specific information about the particular implementation being used) should be provided so that a programmer can ensure the desired space and/or time requirements are met.

With these preliminaries out of the way, I now discuss the practical advantages of declarative programming under five headings: (a) teaching, (b) semantics, (c) programmer productivity, (d) meta-programming, and (e) parallelism.

1.2 Teaching

Typical Computer Science programming courses involve teaching both imperative and declarative languages. On the imperative side, Modula-2 is a common, and excellent, choice. On the declarative side, Haskell is a typical functional language used and Prolog is the usual choice of logic language. Haskell is an excellent language for teaching as, to a very large extent, students only have to concern themselves with the logic component of a program.

In the last couple of years, there has been some discussion in the logic programming community about the use of Prolog as a teaching language and, in particular, as a first language. A remarkable characteristic of this debate is that, while most logic programmers argue that Prolog should be taught *somewhere* in the undergraduate curriculum, almost nobody is arguing that it should be the *first* teaching language! Usually, the reason given for avoiding Prolog as the first language is that its non-logical features make it too complicated for beginning programmers. I think this is true, but I would also add as equally serious flaws its lack of a type system and a module system. This situation is surely a serious indictment of the field of logic programming. In spite of its claims of declarativeness, until recently at least, there hasn't been a single logic programming language sufficiently declarative (and hence simple) to be used successfully as a first teaching language.

I think that the arrival of Gödel could change this situation. Gödel fits much better than Prolog into the undergraduate and graduate curricula since it has a type system and a module

system similar to other commonly used teaching languages such as Haskell and Modula-2. Also most of the problematical non-logical predicates of Prolog simply aren't present in Gödel (they are replaced by declarative counterparts) and so the cause of much confusion and difficulty is avoided. This case has been argued in greater detail elsewhere [10] and I refer the reader to the discussion there. My own experience with teaching Gödel to masters students at Bristol has been very encouraging. For example, a common remark by students at the end of the course is that they find the Gödel type system so helpful that they really don't want to go back to using Prolog! In any case, I would strongly encourage teachers of programming courses to try for themselves the experiment of using Gödel instead of Prolog.

So let us make the assumption that we have a suitable declarative language available and let us see what advantages such a declarative approach brings. The key aspect of my approach is that there should be a natural progression in a programming course (or succession of such courses) from, at first, pure specification, through more detailed consideration of what the system does when running a program (that is, the control aspect), finally finishing up with a detailed study of space and time complexity of various algorithms and the relevant aspects of implementations. It is very advantageous to be able to consider just the logic component at the very beginning. This is because, for many applications, the programmer's only interest is in having the computer carry out some task and the fewer details the programmer has to put into a program, the quicker and more efficiently the program can be written. I am making the assumption here, as is appropriate in many circumstances, that squeezing the last ounce of efficiency out of a program is not necessary and that the most costly component of the computerization process is the programmer's time. Naturally, this approach of concentrating at first only on the logic component of a program requires a declarative language and clearly won't work at all for an imperative language such as Modula-2. This, then, is the major reason I advocate starting first with a declarative language: the *primary* task of *all* programming is the statement of the logic of the task to be solved by the computer and the use of a declarative language makes it possible for student programmers to concentrate at first solely on this task.

By way of illustration, I outline briefly an approach to teaching *beginning* programmers using a declarative language. First, as the concern is with teaching programming to beginners, advantage can be taken of the fact that the applications considered can be carefully controlled. For example, list processing and querying databases make suitable starting points. Then, as for any programming task, the application is modelled by an intended interpretation in the logic underlying the programming language. Beginning programmers should be encouraged to write down the intended interpretation, even if only informally. Given that only simple applications are being considered, this is a feasible, and very instructive, task for the student to carry out.

Next the logic component of the program is written. With carefully chosen applications and appropriate programming language support, programmers can largely ignore control issues in the beginning. As usual, a key requirement is that the intended interpretation should be a model for the program and this should be checked, informally at least, by students. Once again, for simple applications, this is certainly feasible.

Finally, the program is run on various goals to check that everything is working correctly. Typically, it won't be and some debugging will be required. At this point, advantage can again be taken of the fact that the language is declarative by exploiting a debugging technique known as declarative debugging [13], which was introduced and called algorithmic debugging by Shapiro. The main idea is very simple: to detect the cause of missing/wrong answers, it is sufficient for the programmer to know the intended interpretation of the program. (Declarative debugging for Escher is discussed in detail in Chapter 5.) Since I have taken this as a key assumption of this

entire approach to declarative programming, this isn't an unreasonable requirement. Essentially, the programmer presents the debugger with a symptom of a bug and the debugger asks a series of questions about the intended interpretation of the program, finally displaying the bug in the program to the programmer. Note that this approach doesn't handle other kinds of bugs such as infinite loops, floundering, or deadlock, which are procedural in nature and hence must be handled by other methods. Fortunately, as I explain below, for beginning programmers this turns out not to be too much of a limitation.

What programming language support do we need to make all this work? First, the language must be declarative in as strong a sense as possible. Second, types and modules must be supported. These are not declarative features but, as is argued in [10], no modern language can be considered credible without them. Third, the programming system must have considerable autonomous control of the search. This is crucial if the student programmers are to be shielded from having to be concerned about control in the beginning. In its most general form, providing autonomous control to avoid infinite loops and such like is very difficult (in fact, undecidable). However, in the context of teaching beginning programmers, much can be done. For example, breadth-first search of the search tree (in the case of logic programming) can ameliorate the difficulty. This may not be very efficient, but for small programs is unlikely to be prohibitively expensive. Furthermore, simple control preprocessors, such as provided by Naish for the NU-Prolog system [19], can autonomously generate sufficient control to avoid many problems for a wide range of programs. Finally, as I explained above, declarative debugging must be supported.

I believe the declarative approach to teaching programming outlined above has considerable merit and illustrates an important practical advantage of declarative programming.

1.3 Semantics

One problem which has plagued Computer Science over the years, and looks unlikely to be solved in the near future, is the gap between theory and practice. Nowhere is this gap more evident than with programming languages. Typical widely-used programming languages, such as Fortran, C, and Ada, are a nightmare for theoreticians. The semantics of such languages is extremely messy which means that, in practice, it is very difficult to reason, informally or formally, about programs in such languages. Nor is the problem confined to imperative languages – practical Prolog programs are generally only marginally more analyzable than C programs! This gap between the undeniably elegant theory of logic programming and the practice of typical large-scale Prolog programming is just too great and too embarrassing to ignore any longer.

This issue of designing programming languages for which programs have simple semantics is, quite simply, crucial. Until practical programming languages with simple semantics get into widespread use, programming will inevitably be a time-consuming and error-prone task. Having a simple semantics is not simply something nice for theoreticians. It is the key to many techniques, such as program analysis, program optimization, program synthesis, verification, and systematic program construction, which will eventually revolutionize the programming process.

This is another area where declarative programming can make an important practical contribution. Modern functional and logic programming languages, such as Haskell and Gödel, really do have simple semantics compared with the majority of widely-used languages. Of the two, Haskell is probably the cleanest in this regard, but the semantic difficulties that Gödel has are related to its non-deterministic nature which provides facilities and expressive power not possessed by Haskell. In any case, whatever the semantic difficulties of, say, Haskell or Gödel, there are nothing compared to the semantic problems of Fortran, C, or even Prolog.

The complicated semantics of most widely-used programming languages partly manifests itself in current programming practice, which is very far from being ideal. Typically, programmers program at a low level (I regard C and Prolog, say, as low-level languages), they start from scratch for each application, and they have no tools to analyze, transform, or reason about their programs. The inevitable consequence of this is that programming takes much longer than it should and hence is unnecessarily expensive. We must move the programming process to a higher plane where programmers typically can employ substantial pieces of already written code, where concern about low-level implementation issues can be largely avoided, where there are tools available to effectively analyze and optimize programs, and where tools can allow a programmer to effectively reason about the correctness or otherwise of their programs.

All of these facilities are highly desirable and I see no way at all of achieving them with any of the current widely-used languages. The only class of languages which I see having any chance of providing these facilities is the class of declarative languages. If a program really is a theory in some logic, then there is much more chance of being able to analyze the program, transform the program, and so on. There is good evidence to support this claim from the logic programming community. For example, there is a considerable body of work on analysis and transformation of *pure* Prolog which really does work in practice and this work is directly applicable to the entire Gödel language (except for input/output and pruning) and the entire Escher language. I don't mean to imply by this that carrying out such tasks is easy for a declarative language, only that for a declarative language many irrelevant difficulties (for example, assignment in C or assert/retract in Prolog[1]) are swept away and the real difficulties, which genuinely require attention, are exposed. In other words, with a declarative language, the problems are still difficult, but at least time isn't wasted trying to solve problems which shouldn't be there in the first place!

1.4 Programmer Productivity

Some computer scientists spend much of their time trying to discover efficient algorithms. Clearly, this effort is important as the difference for an application of, say, an $O(n \, log \, n)$ versus an $O(n^2)$ algorithm may mean the difference between success and failure. This kind of argument has often been used to justify the low-level programming which typically takes place to implement efficient algorithms. However, there is another cost of programming which can easily be ignored by computer scientists and that is the cost of programmer time and the cost of maintaining and upgrading existing code. Often having the most efficient algorithm isn't so important; often a programmer would be very happy with a programming system which only required the problem be specified in some way and the system itself find a reasonably efficient algorithm. Nor should we ignore the fact that we want to make programming accessible to ordinary people, not just those with a computer science degree. Ordinary people generally aren't interested (and rightly so) in low-level programming details – they just want to express the problem in some reasonably congenial way and let the system get on with solving the problem.

I believe declarative programming has a big contribution to make in improving programmer productivity and making programming available to a wider range of people. Certainly, as long as we continue to program with imperative languages, we will make little progress in this regard. Since, in an imperative language, the logic and control are mixed up together, programmers have no choice but to be concerned about a lot of low-level detail. This adversely affects programmer productivity and precludes most ordinary people from becoming programmers. But once logic

[1]Assignment may very well be present at a low level in an implementation, but it has no place as a language construct because its semantics is too complicated; assert/retract was simply a mistake from the beginning.

and control are split, as they are with declarative languages, the opportunity arises of taking the responsibility for producing the control component away from the programmer. Current declarative languages generally are still only weakly declarative and so programmers still have to specify at least part of the control component. But this situation is improving rapidly are we can look forward to practical languages which are reasonably close to being strongly declarative in the near future.

Having to deal only (or mostly) with the logic component simplifies many things for the programmer. First, (the logic component of) a declarative program is generally easier to write and to understand than a corresponding imperative program. Second, a declarative program is also easier to reason about and to transform, as much current research in functional and logic programming shows. Finally, it sets us on the right road to the ultimate goal of synthesizing efficient programs from specifications.

1.5 Meta-Programming

The essential characteristic of meta-programming is that a meta-program is a program which uses another program (the object program) as data. Meta-programming techniques underlie many of the applications of logic programming. For example, knowledge base systems consist of a number of knowledge bases (the object programs), which are manipulated by interpreters and assimilators (the meta-programs). Other important kinds of software, such as debuggers, compilers, and program transformers, are meta-programs. Thus meta-programming includes a large and important class of programming tasks.

It is rather surprising then that programming languages have traditionally provided very little support for meta-programming. Lisp made much of the fact that data and programs were the same, but it turned out that this didn't really provide much assistance for large-scale meta-programming. Modern functional languages appear to me to effectively ignore this whole class of applications since they provide little direct support. Only logic programming languages have taken meta-programming seriously, but unfortunately virtually all follow the Prolog approach to meta-programming which is seriously flawed. (The argument in support of this claim is given in [10]).

In fact, as the Gödel language shows, providing full-scale meta-programming facilities is a substantial task, both in design and implementation. The key idea is to introduce an abstract data type for the type of a term representing an object program, and then to provide a large number of useful operations on this type. The relevant Gödel system modules, `Syntax` and `Programs`, provide such facilities and contain over 150 predicates. The term representing an object program is obtained from a rather complex ground representation [10], the complexity of which is almost completely hidden from the programmer by the abstractness of the data type. When one sees how much Gödel provides in this regard, it is clear how weak are the meta-programming facilities of functional and imperative languages, all of which should be following the same basic idea as Gödel, as indeed Escher does.

One key property of the Gödel approach to meta-programming is that it is declarative, in contrast to the approach of Prolog. This declarativeness can be exploited to provide significant practical advantages. For example, the declarative approach to meta-programming makes possible advanced software engineering tools such as compiler-generators. To obtain a compiler-generator, one must have an effective self-applicable partial evaluator. The partial evaluator is then partially evaluated with respect to itself as input to produce a compiler-generator. (This exploits the third Futamura projection.) The key to self-applicability is declarativeness of the partial evaluator. If the partial evaluator is sufficiently declarative, then effective self-application is possible, as the SAGE partial evaluator [7] shows. If the partial evaluator is not sufficiently declarative, then effective

self-application is impossible, as much experience with Prolog has shown.

Now why should a programmer be interested in having at hand a partial evaluator such as SAGE and the compiler-generator it provides? First, the partial evaluator is a tool that is essential in carrying forward the move towards higher-level programming. Writing declarative programs inevitably introduces inefficiencies which have to be transformed or compiled away. Partial evaluation is a technique which has proved to be successful in this regard. By way of illustration of this, the use of abstract data types (which is not purely a declarative concept, but nonetheless an important way raising the level of programming) in a logic programming language restricts the possibilities for clause indexing. However, partial evaluation can easily push structures back into the heads of clauses so that the crucial ability to index is regained. In general terms, partial evaluation is a key tool which allows programmers to program declaratively and at a high level, and yet still retain much of the efficiency of low-level imperative programming.

But what about a compiler-generator? Strictly speaking, a compiler-generator is redundant since everything it can achieve can also be achieved with the partial evaluator from which it was derived. However, there is a significant practical advantage of having a compiler-generator which is that it can greatly reduce program development time. For example, suppose a programmer wants to write an interpreter for some specialized proof procedure for a logic programming language. The interpreter is, of course, a meta-program and it can be given as input to a compiler-generator. The result is a specialized version of the partial evaluator which is, in effect, a compiler corresponding to this interpreter. When an object program comes along, it can be given as input to this compiler and the result is a specialized version of the original interpreter, where the specialization is with respect to the object program. Now the final result can be achieved by the partial evaluator alone – simply partially evaluate the interpreter with respect to the object program. But this requires partially evaluating the interpreter over and over again for each object program. By using the compiler-generator, we specialize what we can of the interpreter just once and then complete in an incremental way the specialization by giving the resulting compiler the object program as input. For an interpreter which is going to be run on many object programs, the approach using the compiler-generator can save a considerable amount of time compared with the approach using the partial evaluator directly. Now recall, what was the key to all of this being possible? It was that the partial evaluator be (sufficiently) declarative!

I believe we have hardly begun to scratch the surface of what will be possible through declarative meta-programming. I hope researchers producing analysis and transformation tools will take the ideas of declarative meta-programming more seriously, and that designers of declarative languages will make sure that new languages have adequate facilities for the important class of meta-programming applications.

1.6 Parallelism

Parallelism is currently an area of intense research activity in Computer Science. This is as it should be – many practical problems require huge amounts of computing for their solution and parallelism provides an obvious way of harnessing the computing power required. While many difficult problems associated with parallel computer architectures remain to be solved, I concentrate here on the software problems associated with programming parallel computers and the possible contribution that declarative programming might make towards solving these problems.

It is well known that parallel computers are hard to program. Widely-available programming languages which are used on parallel machines are usually ill-suited to the task. For example, older languages such as Fortran have required retro-fitting of parallel constructs to enable them to be used

effectively on parallel computers. Furthermore, on top of all the other difficulties of programming, on a parallel computer a programmer is also likely to have to cope with deadlock, load balancing, and so on. One, more modern, class of programming languages that can be used on parallel computers are the concurrent languages – those which have explicit facilities in the languages to express concurrency, communication, and synchronization. These clearly have much potential, but I will concentrate instead of the class of declarative languages for which the exploitation of parallelism is implicit. In other words, whether a declarative program is being run on a sequential or parallel computer is transparent to the programmer. The only discernible difference should be that programs run much faster on a parallel machine!

Declarative languages are well-suited for programming parallel computers. This is partly because there is generally much implicit parallelism in declarative programs. For example, a functional program can be run in parallel by applying several reductions at the same time and a logic program can be run in parallel by exploiting (implicit) And-parallelism, Or-parallelism, or even both together. The declarativeness is crucial here as the more declarative the programming language, the more implicit parallelism there is likely to be. This is illustrated by a considerable amount of research on parallelizing Prolog in the logic programming community, which has discovered that the non-logical features of Prolog, such as `var`, `nonvar`, `assert`, and `retract`, greatly inhibit the possible exploitation of parallelism. The lesson here is clear: the more declarative we can make a programming language, the greater will be the amount of implicit parallelism that can be exploited. There is another aspect to the problems caused by the non-logical features of Prolog – not only do they restrict the amount of parallelism that can be exploited, but their parallel implementation takes an inordinate amount of effort (especially if one wants to preserve Prolog's sequential semantics). Once again the lesson is clear: the more declarative we can make a programming language, the easier will be its parallel implementation.

Implicit exploitation of parallelism in a declarative program is very convenient for the programmer who, consequently, has no further difficulties beyond those which are present in programming a sequential computer. But the system itself must now be clever enough to find a way of running the program in the most efficient manner. This is a very hard task in general and the subject of much current research, but the results so far are rather encouraging and suggest that it will indeed be possible in the near future for programmers to routinely run declarative programs on a parallel computer in a more or less transparent way.

I conclude this section with a remark on the problem of debugging programs which are to run on a parallel computer. Clearly, if transparency is to be maintained, it must be possible for programmers to be able to debug their programs without any particular knowledge of the way the system ran them. For the large class of bugs consisting of missing/wrong answers, declarative debugging comes to the rescue since it makes no difference to this debugging method whether programs are run sequentially or in parallel. All the programmer needs to know in either case is the intended interpretation of the program.

1.7 Discussion

Having discussed various specific aspects of the practical advantages of declarative programming, I now turn to some general remarks.

The first is that it is hard to escape the conclusion that many researchers in logic programming, while claiming to be using declarative programming, do not actually take declarativeness all that seriously. There is good evidence for this claim. For example, in spite of the fact that it has been clear for many years that Prolog is not really credible as a declarative language, it is still

by far the dominant logic programming language. The *core* of Prolog (so-called “pure” Prolog) is declarative, but large-scale practical Prolog programs typically use many non-logical facilities and these features destroy the declarativeness of such programs in a rather dramatic way. Furthermore, while many extensions and variations of Prolog have been introduced and studied by the logic programming community over the last 15 years, including concurrent languages, constraint languages, and higher-order languages, these languages have been essentially built on top of Prolog and therefore have inherit many of Prolog’s semantic problems. For example, most of these languages use Prolog’s approach to meta-programming. It seems to me that if logic programmers generally had taken declarativeness seriously, this situation would have been rectified many years ago because as the Gödel language shows the argument that the non-logical features are needed to make logic programming languages practical is simply false. What distinguishes logic programming most from many other approaches to computing and what promises to make it succeed where many other approaches have failed is its declarativeness. Logic programming has so far not succeeded in making the sort of impact in the world of computing that many people, including myself, expected. There are various reasons for this, but I believe very high on the list is the field’s failure to capitalize on its most important asset – its declarative nature.

Perhaps, amidst all the euphoria of the advantages of declarative programming, it would be wise to add some remarks on the limitations of this approach. The forms of declarative programming based on standard logics, such as first order logic or higher-order logic, do not cope at all well with the temporal aspects of many applications. The reason is that the model theory of such logics is essentially “static”. For applications in which there is much interaction with users or processes of various kinds (for example, industrial processes), such declarative languages do not cope so well and often programmers have to resort to rather *ad hoc* techniques to achieve the desired result. For example, in Prolog, a programmer can take advantage of the fact that subgoals are solved in a left to right order. (Actually, most imperative approaches have exactly the same kinds of difficulties, but I don’t think this absolves the declarative programming community from finding good solutions.) We need new “declarative” formalisms which can cope better with the temporal aspects of applications. Obviously, the currently existing temporal logics are a good starting point, but it seems we are still far from having a formalism which generalizes currently existing declarative ones by including temporal facilities in an elegant way. Much more attention needs to be paid to this important aspect of the applications of declarative programming.

I mentioned earlier that the fields of functional and logic programming should be combined. If one forgets for a moment the history of how these fields came into existence, it seems very curious that they are so separated. Apart from periodic bursts of interest in producing an integrated functional and logic programming language (mainly, it seems, by logic programmers), the two fields and the researchers in the fields rarely interact with one another. And yet both fields are trying to solve the same problems, both have the same declarative approach, and the differences between recent languages in each of the fields are not great. Ten years ago, David Turner wrote [20]:

> It would be very desirable if we could find some more general system of assertional programming, of which both functional and logic programming in their present forms could be exhibited as special cases.

This challenge remains one of the most important unsolved research problems in declarative programming today. The Escher language is one response to this challenge.

Chapter 2

Elements of Escher

In this chapter, I introduce the basic facilities of the Escher language and give some illustrative programs.

2.1 Background

Escher is a declarative, general-purpose programming language which integrates the best features of both functional and logic programming languages. It has types and modules, higher-order and meta-programming facilities, and declarative input/output. Escher also has a collection of system modules, providing numerous operations on standard data types such as integers, lists, characters, strings, sets, and programs. The main design aim is to combine in a practical and comprehensive way the best ideas of existing functional and logic languages, such as Gödel, Haskell, and λProlog. Indeed, Escher goes well beyond Gödel in its ability to allow function definitions, its higher-order facilities, its improved handling of sets, and its declarative input/output. Escher also goes well beyond Haskell in its ability to run partly-instantiated predicate calls, a familiar feature of logic programming languages which provides a form of non-determinism, and its more flexible handling of equality. This section contains some background on the various design issues for Escher.

The first question to ask when designing a new declarative language is: what logic should be used? There are plenty of possibilities, but attention can at least be restricted to higher-order logics. Of these, one seems to me to stand out. This is Church's simple theory of types [2], a sublogic of which has already been used successfully in λProlog. In the following, I shall refer to Church's logic as *type theory*. There are also intuitionistic versions of this logic [11] which are particularly interesting. The intuitionistic versions can be interpreted in toposes, instead of sets as for (classical) type theory, and this seems to me to open up an interesting class of applications. However, I haven't explored this in any detail yet and so, in the following, I confine attention to a modest extension of (classical) type theory *á la* Church.

There are several accessible accounts of type theory. For a start, one can read Church's original account [2], a more comprehensive account of higher-order logic in [1], a more recent account, including a discussion of higher-order unification, in [22], or useful summaries in the many papers ([14] and [16], for example) of Miller, Nadathur, and their colleagues, on λProlog. Also, to keep the report self-contained, the main concepts of (a suitable extension of) type theory are presented in Appendix A.

