Programming in an Integrated Functional and Logic Language

J.W. Lloyd
Department of Computer Science
University of Bristol
Bristol BS8 1UB, UK

Abstract

Escher is a general-purpose, declarative 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, concurrency, and declarative input/output. The main design aim is to combine in a practical and comprehensive way the best ideas of existing functional and logic languages, such as Haskell and Gödel. In fact, Escher uses the Haskell syntax and is most straightforwardly understood as an extension of Haskell. Consequently, this paper discusses Escher from this perspective. It provides an introduction to the Escher language, concentrating largely on the issue of programming style and the Escher programming idioms not provided by Haskell. Also the extra mechanisms needed to support these idioms are discussed.

1 Introduction

Escher[1] is a general-purpose, declarative 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, concurrency, and declarative input/output. The main design aim is to combine

[1]M.C.Escher® is a registered trademark of Cordon Art B.V., Baarn, Nederland. Used by permission. All rights reserved.

in a practical and comprehensive way the best ideas of existing functional and logic languages, such as Haskell [17] and Gödel [8]. Indeed, Escher goes beyond Haskell in its ability to run function calls containing variables, its more flexible handling of the connectives and quantifiers, and its set-processing facilities. Escher goes beyond Gödel in its provision of function definitions, its higher-order facilities, its improved handling of sets, and its provision of concurrency and declarative input/output.

Originally, Escher grew out of the Gödel language by moving to a higher-order logic, using equations instead of clauses as statements, and adding function definitions. However, after several years development, it has become clear that the most enlightening and appropriate way to present Escher is as an extension of Haskell which also allows the traditional logic programming style. Syntactically, the extensions needed by Escher as compared with Haskell are rather modest. However, they imply significant changes at the implementation level, particularly in the underlying abstract machine. The extensions needed to turn a Haskell abstract machine into an Escher abstract machine are discussed in [3]. An implementation of the ideas in [3] based on the Brisk system [9] is currently underway at Bristol.

I discuss here only the key ideas of the proposed extensions. There are still a number of less important points that need addressing before the proposed extensions can become fully compatible with Haskell. Most of these points arise because Escher originally began as a language independent from Haskell and with a different treatment of many issues. However, presenting Escher as an extension of Haskell necessitates changing a number of minor aspects of Escher beyond those I have already dealt with. For example, my preferred view of Escher's underlying logic, which is given in Appendix A, is somewhat different from the corresponding Haskell treatment and hence some modifications to the current version of Escher are needed. The operational semantics, given in Appendix B, has been written only in outline and needs to be made more precise. Also there are a few places where I have ignored Haskell's class system. I intend to deal with each of these issues as the Bristol implementation proceeds and the changes required become clearer.

The most important extension of Escher over Haskell is the ability to reduce expressions which contain variables. This extension, together with careful handling of the quantification of local variables, is sufficient to allow Escher to subsume the traditional logic programming style. In addition, Escher provides sophisticated set-processing facilities. A set in a higher-order logic can be identified with a predicate and hence set-processing facilities can be provided by higher-order functions whose definitions contain lambda expressions in the heads of the equations. The bulk of this paper is concerned with elaborating these extensions of Escher over Haskell and showing their utility for programming.

This paper is written from a functional programming perspective. Essentially, the question I pose and answer is: what has to be added to Haskell so that it can provide the facilities made available by typical logic programming languages? For a functional programmer, I believe this question is interesting for at least two reasons.

First, these extensions provide a functional programmer with new programming idioms. In particular, the provision of sets as a data type improves the expressive power of Haskell. Typically, sets are simulated by lists, but this is not usually satisfactory. Sets and lists are distinct data types with distinct properties. It is usually preferable to use precisely the data type required rather than attempt to approximate by another. An alternative approach is to provide an ADT for sets, where the sets are implemented by lists. There are well-known difficulties with this approach and, furthermore, it doesn't provide set abstractions. However, ultimately, the choice of sets or lists is partly a matter of taste and it is indeed true that all the programs in this paper could, with varying degrees of difficulty, be coded successfully in Haskell.

The second and, I believe, more compelling reason for a functional programmer to be interested in these extensions to Haskell is that they substantially increase the likelihood of *logic* programmers using (or switching to) Haskell. If one can provide the familiar logic programming style, as well as all the other sophisticated features of Haskell, the language becomes a very attractive alternative to the languages currently used by logic programmers. Thus Haskell, extended with these Escher features, could become the first widely-used, integrated declarative language – a *lingua franca* for functional and logic programmers. Furthermore, these extensions open the way to adding constraint-solving capabilities to Haskell, thus greatly increasing the potential for industrial and commercial applications of functional programming.