One of the early design decisions which had to be made was whether statements in a program should be equations, as in the functional programming style, or implicational formulas, as in the logic programming style. To some extent this is just a matter of taste, of course. However, there

is a fair bit at stake here since the respective fields have built up distinctive and contrasting programming styles over the decades. In fact, I resolved this issue very quickly in favour of the equational style. Having written numerous Escher programs in this equational style, after having earlier written a lot of Prolog and Gödel code, it became clear that equations are the key to integration as they promote a clear and natural statement of the logic component of a program. The clinching argument is that equations are just behind the scenes in a logic program, anyway. The reason is that, typically, the logic component of a logic program consists of the completed definitions of predicates. Completed definitions are formulas with a biconditional connective at the top level and the biconditional connective is nothing other than equality between boolean expressions! In any case, readers not convinced by these arguments should give the Escher style a solid try before making a judgement.

A distinctive feature of many logic programming languages is their provision of (don't know) non-determinism. For some important application areas such as knowledge base systems, this non-determinism allows these languages to model the applications in a very direct and natural way. Of course, a functional language can also be used but, typically, modelling such applications with a functional language involves a certain amount of encoding (collecting successive answers in a list, and so on), which I wanted to avoid. It turns out that there is an elegant solution to the problem of providing the non-determinism of logic programming in an integrated language and yet keeping the single computational path of functional programming. This involves capturing the non-determinism by explicit disjunctions in the bodies of predicate definitions. Note that a consequence of this approach is that the default behaviour for the Escher system is to return all solutions together.

This approach has a number of advantages. For a start, in concert with the Escher mode system, it gives programmers fine control over the amount of non-determinism that appears in programs and, in particular, helps to reduce the amount of gratuitous non-determinism in programs to the absolute minimum. The key idea is to exploit all information possible about function calls so that definitions may be written as deterministically as possible. The mode system has been designed specifically to facilitate this. In fact, this approach has been so successful that I have managed so far to avoid introducing a pruning operator into the language at all. Since pruning operators are problematic both theoretically and practically, this is a big advantage. Certainly, the bar commit of the Gödel language and concurrent logic programming languages won't be needed. But it's still an open issue whether the one-solution operator will be needed. Note also that the single computation path of Escher computations, in contrast to the search trees of most logic programming languages, provides a simpler computational model for programmers to understand.

Another advantage of the Escher way of handling non-determinism is that it provides much greater flexibility for implementations. A non-deterministic computation *could* be implemented by the traditional depth-first, backtracking search procedure of logic programming languages. But it could also be implemented by a breadth-first procedure or some combination of both. The historical advantage of the depth-first, backtracking search is that it has low memory requirements. However, in these days of ever-improving workstations having larger and larger memories, such implementation considerations carry less weight.

One particularly interesting observation concerning the Escher programming style which became apparent at an early stage was that typical applications naturally require many more (non-predicate) functions than predicates. I believe this provides an exceptionally important motivation for integration. The point is that, when programming with typical logic programming languages, programmers naturally model applications using predicates because only predicate definitions are allowed by such languages. However, when programming the same application with an integrated

language such as Escher, it becomes obvious that many of the predicates are rather unnatural and would be better off being replaced by (non-predicate) functions. There is a similar effect for functional languages. Since partly-instantiated predicate calls are not normally allowed in functional languages, the non-determinism required in some applications has to be captured by various programming tricks which can obscure what is really going on. Thus Escher and similar integrated languages, having both predicate and non-predicate function definitions and allowing partly-instantiated function calls, provide greater expressive power than conventional functional or logic languages because exactly the right kinds of functions can be employed to accurately model the application.

2.2 Basic Facilities

In this section, the most basic features of Escher are outlined. First, some notation needs to be established. The table below shows the correspondence between various symbols and expressions of type theory (as given in Appendix A) in the left column and their equivalent in the notation of Escher in the right column.

$\mathbf{1}$	`One`
o	`Boolean`
$\neg$	`~`
$\wedge$	`&`
$\vee$	`\/`
$\rightarrow$	`->`
$\leftarrow$	`<-`
$\lambda x.E$	`LAMBDA [x] E`
$\exists x.E$	`SOME [x] E`
$\forall x.E$	`ALL [x] E`
$\{x : E\}$	`{x : E}`
$\in$	`IN`

With this notation established, I start with a simple Escher program to illustrate the basic concepts of the language. For this example, I will carry out the design and coding phases of the software engineering cycle in some detail by first giving the intended interpretation of the application and then writing down the program.

The application is concerned with some simple list processing. There are two basic types, `Person`, the type of people, and `Day`, the type of days of the week. In addition, lists of items of such types will be needed. The appropriate constructors are declared as follows.

```
CONSTRUCT Day/0, Person/0, List/1.
```

The `CONSTRUCT` declaration simply declares `Day` and `Person` to be constructors of arity 0 and `List` to be a constructor of arity 1. (In addition, the constructors `One` and `Boolean` of arity 0 are provided automatically by the system via the system module `Booleans` which is discussed in Chapter 4.) Thus, for this application, typical types are `Boolean`, `Day`, `List(Day)`, `List(List(Person))`, and `(List(List(a) * List(a)) -> Day) -> Boolean`, where `a` is a parameter. In the intended interpretation for this application, the domain corresponding to the type `List(Day)`, for example, is the set of all lists of days of the week.

The declarations of the functions for people, days, and list construction are as follows.

```
FUNCTION  Nil : One -> List(a);
          Cons : a * List(a) -> List(a);
          Mon, Tue, Wed, Thu, Fri, Sat, Sun : One -> Day;
          Mary, Bill, Joe, Fred : One -> Person.
```

Each component of the FUNCTION declaration gives the signature of some function. There are only two categories of symbols which a programmer can declare – constructors and functions. Thus what are normally called constants are regarded here as functions which map from the domain of type One and predicates are regarded as functions which map into the domain of type Boolean. This uniform treatment facilitates the synthesis of the functional and logic programming concepts. Note that every function must have an -> at the top-level of its signature.

Functions are either *free* or *defined*. For the current application, the free functions are Nil, Cons, Mon, and so on, appearing in the above FUNCTION declaration. This means that, by default, the "definition" for each of these functions is essentially the corresponding Clark equality theory of syntactic identity. So, for example, the formulas

```
Cons(x, y) ~= Nil
```

and

```
Cons(x, y) = Cons(u, v) -> (x = u) & (y = v)
```

are included in this theory. A term is *free* if every function occurring in it is free and the term doesn't contain any λ-expressions.

On the other hand, defined functions have explicit definitions and take on the equality theory given by their definitions. For the application at hand, there are three defined functions with the following signatures.

```
FUNCTION  Perm : List(a) * List(a) -> Boolean;
          Concat : List(a) * List(a) -> List(a);
          Split : List(a) * List(a) * List(a) -> Boolean.
```

The intended meaning of these functions is as follows. Perm maps <s,t> to True if s and t are lists such that s is a permutation of t; otherwise, Perm maps <s,t> to False. Given lists s and t, Concat maps <s,t> to the list obtained by concatenating s and t (in this order). Given lists r, s, and t, Split maps <r,s,t> to True if r is the result of concatenating s and t (in this order); otherwise, Split maps <r,s,t> to False.

At this point, the intended interpretation has been defined and I can now turn to writing the program. This consists of the above declarations, plus some definitions (and MODE declarations) for the defined functions Perm, Concat, and Split, and is given in the module Permute below. The MODULE declaration simply gives the name of the module. (Details of the Escher module system will be given in Chapter 3.) Note that Perm is a function from the product type List(a) * List(a) to the type Boolean and advantage has been taken of the convention mentioned in Appendix A to write the head of the first statement as Perm(Nil,l) instead of Perm(<Nil,l>).

A *definition* of a function consists of one or more equations, which are called *statements*. The symbol => appearing in statements is simply equality, but I have made it into an arrow to indicate the directionality of the rewrite corresponding to the statement (explained below) and also to give a visual clue to indicate the head and body of a statement. In general, statements have the form

$H \Rightarrow B.$

Here the *head* H is a term of the form

$F(t_1, \ldots, t_n)$

```
MODULE    Permute.

CONSTRUCT Day/0, Person/0, List/1.

FUNCTION  Nil : One -> List(a);
          Cons : a * List(a) -> List(a);
          Mon, Tue, Wed, Thu, Fri, Sat, Sun : One -> Day;
          Mary, Bill, Joe, Fred : One -> Person.

FUNCTION  Perm : List(a) * List(a) -> Boolean.
MODE      Perm(NONVAR, _).
Perm(Nil, l) =>
    l = Nil.
Perm(Cons(h,t), l) =>
    SOME [u,v,r] (Perm(t,r) & Split(r,u,v) & l = Concat(u, Cons(h,v))).

FUNCTION  Concat : List(a) * List(a) -> List(a).
MODE      Concat(NONVAR, _).
Concat(Nil,x) =>
    x.
Concat(Cons(u,x),y) =>
    Cons(u,Concat(x,y)).

FUNCTION  Split : List(a) * List(a) * List(a) -> Boolean.
MODE      Split(NONVAR, _, _).
Split(Nil,x,y) =>
    x=Nil &
    y=Nil.
Split(Cons(x,y),v,w) =>
    (v=Nil & w=Cons(x,y)) \/
    SOME [z] (v = Cons(x,z) & Split(y,z,w)).
```

where F is a function, each t_i is a term, and the *body* B is a term. Note carefully that there is no implicit completion (in the sense of Clark) for predicate definitions in Escher; the theory that is intended is exactly the one that appears in the program (augmented by the default equality theory for the free functions).

Naturally, it must be checked that the intended interpretation is a model of the theory given by the program. For module `Permute`, this involves checking that each of the statements in the definitions is valid in the intended interpretation given above. The details of this are left to the reader. Leaving aside control issues, this completes the design and coding phases for this simple application. Assuming that the process of checking that the intended interpretation is a model of the program has been carried out correctly, the programmer can be now sure that the program is correct (that is, satisfies the specification given by the intended interpretation).

Note the `MODE` declaration for `Perm`. In general, `MODE` declarations restrict the possible calls that can be made to a function. For the `Perm` function, a call can only proceed if the first argument is

a non-variable term. (A *non-variable term* is a term that has a free function at the top level.) The underscore in the second argument indicates that there is no restriction on that argument. Note that a `MODE` declaration for which each argument is an underscore may be dropped.

There are some syntactic restrictions on the form statements may take.

1. Each argument in the head of a statement must be free.

2. Arguments in the head of a statement corresponding to underscores in the `MODE` declaration must be variables.

3. All local variables in a statement must be explicitly quantified.

4. Statements must be pairwise non-overlapping.

The first of these restrictions means that statement heads have a simple structure and also simplifies the matching part of the machinery of function calls (explained below). The second restriction comes about because the head of a statement should be at least as general as a call and an argument in a call corresponding to an underscore in a `MODE` declaration can be a variable. A *local variable* is a variable appearing in the body of a statement but not the head. The third restriction concerning local variables is largely a matter of taste since it would be possible to have a default giving the same effect. However, I think it is far preferable to explicitly give the quantification of the local variables.

Two statements, $H_1 \Rightarrow B_1$ and $H_2 \Rightarrow B_2$, are *non-overlapping* if whenever H_1 and H_2 (after standardization apart) are unifiable by some substitution θ, we have that $B_1\theta$ and $B_2\theta$ are identical (modulo renaming of bound variables). The restriction that statements must be pairwise non-overlapping means that if two or more statements match a call then it doesn't matter which of them is used – the result will be the same. This condition implies the confluence of the rewrite system associated with an Escher program. (See [4, p.252 *et seq.*] for details on rewrite systems in general and below for details on the rewrite system associated with an Escher program.)

Now I turn to the details of function calls. In a function call, a statement is viewed as a rewrite which behaves as follows. A statement $H \Rightarrow B$ *matches* a call H' if there is a substitution θ such that $H\theta$ is syntactically identical to H'. In this case, the call H' is replaced by $B\theta$ and this defines the rewrite given by the statement. (Thus a call is just what is normally called a redex in the terminology of rewrite systems.)

There is a restriction imposed on function calls.

1. No call may proceed unless the corresponding `MODE` declaration is satisfied.

It is a control error if no statement matches a call.

Here are some typical goals to the program consisting of the module `Permute` and their answers. (For these goals, it's convenient to use the usual notational sugar for lists provided by Escher via the `Lists` system module, so that `Nil` is written as `[]`, `Cons(s,t)` is written as `[s|t]`, and `Cons(s,Cons(t,Nil))` is written as `[s,t]`.) The goal

```
Concat([Mon, Tue], [Wed])
```

reduces to the answer

```
[Mon, Tue, Wed].
```

The goal

```
Split([Mon, Tue], x, y)
```

reduces to the answer

```
(x = [] & y = [Mon, Tue]) \/
(x = [Mon] & y = [Tue])   \/
(x = [Mon, Tue] & y = []).
```

The goal

```
~ Split([Mon, Tue], [Tue], y)
```

reduces to the answer

```
True.
```

Finally, the goal

```
Perm([Mon, Tue, Wed], x)
```

reduces to the answer

```
x = [Mon, Tue, Wed]  \/
x = [Tue, Mon, Wed]  \/
x = [Tue, Wed, Mon]  \/
x = [Mon, Wed, Tue]  \/
x = [Wed, Mon, Tue]  \/
x = [Wed, Tue, Mon].
```

How does Escher compute these answers? The first point is that Escher doesn't have a "theorem proving" computational model like the majority of logic programming languages. Instead it has a "rewriting" computational model, in which a goal term is reduced to an equivalent term, which is then given as the answer. Formally, if `s` is the goal term and `t` is the answer term, then the term `s = t` is valid in the intended interpretation. So, the first goal above, `Concat([Mon, Tue], [Wed])`, is reduced by a sequence of rewrites to the term `[Mon, Tue, Wed]`, by means of the computation given in Figure 2.1 below. The computation consists of the successive expressions produced by function calls, the first expression being the goal and the last the answer. The second and third expressions in the computation are obtained by using the second statement in the definition of `Concat`, while the fourth expression, which is the answer, is obtained by using the first statement in the definition of `Concat`. In each expression, the call is underlined. The other three goals for the module `Permute` require the use of statements for the functions `=`, `~`, and `\/` in the module `Booleans`, which will be discussed later.

Recall that, ultimately, a programmer is interested in computing the values of expressions in the intended interpretation. How does Escher assist in this? Since Escher has no direct knowledge of the intended interpretation, it cannot evaluate any expression in the intended interpretation. However, given an expression, it can *simplify* (that is, reduce) the expression, so that the evaluation in the intended interpretation can then be easily done by the programmer. This is evident in the above computation – the expression `Concat([Mon, Tue], [Wed])` is simplified to `[Mon, Tue, Wed]` which can be easily evaluated. Strictly speaking, this view is also appropriate for arithmetic expressions. For example, given the expression `3 + 4`, Escher will reply with the expression `7`. Formally, it hasn't evaluated `3 + 4`, but instead simplified it to `7`. In this case, the distinction

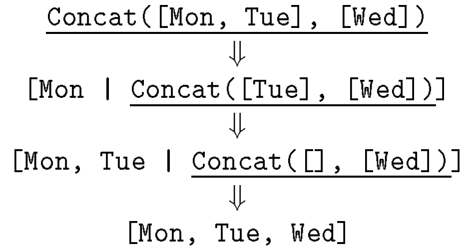


Figure 2.1: An Escher computation

between simplification and evaluation is a bit pedantic. But, in general, it's important to keep in mind this understanding of what Escher is doing.

The reduction process terminates when an expression is reached which contains no redexes. This final expression, the answer, is then in *normal form*. Typically, an answer may contain some defined functions, such as `=`, `&`, or `\/`, which are declared in system modules, and some free functions, such as `Cons` or `Nil`. However, if the answer contains a user-declared defined function, an error has occurred in the sense that a call to a user-declared defined function has never become sufficiently instantiated for its `MODE` declaration to be satisfied. This kind of programming error is called a *flounder*.

Note that there is no *explicit* concept of non-determinism in Escher. Instead, non-determinism is captured implicitly with disjunction. Furthermore, computations return "all answers" and never fail. In Escher, the equivalent of a failure in a conventional logic programming language is to return the answer `False`. For example, for the program consisting of the module `Permute`, the goal

```
Concat([Mon], [Tue]) = [Tue]
```

reduces to the answer

```
False.
```

2.3 Conditional Function

Escher has the usual conditional function `IF_THEN_ELSE`, whose declaration and definition are as follows.

```
FUNCTION  IF_THEN_ELSE : Boolean * a * a -> a.
IF_THEN_ELSE(True, x, y) => x.
IF_THEN_ELSE(False, x, y) => y.
```

Module `Conditional` below illustrates the use of the conditional function. Note that the arguments to `IF_THEN_ELSE` are written infix. `MemberCheck` is a predicate with the following intended meaning. Let `s` be an element and `t` a list. Then `MemberCheck` maps `<s,t>` to `True` if `s` is a member of `t`; otherwise, `MemberCheck` maps `<s,t>` to `False`. For this predicate, it is assumed that both arguments will be known when it is called, so that its definition can be written deterministically. For the program consisting of the module `Conditional`, the goal

```
MemberCheck(Tue, [Mon, Tue])
```

reduces to the answer

```
MODULE    Conditional.

CONSTRUCT Day/0, List/1.

FUNCTION  Nil : One -> List(a);
          Cons : a * List(a) -> List(a);
          Mon, Tue, Wed, Thu, Fri, Sat, Sun : One -> Day.

FUNCTION  MemberCheck : a * List(a) -> Boolean.
MODE      MemberCheck(NONVAR, NONVAR).
MemberCheck(x, Nil) =>
    False.
MemberCheck(x, Cons(y,z)) =>
    IF x = y
    THEN
      True
    ELSE
      Member(x,z).
```

`True.`

The goal

`MemberCheck(Fri, [Mon, Tue])`

reduces to the answer

`False.`

Finally, the goal

`MemberCheck(x, [Mon, Tue])`

has itself as answer because the `MODE` declaration on `MemberCheck` delays this predicate until both arguments are non-variable terms. Here the computation has floundered.

2.4 Local Definitions

It is common in mathematics for a definition to contain definitions of subsidiary concepts. If a subsidiary concept occurs in several places in the main definition, this gives a clearer and more compact main definition. For similar reasons, Escher provides a facility, called a local definition, for giving a subsidiary definition.

Module `LocalDefinition` illustrates the use of a local definition. The function `Halve` takes a list and returns a pair of lists, the first being (approximately) the first half of the original list and the second being the other half. Note that this module makes use of two system modules, `Integers` and `Lists`. The module `Lists` exports, in particular, the function `Take`, which takes the first `n` elements of a list, and the function `Drop`, which drops the first `n` elements of a list, where `n` is the

first argument to each of these functions. `Lists` also exports the function `Length` which returns the length of a list. The standard function `Div` is exported by the module `Integers`. The `IMPORT` declaration makes all these functions available for use in the module `LocalDefinition`. For the program consisting of the modules `LocalDefinition`, `Lists`, `Integers`, and `Booleans`, the goal

```
Halve([8, 1, 4, 23, 2])
```

reduces to the answer

```
<[8, 1], [4, 23, 2]>.
```

```
MODULE    LocalDefinition.

IMPORT    Integers, Lists.

FUNCTION  Halve : List(a) -> (List(a) * List(a)).
Halve(x) =>
    <Take(half, x), Drop(half, x)>  WHERE  half = Length(x) Div 2.
```

I now give the definitions of the concepts needed to understand local definitions. A *local definition* has the form

$$E \texttt{ WHERE } F$$

where E and F are terms. The term E is called the *body* and the term F is called the *qualifier* of the local definition.

To define the meaning of a local definition, I introduce a specific form of qualifier. A qualifier is said to be *standardized* if it has the form

$$v_1 \texttt{ = } E_1 \texttt{ \& } \ldots \texttt{ \& } v_n \texttt{ = } E_n$$

where

- $v_1, \ldots, v_n$ are distinct variables which occur freely in the body and are local to the local definition,
- v_i does not occur in E_j, for $i, j = 1, \ldots, n$, and
- E_i is in normal form, for $i = 1, \ldots, n$.

Then the meaning of a local definition E `WHERE` F, where F is the standardized qualifier v_1 `=` E_1 `&` $\ldots$ `&` v_n `=` E_n, is defined to be

$$(\lambda v_n \ldots (\lambda v_1.E\ E_1)\ \ldots\ E_n).$$

The meaning of a local definition for which the qualifier is not necessarily standardized is obtained by reducing the qualifier until it is standardized, if possible, and then applying the previous definition. (Thus not every local definition is well-defined.) Since

$$(\lambda v_n \ldots (\lambda v_1.E\ E_1)\ \ldots\ E_n)$$

reduces by β-reduction to

$$E\{v_1/E_1, \ldots, v_n/E_n\},$$

the intuitive meaning of a local definition is that the E_i are substituted for the v_i in E.

Typically, qualifiers have the form $v = E$, where v is a variable, and all that needs to be done is to reduce E and make the substitution for v. However, more complicated qualifiers are also useful. For example, a qualifier could have the form `<`v_1, v_2`> =` E, where E is reducible to a pair. In this case, when standardized, the qualifier has the form v_1 `=` E_1 `&` v_2 `=` E_2. As an example of this, suppose the following function is added to the module `LocalDefinition`.

```
FUNCTION  Switch : List(a) -> List(a).
Switch(x) =>
    Concat(z, y)  WHERE  <y, z> = Halve(x).
```

The function `Switch` switches the front and back halves of a list. For instance, the goal

```
Switch([8, 1, 4, 23, 2])
```

reduces to the answer

```
[4, 23, 2, 8, 1].
```

A qualifier could also have the form $F(v_1, v_2)$ `=` E, where F is a free function and E is reducible to a term of the form $F(E_1, E_2)$. In this case, the standardized form is v_1 `=` E_1 `&` v_2 `=` E_2.

The main practical reason for using a local definition is to avoid the inefficiency of computing the value of an expression more than once. Thus, while it makes no difference to the semantics when the substitution obtained from the qualifier is applied to the body, it is important from a practical point of view that it not be done too early (that is, before some E_i has been reduced), otherwise redundant computation may occur. For example, in module `LocalDefinition`, the value of the variable `half`, which is half the length of the list, is only computed once and then the value of `half` is substituted into the body of the local definition. In this case, not much is saved by the use of the local definition. However, in general, the computation of the E_i could be very expensive and there could be several occurrences of each v_i in the body, so that a lot of redundant computation could be avoided by use of the local definition. Later, in Chapter 4, I will give an application of a local definition for the function `Choice` in the module `Sets`, where the local definition is needed not only to avoid redundant computation, but also to ensure the correct semantics.

The concept of a local definition corresponds closely to that of the (non-recursive) `let` construct of functional languages. Since the v_i cannot appear in any E_j, the recursive `letrec` construct is not allowed in Escher. The current functional programming style, which has evolved over several decades, makes heavy use of local definitions. Escher programmers can also use this style, if desired. However, in this report, I will make little use of local definitions, writing more in the logic programming style and saving the use of local definitions for when they are definitely needed. Those who prefer using local definitions may find it instructive to rewrite some of the programs in the report in this style.

2.5 Higher-Order Facilities

This section contains some programs to illustrate the use of higher-order functions and λ-expressions in Escher.

```
MODULE    Higher.

CONSTRUCT Day/0, List/1.

FUNCTION  Nil : One -> List(a);
          Cons : a * List(a) -> List(a);
          Mon, Tue, Wed, Thu, Fri, Sat, Sun : One -> Day.

FUNCTION  Map : (a -> b) * List(a) -> List(b).
MODE      Map(_, NONVAR).
Map(f, Nil) =>
    Nil.
Map(f, Cons(x,xs)) =>
    Cons(f(x), Map(f, xs)).

FUNCTION  Filter : (a -> Boolean) * List(a) -> List(a).
MODE      Filter(_, NONVAR).
Filter(p, Nil) =>
    Nil.
Filter(p, Cons(x,xs)) =>
    IF p(x) THEN  Cons(x, Filter(p, xs)) ELSE Filter(p, xs).

FUNCTION  Next : Day -> Day.
MODE      Next(NONVAR).
Next(Mon) => Tue.
Next(Tue) => Wed.
Next(Wed) => Thu.
Next(Thu) => Fri.
Next(Fri) => Sat.
Next(Sat) => Sun.
Next(Sun) => Mon.

FUNCTION  Weekday : Day -> Boolean.
MODE      Weekday(NONVAR).
Weekday(Mon) => True.
Weekday(Tue) => True.
Weekday(Wed) => True.
Weekday(Thu) => True.
Weekday(Fri) => True.
Weekday(Sat) => False.
Weekday(Sun) => False.
```

Module `Higher` above illustrates the use of higher-order functions. `Map` is (an uncurried form of) the standard map function and `Filter` is a simple filtering function which uses its first argument to do the filtering. The predicate `Weekday` is noteworthy. Use has been made of the fact that its `MODE` declaration will only allow it to be called with its argument instantiated to write it as a collection of simple (deterministic) equations rather than a single (non-deterministic) equation involving disjunction as follows.

```
FUNCTION  Weekday : Day -> Boolean.
MODE      Weekday(_).
Weekday(x) =>
    x = Mon  \/
    x = Tue  \/
    x = Wed  \/
    x = Thu  \/
    x = Fri.
```

In this version of the definition, it is assumed that `Weekday` could be called with its argument a variable. This version of the definition is more flexible, but the price that has to be paid for this flexibility is the non-determinism in the definition in the form of disjunctions in the body. In general, the more information that is known about the mode of calls to a predicate, the more deterministically one can write its definition. Finally, note that in the definition of `Weekday` in module `Higher` an equation is required for every day of the week – both positive and negative information must be given explicitly.

For the program consisting of the modules `Higher` and `Booleans`, the goal

```
Map(Next, [Mon, Tue, Wed])
```

reduces to the answer

```
[Tue, Wed, Thu].
```

Also, the goal

```
Filter(Weekday, [Sun, Mon, Wed])
```

reduces to the answer

```
[Mon, Wed].
```

The final module, `Lambda`, in this chapter, which is taken directly from a λProlog paper [14], illustrates the use of λ-terms. Roughly speaking, the meaning of the function `Rel` is that it is true for an argument r iff either r is a primitive relation or else it is a relation made up of the conjunction of two primitive relations for which the second argument of the first relation coincides with the first argument of the second relation.

For the program consisting of the modules `Lambda` and `Booleans`, the goal

```
Rel(r) & r(John,Mary)
```

reduces to the answer

```
r = LAMBDA [u_1] (u_1 = <John, Mary>).
```

```
MODULE    Lambda.

CONSTRUCT Person/0.

FUNCTION  Jane, Mary, John : One -> Person.

FUNCTION  Mother : Person * Person -> Boolean.
Mother(x,y) =>
    x=Jane &
    y=Mary.

FUNCTION  Wife : Person * Person -> Boolean.
Wife(x,y) =>
    x=John &
    y=Jane.

FUNCTION  PrimitiveRel : (Person * Person -> Boolean) -> Boolean.
PrimitiveRel(r) =>
    r = Mother \/
    r = Wife.

FUNCTION  Rel : (Person * Person -> Boolean) -> Boolean.
Rel(r) =>
    PrimitiveRel(r) \/
    (SOME [r1,r2]
         (r = LAMBDA [u] (SOME [x,y,z] (u = <x,y> & r1(x,z) & r2(z,y))) &
          PrimitiveRel(r1) &
          PrimitiveRel(r2))).
```

The value returned for the variable `r` is the predicate whose domain has type `Person * Person` and which maps to `True` if and only if its argument is the tuple `<John, Mary>`.

Note that the function `Rel` in module `Lambda` could also be defined as follows.

```
Rel(r) =>
    PrimitiveRel(r) \/
    (SOME [r1,r2]
         (r = LAMBDA [u] (SOME [z] (r1(Fst(u),z) & r2(z,Snd(u)))) &
          PrimitiveRel(r1) &
          PrimitiveRel(r2))).
```

Here `Fst` (provided by module `Booleans`) returns the first argument of a tuple and `Snd` returns the second. For this version of `Rel`, the answer is

```
r = LAMBDA [u_1] ((Fst(u_1) = John) & (Snd(u_1) = Mary))
```

which is equivalent to the previous answer.

Chapter 3

Modules

In this chapter, I describe the Escher module system.[1]

3.1 Importing and Exporting

In general, modules consist of two parts, an export part and a local part. The *export part* of a module is indicated by an export declaration, which is either an `EXPORT` or `CLOSED` declaration. The *local part* of a module is indicated by a local declaration, which is either a `LOCAL` or `MODULE` declaration. In these declarations, the keywords `EXPORT`, `CLOSED`, `LOCAL` and `MODULE` are followed by the name of the module. In fact, a module may have a local and an export part, or just a local part, or just an export part. The other kind of module declaration is the import declaration. In this declaration, the keyword `IMPORT` is followed by the name of a module.

The export part of a module begins with an export declaration and contains zero or more import declarations, language declarations, and mode declarations. The local part of a module begins with a local declaration and contains zero or more import declarations, language declarations, mode declarations, and statements. If a module consists *only* of a local part, then this is indicated by using a `MODULE` declaration instead of a `LOCAL` declaration. The use of a `CLOSED` declaration instead of an `EXPORT` declaration will be explained shortly.

I now introduce the concepts of importation, accessibility, and exportation in an informal way. The precise definition of these concepts is given in Section 3.2.

There are two *categories*: constructors and functions. A *symbol* is either a constructor or a function. A part of a module *declares* a symbol if the part contains a language declaration for that symbol. A module *declares* a symbol if either the local or export part of the module declares the symbol. A part of a module *imports* a symbol if the part contains an `IMPORT` declaration with the module name `N`, say, and either the module `N` declares this symbol in its export part or, inductively, `N` imports the symbol into its export part. A symbol is *accessible to* the local (resp., export) part of a module if it is either declared in, or imported into, a part of the module (resp., the export part of the module). A module *exports* a symbol if the symbol is accessible to the export part of the module.

Subject to the module conditions given in Section 3.2, a symbol accessible to a part of a module is available for use in that part of the module. More precisely, a constructor accessible to a part of a module can appear in a `FUNCTION` declaration in that part of the module. Similarly, a function accessible to a part of a module can appear in a statement in the module. Of course, a symbol can only be used according to its language declaration.

[1] The design of the module system is incomplete.

I illustrate these concepts with the modules `M0` and `M1` below. Module `M1` has an export part and a local part. The export part of `M1` makes all the symbols it declares or imports available for use by other modules, such as `M0`. In particular, the declarations for the constructors `Day` and `Person`, the declarations for the functions `Mon`, `Fred`, and so on, and the declaration for the function `Split3` make these symbols available for use by other modules which import `M1`. The `IMPORT` declaration in the export part of `M1` makes the symbols exported by `Integers` and `Lists` available for use in `M1`. Any module which imports `M1` automatically imports all the symbols exported by `Lists` and `Integers` and hence does not need to import `Lists` or `Integers` explicitly to make these available for use. The local part of `M1` contains the definition of the function `Split3`, which uses the definition of `Split` from `Lists`.

```
MODULE     M0.

IMPORT     M1.

FUNCTION   Member2 : a * a * List(a) -> Boolean.
MODE       Member2(_, _, NONVAR).
Member2(x, y, z) =>
    SOME [u,v,w] Split3(z, u, [x | v], [y | w]).
```

```
EXPORT     M1.

IMPORT     Integers, Lists.

CONSTRUCT  Day/0, Person/0.

FUNCTION   Mon, Tue, Wed, Thu, Fri, Sat, Sun : Day;
           Fred, Bill, Mary : Person.

FUNCTION   Split3 : List(a) * List(a) * List(a) * List(a) -> Boolean.
MODE       Split3(NONVAR, _, _, _).
```

```
LOCAL      M1.

Split3(u, x, y, z) =>
    SOME [w] (Split(u, x, w) & Split(w, y, z)).
```

By contrast with module `M1`, module `M0` has only a local part. Hence its first module declaration uses the keyword `MODULE` instead of `LOCAL`. It imports all the symbols exported by `M1`, which include

`Split3` together with all the symbols exported by `Lists` and `Integers`. Module `M0` contains the definition of the predicate `Member2`. `Member2(x,y,z)` maps to true if and only if `z` is a list which contains `x` and `y` as members so that `x` precedes `y` in the list.

Informally, a *program* consists of a set of modules $\{M_i\}_{i=0}^n$ $(n \geq 0)$, where M_0 is a distinguished module called the *main* module and $\{M_i\}_{i=1}^n$ is the set of modules which appear in import declarations in M_0 or appear in import declarations in these modules, and so on. (A formal definition is given in Section 3.3 and Chapter 7.) For the above example, {`M0`, `M1`, `Lists`, `Integers`, `Booleans`} is a program with main module `M0`. (Note that it is not necessary to *explicitly* import `Booleans` into modules `M0` or `M1` as `Booleans` is imported automatically by the system into every module. Also, since `Lists` imports `Integers`, it would be sufficient for `M1` to import just `Lists`.)

Certain modules are provided by the system and hence are called *system modules*. For example, `Lists`, `Integers`, and `Booleans` are system modules. The complete set of system modules is given in Chapter 9. Other modules are called *user modules*. Note that the export declaration in the export part of a system module may contain the keyword `CLOSED` instead of `EXPORT`. A module is *closed* if its export part contains a `CLOSED` declaration and *open* if its export part contains an `EXPORT` declaration. In closed modules the visibility of symbols accessible to the local part of the module is restricted. This effectively increases the range of implementations available. For example, the local part of a closed module does not have to be implemented in Escher. An implementation of Escher may make some or all of the system modules closed.

3.2 Module Declarations and Conditions

I now give a precise meaning to the various module declarations and also give the module conditions which every module must satisfy.

A *symbol* is a constructor or function.

A part of a module *refers to* a module N if it contains a declaration of the form

`IMPORT` N.

A module M *refers to* a module N if either the local or export part of M refers to N.

A part of a module M *1-imports* a symbol S *via* a module N if S has a declaration in the export part of N and the part of M refers to N.

A part of a module M *n-imports*, $n > 1$, a symbol S *via* a module N if there is a module L such that the export part of L $(n-1)$-imports S via N and the part of M refers to L.

A part of a module M *imports* a symbol S *via* a module N if the part of M n-imports S via N, for some $n \geq 1$.

A module M *imports* a symbol S *via* a module N if either the local or export part of M imports S via N.

A part of a module *declares* a symbol if it contains a declaration for the symbol.

A module *declares* a symbol if either the local or export part of the module declares the symbol.

A part of a module M *imports* a symbol S *from* a module N if the part of M imports S via N and the export part of N declares S.

A module M *imports* a symbol S *from* a module N if either the local or export part of M imports S from N.

A symbol is *accessible to* the local (resp., export) part of a module if it is either declared in, or imported into, the module (resp., export part of the module).

A module *exports* a symbol if the symbol is accessible to the export part of the module.

Now I turn to the *module conditions*, M1 to M4, given below, which are enforced by the system. The first of these ensures that the module structure of a program is non-circular and hence greatly simplifies compilation. To state this condition, I introduce a new concept. The relation *depends upon* between modules is the transitive closure of the relation "refers to". So, for example, module `M0` depends upon the modules `M1`, `Lists`, `Integers`, and `Booleans`.

M1: No module may depend upon itself.

The next module condition ensures that, in any given module part, the only names that can appear are names of symbols that are accessible.

M2: For every name appearing in a part of a module, there must be a symbol having that name accessible to that part.

Overloading is a useful programming language facility. For example, it is convenient to use the name `-` for both unary and binary minus, and the name `+` for addition of integer, rational, and floating-point numbers. For this reason, Escher has a flexible scheme for overloading. The only condition on naming symbols enforced by the system is the following module condition.

M3: Distinct symbols cannot be declared in the same module with the same category, name, and arity.

The final module condition ensures that no definition can be split across several modules.

M4: A module must declare every function defined in that module.

3.3 Programs, Goals, and Answers

A *program* consists of a set of modules $\{M_i\}_{i=0}^{n}$ $(n \geq 0)$, where $\{M_i\}_{i=1}^{n}$ is the set of modules upon which M_0 depends and where each module satisfies the module conditions given in Section 3.2. The module M_0 is called the *main* module of the program.

To define a statement, I introduce the language of a module in a program.

- The *language* of a module M in a program is the language given by the language declarations of all symbols accessible to the local part of M.

The export language of a module will also be needed.

- The *export language* of a module M in a program is the language given by the language declarations of all symbols accessible to the export part of M.

To define a goal, the goal language of a program will be needed.

- The *goal language* of a program P is the language of the main module M_0 of P, if M_0 is open, or the export language of M_0, if M_0 is closed.

A *statement* in a module M in a program is a term in the language of M having the form $H \Rightarrow B$. Here the *head* H is a term of the form $F(t_1, \ldots, t_n)$ where F is a function, each t_i term, and the *body* B is a term.

A *goal* for a program is a term in the goal language of the program.

An *answer* for a program and goal is the term obtained by reducing the goal to normal form using the rewrites associated with the statements in the program.

As an example, typical goals for the program {`M0`, `M1`, `Lists`, `Integers`, `Booleans`} could be

```
Member2(Fred, Mary, [Fred, Bill, Mary])
```

which reduces to the answer

```
True
```

and

```
Member2(x, y, [1, 2, 3])
```

which reduces to the answer

```
((x = 1) & (y = 2)) \/
(((x = 1) & (y = 3)) \/
((x = 2) & (y = 3))).
```

Also the goal

```
Split3([Mon, Tue], x, y, z)
```

reduces to the answer

```
((x = []) & ((y = []) & (z = [Mon, Tue]))) \/
(((x = []) & ((y = [Mon]) & (z = [Tue]))) \/
(((x = []) & ((y = [Mon, Tue]) & (z = []))) \/
(((x = [Mon]) & ((y = []) & (z = [Tue]))) \/
(((x = [Mon]) & ((y = [Tue]) & (z = []))) \/
((x = [Mon, Tue]) & ((y = []) & (z = [])))))))).
```

Chapter 4

System Types

In this chapter, I describe various types provided by Escher system modules. Full versions of the export parts of all system modules are given in Chapter 9.

4.1 Booleans

The most basic of the types provided by the Escher module system is the type `Boolean`. Various functions operating on booleans, including the connectives and quantifiers of type theory, are provided the the system module `Booleans`. In fact, `Booleans` is such a fundamental module that it is imported into every other module without an explicit `IMPORT` statement. A brief version of the export part of `Booleans` appears below and (a possible implementation of) the local part is given in Appendix B.

```
EXPORT    Booleans.

CONSTRUCT One/0, Boolean/0.

FUNCTION  = : a * a -> Boolean;
          True : One -> Boolean;
          False : One -> Boolean;
          & : Boolean * Boolean -> Boolean;
          \/ : Boolean * Boolean -> Boolean;
          -> : Boolean * Boolean -> Boolean;
          <- : Boolean * Boolean -> Boolean;
          <-> : Boolean * Boolean -> Boolean;
          ~ : Boolean -> Boolean;
          SIGMA : (a -> Boolean) -> Boolean;
          PI : (a -> Boolean) -> Boolean;
          IF_THEN_ELSE : Boolean * a * a -> a;
          Fst : a * b -> a;
          Snd : a * b -> b.
```

The function `=` is equality (which is interpreted as the identity relation). As is clear from the

local part of `Booleans`, the definition for `=` provides the functionality of the (first order) unification algorithm. So far, no *explicit* functionality of the higher-order unification algorithm beyond the first-order case is provided by Escher. However, Escher can cope with equality between terms involving λ-expressions in some situations, especially for sets. (See Section 4.5.)

`True` and `False` provide the truth values. Each of these is a free function. Next come the connectives, conjunction (`&`), disjunction (`\/`), implication (`->`), reverse implication (`<-`), biconditional (`<->`), and negation (`~`). Their respective definitions in the local part of `Booleans` capture the standard properties of these connectives. After the connectives are the (generalized) quantifiers, `SIGMA` and `PI`. In fact, these never appear without encompassing a λ-expression and so appear only in disguised form as `SOME` and `ALL`. Consequently, the definition for `SIGMA` provides some statements with `SOME` in the head and the definition for `PI` provides a statement with `ALL` in the head. Note that `ALL` is treated classically. Escher also provides the conditional function `IF-THEN-ELSE` and the projection functions `Fst` and `Snd`, each having their natural definitions. There is also an axiom (schema) corresponding to β-reduction.

As an example of the capabilities provided by `Booleans`, for any program, the goal

```
ALL [x] (y \/ ~y)
```

reduces to the answer

```
True
```

and the goal

```
((LAMBDA [x] x = True) False)
```

reduces to the answer

```
False
```

by means of β-reduction.

In addition, Escher has some very useful notational sugar for a form of conditional introduced by Naish and first made available in NU-Prolog [19]. This notation has the form

`IF SOME` $[x_1, \ldots, x_n]$ *Cond* `THEN` *Form1* `ELSE` *Form2*

where *Cond*, *Form1* and *Form2* are formulas. This conditional is defined to mean

`(SOME` $[x_1, \ldots, x_n]$ `(`*Cond* `&` *Form1* `)) \/ (~ SOME` $[x_1, \ldots, x_n]$ *Cond* `&` *Form2*`).`

As was confirmed by the Gödel language, this form of conditional is extremely useful. The main advantages of using the notational sugar (instead of its meaning) are that the notation provides a compact and expressive notation for a common programming idiom and the Escher system avoids the inefficiency of computing the condition twice.

Anticipating a little the introduction of the `Integers`, `Lists`, and `Text` system modules later in this chapter, module `AssocList` below illustrates the use of this form of conditional. In this module, an association list is a list of entries, each of which is a tuple consisting of an integer which is the key and a string which is the data. The function `Lookup` maps a quadruple to `True` if the first argument is an integer, the second is a string, the third is an association list, and the fourth is the association list of the third argument, augmented with the entry consisting of the tuple of the first and second arguments if and only if the key of this entry is not in the association list in the third argument; otherwise, `Lookup` maps a quadruple to `False`.

For the program {`AssocList`, `Text`, `Lists`, `Integers`, `Booleans`}, the goal

```
MODULE    AssocList.

IMPORT    Integers, Lists, Text.


FUNCTION  Lookup : Integer * String * List(Integer * String) *
                   List(Integer * String) -> Boolean.

MODE      Lookup(NONVAR,_,NONVAR,_).

Lookup(key, value, assoc_list, new_assoc_list) =>
    IF SOME [v] Member(<key,v>, assoc_list)
    THEN
      value = v &
      new_assoc_list = assoc_list
    ELSE
      new_assoc_list = Cons(<key,value>, assoc_list).
```

```
Lookup(5, value, [<4,"How">, <5,"You">], list)
```

reduces to the answer

```
(value = "You") & (list = [<4, "How">, <5, "You">])
```

and the goal

```
Lookup(1, "Are", [<4,"How">, <5,"You">, <5,"Then">], list)
```

reduces to the answer

```
list = [<1, "Are">, <4, "How">, <5, "You">, <5, "Then">].
```

Finally, note that the syntax of statements allowed in system modules is more liberal than that allowed in user modules in that the head of a statement in a system module can be an arbitrary term. An example of this is the axiom for β-reduction which has an application of a λ-expression to an argument in the head. Furthermore, the control available in system modules is more flexible than that provided by `MODE` declarations. For example, certain statements in system modules can only be invoked if an argument in a call is a variable. The reason for the decision to restrict the syntax available to users is that the form of heads and the mode system available to users seem quite adequate for the vast majority of applications and avoid various complications that would arise if these restrictions were dropped.

4.2 Integers

A brief version of the export part of the system module `Integers` is given below. There is a constructor `Integer` of arity 0 giving the type of the integers. Furthermore, there are declared infinitely many free functions `0`, `1`, `2`, ... mapping from `One` to `Integer` which are in one-to-one

correspondence with the non-negative integers. This declaration is commented out, since the syntax of Escher doesn't support the declaration of infinitely many functions. However, this doesn't cause any difficulty for the underlying logic and the implementation can handle such a "declaration".

```
EXPORT    Integers.

CONSTRUCT Integer/0.

% FUNCTION  0, 1, 2, ... : One -> Integer.

FUNCTION  = : Integer * Integer -> Boolean;
          + : Integer * Integer -> Integer;
          - : Integer * Integer -> Integer;
          - : Integer -> Integer;
          * : Integer * Integer -> Integer;
          Div : Integer * Integer -> Integer;
          Mod : Integer * Integer -> Integer;
          ^ : Integer * Integer -> Integer;
          =< : Integer * Integer -> Boolean;
          >= : Integer * Integer -> Boolean;
          < : Integer * Integer -> Boolean;
          > : Integer * Integer -> Boolean.
```

The module provides the usual arithmetic operations and predicates for integers. Furthermore, Escher is intended to eventually support constraint solving in the domain of the integers. However, the current implementation supports only limited constraint-solving facilities. For example, for the program {`Integers`, `Booleans`}, the goal

```
12 = x*4
```

reduces to the answer

```
x = 3,
```

the goal

```
9 = 4*x
```

reduces to the answer

```
False,
```

the goal

```
32*4 >= (130 Mod 4)
```

reduces to the answer

```
True,
```

the goal

```
x + 43 = 73 + (34 Mod 4)
```

reduces to the answer

```
x = 32,
```

the goal

```
0 =< x =< 10  &  0 < y =< 10  &  35*x + 33*y =< 34
```

reduces to the answer

```
(x = 0) & (y = 1),
```

and the goal

```
0 < x =< 10  &  x ~= 2  &  x^2 < 12
```

reduces to the answer

```
(x = 1) \/
(x = 3).
```

One can also find Pythagorean numbers using, for example, the goal

```
x^2 + y^2 = z^2  &  0 < x < 10  &  0 < y < 10  &  0 < z
```

which reduces to the answer

```
((x = 3) & ((y = 4) & (z = 5))) \/
(((x = 4) & ((y = 3) & (z = 5))) \/
(((x = 6) & ((y = 8) & (z = 10))) \/
((x = 8) & ((y = 6) & (z = 10))))).
```

4.3 Lists

Escher provides some standard list-processing functions via the system module `Lists`. A brief version of the export part of `Lists` appears below. Escher also provides facilities for list comprehension familiar from functional languages. Thus the comprehension

```
[s : x <-- t]
```

means

```
Map(LAMBDA [x] s, t),
```

the comprehension

```
[s : t ; r]
```

means

```
Join([[s : r] : t]),
```

```
EXPORT    Lists.

IMPORT    Integers.

CONSTRUCT List/1.

FUNCTION  Nil : One -> List(a);
          Cons : a * List(a) -> List(a);
          Member : a * List(a) -> Boolean;
          Concat : List(a) * List(a) -> List(a);
          Split : List(a) * List(a) * List(a) -> Boolean;
          Append : List(a) * List(a) * List(a) -> Boolean;
          Permutation : List(a) * List(a) -> Boolean;
          Delete : a * List(a) * List(a) -> Boolean;
          DeleteFirst : a * List(a) * List(a) -> Boolean;
          Sorted : List(Integer) -> Boolean;
          Sort : List(Integer) -> List(Integer);
          Foldr : (a * b -> b) *  b * List(a) -> b;
          Join : List(List(a)) -> List(a);
          Map : (a -> b) * List(a) -> List(b);
          Empty : List(a) -> Boolean;
          Head : List(a) -> a;
          Tail : List(a) -> List(a);
          Length : List(a) -> Integer;
          Take : Integer * List(a) -> List(a);
          Drop : Integer * List(a) -> List(a).
```

and the comprehension

```
[s : t]
```

means

```
IF t THEN [s] ELSE [].
```

In this notation, a term appears before the colon and one or more *qualifiers* appear after the colon. A qualifier can be either a *generator* or a *test*. Generators have the form `x <-- s` and tests are formulas. The semicolon indicates *composition* of qualifiers.