How should a logic programmer go about trying to appreciate the advantages of what is proposed in this paper? If the programmer knows Haskell, then the only problem is to investigate the extent to which the logic programming style is provided by Escher. Perhaps the main difference between Escher and typical logic programming languages, such as Prolog, is Escher's use of equality at the top level of a statement where Prolog uses implication. This difference means that a little massaging is needed to convert Prolog code into Escher code. However, once some experience is gained in this, a logic programmer will, I believe, come to the conclusion as I have done that equality is easier to work with than implication. In other words, the equations of Escher are simpler and easier to write than the clauses of Prolog. For a logic programmer who doesn't know Haskell or any other modern functional language, the job is harder. However, once the power of types, modules, higher-order functions, and so on, is appreciated, it is very difficult to go back to Prolog!

Much useful background material on the issue of integration including a comprehensive list of references up to 1994 can be found in the survey paper by Hanus [4]. Also a general discussion of the advantages of declarative programming can be found in the first chapter of [13]. The original proposal for Escher appeared in [12]. Some discussion of the motivation for the design of Escher is given in section 2.1 of [13]. More detail concerning the logic underlying Escher appears in Appendix A of [13]. An approach to debugging Escher programs appears in [14]. Also a discussion of concurrency in Escher is given in [11]. An integrated functional logic language called Curry ([5], [6]) is also under development by a group of researchers from both functional and logic programming. It is intended to serve the same role for the declarative programming community as Haskell does for the functional community. Curry has similar facilities to Escher but uses a generalized form of narrowing as the operational semantics, whereas Escher uses rewriting.

The next section introduces the key ideas of the Escher language. The third section contains a variety of programs to illustrate the Escher programming style. The fourth section shows how the extensions proposed here are incorporated into entire programs. The fifth section contains a more detailed discussion of the definitions in the `Booleans` module which contains the extensions of Escher over Haskell concerning the connectives, quantifiers, and sets. The last section contains some conclusions. To make the paper self-contained, Appendix A gives a brief introduction to the logic underlying Escher and a table summarizing the notational conventions, Appendix B contains the operational semantics, and Appendix C contains the `Booleans` module.

2 Elements of Escher

In this section, the basic features of Escher are outlined.

First, I briefly describe the resources provided by the `Booleans` module. This module is presented here separately for reasons of clarity in the exposition, but for the Escher implementation the intention is to absorb this module into the Haskell Prelude. The data constructors `True` and `False` are the truth values. The data constructor `Inc` is used in the extensional representation of sets. The predicate `==` is equality and `/=` is disequality. Next come the connectives, conjunction (`&&`), disjunction (`||`), and negation (`not`). Their respective definitions in `Booleans` capture standard properties of these connectives. Note that `y{x/u}` denotes the result of applying the substitution `{x/u}` to `y`. The connective implication (`==>`) is available for limited use but does not have a definition. After the connectives are Church's generalized quantifiers, `exists` and `forall`. A variety of standard set-processing functions is provided. 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.

Next, I present an Escher program chosen just to illustrate the basic concepts of the language. 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 type of lists is denoted using `[` and `]`. Thus, for this application, typical types are `Bool`, `Day`, `[Day]`, `[[Person]]`, and `([([a], [a])] -> Day) -> Bool`, where `a` is a type variable. In the intended interpretation for this application, the domain corresponding to the type `[Day]`, for example, is the set of all lists of days of the week.

The data constructors for forming lists are `[]` and `:`, the data constructors of type `Day` are

```
Mon, Tue, Wed, Thu, Fri, Sat, Sun
```

and those of type `Person` are

```
Mary, Bill, Joe, Fred.
```

For this application, there are three functions with the following signatures.

```
permute :: ([a], [a]) -> Bool;
concatenate :: ([a], [a]) -> [a];
split :: ([a], [a], [a]) -> Bool;
```

The intended meaning of these functions is as follows. The predicate `permute` maps `(s,t)` to `True` if `s` and `t` are lists such that `s` is a permutation of `t`; otherwise, `permute` maps `(s,t)` to `False`. Given lists `s` and `t`, `concatenate` 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 for the functions `permute`, `concatenate`, and `split`, and is given in the module `Permute`. Note that one could easily rewrite the functions in the module `Permute` in a curried style. Throughout the paper, except where compatibility with Haskell forced the issue, I have deliberately mixed the (functional) curried and (logic) uncurried styles to show both are possible.