As an illustration of the use of list comprehension, consider the module `ListProcessing` below. For the program {`ListProcessing`, `Lists`, `Integers`, `Booleans`}, the goal

```
[Next(x) : x <-- [Mon, Tue]]
```

reduces to the answer

```
[Tue, Wed],
```

the goal

```
[<x,y> : x <-- [Mon, Tue]; y <-- [Wed, Thu]]
```

reduces to the answer

```
[<Mon, Wed>, <Mon, Thu>, <Tue, Wed>, <Tue, Thu>],
```

and the goal

```
[x : x <-- [Mon, Tue, Sun]; Weekday(x)]
```

reduces to the answer

```
[Mon, Tue].
```

```
MODULE    ListProcessing.

IMPORT    Integers, Lists.

CONSTRUCT Day/0.

FUNCTION  Mon, Tue, Wed, Thu, Fri, Sat, Sun : One -> Day.

FUNCTION  Next : Day -> Day.
MODE      Next(NONVAR).
Next(Mon) => Tue.
Next(Tue) => Wed.
Next(Wed) => Thu.
Next(Thu) => Fri.
Next(Fri) => Sat.
Next(Sat) => Sun.
Next(Sun) => Mon.

FUNCTION  Weekday : Day -> Boolean.
MODE      Weekday(NONVAR).
Weekday(Mon) => True.
Weekday(Tue) => True.
Weekday(Wed) => True.
Weekday(Thu) => True.
Weekday(Fri) => True.
Weekday(Sat) => False.
Weekday(Sun) => False.
```

The functions provided by `Lists` are rather unremarkable, except perhaps for `Append`, `Concat`, and `Split`. The function `Append` is the traditional logic programming predicate and, as its mode declaration shows, can be called with any instantiation pattern of its arguments. However, typically `Append` is called in one of two modes, either to concatenate two given lists or to split a given list into two sublists. Hence Escher also provides two functions to handle this – `Concat` to concatenate

two lists and Split to split a list, each with an appropriate mode declaration. In fact, Append is really only provided for completeness, as I believe one hardly ever uses Append for anything other than concatenating or splitting. The use of Concat or Split, as appropriate, is not only more efficient than using Append, but is also significantly clearer for someone trying to understand the program.

4.4 Characters and Strings

The system module Text provides functions for processing characters and strings. The export part of Text appears, in brief, below. The CONSTRUCT declaration introduces the types Char and String. To represent the characters, finitely many functions from One to Char, in one-to-one correspondence with the characters, are introduced. Associated with this is some notational sugar. Escher uses single quotes around the character itself to denote these functions. Thus, 'z' is notational sugar for the function whose denotation is the character z. Similarly, there are infinitely many functions from One to String in one-to-one correspondence with all possible strings. The notational sugar for string uses the usual double quotes.

As an illustration of some of the functions in Text, for the program {Text, Lists, Integers, Booleans}, the goal

```
Chr(100)
```

reduces to the answer

```
'd',
```

the goal

```
Ord('d')
```

reduces to the answer

```
100,
```

the goal

```
StringToAscii("abcd")
```

reduces to the answer

```
[97, 98, 99, 100],
```

and the goal

```
"ABC" ++ "DEF"
```

reduces to the answer

```
"ABCDEF".
```

```
EXPORT    Text.

IMPORT    Integers, Lists.

CONSTRUCT Char/0, String/0.

%FUNCTION  Finitely many functions from One to Char which are in one-to-one
%          correspondence with all possible characters.  Syntax: e.g., 'z'.

%FUNCTION  Infinitely many functions from One to String which are in one-to-one
%          correspondence with all possible strings.  Syntax: e.g., "A string".

FUNCTION  = : Char * Char -> Boolean;
          = : String * String -> Boolean;
          CharToString : Char -> String;
          StringToChar : String -> Char;
          =< : Char * Char -> Boolean;
          >= : Char * Char -> Boolean;
          < : Char * Char -> Boolean;
          > : Char * Char -> Boolean;
          Ord : Char -> Integer;
          Chr : Integer -> Char;
          ++ : String * String -> String;
          =< : String * String -> Boolean;
          >= : String * String -> Boolean;
          < : String * String -> Boolean;
          > : String * String -> Boolean;
          StringToAscii : String -> List(Integer);
          AsciiToString : List(Integer) -> String.
```

4.5 Sets

Next I turn to the set-processing facilities of Escher. Because the underlying logic of Escher is higher order, sets are handled in a particularly simple and satisfying way. Essentially, one identifies a set with a predicate. More precisely, a set is identified with the predicate on the same domain as the set which maps an element of the domain to `True` if and only if the element is a member of the set. Having made this identification, the usual set operations such as intersection and union are simply higher-order functions whose arguments are predicates.

A brief version of the export part of the system module `Sets` is given below. As an illustration of the use of this module, consider the module `SportsDB` below. Because the function `Likes` is intended to be called with either argument or even both arguments uninstantiated, its definition must be given by a single statement in which the non-determinism is captured by the disjunctions. The statement is essentially the (Clark) completion of the obvious set of facts. Compare this with the treatment of the function `Weekday` in the module `Higher` for which more was known (or at least assumed) about the argument in function calls. These two functions highlight a crucial aspect of

```
EXPORT    Sets.

IMPORT    Integers.

FUNCTION  = : (a -> Boolean) * (a -> Boolean) -> Boolean;
          UNION : (a -> Boolean) * (a -> Boolean) -> (a -> Boolean);
          INTERS : (a -> Boolean) * (a -> Boolean) -> (a -> Boolean);
          MINUS : (a -> Boolean) * (a -> Boolean) -> (a -> Boolean);
          SUBSET :  (a -> Boolean) * (a -> Boolean) -> Boolean;
          SUPERSET :  (a -> Boolean) * (a -> Boolean) -> Boolean;
          IN : a * (a -> Boolean) -> Boolean;
          Size : (a -> Boolean) -> Integer;
          Select : (a -> Boolean) -> a;
          Choice : (a -> Boolean) -> a.
```

the Escher mode system. The more that is known about the mode of a function, the more scope there is for writing the function deterministically. The mode system has been specifically designed to support this.

For the program {`SportsDB`, `Sets`, `Integers`, `Booleans`}, the goal

```
{s : Likes(Fred, s)}
```

reduces to the answer

```
{},
```

since `Likes(Fred, s)` reduces to `False`. The goal

```
Fred IN ({Joe, Fred} UNION x)
```

reduces to the answer

```
True.
```

The goal

```
({Mary, Bill} UNION x) = ({Fred, Bill} UNION x)
```

reduces to the answer

```
(Mary IN x) & (Fred IN x).
```

The goal

```
ALL [y] (y IN {Cricket, Tennis} -> Likes(x,y))
```

reduces to the answer

```
(x = Mary) \/
(x = Bill).
```

Finally, the goal

```
{Mary, Joe} =  {x, y}
```

reduces to the answer

```
(x = Mary & y = Joe)  \/
(x = Joe & y = Mary).
```

```
MODULE    SportsDB.

IMPORT    Sets.

CONSTRUCT Person/0, Sport/0.

FUNCTION  Mary, Bill, Joe, Fred : One -> Person;
          Cricket, Football, Tennis : One -> Sport.

FUNCTION  Likes : Person * Sport -> Boolean.
Likes(x, y) =>
       (x = Mary & y = Cricket) \/
       (x = Mary & y = Tennis)  \/
       (x = Bill & y = Cricket) \/
       (x = Bill & y = Tennis)  \/
       (x = Joe  & y = Tennis)  \/
       (x = Joe  & y = Football).
```

As a perusal of the local part of `Sets` in Appendix B shows, Escher has considerable resources for dealing with sets. The many statements about set equality in this module amount to rather sophisticated constraint-solving facilities involving sets. To take one example of this, consider the statement

```
(y UNION x) = (z UNION x)  =>  x SUPERSET ((y UNION z) MINUS (y INTERS z)).
%
%   where x is a variable.
```

Supposing that `y` and `z` are known sets, the head of this statement is a constraint on the value of `x`. After the statement is called, the constraint in the head with two occurrences of `x` is replaced by the simpler constraint in the body with one occurrence of `x`. Thus progress towards solving the larger constraint system in which an instance of the head of the statement appears has been made. Many other statements in the local part of `Sets` provide similar capabilities.

Finally, I discuss the functions `Select` and `Choice` in `Sets`, which are closely related, but subtly different, functions. First, the function `Select` chooses the (unique) element in a singleton set and is undefined if its argument has more than one member. `Select` corresponds to the "definite description" operator of type theory. (See [1, p. 162].) This function is declarative – it is implemented by a single statement which chooses the element of a singleton set (see the local part of `Sets`) and its denotation is some fixed function whose value for a singleton set is the element in

the set and whose value for a non-singleton set is some definite, but unspecified, value. Whatever the denotation settled upon, it *does* satisfy the statement.

The function `Choice` (non-deterministically) chooses an element from a set, but now the set doesn't have to be a singleton. Although convenient in some applications (one is given below), `Choice` is problematical because it is not declarative. More precisely, while it is possible to write down statements to implement `Choice` (see the local part of `Sets`), it is not possible to give a denotation for this function. The trouble is in the third statement for which different computational paths could lead to different elements being chosen. Thus the denotation of `Choice` is not certain to satisfy the third statement.

However, while `Choice` is non-declarative, it is very useful if used carefully. To give an example of this, consider the task of writing a function which computes the sum of the integers in a (finite) integer-valued set. One can write this as follows.

```
FUNCTION  Sum : (Integer -> Boolean) -> Integer.
Sum(set) =>
    Select({sum : Sum1(set, sum)}).

FUNCTION  Sum1 : (Integer -> Boolean) * Integer -> Boolean.
Sum1(set, sum) =>
    IF set = {}
    THEN
      False
    ELSE
      IF SOME [x] set = {x}
      THEN
        sum = x
      ELSE
        SOME [z, sum1] (z IN set & Sum1(set MINUS {z}, sum1) & sum = z + sum1).
```

This definition of `Sum` is declarative but highly inefficient as it will calculate the sum of the elements in the set in $n!$ ways, if there are n elements in the set. Unfortunately, this kind of unwanted non-determinism can plague set-processing functions. There really isn't any good solution to this problem – one way or another the unwanted solutions have to be pruned away. For the particular example being considered, `Choice` can be used to provide a neat, if non-declarative, solution. Here is another version of the `Sum` function using `Choice`.

```
FUNCTION  Sum : (Integer -> Boolean) -> Integer.
Sum(set) =>
    IF set = {}
    THEN
      Error("Error: Sum has empty set as argument", Sum({}))
    ELSE
      IF SOME [x] set = {x}
      THEN
        Choice(set)
      ELSE
        (y + Sum(set MINUS {y})) WHERE y = Choice(set).
```

This version of `Sum` is efficient – the sum will only be computed once. However, it is not declarative because it uses `Choice`. The way to handle this kind of problem is to build a barrier around the non-declarativeness. As the first definition showed, `Sum` *can* be written in a declarative way. So, provided program analysis and optimization tools take extreme care inside the second, more efficient, definition of `Sum`, it can be used instead. It would be possible to enforce this kind of barrier to non-declarative code by specific language facilities, but for the moment I have not done this.

One other point of interest is the use of the `WHERE` clause in the second definition of `Sum`. Since there are two occurrences of the variable `y`, the use of the `WHERE` clause becomes mandatory. If each occurrence of `y` were replaced by `Choice(set)`, we not could be sure that the two different calls to `Choice(set)` would return the same answer!

4.6 Programs

Escher is intended to provide sophisticated meta-programming facilities via an abstract data type `Program` for which many operations need to be provided. The basic ideas here have already been worked out in detail for the Gödel language [10] and so it is only necessary to transfer those ideas to Escher. There will be a few differences between Escher and Gödel in their respective meta-programming facilities. In the case of Escher, a ground representation for its higher-order logic will have to be provided and this will differ in some details from the Gödel case where the logic is first order. Overall, the ground representation for Escher should be simpler as the treatment of sets in Escher is simpler and there are only two categories of symbols (compared to six for Gödel). The details of all this will be given in a future version of this report.

4.7 Input/Output

In this section, I describe the input/output facilities provided by Escher. The higher-order nature of Escher can be used to provide declarative input/output. The key idea to achieving this is to provide a suitable abstract data type (ADT) for the "world" and then restrict the operations on this type so that declarativeness can be enforced. This approach to input/output is called monadic IO because its theoretical basis can be found in the category-theoretic concept of a monad. Fortunately, it is not necessary to investigate the category-theoretic foundations to understand the key ideas – it's sufficient to merely appreciate how the ADT works. The idea of monadic IO has been developed over recent years in the functional community ([12], [17], [21]) and is being incorporated into the next version of the Haskell language.

The starting point is an ADT called `World`. A programmer can think of items of type `World` as being a reflection of (part of) the actual world, for example, the state of a file system in an operating system. In fact, `World` is abstract in a very strong sense as programmers cannot refer directly to this type at all! But it is the foundation on which monadic IO is built. What a programmer *can* manipulate are called IO *transformers*, which are mappings from one world to another world that return a value as well. The appropriate constructor for IO transformers is `IO` of arity 1. In fact, as the export part of the system module `IO` makes clear, the "constructor" `IO` is not independently declared but instead is defined in terms of `World` by

```
IO(a) = World -> a * World,
```

where `a` is a parameter. Thus an IO transformer of type `IO(a)` is a function that takes a state of the world and returns a new state of the world together with a value of type `a`.

```
EXPORT    IO.

IMPORT    Text, Integers.

% CONSTRUCT World/0                          % Declared locally.
%
% DEFINE    IO(a) = World -> a * World.      % Defined locally.
%
% FUNCTION  New : One -> World.              % Used by system wrapper.

DEFINE    EOF = -1.

CONSTRUCT InputStream/0, OutputStream/0, Result/0, IO/1.

FUNCTION  In : InputStream -> Result;
          Out : OutputStream -> Result;
          NotFound : One -> Result;
          StdIn : One -> InputStream;
          StdOut, StdErr : One -> OutputStream;
          FindInput : String -> IO(Result);
          FindOutput : String -> IO(Result);
          Get : InputStream -> IO(Integer);
          Put : OutputStream * Integer -> IO(One);
          WriteString : OutputStream * String -> IO(One);
          Unit : a -> IO(a);
          >>> : IO(a) * (a -> IO(b)) -> IO(b);
          Done : IO(One);
          >> : IO(a) * IO(b) -> IO(b).
```

IO transformers are the accessible building blocks of the monadic approach to input/output. Corresponding to IO operations such as reading or writing a file, there are IO transformers. Furthermore, the IO transformers can only be combined in certain ways which are flexible enough to allow programmers to easily program the standard kinds of things one needs for input/output, but are restricted enough to ensure declarativeness via a single-threadedness property.

For example, reading a file is handled by the function

```
Get : InputStream -> IO(Integer)
```

which takes an input stream as an argument and returns the IO transformer which takes a world as input and returns a new world in which the file pointer of the file corresponding to the input stream has been advanced one character and the ASCII code of the character to which the file pointer originally pointed is returned as the value. As a side-effect, the actual file in the file system has a character read from it. So far, it is not clear how one arranges for the IO transformer to be applied to the current state of the world – this will become clear shortly.

IO transformers can be composed by the function `>>>`. This takes an IO transformer of type `IO(a)` and a function from `a` to `IO(b)` as input and returns an IO transformer of type `IO(b)` as a

result. This latter IO transformer behaves as follows. Given a state of the world it first applies the first argument of `>>>` to this world to produce an intermediate state of the world and a value `v`, say, of type `a`. The second argument is then applied to the value `v` to produce an IO transformer of type `IO(b)` which is then applied to the intermediate state of the world to produce a final state of the world and some value of type `b`. It turns out to be very useful for the second IO transformer applied to be dependent on the value returned by the first. Naturally, there is a unit element corresponding to composition of transformers. This is given by the function

```
Unit : a -> IO(a)
```

which takes a value of type `a` and returns the IO transformer which leaves the state of the world unchanged and returns this value.

Special cases of `>>>` and `Unit` are also useful. The function

```
>> : IO(a) * IO(b) -> IO(b)
```

is similar to `>>>`, except that the value returned by the first transformer is discarded and the second transformer does not depend on this value. Similarly,

```
Done : IO(One)
```

is the identity transformer which leaves the state of the world unchanged and returns the empty tuple as value.

As an illustration of the use of monadic IO, consider the module `Copy` below. The main function in `Copy` is `CopyFile` which takes as input a file to be read and another file to be written and returns an IO transformer which opens the first file for input, opens the second file for output, and then copies the contents of the first file to the second.

But `CopyFile` needs a state of the world in order to do anything at all. So from where does it get this? Certainly not from the program which has no (direct) access to items of type `World`. What happens is that the system itself provides a *system wrapper* in the form of a call to the function `New` which creates an initial world that is given to the programmer's IO transformer as an argument and around all of this a call to `Fst` to return just the value returned by the transformer (discarding the new world created). In other words, a call such as

```
CopyFile("file1", "file2")
```

is changed into

```
Fst(CopyFile("file1", "file2")(New))
```

by the system wrapper. In this form, the call can now proceed, having a world to which the IO transformer can be applied. When the call is made, as a side-effect, the contents of `file1` will be copied into `file2` and the empty tuple will be returned as the value of the call.

This still leaves some mysteries. How can `New` be implemented, for example? It would be disastrous if one really had to create an actual representation of the state of the file system to implement `New`! In fact, the implementation of the type `World` can be faked. It is merely necessary for `New` to return some constant as a token to instantiate the input argument of the IO transformer. Once this argument is instantiated, the transformer can be run, side-effecting the file system, and returning a new state of the world which is simply this same token again. If there are several transformers composed together, this token gets passed from world argument to world argument, firing up one IO transformer after another in a single-threaded way. The restricted form of IO

```
MODULE    Copy.

IMPORT    IO.

FUNCTION  CopyFile : String * String -> IO(One).

CopyFile(s, t) =>
    FindInput(s) >>>
    LAMBDA [n] (FindOutput(t) >>> LAMBDA [m] CopyStream(n, m)).

FUNCTION  CopyStream : Result * Result -> IO(One).

CopyStream(In(n), Out(m)) =>
    Get(n) >>>
    LAMBDA [c] (IF c = EOF
                THEN
                  DoneIO
                ELSE
                  Put(m, c) >> CopyStream(In(n), Out(m))).
```

transformers and the restricted way they can be composed ensures the single-threadedness of the sequence of world arguments and allows the function `New` to be faked in this way. At the same time, all the functions involved are declarative. One can imagine the type `World` actually is the type of a representation of the file system and the function `New` is actually implemented to give an initial state of the file system. Then all the functions in the module `IO` do have fully declarative readings.

These monadic ideas have also been applied by functional programmers to other situations where state is involved, for example, array operations. But it seems that the ideas work best for the case of IO, partly because the restrictions of the monadic style still easily allow for the kind of things one wants to program for input/output. In other situations, the monadic style seems to get a little too much in the way.

Chapter 5

Declarative Debugging

In this chapter, I describe the debugging facilities provided for Escher.

5.1 Introduction

One of the substantial advantages of declarative programming languages is the possibility of employing declarative debugging for repairing incorrect programs. Declarative debugging was introduced (under the name algorithmic debugging) by Shapiro in 1983 in [18] and was studied by a number of authors in subsequent years. (A fairly complete bibliography up to 1987 is given in [13].) More recently, there have been several conferences on debugging at which further developments of the declarative approach have been presented.

The basic idea of declarative debugging (at least in the reconstruction in [13] of Shapiro's original framework) is that to debug an incorrect program, all a programmer needs to know is the intended interpretation of the program. In particular, knowledge of the procedural behaviour of the system running the program is unnecessary. What happens is that the programmer gives a symptom of a bug to the debugger which then proceeds to ask a series of questions about the validity of expressions in the intended interpretation, finally presenting an incorrect statement to the programmer. The terminology used is that an *oracle* answers the queries about the intended interpretation. The oracle is typically the programmer, but it could also be a formal specification of the program, or some combination of both. The approach only applies to *incorrect* programs, that is, those for which the intended interpretation is not a model of the program. Thus other kinds of errors such as infinite loops, deadlocks, or flounders have to be handled by more procedural methods. However, the class of errors associated with an incorrect program is certainly the largest such class, so that declarative debugging does address the major aspect of the debugging problem. Thus, overall, declarative debugging is an extremely attractive approach to debugging.

This last claim makes the obvious lack of practical success of declarative debugging something that needs to be explained! In fact, it's failure can be largely put down to the non-declarative nature of widely-used logic programming languages. My own experience in the mid 1980's suggested strongly that, however attractive declarative debugging may be, it certainly does not work well for Prolog, for example. The main difficulty by far in this regard is handling the non-declarative features of Prolog. Thus, not surprisingly, the major prerequisite for declarative debugging to be successful is to apply it to a (sufficiently) declarative language! Escher is such a language.

This chapter investigates a framework for declarative debugging in Escher. As well as presenting the keys concepts and debugging algorithm, I give examples of the use of an implementation of these ideas. One outstanding feature of the Escher debugging framework is its simplicity. Compared

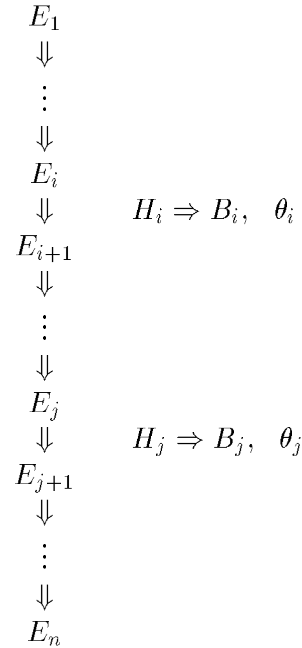
with the debugging framework for Prolog-like languages based on SLDNF-resolution given in [13], the framework presented here is much simpler. The main reasons for the simplicity of the Escher framework are the use of equations rather than implicational formulas for statements, the single computation path rather than an (explicit) search tree, and the avoidance here of negation as failure.

5.2 Principles of Debugging

Throughout this chapter, a "program" will mean an Escher program.

Definition Let P be a program and I the intended interpretation for P. Then P is *incorrect* if I is not a model for P. A statement in P is *incorrect* if it is not valid in I.

If a program is incorrect, then some statement in the program is incorrect. The task of (declarative) debugging is to take an incorrect program and locate an incorrect statement in the program. The basic algorithm for achieving this was introduced by Shapiro in [18] and is called *divide-and-query*. To understand how divide-and-query works, consider a typical computation.



In this computation, E_1 is the goal, E_n is the answer, and $H_i \Rightarrow B_i$ is a statement used in a function call in the expression E_i with associated substitution θ_i. Now suppose the computation is buggy. This will show up because of a bug symptom, which is formalized as follows.

Definition Let C be a computation with goal E_1 and answer E_n. Then C is *incorrect* if $E_1 = E_n$ is not valid in the intended interpretation.

Informally, divide-and-query proceeds as follows. Consider the computation with goal E_1 and answer E_n. Having confirmed with the oracle that the computation is indeed incorrect, the debugging algorithm then chooses the expression $E_{\lfloor (n+1)/2 \rfloor}$ at the midpoint of the computation and asks the oracle whether $E_1 = E_{\lfloor (n+1)/2 \rfloor}$ is valid (in the intended interpretation). If the answer is no, the algorithm discards the bottom half of the computation and continues with the segment from E_1 to $E_{\lfloor (n+1)/2 \rfloor}$. If the answer is yes, then the algorithm discards the top half of the computation and continues with the segment from $E_{\lfloor (n+1)/2 \rfloor}$ to E_n. Eventually, this process ends with the

identification of an incorrect statement. The number of oracle queries required is logarithmic in the length of the computation.

There are several important optimizations of this basic algorithm. The first is to exploit the fact that statements in system modules are correct. Hence the algorithm can ignore steps in the computation which use statements from a system module. This leads to the definition of the following concept.

Definition Let C be a computation with goal E_1. The *debugging trace* is the subsequence of the computation formed from E_1 together with all expressions E_i in the computation which resulted from the (direct) use of a statement from a user module.

In the debugging trace below, E_1 is the goal and E'_{i+1} is the expression which resulted from the function call using the (user) statement $H'_i \Rightarrow B'_i$ with associated substitution θ'_i.

$$
\begin{array}{cc}
E_1 & \\
E'_2 & (H'_1 \Rightarrow B'_1, \quad \theta'_1) \\
E'_3 & (H'_2 \Rightarrow B'_2, \quad \theta'_2) \\
\vdots & \vdots \\
E'_{m+1} & (H'_m \Rightarrow B'_m, \quad \theta'_m)
\end{array}
$$

From this point onwards, I deal with debugging traces rather than computations. Here is a preliminary version of the debugging algorithm.

Debugging Algorithm (Preliminary Version)

Input: The debugging trace $E_1, \ldots, E_m$ from an incorrect computation.
Output: An incorrect statement.

begin
    $i := 1;$
    $j := m;$
    while $i + 1 < j$ **do**
    begin
        if $E_i = E_{\lfloor (i+j)/2 \rfloor}$ is valid in the intended interpretation
        then $i := \lfloor (i+j)/2 \rfloor$
        else $j := \lfloor (i+j)/2 \rfloor$
    end;
    incorrect_statement := the statement used to derive E_j
end

In fact, the algorithm can also return an instance of a statement which is not valid in the intended interpretation (the instance of the incorrect statement given by the substitution used at that step).

The next optimization concerns the form of queries presented to the oracle. Often expressions in a computation can be large and complex. Hence, in the case when the oracle is the programmer, some effort must be put into making oracle queries as simple as possible. One obvious idea is to exploit the fact that the programmer will surely know the denotation in the intended interpretation of the goal E_1 and that it is possible to replace the query about $E_i = E_{\lfloor (i+j)/2 \rfloor}$ with one about $E_1 = E_{\lfloor (i+j)/2 \rfloor}$. This idea cuts the complexity of oracle queries dramatically. The last optimization

is to simplify the expression $E_{\lfloor(i+j)/2\rfloor}$ using only statements in system modules before presenting it to the oracle. This often reduces the complexity of oracle queries as well. With these optimizations, I now give the final form of the debugging algorithm.

Debugging Algorithm

Input: The debugging trace $E_1, \ldots, E_m$ from an incorrect computation.
Output: An incorrect statement.

begin
  $i := 1$;
  $j := m$;
  while $i + 1 < j$ **do**
  begin
    $F :=$ simplified form of $E_{\lfloor(i+j)/2\rfloor}$;
    if $E_1 = F$ is valid in the intended interpretation
    then $i := \lfloor(i+j)/2\rfloor$
    else $j := \lfloor(i+j)/2\rfloor$
  end;
  incorrect_statement := the statement used to derive E_j
end

This algorithm has some important properties which are easily established.

Theorem Under the assumption that the oracle is perfect, the debugging algorithm has the following properties.

1. It always terminates.

2. It is sound and complete (that is, a statement returned by the algorithm is incorrect; and, if the debugging trace comes from an incorrect computation, the algorithm will return an incorrect statement).

3. If there are m expressions in the debugging trace of the computation, then the number of oracle queries required is bounded by $\lceil log_2 m \rceil$.

5.3 Debugging Example

As an illustration of the use of the debugging algorithm, consider the module `Eratosthenes` below. This is intended to compute the list of prime numbers up to some given number. However, there is a bug in the last statement where the `THEN` and `ELSE` parts of a conditional have been interchanged. As a result, the goal

```
Primes(5, x)
```

reduces incorrectly to the answer

```
x = [2, 4].
```

Here then is the listing of a session using the debugger to locate the incorrect statement. The `n` or `y` after a colon is an oracle answer, where `y` indicates that the equation is valid in the intended interpretation and `n` indicates that it is not.

```
MODULE   Eratosthenes.

IMPORT   Integers, Lists.

FUNCTION Primes : Integer * List(Integer) -> Boolean.
MODE     Primes(NONVAR, _).
Primes(limit, ps) =>
    SOME [is] (Range(2, limit, is) & Sift(is, ps)).

FUNCTION Range : Integer * Integer * List(Integer) ->  Boolean.
MODE     Range(NONVAR, NONVAR, _).
Range(low, high, l) =>
    IF low =< high
    THEN
      SOME [rest] (Range(low + 1, high, rest) & l = [low | rest])
    ELSE
      l = [].

FUNCTION Sift : List(Integer) * List(Integer) -> Boolean.
MODE     Sift(NONVAR, _).
Sift([], l) =>
    l = [].
Sift([i | is], l) =>
    SOME [new, ps] (Remove(i, is, new) & Sift(new, ps) & l = [i | ps]).

FUNCTION Remove : Integer * List(Integer) * List(Integer) -> Boolean.
MODE     Remove(NONVAR, NONVAR, _).
Remove(x, [], l) =>
    l = [].
Remove(x, [i | is], l) =>
    SOME [new] (Remove(x, is, new) &
                IF i Mod x = 0
                THEN
                  l = [i | new]     % should be  l = new
                ELSE
                  l = new).         % should be  l = [i | new]
```

```
Debug("Eratosthenes", "Primes(5, x)").

Primes(5, x)
 =
x = [2, 4] ?
|: n


Primes(5, x)
 =
SOME [new_7, ps_7] ((Remove(2, [3, 4, 5], new_7) &
          (Sift(new_7, ps_7) & (x = [2 | ps_7])))) ?
|: y


Primes(5, x)
 =
SOME [new_7, ps_7] ((SOME [new_9]
  ((Remove(2, [], new_9) & (new_7 = [4 | new_9]))) &
          (Sift(new_7, ps_7) & (x = [2 | ps_7])))) ?
|: n


Primes(5, x)
 =
SOME [new_7, ps_7] ((Remove(2, [4, 5], new_7) &
          (Sift(new_7, ps_7) & (x = [2 | ps_7])))) ?
|: n


Incorrect statement instance:

Remove(2, [3, 4, 5], new_7) =>
    SOME [new_8] ((Remove(2, [4, 5], new_8) &
                  (IF ((3 Mod 2) = 0)
                   THEN (new_7 = [3 | new_8])
                   ELSE (new_7 = new_8))))

Corresponding statement:

Remove(x, [i | is], l) =>
    SOME [new] ((Remove(x, is, new) &
                (IF ((i Mod x) = 0)
                 THEN (l = [i | new])
                 ELSE (l = new))))
```

5.4 Practicalities of Declarative Debugging

Unfortunately, to build a practical debugging system, much more needs to be done than simply implement the algorithm of the previous section. The problem is that, while a programmer may, *in principle*, know the intended interpretation, he or she may not be able, *in practice*, to answer an oracle query, simply because an expression in the query may be very large and/or complex. Actually, this problem is not unique to declarative debugging – every debugging method suffers from it. However, in declarative debugging, there is certainly a lot at stake when an oracle query is answered, as giving the wrong answer is likely to lead the debugger astray. Thus, in this section, I discuss some of the issues, none of which are conceptually difficult, that need to be addressed to build a practical declarative debugger.

The main difficulties are centered around the "presentation" problem, which is concerned with finding ways of presenting potentially large and complex oracle queries in such a way that the programmer is likely to be able to answer correctly. I now discuss various aspects of the presentation problem.

The first is that one of the expressions in the query may be very large. If the query concerns the equation $E_1 = F$, where E_1 is the goal, then typically it will be F that is large. This is probably the most troublesome aspect of the presentation problem and there may be little that can be done about it. One possibility is for the programmer to head the problem off altogether by finding a "small" goal for which the bug manifests itself. This is good debugging practice anyway. Failing this, the debugger may be able to break the query up into smaller subqueries, the answers to which can answer the whole query. This kind of thing was implemented for the top-down debugger reported in [13], but it is not clear to me how it might be applied to the different context of the debugger of this chapter.

The next issue concerns the use of abstract data types (ADT's). The problem here is that a subexpression in the query may involve functions hidden inside an ADT (in the local part of a system module, say). Since there is no way of directly displaying such an expression, the query in which it appears cannot even be properly (directly) presented. The solution to this kind of problem is rather straightforward – for each system ADT, a method has to be implemented for displaying in a suitable format expressions of that type to the user. For user ADT's, the language can provide a suitable mechanism for allowing a programmer to specify how a hidden expression should be displayed. A typical situation where such a problem arises is for meta-programming. Here the object program is represented by its ground representation via the ADT `Program`, which uses functions entirely hidden from the programmer. In such a case, the difficulty is overcome by displaying the object program in source form, which the programmer will certainly understand.

One useful technique which a debugger can employ is to build up a partial knowledge of the intended interpretation from the programmer's answers to oracle queries, so that it can avoid repeating queries. Typically, a large program is developed over a period of time, so one can easily imagine the debugger recording answers to oracle queries to build a more and more complete picture of the intended interpretation. If this idea is going to be practical, the debugger must also easily allow a programmer to update this partial intended interpretation, either because the programmer realised afterwards that an answer to an oracle query was wrong or because the data structures, and hence the constructors and functions, employed by the program somehow changed.

Finally, I emphasize again that declarative programming can only cope with programs that have some declarative error (that is, are not correct) and hence one needs other techniques to deal with procedural errors, such as infinite loops, deadlocks, and flounders.

Chapter 6

Example Programs

In this chapter, I give a variety of example programs to further illustrate the Escher programming style.

6.1 Binary Search Trees

The module `BinaryTrees` contains some functions which operate on binary search trees, the nodes of which contain strings. The function `Root` maps a (binary search) tree to its root, `Left` maps a tree to its left subtree, and `Right` maps a tree to its right subtree. `IsEmpty` maps a tree to `True` if the tree is empty; otherwise, it maps to `False`. `Insert` maps a string and a tree to the tree obtained by inserting this string into this tree. `Delete` maps a string and a tree to the tree obtained by deleting this string from this tree. `IsIn` maps a string and a tree to `True` if the string is in the tree; otherwise, it maps to `False`. `Greatest` maps a tree to the (lexically) greatest string appearing in it and `Smallest` maps a tree to the smallest string appearing in it.

For the program {`BinaryTrees`, `Text`, `Lists`, `Integers`, `Booleans`}, the goal

```
Left(Tree(Tree(Tree(Empty, "F", Empty), "G", Tree(Empty, "I", Empty)), "S", Empty))
```

reduces to the answer

```
Tree(Tree(Empty, "F", Empty), "G", Tree(Empty, "I", Empty)),
```

the goal

```
Insert("I",Insert("F",Insert("G", Insert("S",Empty))))
```

reduces to the answer

```
Tree(Tree(Tree(Empty, "F", Empty), "G", Tree(Empty, "I", Empty)), "S", Empty),
```

the goal

```
Smallest(Tree(Tree(Tree(Empty, "F", Empty), "G", Tree(Empty, "I", Empty)), "S",
Empty))
```

reduces to the answer

```
"F",
```

the goal

```
IsIn("T", Tree(Tree(Tree(Empty, "F", Empty), "G", Tree(Empty, "I", Empty)), "S",
Empty))
```

reduces to the answer

```
False,
```

and the goal

```
Delete("F", Tree(Tree(Tree(Empty, "F", Empty), "G", Tree(Empty, "I", Empty)), "S",
Empty))
```

reduces to the answer

```
Tree(Tree(Empty, "G", Tree(Empty, "I", Empty)), "S", Empty).
```

```
MODULE    BinaryTrees.

IMPORT    Text.

CONSTRUCT BinaryTree/0.

FUNCTION  Empty : One -> BinaryTree;
          Tree : BinaryTree * String * BinaryTree -> BinaryTree.

FUNCTION  Root : BinaryTree -> String.
MODE      Root(NONVAR).
Root(Empty) =>
    Error("Error: Root has empty tree as argument", Root(Empty)).
Root(Tree(left, item, right)) =>
    item.

FUNCTION  Left : BinaryTree -> BinaryTree.
MODE      Left(NONVAR).
Left(Empty) =>
    Error("Error: Left has empty tree as argument", Left(Empty)).
Left(Tree(left, item, right)) =>
    left.

FUNCTION  Right : BinaryTree -> BinaryTree.
MODE      Right(NONVAR).
Right(Empty) =>
    Error("Error: Right has empty tree as argument", Right(Empty)).
Right(Tree(left, item, right)) =>
    right.

FUNCTION  IsEmpty : BinaryTree -> Boolean.
MODE      IsEmpty(NONVAR).
IsEmpty(Empty) =>
    True.
IsEmpty(Tree(left, item, right)) =>
    False.

FUNCTION  Insert : String * BinaryTree -> BinaryTree.
MODE      Insert(_, NONVAR).
Insert(item, Empty) =>
    Tree(Empty, item, Empty).
Insert(new_item, Tree(left, item, right)) =>
    IF new_item = item
    THEN
      Tree(left, item, right)
    ELSE
```

```
      IF new_item < item
      THEN
        Tree(Insert(new_item, left), item, right)
      ELSE
        Tree(left, item, Insert(new_item, right)).

FUNCTION  Delete : String * BinaryTree -> BinaryTree.
MODE      Delete(_, NONVAR).
Delete(_, Empty) =>
    Error("Error: Delete has empty tree as argument", Delete(_, Empty)).
Delete(del_item, Tree(left, item, right)) =>
    IF del_item = item
    THEN
       (IF IsEmpty(left)
        THEN
          right
        ELSE
          IF IsEmpty(right)
          THEN
            left
          ELSE
            Tree(left, Smallest(right), Delete(Smallest(right), right)))
    ELSE
      IF del_item < item
      THEN
        Tree(Delete(del_item, left), item, right)
      ELSE
        Tree(left, item, Delete(del_item, right)).

FUNCTION  IsIn : String * BinaryTree -> Boolean.
MODE      IsIn(_,NONVAR).
IsIn(in_item, Empty) =>
    False.
IsIn(in_item, Tree(left, item, right)) =>
    IF in_item = item
    THEN
      True
    ELSE
      IF in_item < item
      THEN
        IsIn(in_item, left)
      ELSE
        IsIn(in_item, right).

FUNCTION  Greatest : BinaryTree -> String.
MODE      Greatest(NONVAR).
Greatest(Empty) =>
```

```
    Error("Error: Greatest has empty tree as argument", Greatest(Empty)).
Greatest(Tree(left, item, right)) =>
    IF IsEmpty(right)
    THEN
      item
    ELSE
      Greatest(right).

FUNCTION  Smallest : BinaryTree -> String.
MODE      Smallest(NONVAR).
Smallest(Empty) =>
    Error("Error: Smallest has empty tree as argument", Smallest(Empty)).
Smallest(Tree(left, item, right)) =>
    IF IsEmpty(left)
    THEN
      item
    ELSE
      Smallest(left).
```

6.2 Laziness

The module `Lazy` below is a small example to illustrate the use of laziness in Escher. The function `First` maps an integer n and a list to the list consisting of the first n elements of this list. The function `From` maps an integer n to the (infinite) list of integers with successive elements n, $n+1$, and so on. For the program {`Lazy`, `Lists`, `Integers`, `Booleans`}, the goal

```
First(4, From(2))
```

reduces to the answer

```
[2, 3, 4, 5].
```

The point of the example, of course, is that Escher only computes enough of the list created by `From` as is essential to reduce the goal.

```
MODULE    Lazy.

IMPORT    Lists.

FUNCTION  First : Integer * List(a) -> List(a).
First(n, x) =>
    IF n = 0
    THEN
      []
    ELSE
      [Head(x) | First(n-1, Tail(x))].

FUNCTION  From : Integer -> List(Integer).
From(n) =>
    [n | From(n+1)].
```

6.3 Map

Module `Map` below contains a function `Map1`, which is a variation of the `Map` function in `Lists`. The function `Map1` maps a function F to the function which maps a list of elements of the domain type of F to the list obtained by applying F to each element of this list. For the program {`Map`, `Lists`, `Integers`, `Booleans`}, the goal

```
Map1(Next)([Mon, Tue])
```

reduces to the answer

```
[Tue, Wed],
```

and the goal

```
Map1(f)([])
```

reduces to the answer

```
[].
```

```
MODULE    Map.

IMPORT    Lists.

CONSTRUCT Day/0.

FUNCTION  Mon, Tue, Wed, Thu, Fri, Sat, Sun : One -> Day.

FUNCTION  Map1 : (a -> b) -> (List(a) -> List(b)).
Map1(f) =>
    LAMBDA [x] (IF Empty(x) THEN [] ELSE [f(Head(x)) | Map1(f)(Tail(x))]).

FUNCTION  Next : Day -> Day.
MODE      Next(NONVAR).
Next(Mon) => Tue.
Next(Tue) => Wed.
Next(Wed) => Thu.
Next(Thu) => Fri.
Next(Fri) => Sat.
Next(Sat) => Sun.
Next(Sun) => Mon.
```

6.4 Relations

The module `Relational` below is the Escher version of a λProlog program [16], which is written in the relational programming style. The function `MapPred` is a relational version of the `Map` function in `Lists`. The function `ForEvery` maps a predicate and a list to `True` if the predicate is true for each element of the list; otherwise, it maps to `False`.

For the program {`Relational`, `Lists`, `Integers`, `Booleans`}, the goal

```
MapPred(Age, [Bob, Sue], x)
```

reduces to the answer

```
x = [24, 23],
```

the goal

```
MapPred(Parent, [Bob, Dick], x)
```

reduces to the answer

```
(x = [John, Kate]) \/
(x = [Dick, Kate]),
```

the goal

```
MapPred(r, [Bob, Sue], [24, 23])
```

reduces to the answer

```
r(Bob, 24) & r(Sue, 23),
```

the goal

```
MapPred(LAMBDA [z] (SOME [x,y] (z = <x,y> & Age(y,x))), [24, 23], w)
```

reduces to the answer

```
(w = [Bob, Sue]) \/
(w = [Bob, Ned]),
```

the goal

```
(LAMBDA [x] Age(x,24))(Bob)
```

reduces to the answer

```
True,
```

the goal

```
ForEvery(LAMBDA [x] Age(x,y), [Ned, Bob, Sue])
```

reduces to the answer

```
False,
```

the goal

```
ForEvery(LAMBDA [x] Age(x,y), [Ned, Sue])
```

reduces to the answer

```
y = 23,
```

and the goal

```
ForEvery(LAMBDA [x] (SOME [y] Age(x,y)), [Ned, Bob, Sue])
```

reduces to the answer

```
True.
```

```
MODULE    Relational.

IMPORT    Lists.

CONSTRUCT Person/0.

FUNCTION  Bob, John, Mary, Sue, Dick, Kate, Ned : One -> Person.

FUNCTION  Parent : Person * Person -> Boolean.
Parent(x,y) =>
    (x=Bob & y=John)  \/
    (x=Bob & y=Dick)  \/
    (x=John & y=Mary) \/
    (x=Sue & y=Dick)  \/
    (x=Dick & y=Kate).

FUNCTION  Age : Person * Integer -> Boolean.
Age(x,y) =>
    (x=Bob & y=24)  \/
    (x=John & y=7)  \/
    (x=Mary & y=13) \/
    (x=Sue & y=23)  \/
    (x=Dick & y=53) \/
    (x=Kate & y=11) \/
    (x=Ned & y=23).

FUNCTION  MapPred : (a * b -> Boolean) * List(a) * List(b) -> Boolean.
MODE      MapPred(_, NONVAR, _).
MapPred(p, [], z) =>
    z = [].
MapPred(p, [x|xs], z) =>
    SOME [y,ys] (p(x,y) & MapPred(p,xs,ys) & z = [y|ys]).

FUNCTION  ForEvery : (a -> Boolean) * List(a) -> Boolean.
MODE      ForEvery(_, NONVAR).
ForEvery(p, []) =>
    True.
ForEvery(p, [x|y]) =>
    p(x) &
    ForEvery(p, y).
```

6.5 Set Processing

Finally, in this chapter, I give some further illustrations of the set-processing capabilities of Escher using the module `SportsDB` from Chapter 4 (listed below). For the program {`SportsDB`, `Sets`, `Integers`, `Booleans`}, the goal

```
Likes(Mary, x)
```

reduces to the answer

```
(x = Cricket) \/
(x = Tennis),
```

the goal

```
(x = {p : Likes(p, s)}) &  (s = Cricket \/ s = Football)
```

reduces to the answer

```
((x = {Mary, Bill}) & (s = Cricket)) \/
((x = {Joe}) & (s = Football)),
```

the goal

```
{Mary, Bill} INTERS {Joe, Bill}
```

reduces to the answer

```
{Bill},
```

the goal

```
{x, Bill} INTERS {Joe, x}
```

reduces to the answer

```
{x},
```

the goal

```
{Bill} MINUS {Joe, Bill}
```

reduces to the answer

```
{},
```

the goal

```
{Mary, x} SUBSET {Joe, Mary}
```

reduces to the answer

```
(x = Joe) \/
(x = Mary),
```

the goal

```
{Mary, Joe} = {x}
```

reduces to the answer

```
False,
```

the goal

```
(x INTERS {Mary}) = ({Fred} INTERS y)
```

reduces to the answer

```
((x INTERS {Mary}) = {}) & ((y INTERS {Fred}) = {}),
```

and the goal

```
({Mary} INTERS x) = x
```

reduces to the answer

```
x SUBSET {Mary}.
```

```
MODULE    SportsDB.

IMPORT    Sets.

CONSTRUCT Person/0, Sport/0.

FUNCTION  Mary, Bill, Joe, Fred : One -> Person;
          Cricket, Football, Tennis : One -> Sport.

FUNCTION  Likes : Person * Sport -> Boolean.
Likes(x, y) =>
       (x = Mary & y = Cricket) \/
       (x = Mary & y = Tennis)  \/
       (x = Bill & y = Cricket) \/
       (x = Bill & y = Tennis)  \/
       (x = Joe  & y = Tennis)  \/
       (x = Joe  & y = Football).
```

Part II

DEFINITION

Chapter 7

Syntax

This chapter will contain an account of the syntax of Escher.