The notation `exists \z -> ...` is an extension of the Haskell syntax. Here `exists` is the Church existential quantifier which expects an immediately following λ. The notation

```
exists \u v r -> ...
```

in the definition of `permute` is understood to mean

```
exists \u -> (exists \v -> (exists \r -> ...)).
```

A *definition* of a function consists of one or more equations, which are called *statements*. The top-level equality symbol is denoted by `=` to distinguish it from other occurrences of equality which are denoted by `==`. In general, statements have the form
$h = b$.
Here the *head* h is a term of the form
$f\ t_1 \ldots t_n$
where f is a function, each t_i is a term, and the *body* b is a term. Note that all

```
module Permute(Day(..), Person(..), permute) where {

data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;

data Person = Mary | Bill | Joe | Fred;

permute :: ([a], [a]) -> Bool;
permute([], l) =
    l == [];
permute(h : t, l) =
    exists \u v r -> permute(t, r) && split(r, u, v) &&
                     l == concatenate(u, h : v);

concatenate :: ([a], [a]) -> [a];
concatenate([], x) =
    x;
concatenate(u : x, y) =
    u : concatenate(x, y);

split :: ([a], [a], [a]) -> Bool;
split([], x, y) =
    x == [] &&
    y == [];
split(x : y, v, w) =
    (v == [] && w == x : y) ||
    exists \z -> v == x : z && split(y, z, w);

}
```

local variables in a statement must be explicitly quantified. (A *local variable* is a variable appearing in the body of a statement but not the head.)

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. 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).

Next I discuss function calls. (More details are given in Appendix B.) A redex is a subterm of the form $f\ t_1 \ldots t_n$, for some function f, which is identical to the head of an instance of some statement. Escher selects some redex in the current term on which to make the function call. In a function call, a statement is viewed as a rewrite which behaves as follows. Suppose there is some instance $h = b$ of a statement such that h is identical to a redex r. Then the redex r in the current term is replaced by b to give the next term in the computation.

It is intended that Escher will have a documentation tool which extracts signature and mode information for each function exported from a module and makes this information available to programmers. The mode documentation should indicate the expected usage of each function. For example, consider the predicate `split` in the module `Permute`. The mode documentation should indicate that it is intended that some list be given in the first argument and `split` will return all possible ways of splitting the list. This information can easily be obtained by analysis of the definition of `split`.

The overall view of Escher computations is given in Section 4. Until then, I concentrate mainly on low-level details of computations, which are concerned with reducing terms, called *goal* terms, to "simpler" terms, called *answer* terms. Here are some typical goal terms and their corresponding answer terms for the module `Permute`. The term

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

reduces to

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

The term

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

reduces to

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

The term

```
not(split([Mon, Tue], [Tue], y))
```

reduces to

```
True.
```

Finally, the term

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

reduces to

```
x == [Mon, Tue, Wed]  ||
x == [Tue, Mon, Wed]  ||
x == [Tue, Wed, Mon]  ||
x == [Mon, Wed, Tue]  ||
x == [Wed, Mon, Tue]  ||
x == [Wed, Tue, Mon].
```

Escher has a *rewriting* computational model in which a computation proceeds by choosing at each step a redex in the current term and replacing this redex by an equivalent expression obtained from a statement to give the next term in the computation. Note that Escher computations are demand-driven (or lazy), that is, the evaluation of a subterm is only performed if it is demanded by a higher-level evaluation. Thus, in the examples throughout the paper, I assume that each goal is actually a subterm of a term whose evaluation demands the “full” evaluation of the goal.