7.1 Notation

7.2 Programs

Chapter 8

Semantics

This chapter will contain an account of the semantics of Escher.

8.1 Declarative Semantics

8.2 Procedural Semantics

Chapter 9

System Modules

This chapter contains the export parts of the Escher system modules, `Booleans`, `Integers`, `Lists`, `Text`, `Sets`, and `IO`.

9.1 Booleans

```
EXPORT    Booleans.

% This module exports the basic types One and Boolean; the functions True,
% False, and equality; the conditional and projection functions; and the
% connectives and (generalised) quantifiers.
%
% It is imported into every module in a program by default (without an explicit
% IMPORT declaration).


CONSTRUCT One/0, Boolean/0.


FUNCTION  True : One -> Boolean.
%
% Truth. This is a free function.


FUNCTION  False : One -> Boolean.
%
% Falsity. This is a free function.


FUNCTION  = : a * a -> Boolean.
%
% Equality.
```

```
FUNCTION  & : Boolean * Boolean -> Boolean.
%
% Conjunction.

FUNCTION  \/ : Boolean * Boolean -> Boolean.
%
% Disjunction.

FUNCTION  -> : Boolean * Boolean -> Boolean.
%
% Implication.

FUNCTION  <- : Boolean * Boolean -> Boolean.
%
% Reverse implication.

FUNCTION  <-> : Boolean * Boolean -> Boolean.
%
% Biconditional. This is a synonym for equality on Boolean types.

FUNCTION  ~ : Boolean -> Boolean.
%
% Negation.

FUNCTION  SIGMA : (a -> Boolean) -> Boolean.
%
% Generalised existential quantifier.
%
% SIGMA LAMBDA [x] F is abbreviated to SOME [x] F.

FUNCTION  PI : (a -> Boolean) -> Boolean.
%
% Generalised universal quantifier.
%
% PI LAMBDA [x] F is abbreviated to ALL [x] F.

FUNCTION  IF_THEN_ELSE : Boolean * a * a -> a.
%
% Conditional function.

FUNCTION  Fst : a * b -> a.
%
% Projection onto first argument of a pair.

FUNCTION  Snd : a * b -> b.
%
% Projection onto second argument of a pair.
```

9.2 Integers

```
EXPORT    Integers.


CONSTRUCT Integer/0.


% FUNCTION  0, 1, 2, ... : One -> Integer.

FUNCTION  = : Integer * Integer -> Boolean.
%
% Equality for integers.

FUNCTION  + : Integer * Integer -> Integer.
%
% Addition.

FUNCTION  - : Integer * Integer -> Integer.
%
% Subtraction.

FUNCTION  - : Integer -> Integer.
%
% Unary minus.

FUNCTION  * : Integer * Integer -> Integer.
%
% Multiplication.

FUNCTION  Div : Integer * Integer -> Integer.
%
% Div.

FUNCTION  Mod : Integer * Integer -> Integer.
%
% Mod.

FUNCTION  ^ : Integer * Integer -> Integer.
%
% Power.
```

```
FUNCTION  =< : Integer * Integer -> Boolean.
%
% Less than or equal to.


FUNCTION  >= : Integer * Integer -> Boolean.
%
% Greater than or equal to.


FUNCTION  < : Integer * Integer -> Boolean.
%
% Less than.


FUNCTION  > : Integer * Integer -> Boolean.
%
% Greater than.
```

9.3 Lists

```
EXPORT    Lists.


IMPORT    Integers.


CONSTRUCT List/1.


% List comprehensions:
%
% [s : x <-- t]   means  Map(LAMBDA [x] s, t)
%
% [s : t ; r]     means  Join([[s : r] : t])
%
% [s : t]         means  IF t THEN [s] ELSE []

FUNCTION  Nil : One -> List(a).
%
% Empty list. This is a free function.


FUNCTION  Cons : a * List(a) -> List(a).
%
% List constructor. This is a free function.


FUNCTION  Member :

   a             % An element.
*  List(a)       % A list.
-> Boolean.      % True if the element is in the list; otherwise, false.

MODE      Member(_, NONVAR).


FUNCTION  Concat :

   List(a)       % A list.
*  List(a)       % A list.
-> List(a).      % The list resulting from the concatenation of the arguments
                 % (in the order they appear).

MODE      Concat(NONVAR, _).
```

```
FUNCTION  Split :

   List(a)        % A list.
*  List(a)        % A list.
*  List(a)        % A list.
-> Boolean.       % True if the first argument is the result of concatenating
                  % the second and third arguments (in the order they appear);
                  % otherwise, false.

MODE      Split(NONVAR, _, _).


FUNCTION  Append :

   List(a)        % A list.
*  List(a)        % A list.
*  List(a)        % A list.
-> Boolean.       % True if the third argument is the result of concatenating
                  % the first and second arguments (in the order they appear);
                  % otherwise, false.

MODE      Append(_, _, _).


FUNCTION  Permutation :

   List(a)        % A list.
*  List(a)        % A list.
-> Boolean.       % True if the second argument is a permutation of the first;
                  % otherwise; false.

MODE      Permutation(NONVAR, _).


FUNCTION  Delete :

   a              % An element.
*  List(a)        % A list.
*  List(a)        % A list.
-> Boolean.       % True if the third argument is a list which results from
                  % deleting an occurrence of the element in the first argument
                  % from the second argument; otherwise; false.

MODE      Delete(_, NONVAR, _).


FUNCTION  DeleteFirst :

   a              % An element.
```

```
*  List(a)       % A list.
*  List(a)       % A list.
-> Boolean.      % True if the third argument is the list which results from
                 % deleting the first occurrence of the element in the first
                 % argument from the second argument; otherwise; false.

MODE      DeleteFirst(NONVAR, NONVAR, _).

FUNCTION  Sorted :

   List(Integer) % A list of integers.
-> Boolean.      % True if the list is (increasingly) sorted; otherwise, false.

MODE      Sorted(NONVAR).

FUNCTION  Sort :

   List(Integer) % A list of integers.
-> List(Integer).% The list consisting of the elements of the list in the
                 % argument (increasingly) sorted.

MODE      Sort(NONVAR).

FUNCTION  Foldr :

   (a * b -> b)  % A function F.
*  b             % An element e.
*  List(a)       % A list [e1,...,em].
-> b.            % If m = 0 then e else F(e1,F(e2,...,F(em,e)...)

MODE      Foldr(_, _, NONVAR).

FUNCTION  Join :

   List(List(a)) % A list of lists.
-> List(a).      % The list resulting from the concatenation of the elements
                 % (in the order they appear) of the first argument.

MODE      Join(_).

FUNCTION  Map :

   (a -> b)      % A function F.
*  List(a)       % A list [e1,...,em].
-> List(b).      % [F(e1),...,F(em)].

MODE      Map(_, NONVAR).
```

```
FUNCTION  Empty :

   List(a)       % A list.
-> Boolean.      % True if the list is empty; otherwise, false.

MODE      Empty(_).

FUNCTION  Head :

   List(a)       % A list.
-> a.            % The first element of the list. (It is an error if the list
                 % is empty.)

MODE      Head(NONVAR).

FUNCTION  Tail :

   List(a)       % A list.
-> List(a).      % The tail of the list. (It is an error if the list is empty.)

MODE      Tail(NONVAR).

FUNCTION  Length :

   List(a)       % A list.
-> Integer.      % The length of the list.

MODE      Length(NONVAR).

FUNCTION  Take :

   Integer       % An integer n.
*  List(a)       % A list.
-> List(a).      % The list resulting from taking the first n elements of
                 % the second argument. (It is an error if n is larger than
                 % the length of the second argument.)

MODE      Take(_, _).

FUNCTION  Drop :

   Integer       % An integer n.
*  List(a)       % A list.
-> List(a).      % The list resulting from dropping the first n elements of
                 % the second argument. (It is an error if n is larger than
                 % the length of the second argument.)

MODE      Drop(_, _).
```

9.4 Text

```
EXPORT    Text.


IMPORT    Integers, Lists.


CONSTRUCT Char/0, String/0.


% Each function (except equality) in this module is moded in such a way that
% each argument must be an actual character or string (as appropriate) before
% the function can be called.


%FUNCTION  Finitely many functions from One to Char which are in one-to-one
%          correspondence with all possible characters.  Syntax: e.g., 'z'.

%FUNCTION  Infinitely many functions from One to String which are in one-to-one
%          correspondence with all possible strings.  Syntax: e.g., "A string".

FUNCTION  = : Char * Char -> Boolean.
%
% Equality for characters.


FUNCTION  = : String * String -> Boolean.
%
% Equality for strings.


FUNCTION  CharToString : Char -> String.
%
% Conversion from character to string of length one.


FUNCTION  StringToChar : String -> Char.
%
% Conversion from string of length one to character.


FUNCTION  =< : Char * Char -> Boolean.
%
% Lexical less than or equal to.
```

```
FUNCTION  >= : Char * Char -> Boolean.
%
% Lexical greater than or equal to.

FUNCTION  < : Char * Char -> Boolean.
%
% Lexical less than.

FUNCTION  > : Char * Char -> Boolean.
%
% Lexical greater than.

FUNCTION  Ord : Char -> Integer.
%
% Character to ASCII code.

FUNCTION  Chr : Integer -> Char.
%
% ASCII code to character.

FUNCTION  ++ : String * String -> String.
%
% String concatenation.

FUNCTION  =< : String * String -> Boolean.
%
% Lexical less than or equal to.

FUNCTION  >= : String * String -> Boolean.
%
% Lexical greater than or equal to.

FUNCTION  < : String * String -> Boolean.
%
% Lexical less than.

FUNCTION  > : String * String -> Boolean.
%
% Lexical greater than.

FUNCTION  StringToAscii : String -> List(Integer).
%
% String to list of ASCII codes.

FUNCTION  AsciiToString : List(Integer) -> String.
%
% List of ASCII codes to string.
```

9.5 Sets

```
EXPORT    Sets.

IMPORT    Integers.

FUNCTION  = : (a -> Boolean) * (a -> Boolean) -> Boolean.
%
% Equality for sets.

FUNCTION  UNION : (a -> Boolean) * (a -> Boolean) -> (a -> Boolean).
%
% Set-theoretic union.

FUNCTION  INTERS : (a -> Boolean) * (a -> Boolean) -> (a -> Boolean).
%
% Set-theoretic intersection.

FUNCTION  MINUS : (a -> Boolean) * (a -> Boolean) -> (a -> Boolean).
%
% Set-theoretic difference.

FUNCTION  SUBSET :  (a -> Boolean) * (a -> Boolean) -> Boolean.
%
% Subset relation.

FUNCTION  SUPERSET :  (a -> Boolean) * (a -> Boolean) -> Boolean.
%
% Superset relation.

FUNCTION  IN : a * (a -> Boolean) -> Boolean.
%
% Set membership.

FUNCTION  Size : (a -> Boolean) -> Integer.
%
% Cardinality of set.
```

```
FUNCTION  Select : (a -> Boolean) -> a.
%
% Select returns the (unique) element in a singleton set.
%
% For an argument which is not a singleton set, the value of Select is
% fixed but unspecified.  Select flounders for such an argument.


FUNCTION  Choice : (a -> Boolean) -> a.
%
% Non-deterministic choice of an element of a (non-empty) set.
% If called with the empty set as argument, an error is generated.
%
% This "function" is not declarative.
```

9.6 IO

```
EXPORT    IO.

IMPORT    Integers, Text.

% CONSTRUCT World/0                         % Declared locally.
%
% DEFINE    IO(a) = World -> a * World.     % Defined locally.
%
% FUNCTION  New : One -> World.             % Used by system wrapper.

DEFINE    EOF = -1.

CONSTRUCT InputStream/0, OutputStream/0, Result/0, IO/1.

FUNCTION  In : InputStream -> Result;
          Out : OutputStream -> Result;
          NotFound : One -> Result;
          StdIn : One -> InputStream;
          StdOut, StdErr : One -> OutputStream.

FUNCTION  FindInput : String -> IO(Result).
%
% Given a file name, FindInput returns the IO transformer which opens the file
% for input.

FUNCTION  FindOutput : String -> IO(Result).
%
% Given a file name, FindOutput returns the IO transformer which opens the file
% for output.

FUNCTION  Get : InputStream -> IO(Integer).
%
% Given an input stream, Get returns the IO transformer which reads (the ASCII
% code of) the next character on the input stream.
```

```
FUNCTION  Put : OutputStream * Integer -> IO(One).
%
% Given an output stream and (the ASCII code of) a character, Put returns the
% IO transformer which writes the character to the output stream.


FUNCTION  WriteString : OutputStream * String -> IO(One).
%
% Given an output stream and a string, WriteString returns the IO transformer
% which writes the string to the output stream.


FUNCTION  Unit : a -> IO(a).
%
% Given a value, Unit returns the IO transformer which leaves the world
% unchanged and returns the value.


FUNCTION  >>> : IO(a) * (a -> IO(b)) -> IO(b).
%
% Composition of IO transformers:
%
% (m >>> k)(w) => k(x)(w')  where  <x, w'> = m(w).


FUNCTION  Done : IO(One).
%
% The IO transformer which leaves the world unchanged and returns the empty
% tuple.


FUNCTION  >> : IO(a) * IO(b) -> IO(b).
%
% Given two IO transformers, >> returns the IO transformer which gives the
% world obtained by applying the second transformer to the world obtained
% from the first transformer, and returns the value of the second transformer.
```

Appendix A

Type Theory

This appendix contains a brief account of the higher-order logic underlying Escher – Church's simple theory of types or, more briefly, type theory. The original account of type theory appeared in [2] and the model theory was given by Henkin in [9]. A more recent and comprehensive account appears in [1]. See also [22].

In fact, the version of type theory given here extends Church's original treatment in that it is polymorphic, many-sorted, and includes product types. The polymorphism introduced is a simple form of parametric polymorphism. A declaration for a polymorphic function is understood as a schema for the declarations of the (monomorphic) functions which can be obtained by instantiating all parameters in the polymorphic declaration with closed types. Similarly, a polymorphic term can be regarded as a schema for a collection of (monomorphic) terms. Furthermore, the model-theoretic semantics of polymorphic type theory can easily be reduced to that of (monomorphic) type theory by adopting this schema view of the polymorphism. Another difference compared to the original formulation of type theory is that the proof theory developed by Church (and others) is modified slightly here to give a more direct form of equational reasoning which is better suited to the application of the logic as a foundation for Escher.

One other possible extension which I have yet to explore fully is to move to an intuitionistic version of the logic, as in [11], for example. In intuitionistic type theory, one interprets a theory in a topos, which is a particular kind of cartesian-closed category that generalizes, and shares key properties with, the category of sets. The effect of such a change would be to extend the range of applications of Escher, as a programmer would no longer be confined to interpreting their programs in a particular topos, the category of sets, but could instead interpret in an arbitrary topos. Since there are many interesting toposes other than the category of sets which could be used to model applications, this extension would increase the expressive power of Escher.

A.1 Monomorphic Type Theory

This section outlines the main concepts of (monomorphic) type theory.

An *alphabet* consist of two sets, a set $\mathcal{C}$ of constructors of various arities and a set $\mathcal{F}$ of functions of various signatures. The set $\mathcal{C}$ always includes the constructors $\mathbf{1}$ and o both of arity 0.[1] The main purpose of having $\mathbf{1}$ is so that "constants" can be given signatures in a uniform way as for

[1] In the model theory, the domain corresponding to $\mathbf{1}$ is some canonical singleton set and the domain corresponding to o is the set containing just *True* and *False*. In an intuitionistic version of the logic, the domain corresponding to o would be some other lattice.

"functions". The constructor o gives the type of propositions. For any particular application, the alphabet is assumed fixed and all definitions are relative to the alphabet.

Types are built up from the set $\mathcal{C}$ of constructors, using the symbols $\rightarrow$ (for function types) and $\times$ (for product types).

Definition A *type* is defined inductively as follows.

1. If c is a constructor in $\mathcal{C}$ of arity n and $\alpha_1, \ldots, \alpha_n$ are types, then $c(\alpha_1, \ldots, \alpha_n)$ is a type. (For $n = 0$, this reduces to a constructor of arity 0 being a type.)

2. If α and β are types, then $\alpha \rightarrow \beta$ is a type.

3. If $\alpha_1, \ldots, \alpha_n$ are types, then $\alpha_1 \times \ldots \times \alpha_n$ is a type. (For $n = 0$, this reduces to $\mathbf{1}$ being a type.)

The $\rightarrow$ is right associative, so that $\alpha \rightarrow \beta \rightarrow \gamma$ means $\alpha \rightarrow (\beta \rightarrow \gamma)$. Throughout, $\mathcal{T}$ denote the set of all types obtained from the alphabet.

Associated with an alphabet is a set of *variables*. More precisely, for each type α, there are denumerably many variables x_α, y_α, z_α, ... of type α.

Definition A *signature* is a type of the form $\alpha \rightarrow \beta$, for some α and β.

The set $\mathcal{F}$ always includes the following functions.

1. $=_\alpha$, having signature $\alpha \times \alpha \rightarrow o$, where α varies over all possible types.

2. $\top$ and $\bot$, having signature $\mathbf{1} \rightarrow o$.

3. $\neg$, having signature $o \rightarrow o$.

4. $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$, and $\leftrightarrow$, having signature $o \times o \rightarrow o$.

5. Σ_α and Π_α, having signature $(\alpha \rightarrow o) \rightarrow o$, where α varies over all possible types.

The terms of type theory are the terms of the simply typed λ-calculus, which are formed by abstraction, application, and tupling from the given set $\mathcal{F}$ of functions and the set of variables.

Definition A *term* is defined inductively as follows.

1. A variable of type α is a term of type α.

2. A function in $\mathcal{F}$ having signature $\alpha \rightarrow \beta$ is a term of type $\alpha \rightarrow \beta$.

3. (Abstraction) If t is a term of type β and x_α is a variable of type α, then $\lambda x_\alpha.t$ is a term of type $\alpha \rightarrow \beta$.

4. (Application) If s is a term of type $\alpha \rightarrow \beta$ and t is a term of type α, then $(s\ t)$ is a term of type β.

5. (Tupling) If $t_1, \ldots, t_n$ are terms of type $\alpha_1, \ldots, \alpha_n$, respectively, for some $n \geq 0$, then $< t_1, \ldots, t_n >$ is a term of type $\alpha_1 \times \ldots \times \alpha_n$.

   If $n = 1$, $< t_1 >$ is defined to be t_1. If $n = 0$, the term obtained is the empty tuple, $<>$, which is a term of type $\mathbf{1}$.

A term of type o is called a *formula.*

An occurrence of a variable x_α in a term is *bound* if it occurs within a subterm of the form $\lambda x_\alpha.t$. Otherwise, the occurrence is free. A variable in a term is *free* if it has a free occurrence. Similarly, a variable in a term is *bound* if it has a bound occurrence. Of course, a variable can be both bound and free in a term. A term is *closed* if it contains no free variable.

I adopt various standard syntactic conventions. Terms of the form $\Sigma_\alpha(\lambda x_\alpha.t)$ are written as $\exists_\alpha x_\alpha.t$ and terms of the form $\Pi_\alpha(\lambda x_\alpha.t)$ are written as $\forall_\alpha x_\alpha.t$. In addition, if t is of type o, the abstraction $\lambda x_\alpha.t$ is written $\{x_\alpha : t\}$ to emphasize its intended meaning as a set. A set abstraction of the form $\{x_\alpha : (x_\alpha = t_1) \vee \ldots \vee (x_\alpha = t_n)\}$ is abbreviated to $\{t_1, \ldots, t_n\}$, where x_α is not free in t_i, $i = 1, \ldots, n$. The notation $s \in t$ means $(t\ \ s)$, where t has type $\alpha \to o$ and s has type α. If F is a function, the application $(F\ \ t)$ is written $F(t)$. Also the term $F(< t_1, \ldots, t_n >)$ is abbreviated to $F(t_1, \ldots, t_n)$, where F is a function. Thus, although all functions are unary, one can effectively use the more common syntax of n-ary functions and I sometimes refer to the "arguments" of a function (rather than the argument). Functions having signature $\mathbf{1} \to \alpha$, for some α, have their argument omitted.

Definition The *language* given by an alphabet consists of the set of all terms constructed from the symbols of the alphabet.

Definition A *theory* is a set of formulas.

Type theory incorporates the three rules of λ-conversion for the simply typed λ-calculus which are α-conversion, β-reduction, and η-reduction. I now define these rules and recall two of their most important properties. Escher makes use of both α-conversion and β-reduction. However, it doesn't use η-reduction at all. Also the two properties are not needed for the foundation of Escher and are only presented here to provide some general background.

A term s is *free for* a variable x_α *in* a term t if, for every subterm $\lambda y_\beta.r$ of t, y_β is free in s implies x_α is not free in r. The notation $\{x_\alpha/t\}$ stands for the substitution in which the variable x_α has the binding t, where x_α and t both have type α. Thus, if s is a term, $s\{x_\alpha/t\}$ is the term obtained from s by simultaneously replacing each free occurrence of x_α in s by t. In the following, α-conversion (resp., β-reduction, η-reduction) is denoted by $\succ_\alpha$ ($\succ_\beta$, $\succ_\eta$).

Definition The rules of λ-conversion are as follows.

1. (α-conversion) If y_α is a variable not occurring in a term t, then $\lambda x_\alpha.t \ \succ_\alpha\ \lambda y_\alpha.(t\{x_\alpha/y_\alpha\})$.
2. (β-reduction) If t is free for x_α in s, then $(\lambda x_\alpha.s\ \ t) \ \succ_\beta\ s\{x_\alpha/t\}$.
3. (η-reduction) If x_α is a variable not free in a term t, then $\lambda x_\alpha.(t\ \ x_\alpha) \ \succ_\eta\ t$.

Let $e[s]$ denote a term with some distinguished occurrence of a subterm s and $e[t]$ the result of replacing this single subterm by the term t having the same type as s. Define $\longrightarrow_\alpha$ as follows. $e[s] \longrightarrow_\alpha e[t]$ if $s \ \succ_\alpha\ t$. Similarly, define $\longrightarrow_\beta$ and $\longrightarrow_\eta$. Define $\longrightarrow_{\beta\eta}$ as $\longrightarrow_\beta \cup \longrightarrow_\eta$. The equivalence relation $\stackrel{*}{\longleftrightarrow}_{\beta\eta}$ is the reflexive, symmetric, and transitive closure of $\longrightarrow_{\beta\eta}$.

Theorem (Strong Normalisation) Every sequence of $\beta\eta$-reductions is finite.

Theorem (Church-Rosser) If $s \stackrel{*}{\longleftrightarrow}_{\beta\eta} t$ for terms s and t, then there exists a term u such that $s \stackrel{*}{\longrightarrow}_{\beta\eta} u$ and $t \stackrel{*}{\longrightarrow}_{\beta\eta} u$.

Consequently, for any term t there exists a unique (up to α-conversion) term t' such that $t \stackrel{*}{\longrightarrow}_{\beta\eta} t'$ with t' in $\beta\eta$-normal form (that is, no β- or η-reduction can be applied to t'). Note that both theorems hold when $\beta\eta$ is replaced by just β or just η.

Next I turn to the model theory of type theory. The key idea, introduced by Henkin in his paper [9] which proved the completeness of type theory, is that of a *general model*, here called an *interpretation*. Such interpretations are a natural generalization of first-order interpretations and are the appropriate concept for capturing the *intended interpretation* of an application.

Definition A *frame* for an alphabet $\mathcal{A}$ is a collection of non-empty sets $\{\mathcal{D}_\alpha\}_{\alpha \in \mathcal{T}}$ (where $\mathcal{T}$ is the set of types obtained from $\mathcal{A}$) such that the following conditions are satisfied.

1. $\mathcal{D}_{\mathbf{1}}$ is some canonical singleton set.
2. $\mathcal{D}_o$ is the set $\{\mathtt{T}, \mathtt{F}\}$.
3. If α has the form $\beta \to \gamma$, for some β and γ, then $\mathcal{D}_\alpha$ is a collection of mappings from $\mathcal{D}_\beta$ to $\mathcal{D}_\gamma$.
4. If α has the form $\alpha_1 \times \ldots \times \alpha_n$, for some $n > 0$, then $\mathcal{D}_\alpha$ is the cartesian product $\mathcal{D}_{\alpha_1} \times \ldots \times \mathcal{D}_{\alpha_n}$.

Definition A *structure* $< \{\mathcal{D}_\alpha\}_{\alpha \in \mathcal{T}}, \mathcal{J} >$ for an alphabet $\mathcal{A}$ consists of a frame and a mapping $\mathcal{J}$ which maps each function in $\mathcal{A}$ having signature $\alpha \to \beta$ to a member of $\mathcal{D}_{\alpha \to \beta}$ (called its *denotation*) such that the following conditions are satisfied.

1. $\mathcal{J}(=_\alpha)$ is the mapping from $\mathcal{D}_\alpha \times \mathcal{D}_\alpha$ into $\{\mathtt{T}, \mathtt{F}\}$ such that
   $\mathcal{J}(=_\alpha)(x, x) = \mathtt{T}$
   $\mathcal{J}(=_\alpha)(x, y) = \mathtt{F}$, if x and y are distinct.
2. $\mathcal{J}(\top)$ is the mapping from $\mathcal{D}_{\mathbf{1}}$ to $\mathcal{D}_o$ which maps the single element of the domain to T. $\mathcal{J}(\bot)$ is the mapping from $\mathcal{D}_{\mathbf{1}}$ to $\mathcal{D}_o$ which maps the single element of the domain to F.
3. The denotation under $\mathcal{J}$ of the connective $\neg$ is given by the following table:

x	$\mathcal{J}(\neg)(x)$
T	F
F	T

4. The denotations under $\mathcal{J}$ of each of the connectives $\wedge$, $\vee$, $\to$, $\leftarrow$, and $\leftrightarrow$ are given by the following table:

x	y	$\mathcal{J}(\wedge)(x,y)$	$\mathcal{J}(\vee)(x,y)$	$\mathcal{J}(\to)(x,y)$	$\mathcal{J}(\leftarrow)(x,y)$	$\mathcal{J}(\leftrightarrow)(x,y)$
T	T	T	T	T	T	T
T	F	F	T	F	T	F
F	T	F	T	T	F	F
F	F	F	F	T	T	T

5. $\mathcal{J}(\Sigma_\alpha)$ is the mapping from $\mathcal{D}_{\alpha \to o}$ to $\mathcal{D}_o$ which maps an element f of $\mathcal{D}_{\alpha \to o}$ to T if f maps at least one element of $\mathcal{D}_\alpha$ to T; otherwise, it maps f to F.
6. $\mathcal{J}(\Pi_\alpha)$ is the mapping from $\mathcal{D}_{\alpha \to o}$ to $\mathcal{D}_o$ which maps an element f of $\mathcal{D}_{\alpha \to o}$ to T if f maps every element of $\mathcal{D}_\alpha$ to T; otherwise, it maps f to F.

$\mathcal{D}_\alpha$ is called the *domain* corresponding to the type α.

Definition An *assignment* into a frame $\{\mathcal{D}_\alpha\}_{\alpha \in \mathcal{T}}$ for an alphabet is a function ϕ from the set of all variables into the (disjoint) union of all the $\mathcal{D}_\alpha$ such that $\phi(x_\alpha) \in \mathcal{D}_\alpha$, for each variable x_α. An assignment into a structure is an assignment into the frame of the structure.

Given an assignment ϕ, a variable x_α, and $d \in \mathcal{D}_\alpha$, let $\phi[x_\alpha/d]$ be the assignment defined by $\phi[x_\alpha/d](x) = \phi(x)$, for all x distinct from x_α, and $\phi[x_\alpha/d](x_\alpha) = d$.

Definition A structure $I = < \{\mathcal{D}_\alpha\}_{\alpha \in \mathcal{T}}, \mathcal{J} >$ is an *interpretation* if there is a binary function $\mathcal{V}$ dependent upon I such that, for each assignment ϕ into I, the following conditions are satisfied.

1. $\mathcal{V}(\phi, t) \in \mathcal{D}_\alpha$, for each term t of type α.
2. $\mathcal{V}(\phi, x_\alpha) = \phi(x_\alpha)$, for each variable x_α.
3. $\mathcal{V}(\phi, F) = \mathcal{J}(F)$, for each function $F \in \mathcal{F}$.
4. $\mathcal{V}(\phi, \lambda x_\alpha.t) =$ the function from $\mathcal{D}_\alpha$ into $\mathcal{D}_\beta$ whose value for each $d \in \mathcal{D}_\alpha$ is $\mathcal{V}(\phi[x_\alpha/d], t)$, where t has type β.
5. $\mathcal{V}(\phi, (s\ t)) = \mathcal{V}(\phi, s)(\mathcal{V}(\phi, t))$.
6. $\mathcal{V}(\phi, < t_1, \ldots, t_n >) = < \mathcal{V}(\phi, t_1), \ldots, \mathcal{V}(\phi, t_n) >$.

If a structure I is an interpretation, then the associated function $\mathcal{V}$ is uniquely determined. The value $\mathcal{V}(\phi, t)$ is called the *denotation* of t with respect to I and ϕ. If t is a closed term, then $\mathcal{V}(\phi, t)$ is independent of ϕ. In this case, the denotation is written as $\mathcal{V}(t)$.

An interpretation for which each $\mathcal{D}_{\alpha \to \beta}$ consists of *all* the mappings from $\mathcal{D}_\alpha$ to $\mathcal{D}_\beta$ is called a *standard* interpretation. In case I is a standard interpretation, the function $\mathcal{V}$ can be *defined* by induction on the structure of terms. It would appear that, in practice, intended interpretations are usually standard.

Definition Let t be a formula, I an interpretation, and ϕ an assignment into I.

1. ϕ *satisfies* t *in* I if $\mathcal{V}(\phi, t) = \mathbf{T}$.
2. t is *satisfiable in* I if there is an assignment which satisfies t in I.
3. t is *valid in* I if every assignment satisfies t in I.
4. A closed formula t is *true in* I if $\mathcal{V}(t) = \mathbf{T}$, and *false in* I if $\mathcal{V}(t) = \mathbf{F}$.
5. t is *valid* (resp., *valid in the standard sense*) if t is valid in every interpretation (every standard interpretation).
6. A *model* for a set S of formulas is an interpretation in which each formula in S is valid.

Definition A theory is *consistent* if it has a model.

A.2 Polymorphic Type Theory

This section contains an account of a polymorphic version of the type theory of the previous section. (Polymorphic) type theory is the higher-order logic which provides the foundation for Escher.

An *alphabet* consists of two sets, a set $\mathcal{C}$ of constructors of various arities and a set $\mathcal{F}$ of functions of various signatures. The set $\mathcal{C}$ always includes the constructors $\mathbf{1}$ and o both of arity 0. Associated with an alphabet is a denumerable set of *parameters* a, b, c, ... and a denumerable set of *variables* x, y, z, ... Parameters are type variables.

Types are built up from the set $\mathcal{C}$ of constructors and the set of parameters, using the symbols $\rightarrow$ and $\times$.

Definition A *type* is defined inductively as follows.

1. A parameter is a type.
2. If c is a constructor in $\mathcal{C}$ of arity n and $\alpha_1, \ldots, \alpha_n$ are types, then $c(\alpha_1, \ldots, \alpha_n)$ is a type. (For $n = 0$, this reduces to a constructor of arity 0 being a type.)
3. If α and β are types, then $\alpha \rightarrow \beta$ is a type.
4. If $\alpha_1, \ldots, \alpha_n$ are types, then $\alpha_1 \times \ldots \times \alpha_n$ is a type. (For $n = 0$, this reduces to $\mathbf{1}$ being a type.)

A type is *closed* if it contains no parameters.

Definition A *signature* is a type of the form $\alpha \rightarrow \beta$, for some α and β.

Definition A function with signature $\alpha \rightarrow \beta$ is *transparent* if every parameter appearing in α also appears in β.

The set $\mathcal{F}$ always includes the following functions (where a is a parameter):

1. $=$, having signature $a \times a \rightarrow o$.
2. $\top$ and $\bot$, having signature $\mathbf{1} \rightarrow o$.
3. $\neg$, having signature $o \rightarrow o$.
4. $\wedge$, $\vee$, $\rightarrow$, $\leftarrow$, and $\leftrightarrow$, having signature $o \times o \rightarrow o$.
5. Σ and Π, having signature $(a \rightarrow o) \rightarrow o$.

I now define the interrelated concepts of a term of some type, the relative type of an occurrence of a variable or function in a term, and the free or bound occurrence of a variable in a term.

Definition A *term* is defined inductively as follows.

1. A variable x is a term of type a, where a is a parameter.

   The occurrence of the variable x has the relative type a in x and the occurrence of the variable x in the term x is free.

2. A function F in $\mathcal{F}$ having signature $\alpha \rightarrow \beta$ is a term of type $\alpha \rightarrow \beta$.

   The function F has relative type $\alpha \rightarrow \beta$ in the term F.

3. (Abstraction) If t is a term of type β and x is a variable which has a free occurrence in t of relative type α in t, then $\lambda x.t$ is a term of type $\alpha \rightarrow \beta$. If x is a variable not occurring free in t, then $\lambda x.t$ is a term of type $a \rightarrow \beta$, where a is a new parameter.

   The occurrence of x immediately after the λ in $\lambda x.t$ has relative type α in $\lambda x.t$, in case x occurs free in t; otherwise, this occurrence of x has relative type a in $\lambda x.t$. Every other occurrence of a variable in $\lambda x.t$ has the same relative type in $\lambda x.t$ as the corresponding occurrence in t. All occurrences of the variable x in the term $\lambda x.t$ are bound. An occurrence of another variable in $\lambda x.t$ is free (resp., bound) if the corresponding occurrence in t is free (resp., bound) in t.

   An occurrence of a function in $\lambda x.t$ has the same relative type in $\lambda x.t$ as the corresponding occurrence in t.

4. (Application) Let s be a term of type $\alpha \rightarrow \beta$ and t a term of type γ. Suppose that the parameters in $\alpha \rightarrow \beta$, taken together with the parameters in the relative types in s of each of the variables in s, and the parameters in γ, taken together with the parameters in the relative types in t of each of the variables in t, are standardized apart. Consider the equation
   $\alpha = \gamma$,
   augmented with equations of the form
   $\rho = \delta$
   for each variable having a free occurrence in both of the terms s and t and where the variable has relative type ρ in s and δ in t. Then $(s\ t)$ is a term if and only if this set of equations has a most general unifier θ, say. In this case, $(s\ t)$ has type $\beta\theta$.

   An occurrence of a variable in $(s\ t)$ has relative type $\sigma\theta$ in $(s\ t)$ if the corresponding occurrence in s or t has relative type σ. An occurrence of a variable in $(s\ t)$ is free (resp., bound) if the corresponding occurrence in s or t is free (resp., bound).

   An occurrence of a function in $(s\ t)$ has relative type $\sigma\theta \rightarrow \delta\theta$ in $(s\ t)$ if the corresponding occurrence in s or t has relative type $\sigma \rightarrow \delta$.

5. (Tupling) Let $t_1, \ldots, t_n$ be terms of type $\alpha_1, \ldots, \alpha_n$, respectively, for some $n \geq 0$. Suppose that the parameters of each α_i, taken together with the parameters in the relative types in t_i of each of the variables of t_i, are standardized apart. Consider the set of equations of the form
   $\rho_{i_1} = \rho_{i_2} = \ldots = \rho_{i_k}$
   for each variable having a free occurrence in the terms $t_{i_1}, \ldots, t_{i_k}$ ($\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$, $k > 1$), say, and where the variable has relative type ρ_{i_j} in t_{i_j} ($j = 1, \ldots, k$). Then $< t_1, \ldots, t_n >$ is a term if and only if this set of equations has a most general unifier θ, say. In this case, $< t_1, \ldots, t_n >$ has type $\alpha_1\theta \times \ldots \times \alpha_n\theta$.

   An occurrence of a variable in $< t_1, \ldots, t_n >$ has relative type $\sigma\theta$ in $< t_1, \ldots, t_n >$ if the corresponding occurrence in t_i, for some i, has relative type σ in t_i. An occurrence of a variable in $< t_1, \ldots, t_n >$ is free (resp., bound) if the corresponding occurrence in t_i, for some i, is free (resp., bound).

   An occurrence of a function in $< t_1, \ldots, t_n >$ has relative type $\sigma\theta \rightarrow \delta\theta$ in $< t_1, \ldots, t_n >$ if the corresponding occurrence in t_i, for some i, has relative type $\sigma \rightarrow \delta$ in t_i.

   If $n = 1$, $< t_1 >$ is defined to be t_1. If $n = 0$, the term obtained is the empty tuple, $<>$, which is a term of type $\mathbf{1}$.

A term of type o is called a *formula*.

It can be shown by induction on the structure of a term that an occurrence of a variable x in a term is bound if and only if it occurs within a subterm of the form $\lambda x.t$. Otherwise, the occurrence is free. A variable in a term is *free* if it has a free occurrence. Similarly, a variable in a term is *bound* if it has a bound occurrence. It can also be shown by induction on the structure of a term that all free occurrences of a variable have the same relative type in the term.

Terms of the form $\Sigma(\lambda x.t)$ are written as $\exists x.t$ and terms of the form $\Pi(\lambda x.t)$ are written as $\forall x.t$. In addition, if t is of type o, the abstraction $\lambda x.t$ is written $\{x : t\}$. A set abstraction of the form $\{x : (x = t_1) \vee \ldots \vee (x = t_n)\}$ is abbreviated to $\{t_1, \ldots, t_n\}$, where x is not free in t_i, $i = 1, \ldots, n$. The notation $s \in t$ means $(t \;\; s)$, where t has type $\alpha \rightarrow o$ and s has type α. If F is a function, the application $(F \;\; t)$ is written $F(t)$. Also the term $F(< t_1, \ldots, t_n >)$ is abbreviated to $F(t_1, \ldots, t_n)$, where F is a function. Functions having signature $\mathbf{1} \rightarrow \alpha$, for some α, have their argument omitted.

Definition The *language* given by an alphabet consists of the set of all terms constructed from the symbols of the alphabet.

Definition A *theory* is a collection of formulas.

An Escher program is a theory for which each formula is a particular kind of equation.

A polymorphic term can be intuitively understood as representing a collection of (monomorphic) terms. I now make this idea more precise. For this purpose, I will be need to be more careful with the terminology. Alphabets, terms, languages, interpretations, and so on, for polymorphic type theory will often have the qualifier "polymorphic" added, while the analogous concepts from monomorphic type theory will often have the qualifier "monomorphic" added.

Definition Let $\mathcal{A}$ be a polymorphic alphabet. The *underlying alphabet* of $\mathcal{A}$ is the (monomorphic) alphabet $\mathcal{A}^*$ defined as follows.

1. The constructors of $\mathcal{A}^*$ are the constructors of $\mathcal{A}$.

2. For each function F having signature $\alpha \rightarrow \beta$ in $\mathcal{A}$ and closed instance $\delta \rightarrow \sigma$ of $\alpha \rightarrow \beta$, there is a function $F_{\delta \rightarrow \sigma}$ having signature $\delta \rightarrow \sigma$ in $\mathcal{A}^*$.

As usual, there is a set of variables associated with the alphabet $\mathcal{A}^*$. More precisely, for each closed type δ obtained from $\mathcal{A}$, there are denumerably many variables x_δ, y_δ, z_δ, ... in $\mathcal{A}^*$.

Definition Let $\mathcal{L}$ be the language obtained from a polymorphic alphabet $\mathcal{A}$. The *underlying language* of $\mathcal{L}$ is the (monomorphic) language $\mathcal{L}^*$ obtained from $\mathcal{A}^*$.

Definition Let t be a term in a polymorphic language. A *grounding type substitution* is a type substitution which binds all the parameters appearing in relative types in t of variables and functions occurring in t to closed types.

Definition Let t be a term in a polymorphic language $\mathcal{L}$ and Ψ be a grounding type substitution for t. The term t_Ψ in the language $\mathcal{L}^*$ is obtained by replacing all occurrences of symbols in t as follows.

1. For an occurrence in t of a variable x having relative type τ in t, replace x by the variable $x_{\tau\Psi}$.

2. For an occurrence in t of a function F having relative type $\delta \rightarrow \sigma$ in t, replace F by the function $F_{\delta\Psi \rightarrow \sigma\Psi}$.

Now a precise meaning can be given to the intuitive concept of a polymorphic term representing a set of (monomorphic) terms.

Definition Let t be a term in a polymorphic language. The set

$$\{t_\Psi : \Psi \text{ is a grounding type substitution for } t\}$$

is called the *set of (monomorphic) terms underlying* t.

Next I turn to the model theory of polymorphic type theory.

Definition A *frame* (resp., *structure*, *interpretation*, *assignment*) of a polymorphic alphabet $\mathcal{A}$ is a frame (structure, interpretation, assignment) of the underlying alphabet $\mathcal{A}^*$

If t is a term in a polymorphic language, I is an interpretation, and ϕ is an assignment, one can ask what is the denotation of t with respect to I and ϕ. In the context of a polymorphic alphabet, the answer depends on which term t_Ψ underlying t is chosen, as t will have a denotation that (generally) depends upon the choice of grounding type substitution Ψ. Thus, in the polymorphic case, the function $\mathcal{V}$ is a ternary function also taking Ψ as an argument and $\mathcal{V}(\Psi, \phi, t)$ is defined to be $\mathcal{V}(\phi, t_\Psi)$.

The definitions of satisfaction, validity, and so on, are made in the obvious way.

Definition Let t be a formula, I an interpretation, and ϕ an assignment into I.

1. ϕ *satisfies* t *in* I if ϕ satisfies t_Ψ in I, for each grounding type substitution Ψ.
2. t is *satisfiable in* I if t_Ψ is satisfiable in I, for each grounding type substitution Ψ.
3. t is *valid in* I if t_Ψ is valid in I, for each grounding type substitution Ψ.
4. A closed formula t is *true in* I if t_Ψ is true in I, for each grounding type substitution Ψ.
5. A closed formula t is *false in* I if t_Ψ is false in I, for each grounding type substitution Ψ.
6. t is *valid* (resp., *valid in the standard sense*) if t is valid in every interpretation (every standard interpretation).
7. A *model* for a set S of formulas is an interpretation in which each formula in S is valid.

Definition A theory is *consistent* if it has a model.

The rules of λ-conversion can be incorporated into (polymorphic) type theory in a straightforward way. I now give the details of this.

A term s is *free for* a variable x *in* a term t if, for every subterm $\lambda y.r$ of t, y is free in s implies x is not free in r. The notation $\{x/t\}$ stands for the substitution in which the variable x has the binding t. If s is a term, $s\{x/t\}$ is the term obtained from s by simultaneously replacing each free occurrence of x in s by t (whenever $s\{x/t\}$ is a well-defined term).

Definition The rules of λ-conversion are as follows.

1. (α-conversion) If y is a variable not occurring in a term t, then $\lambda x.t \;\succ_\alpha\; \lambda y.(t\{x/y\})$.
2. (β-reduction) If t is free for x in s, then $(\lambda x.s\ t) \;\succ_\beta\; s\{x/t\}$.
3. (η-reduction) If x is a variable not free in a term t, then $\lambda x.(t\ x) \;\succ_\eta\; t$.

Note that each rule is well-defined. For example, for β-reduction, the term $s\{x/t\}$ is well-defined.

Finally, I turn to the procedural aspects of type theory. As mentioned earlier, for the version of type theory adopted here, the standard proof theory is slightly modified in favour of more direct equational reasoning which is better suited to the requirements of Escher. More precisely, taking the presentation of the proof theory as given by Andrews in [1, p. 164] as the standard, the relationship between the two is as follows, In the presentation of Andrews, there are 5 axioms (one of which, the axiom corresponding to β-reduction, is split into 4 parts) and a single inference rule (rule R). The first 3 of these axioms are only partly present in Escher via some statements in the modules `Booleans` and `Sets`. Axiom 4 (the axiom corresponding to the β-reduction rule) appears in `Booleans` and axiom 5 (the axiom of descriptions) appears in `Sets` as the statement defining the function `Select`. Finally, rule R and the function call mechanism of Escher are very similar.

The main technical result needed is that of the soundness of function calls. This is given by the following theorem.

Theorem Let $r[s']$ be a term with some distinguished occurrence of a subterm s' and let $s = t$ be a term. Suppose there exists a substitution θ such that s' is identical to $s\theta$ (modulo renaming of bound variables) and that, for all bindings x/u in θ, u is free for x in s and t. If $s = t$ is valid in some interpretation I, then $r[s'] = r[t\theta]$ is also valid in I (where $r[t\theta]$ is the result of replacing the occurrence of s' in r by $t\theta$).

Proof Note that $s\theta = t\theta$ is valid in I. The result now follows by a straightforward induction on the structure of $r[s']$.

This theorem is applied in the following way. Let I be the intended interpretation of a program. Then each statement in the program is valid in I. Let the term $s = t$ of the theorem be a statement with head s and body t used in some function call. The theorem guarantees that if u is the term to which the function call is applied and v is the term resulting from the function call, then $u = v$ is valid in the intended interpretation.