As an example, the goal, `concatenate([Mon, Tue], [Wed])`, is reduced by a sequence of rewrites to the term `[Mon, Tue, Wed]`, by means of the computation given in Figure 1. The computation consists of the successive terms produced by function calls, the first term being the goal and the last the answer. The second and third terms in the computation are obtained by using the second statement in the definition of `concatenate`, while the fourth term, which is the answer, is obtained by using the first statement in the definition of `concatenate`. In each term, the redex is underlined. The other three goals for the module `Permute` require the use of statements for some connectives and existential quantification in the module `Booleans`, which will be discussed later.

$$\begin{array}{c}
\underline{\texttt{concatenate([Mon, Tue], [Wed])}} \\
\Downarrow \\
\texttt{Mon : }\ \underline{\texttt{concatenate([Tue], [Wed])}} \\
\Downarrow \\
\texttt{Mon : Tue : }\ \underline{\texttt{concatenate([], [Wed])}} \\
\Downarrow \\
\texttt{[Mon, Tue, Wed]}
\end{array}$$

Figure 1: An Escher computation

By way of contrast to Figure 1, the evaluation of `head(concatenate([Mon, Tue], [Wed]))` only requires that `concatenate([Mon, Tue], [Wed])` be reduced to `Mon :  concatenate([Tue], [Wed])`, at which point `head` can return the answer `Mon`.

Escher computations are sound, in the following sense: If `s` is the goal term and `t` is the answer term of a computation, then `s == t` is a logical consequence of the program. Thus, in the intended interpretation, `s` and `t` have the same value.

Using this result, one can interpret an Escher computation in the following way. Suppose a programmer wants to find the value of some complicated term in the intended interpretation. Since Escher has no direct knowledge of the intended interpretation, it cannot evaluate any term in the intended interpretation. However, it can *simplify* (that is, reduce) the term, so that the evaluation in the intended interpretation can then be easily done by the programmer. This is evident in the above computation – the term `concatenate([Mon, Tue], [Wed])` is simplified to `[Mon, Tue, Wed]` which can be easily evaluated in the intended interpretation. Strictly speaking, this view is also appropriate for arithmetic terms. For example, given the term `3 + 4`, Escher will reply with the term `7`. Formally, it hasn't evaluated `3 +`

`4`, but instead simplified it to `7`. In this case, the distinction 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.

By default, Escher computations for logic programming examples return "all answers". Also they 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 module `Permute`, the term

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

reduces to

```
False.
```

From a programming language perspective, Escher has an unusual treatment of variables in statements, which are regarded as syntactical variables. (This treatment is common in the presentation of logics. For a typical example of this, see [1, page 164].) This comes about because of the particular form certain statements can have and, even though not all statements have this form, I have decided for the sake of simplicity to treat all statements in the same way. In fact, what I have called statements up to now are actually statement *schemas*. In statement schemas, identifiers beginning with a lower-case letter (and not having a signature) are syntactical variables ranging over appropriately typed (object) terms. An *instance* of a statement schema is an (object) formula, called a statement, which is obtained by instantiating syntactical variables by (object) terms (according to certain restrictions given below). One can think of a statement schema as an expression in a meta-language which specifies a (potentially infinite) collection of statements, each obtained as an instance of the statement schema.

There are some conventions regarding the use of syntactical variables.

- Syntactical variables appearing immediately after a λ are restricted to range over (object) variables.

- If a syntactical variable u occurs both inside the scope of some $\lambda\, x$ and also outside the scope of all $\lambda\, x$'s, then u cannot contain x as a free variable.

- $\{t_1, \ldots, t_n\}$ means $\{x \mid (x = t_1) \vee \ldots \vee (x = t_n)\}$, for which x cannot be free in the t_i's. In particular, $\{\}$ means $\{x \mid \mathit{False}\}$ and $\{t\}$ means $\{x \mid x = t\}$.

These conventions has been written somewhat informally because a precise account is rather clumsy. For example, a precise statement of the second convention is as follows. If a syntactical variable occurs both inside the scope of some $\lambda\, x$ and also outside the scope of all $\lambda\, x$'s, then it cannot be instantiated by an (object) term containing as a free variable the (object) variable instantiating x. Throughout the paper, I use the informal presentation.

Here are some examples to illustrate these conventions. First consider the following statement schema from the definition of `split`.

```
split(x : y, v, w) =
    (v == [] && w == x : y) ||
    exists \z -> v == x : z && split(y, z, w);
```

The syntactical variable `x` occurs both inside and outside the scope of the λ in `exists \z`. Consequently, `x` cannot contain `z` as a free variable.

The next statement schema comes from the definition of the function `subset` which has the signature
`subset :: (a -> Bool) -> (a -> Bool) -> Bool`.

```
{x | u || v} ‘subset‘ s  =
    ({x | u} ‘subset‘ s) && ({x | v} ‘subset‘ s);
```

Here the syntactical variable `u` occurs only inside the scope of λ`x`'s. Hence it is possible for `u` to contain `x` as a free variable.

The next statement schema comes from the definition of `&&`.

```
x && (exists \x1 ... xn -> v) && y  =
    exists \x1 ... xn -> x && v && y;
```

Here each `xi` is not free in `x` and `y`.

The following is (a special case of) the last statement schema in the definition of `exists`.

```
exists \x1 -> x && (x1 == u) && y  =  x{x1/u} && y{x1/u};
```

In this case, the `x` and `y` in the body are not regarded as occurrences to which the second convention above is relevant since in both cases they have a substitution applied. In other words, `x{x1/u}` and `y{x1/u}` are treated as indivisible terms. Thus `x` and `y` can, indeed typically do, contain `x1` as a free variable.

Finally, the statement schema

```
card {t} =
    1;
```

means

```
card {x | x == t} =
    1;
```

Here `x` cannot be free in `t`.

Having made clear the meaning of statement schemas, for brevity I will usually call them "statements" in what follows.

3 Programming Examples

This section contains a variety of programs to illustrate the Escher programming style. As the Haskell part of Escher is taken for granted in this paper, the examples all concentrate on the extensions that Escher provides over Haskell, especially the treatment of the connectives and quantifiers, and set-processing.

3.1 Set-processing

First I turn to the set-processing facilities of Escher. As an illustration of this, consider the module `SportsDB`. The resources of the `Booleans` module are made available to `SportsDB` and will be needed to answer the queries below.

For the module `SportsDB`, the term

```
likes(Mary, x)
```

reduces to

```
(x == Cricket) ||
(x == Tennis).
```

The term

```
{s | likes(Fred, s)}
```

reduces to

```
module SportsDB(Person(..), Sport(..), likes) where {

data Person = Mary | Bill | Joe | Fred;

data Sport = Cricket | Football | Tennis;

likes :: (Person, Sport) -> Bool;
likes =
      {(Mary, Cricket),
       (Mary, Tennis),
       (Bill, Cricket),
       (Bill, Tennis),
       (Joe, Tennis),
       (Joe, Football)};

}
```

```
{},
```

since `likes(Fred, s)` reduces to `False`.

The term

```
Fred ‘in‘ ({Joe, Fred} ‘union‘ x)
```

reduces to

```
True.
```

The term

```
forall \y -> y ‘in‘ {Cricket, Tennis} ==> likes(x,y)
```

reduces to

```
(x == Mary) ||
(x == Bill).
```

The term

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

reduces to

```
(x == Mary && y == Joe) ||
(x == Joe && y == Mary).
```

The term

```
(x == {p | likes(p, s)}) &&  (s == Cricket || s == Football)
```

reduces to

```
((x == {Mary, Bill}) && (s == Cricket)) ||
((x == {Joe}) && (s == Football)).
```

The term

```
{Mary, Bill} `inters` {Joe, Bill}
```

reduces to

```
{Bill}.
```

The term

```
{Bill} `minus` {Joe, Bill}
```

reduces to

```
{}.
```

The term

```
{Mary, x} `subset` {Joe, Mary}
```

reduces to

```
(x == Joe) ||
(x == Mary).
```

The term

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

reduces to

```
False.
```

The term

```
power {x | likes(Mary, x)}
```

reduces to

```
{{}, {Tennis}, {Cricket}, {Tennis, Cricket}}.
```

The definition of `power` in `Booleans` is typical of set-processing functions. The first statement covers the case when the top-level function in the body of the set abstraction is `False`, the second when it is `==`, and the third when it is `||`. Note the parallel here with list-processing functions which typically have a statement for the `[]` case and one for the `:` case.

The function `mapset` in `Booleans` is the analogue for sets of the `map` function for lists. For example, if the function `square` is defined by

```
square :: Int -> Int;
square x =
    x * x;
```

then the term

```
mapset square {1, 2, -1}
```

reduces to

```
{1, 4}.
```

An alternative definition of `mapset` to the one given in `Booleans` is

```
mapset f s =
    {y | exists \x -> x `in` s && y == f x}.
```

Consider now the task of computing the cardinality of a (finite) set. As a first try at a definition of the appropriate function `card`, one might write the following definition.

```
card {} =
    0;
card {t} =
    1;
card {x | u || v} =
    card {x | u} + card {x | v};
```

However, this definition is flawed as it doesn't cope with "duplicate" elements. For example, `card {1, 1}` reduces to the answer `2`.