Appendix B

Local Parts of 3 System Modules

This appendix contains the local parts of the system modules `Booleans`, `Lists`, and `Sets`.

Note that, of these three modules, only the local part of `Lists` can be written completely in the Escher syntax which is available to programmers. The local parts of both `Booleans` and `Sets` rely heavily on mode conditions which cannot be expressed in the Escher mode system. For example, in these modules, mode conditions are often local to a particular statement (in contrast to a `MODE` declaration which is global to the definition of a function) and, in some cases, a statement can only be used if some argument is a variable, a condition which cannot be expressed at all by a `MODE` declaration. Mode conditions in these two modules are given by comments attached to the statements to which the condition applies.

B.1 Booleans

```
LOCAL    Booleans.

%FUNCTION  = : a * a -> Boolean.

x = x => True.

x = y => False.
%
%  where x is freely embedded in y; x is distinct from y.

F(x1,...,xn) = F(y1,...,yn)  =>  (x1 = y1) & ... & (xn = yn).
%
%  where n > 0; F is free and transparent.

<x1,...,xn> = <y1,...,yn>  =>  (x1 = y1) & ... & (xn = yn).

F(x1,...,xn) = G(y1,...,ym)  =>  False.
%
%  where F and G are free; F is distinct from G.
```

```
y = x => x = y.
%
%  where x is a variable and y is not a variable.

%FUNCTION  & : Boolean * Boolean -> Boolean.

(x & y) & z => x & (y & z).

x & (x & y) => x & y.

x & x => x.

True & x => x.

x & True => x.

False & x => False.

x & False => False.

~x & x => False.

x & ~x => False.

y & (x = u) & z  =>  y{x/u} & (x = u) & z{x/u}.
%
%  where x is a variable; x is not free in u; x is free in y or z.

(x \/ y) & (z \/ x) => x \/ (y & z).

(x \/ y) & (x \/ z) => x \/ (y & z).

(x \/ y) & (z \/ y) => y \/ (x & z).

(x \/ y) & (y \/ z) => y \/ (x & z).

%FUNCTION  \/ : Boolean * Boolean -> Boolean.

(x \/ y) \/ z => x \/ (y \/ z).

x \/ (x \/ y) => x \/ y.

x \/ x => x.

True \/ x => True.
```

```
x \/ True => True.

False \/ x => x.

x \/ False => x.

~x \/ x => True.

x \/ ~x => True.


%FUNCTION  -> : Boolean * Boolean -> Boolean.

x -> y => ~x \/ y.


%FUNCTION  <- : Boolean * Boolean -> Boolean.

x <- y => x \/ ~y.


%FUNCTION  <-> : Boolean * Boolean -> Boolean.

x <-> y => x = y.


%FUNCTION  ~ : Boolean -> Boolean.

~False => True.

~True => False.

~~x  => x.

~(x \/ y) => ~x & ~y.

~(x & y) => ~x \/ ~y.

~(x -> y) => x & ~y.

~(x <- y) => ~x & y.

~ ALL [x1,...,xn] x => SOME [x1,...,xn] ~x.


%FUNCTION  SIGMA : (a -> Boolean) -> Boolean.

SOME [x1,...,xn] True => True.
```

```
SOME [x1,...,xn] False => False.

SOME [x1,...,xn] (x \/ y) => (SOME [x1,...,xn] x) \/ (SOME [x1,...,xn] y).

SOME [x1,...,xn] (x & (xi = u) & y)  =>
    SOME [x1,...,xi-1,xi+1,...,xn] (x{xi/u} & y{xi/u}).
%
%  where xi is not free in u.

%FUNCTION  PI : (a -> Boolean) -> Boolean.

ALL [x1,...,xn] x  => ~ SOME [x1,...,xn] ~ x.

%FUNCTION  IF_THEN_ELSE : Boolean * a * a -> a.

IF True THEN x ELSE y => x.

IF False THEN x ELSE y => y.

%FUNCTION  Fst : a * b -> a.

Fst(x, y) => x.

%FUNCTION  Snd : a * b -> b.

Snd(x, y) => y.

% Beta-reduction

((LAMBDA [x] y) z) => y{x/z}
%
%  where z is free for x in y.

% Rewrites for the conditional syntactic sugar.

IF SOME [x1,...,xn] True THEN x ELSE y => SOME [x1,...,xn] x.

IF SOME [x1,...,xn] False THEN x ELSE y => y.

IF SOME [x1,...,xn] (x & (xi = u) & y) THEN z ELSE v  =>
    IF SOME [x1,...,xi-1,xi+1,...,xn] (x{xi/u} & y{xi/u}) THEN z{xi/u} ELSE v.
%
%  where xi is not free in u.

IF SOME [x1,...,xn] (x \/ (xi = u) \/ y) THEN z ELSE v  =>
    SOME [x1,...,xn] ((x \/ (xi = u) \/ y) & z).
```

B.2 Lists

```
LOCAL    Lists.

%FUNCTION  Member : a * List(a) -> Boolean.

%MODE      Member(_, NONVAR).

Member(x, []) =>
    False.

Member(x, [y | z]) =>
    (x = y) \/ Member(x, z).


%FUNCTION  Concat : List(a) * List(a) -> List(a).

%MODE      Concat(NONVAR, _).

Concat([], x) =>
    x.

Concat([x | y], z) =>
    [x | Concat(y, z)].


%FUNCTION  Split : List(a) * List(a) * List(a) -> Boolean.

%MODE      Split(NONVAR, _, _).

Split([], x, y) =>
    x = [] & y = [].

Split([x | y], v, w) =>
    (v = [] & w = [x | y]) \/
    SOME [z] (v = [x | z] & Split(y, z, w)).


%FUNCTION  Append : List(a) * List(a) * List(a) -> Boolean.

Append(u, v, w) =>
    (u = [] & v = w) \/
    SOME [r,x,y] (u = [r | x] & w = [r | y] & Append(x, v, y)).
```

```
%FUNCTION  Permutation : List(a) * List(a) -> Boolean.

%MODE      Permutation(NONVAR, _).

Permutation([], x) =>
    x = [].

Permutation([x | y], w) =>
    SOME [u,v,z] (w = [u | v] & Delete(u, [x | y], z) & Permutation(z, v)).

%FUNCTION  Delete : a * List(a) * List(a) -> Boolean.

%MODE      Delete(_, NONVAR, _).

Delete(x, [], y) =>
    False.

Delete(x, [y | z], w) =>
    (x = y & w = z) \/
    SOME [v] (w = [y | v] & Delete(x, z, v)).

%FUNCTION  DeleteFirst : a * List(a) * List(a) -> Boolean.

%MODE      DeleteFirst(NONVAR, NONVAR, _).

DeleteFirst(x, [], y) =>
    False.

DeleteFirst(x, [y | z], w) =>
    IF x = y
    THEN
      w = z
    ELSE
      SOME [v] (w = [y | v] & DeleteFirst(x, z, v)).

%FUNCTION  Sorted : List(Integer) -> Boolean.

%MODE      Sorted(NONVAR).

Sorted([]) =>
    True.

Sorted([x]) =>
    True.

Sorted([x, y | z]) =>
    x =< y &
    Sorted([y | z]).
```

```
%FUNCTION  Sort : List(Integer) -> List(Integer).

%MODE      Sort(NONVAR).

Sort([]) =>
    [].

Sort([x | y]) =>
    Insert(x, Sort(y)).

FUNCTION  Insert : Integer * List(Integer) -> List(Integer).

MODE      Insert(NONVAR, NONVAR).

Insert(x, []) =>
    [x].

Insert(x, [y | z]) =>
    IF x =< y
    THEN
      [x, y | z]
    ELSE
      [y | Insert(x, z)].

%FUNCTION  Foldr : (a * b -> b) * b * List(a) -> b.

%MODE      Foldr(_, _, NONVAR).

Foldr(f, a, []) =>
    a.

Foldr(f, a, [x | xs]) =>
    f(x, Foldr(f, a, xs)).

%FUNCTION  Join : List(List(a)) -> List(a).

Join(x) =>
    Foldr(Concat, [], x).

%FUNCTION  Map : (a -> b) * List(a) -> List(b).

%MODE      Map(_, NONVAR).

Map(f, []) =>
    [].

Map(f, [x | xs]) =>
    [f(x) | Map(f, xs)].
```

```
%FUNCTION  Empty : List(a) -> Boolean.

Empty(x) =>
    x = [].


%FUNCTION  Head : List(a) -> a.

%MODE      Head(NONVAR).

Head([]) =>
    Error("Error: Head has empty list as argument", Head([])).

Head([x | y]) =>
    x.


%FUNCTION  Tail : List(a) -> List(a).

%MODE      Tail(NONVAR).

Tail([]) =>
    Error("Error: Tail has empty list as argument", Tail([])).

Tail([x | y]) =>
    y.


%FUNCTION  Length : List(a) -> Integer.

%MODE      Length(NONVAR).

Length([]) =>
    0.

Length([x | y]) =>
    1 + Length(y).


%FUNCTION  Take : Integer * List(a) -> List(a).

Take(n, x) =>
    IF n = 0
    THEN
      []
    ELSE
      [Head(x) | Take(n-1, Tail(x))].
```

```
%FUNCTION  Drop : Integer * List(a) -> List(a).

Drop(n, x) =>
    IF n = 0
    THEN
      x
    ELSE
      Drop(n-1, Tail(x)).
```

B.3 Sets

```
LOCAL     Sets.

%FUNCTION  = : (a -> Boolean) * (a -> Boolean) -> Boolean.

x = y  =>  (x SUBSET y) & (y SUBSET x).
%
%   where x and y are sets.

x = (y UNION x)  =>  x SUPERSET y.
%
%   where x is a variable.

x = (x UNION y)  =>  x SUPERSET y.
%
%   where x is a variable.

(y UNION x) = x  =>  x SUPERSET y.
%
%   where x is a variable.

(x UNION y) = x  =>  x SUPERSET y.
%
%   where x is a variable.

(y UNION x) = (z UNION x)  =>  x SUPERSET ((y UNION z) MINUS (y INTERS z)).
%
%   where x is a variable.

(x UNION y) = (z UNION x)  =>  x SUPERSET ((y UNION z) MINUS (y INTERS z)).
%
%   where x is a variable.

(y UNION x) = (x UNION z)  =>  x SUPERSET ((y UNION z) MINUS (y INTERS z)).
%
%   where x is a variable.

(x UNION y) = (x UNION z)  =>  x SUPERSET ((y UNION z) MINUS (y INTERS z)).
%
%   where x is a variable.

s UNION x = u UNION y  =>
   SOME [z] ((x SUPERSET ((u MINUS s) UNION z)) &
             ((y SUPERSET (s MINUS u) UNION z)) &
```

```
             ((x SUBSET (u UNION z)) &
             ((y SUBSET (s UNION z))).
%
%   where x and y are variables; x is distinct from y.

x UNION s = u UNION y  =>
   SOME [z] ((x SUPERSET ((u MINUS s) UNION z)) &
             ((y SUPERSET (s MINUS u) UNION z)) &
             ((x SUBSET (u UNION z)) &
             ((y SUBSET (s UNION z))).
%
%   where x and y are variables; x is distinct from y.


s UNION x = y UNION u  =>
   SOME [z] ((x SUPERSET ((u MINUS s) UNION z)) &
             ((y SUPERSET (s MINUS u) UNION z)) &
             ((x SUBSET (u UNION z)) &
             ((y SUBSET (s UNION z))).
%
%   where x and y are variables; x is distinct from y.

x UNION s = y UNION u  =>
   SOME [z] ((x SUPERSET ((u MINUS s) UNION z)) &
             ((y SUPERSET (s MINUS u) UNION z)) &
             ((x SUBSET (u UNION z)) &
             ((y SUBSET (s UNION z))).
%
%   where x and y are variables; x is distinct from y.

x = (y INTERS x)  =>  x SUBSET y.

x = (x INTERS y)  =>  x SUBSET y.

(y INTERS x) = x  =>  x SUBSET y.

(x INTERS y) = x  =>  x SUBSET y.

(s INTERS x) = (u INTERS x)  =>  x INTERS ((s UNION u) MINUS (s INTERS u)) = {}.
%
%   where x is a variable.

(x INTERS s) = (u INTERS x)  =>  x INTERS ((s UNION u) MINUS (s INTERS u)) = {}.
%
%   where x is a variable.

(s INTERS x) = (x INTERS u)  =>  x INTERS ((s UNION u) MINUS (s INTERS u)) = {}.
```

```
%
%   where x is a variable.

(x INTERS s) = (x INTERS u)  =>  x INTERS ((s UNION u) MINUS (s INTERS u)) = {}.
%
%   where x is a variable.

s INTERS x = u INTERS y  =>
    SOME [z,z1,z2] (x = (z UNION z1) &
                    y = (z UNION z2) &
                    z SUBSET (s INTERS u) &
                    z1 INTERS s = {} &
                    z2 INTERS u = {}).
%
%   where x and y are variables; x distinct from y.


x INTERS s = u INTERS y  =>
    SOME [z,z1,z2] (x = (z UNION z1) &
                    y = (z UNION z2) &
                    z SUBSET (s INTERS u) &
                    z1 INTERS s = {} &
                    z2 INTERS u = {}).
%
%   where x and y are variables; x distinct from y.


s INTERS x = y INTERS u  =>
    SOME [z,z1,z2] (x = (z UNION z1) &
                    y = (z UNION z2) &
                    z SUBSET (s INTERS u) &
                    z1 INTERS s = {} &
                    z2 INTERS u = {}).
%
%   where x and y are variables; x distinct from y.


x INTERS s = y INTERS u  =>
    SOME [z,z1,z2] (x = (z UNION z1) &
                    y = (z UNION z2) &
                    z SUBSET (s INTERS u) &
                    z1 INTERS s = {} &
                    z2 INTERS u = {}).
%
%   where x and y are variables; x distinct from y.


%FUNCTION  UNION : (a -> Boolean) * (a -> Boolean) -> (a -> Boolean).
```

```
{} UNION x  =>  x.

x UNION {}  =>  x.

{x : u} UNION {y : v}  =>  {z : u{x/z} \/ v{y/z}}.
%
%   where z is a new variable.

x UNION (y UNION z)  =>  (x UNION y) UNION z.
%
%   where x and y are sets; z is a variable.

x UNION (y UNION z)  =>  (x UNION z) UNION y.
%
%   where x and z are sets; y is a variable.

(x UNION y) UNION z  =>  (x UNION z) UNION y.
%
%   where x and z are sets; y is a variable.

(x UNION y) UNION z  =>  (y UNION z) UNION x.
%
%   where y and z are sets; x is a variable.


%FUNCTION  INTERS : (a -> Boolean) * (a -> Boolean) -> (a -> Boolean).

{} INTERS x  =>  {}.

x INTERS {}  =>  {}.

{x : u} INTERS {y : v}  =>  {z : u{x/z} & v{y/z}}.
%
%   where z is a new variable.

x INTERS (y INTERS z)  =>  (x INTERS y) INTERS z.
%
%   where x and y are sets; z is a variable.

x INTERS (y INTERS z)  =>  (x INTERS z) INTERS y.
%
%   where x and z are sets; y is a variable.

(x INTERS y) INTERS z  =>  (x INTERS z) INTERS y.
%
%   where x and z are sets; y is a variable.
```

```
(x INTERS y) INTERS z  =>  (y INTERS z) INTERS x.
%
%   where y and z are sets; x is a variable.


%FUNCTION  MINUS : (a -> Boolean) * (a -> Boolean) -> (a -> Boolean).

{} MINUS x  =>  {}.

x MINUS {}  =>  x.

{x : u} MINUS {y : v}  =>  {z : u{x/z} & ~(v{y/z})}.
%
%   where z is a new variable.


%FUNCTION  SUBSET :  (a -> Boolean) * (a -> Boolean) -> Boolean.

x SUBSET {}  =>  x = {}.
%
%   where x is a variable.

x SUBSET y  =>  ALL [z] (z IN x -> z IN y).
%
%   where x is a set.


%FUNCTION  SUPERSET :  (a -> Boolean) * (a -> Boolean) -> Boolean.

{} SUPERSET x  =>  x = {}.
%
%   where x is a variable.

x SUPERSET y  =>  ALL [z] (z IN x <- z IN y).
%
%   where y is a set.


%FUNCTION  IN : a * (a -> Boolean) -> Boolean.

z IN {x : y}  =>  y{x/z}.

x IN (y UNION z)  =>  (x IN y) \/ (x IN z).

x IN (y INTERS z)  =>  (x IN y) & (x IN z).

x IN (y MINUS z)  =>  (x IN y) & ~(x IN z).
```

```
%FUNCTION  Size : (a -> Boolean) -> Integer.

Size(set)  =>
    IF set = {}
    THEN
      0
    ELSE
      (1 + Size(set MINUS {y}))  WHERE  y = Choice(set).


%FUNCTION  Select : (a -> Boolean) -> a.

Select({x : x = t}) =>
    t.


%FUNCTION  Choice : (a -> Boolean) -> a.
%
% Non-declarative.

Choice({x : False}) =>
    Error("Error: Choice has empty set as argument", Choice({x : False})).

Choice({x : x = t}) =>
    t.

Choice({x : (x = t) \/ w}) =>
    t.
```

Bibliography

[1] P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[2] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[3] D. DeGroot and G. Lindstrom, editors. *Logic Programming: Relations, Functions and Equations.* Prentice-Hall, 1986.

[4] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 6, pages 243–320. Elsevier, 1990.

[5] J.H. Fasel, P. Hudak, S. Peyton-Jones, and P. Wadler. Special issue on the functional programming language Haskell. *ACM SIGPLAN Notices*, 27(5), 1992.

[6] A.J. Field and P.G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[7] C.A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[8] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[9] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.

[10] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994. Logic Programming Series.

[11] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, 1986.

[12] J. Launchbury and S.L. Peyton-Jones. Lazy functional state threads. In *Proceedings of ACM Conference on Programming Languages Design and Implementation (PLDI)*, 1994.

[13] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[14] G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD thesis, University of Pennsylvania, 1987.

[15] G. Nadathur and D. Miller. An overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming,* Seattle, pages 810–827. MIT Press, 1988.

[16] G. Nadathur and D.A. Miller. Higher-order logic programming. Technical Report CS-1994-38, Department of Computer Science, Duke University, 1994. To appear in The Handbook of Logic in Artificial Intelligence and Logic Programming, D. Gabbay, C. Hogger, and J.A. Robinson (Eds.), Oxford University Press.

[17] S.L. Peyton-Jones and P. Wadler. Imperative functional programming. In *Proceedings of ACM Symposium on Principle of Programming Languages (POPL)*, pages 71–84, 1993.

[18] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[19] J. A. Thom and J. Zobel. Nu-prolog reference manual, version 1.3. Technical report, Machine Intelligence Project, Department of Computer Science, University of Melbourne, 1988.

[20] D.A. Turner. Functional programs as executable specifications. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54. Prentice-Hall, 1985.

[21] P. Wadler. The essence of functional programming. In *Proceedings of 19th Annual Symposium on Principles of Programming Languages (POPL)*, 1992.

[22] D.A. Wolfram. *The Clausal Theory of Types.* Cambridge University Press, 1993.

System Index

&, 78
*, 79
+, 79
++, 86
-, 79
->, 78
<, 80, 86
<-, 78
<->, 78
=, 77, 79, 85, 87
=<, 80, 85, 86
>, 80, 86
>=, 80, 86
>>, 90
>>>, 90
\/, 78
^, 79
~, 78

Append, 82
AsciiToString, 86

CharToString, 85
Choice, 88
Chr, 86
Concat, 81
Cons, 81

Delete, 82
DeleteFirst, 82
Div, 79
Done, 90
Drop, 84

Empty, 84

False, 77
FindInput, 89
FindOutput, 89
Foldr, 83
Fst, 78

Get, 89

Head, 84

IF_THEN_ELSE, 78
IN, 87
In, 89
INTERS, 87

Join, 83

Length, 84

Map, 83
Member, 81
MINUS, 87
Mod, 79

Nil, 81
NotFound, 89

Ord, 86
Out, 89

Permutation, 82
PI, 78
Put, 90

Select, 88
SIGMA, 78
Size, 87
Snd, 78
Sort, 83
Sorted, 83
Split, 82
StdErr, 89
StdIn, 89
StdOut, 89
StringToAscii, 86
StringToChar, 85
SUBSET, 87
SUPERSET, 87

Tail, 84
Take, 84
True, 77

UNION, 87
Unit, 90

WriteString, 90

General Index

1-import, 31

accessible to, 29, 31
alphabet, 91, 96
answer, 32
arity, 91, 96
assignment, 95, 99

body, 32
body of local definition, 24
`Booleans`, 77, 101
bound occurrence, 93
bound occurrence of a variable, 96
bound variable, 93, 98

call, 20
category, 29
`CLOSED`, 29
closed module, 31
closed term, 93
closed type, 96
composition, 40
consistent, 95, 99
constructor, 91, 96

debugging trace, 53
declarative debugging, 7
declarative programming, 3
declare a symbol, 29, 31
defined function, 18
definition, 18
denotation, 94, 95
depends upon, 32
divide-and-query algorithm, 52
domain, 95

`EXPORT`, 29
export a symbol, 29, 31
export declaration, 29
export language of a module, 32
export part, 29

false, 95, 99
flounder, 22
frame, 94, 99
free for a variable, 93, 99
free function, 18
free occurrence of a variable, 96
free term, 18
free variable, 93, 98
function, 91, 96
function call, 20

general model, 94
generator, 40
goal, 32
goal language of a program, 32
grounding type substitution, 98

head, 32

`IMPORT`, 29
import, 29, 31
import a symbol, 29, 31
import declaration, 29
import from, 31
import via, 31
incorrect computation, 52
incorrect program, 52
incorrect statement, 52
`Integers`, 79
intended interpretation, 4, 94
interpretation, 94, 95, 99
`IO`, 89
IO transformer, 47

language, 93, 98
language of a module, 32
`Lists`, 81, 105
`LOCAL`, 29
local declaration, 29
local definition, 24
local part, 29

local variable, 20

main module, 32
model, 95, 99
`MODULE`, 29
module, 29
module conditions, 31
monadic IO, 47

n-import, 31
non-overlapping statements, 20
non-variable term, 20
normal form, 22

open module, 31
oracle, 51
overloading, 32

parameter, 96
parametric polymorphism, 91
polymorphism, 91
program, 31, 32

qualifier, 40
qualifier of local definition, 24

redex, 20
refer to, 31
rewrite system, 20

satisfiable, 95, 99
satisfies, 95, 99
`Sets`, 87, 110
signature, 91, 92, 96
standard interpretation, 95
standardized qualifier, 24
statement, 18, 32
structure, 94, 99
symbol, 29, 31
system module, 31
system wrapper, 49

term, 92, 96
term of some type, 96
test, 40
`Text`, 85
theory, 93, 98
transparent function, 96
true, 95, 99
type, 92, 96
type of a term, 96
type of a variable in a term, 96
type theory, 15, 91, 96

underlying alphabet, 98
underlying language, 98
user module, 31

valid, 95, 99
valid in the standard sense, 95, 99
variable, 92, 96