This task exposes a source of difficulty in set processing which is that of handling duplicate elements. Of the many possible ways of handling duplicate elements, I have chosen the following approach for Escher. First, the data constructor

```
Inc :: a -> (a -> Bool) -> (a -> Bool)
```

is included in the `Booleans` module. The meaning of `Inc` is that `Inc x s` is equal to `{x} `union` s`. The constructor `Inc` provides a second representation for sets, called the *extensional* representation. For clarity, the set representation used up to this point is sometimes referred to as the *standard* representation. In the extensional representation, the set `{1, 2, 3}` could be represented by

```
Inc(1, Inc(2, Inc(3, {}))).
```

Essentially, the extensional representation provides a "linear" representation for extensional sets which is convenient for removing duplicates.

In addition, the `Booleans` module provides some functions for dealing with sets in their extensional representation. There is the function `linearise` which converts a standard representation of a set to an extensional representation, the function `remove` which deletes an element from a set, the function `deletedup` which deletes duplicate elements from a set, and the function `delinearise` which converts an extensional representation to a standard representation. Semantically, `linearise`, `delinearise` and `deletedup` are just the identity function. The intention is that, for a set-processing function which needs to detect duplicate elements, one first linearises the set and then applies a suitable version of the function to the linearised representation.

A definition of the function `card` using this approach is given in the module `SetProcessing`. For example, the term

```
card {1, 2, 3, 2}
```

reduces to

```
3.
```

Also the term

```
card {x | likes(Mary, x)}
```

reduces to

```
2.
```

The function `card` has a time complexity which is quadratic in the number of elements, including duplicates, in the set. This is the best one can achieve without having a total ordering on the domain of the elements in the argument to `card`.

Similarly, the function `sum` in `SetProcessing` returns the sum of the elements in a set containing integers. For example, the term

```
sum {1, 2, 3, 2}
```

reduces to

```
6.
```

A criticism that one could make of Escher's approach to set processing is that *two* set representations are required. However, none of the solutions of the duplicate element problem using just the standard representation that are known to me are nearly as satisfactory as the one I have just given, which is both declarative and efficient. Furthermore, I believe programmers will quickly understand the part played by each representation – mostly one uses the standard representation, but for functions like `card` and `sum` one first linearises the set and then applies the function to an extensional representation of the set.

```
module SetProcessing(card, sum) where {

card :: (a -> Bool) -> Int;
card s =
    cardl (linearise s);

cardl :: (a -> Bool) -> Int;
cardl {} =
    0;
cardl (Inc x s) =
    1 + cardl (remove x s);

sum :: (Int -> Bool) -> Int;
sum s =
    suml (linearise s);

suml :: (Int -> Bool) -> Int;
suml {} =
    0;
suml (Inc x s) =
    x + suml (remove x s);

}
```

I conclude this subsection with a remark about Escher's semantics. It may appear to be possible with the facilities introduced so far to write non-deterministic functions in Escher. For example, here is an attempted definition for the function `choice` which non-deterministically chooses an element from a set.

```
choice :: (a -> Bool) -> a;
choice {t} =
    t;
choice {x | (u || v) || w} =
    choice {x | u || (v || w)};
choice {x | (x == t) || u} =
    t.
```

If run, this function *will* choose an element from a set. However, according to the semantics of Escher, it is wrong. The reason is that it is not possible to give the meaning of the function `choice`. That is, it is not possible to given a concrete mapping for `choice` which satisfies the above equations. Unfortunately, this kind of modelling error cannot be detected by the compiler. It is the programmer's responsibility to make sure the intended interpretation is well-defined and is a model for the program.

Of course, one could proceed differently by admitting non-deterministic functions and providing a suitable semantics for them, as is proposed in [6]. However, for the sake of simplicity, I prefer to stay within the conventional logical semantics as much as possible. It is not until one reaches the Escher concurrency facilities [11] that non-determinism appears. However, one can at least argue that many important concurrent applications are intrinsically non-deterministic and hence it is impossible for the concurrency facilities to avoid non-determinism if they are to be powerful enough to model these applications.

3.2 Queens problem

The next program is a solution of the queens problem (for 5 queens) in traditional logic programming style. For the module `Queens`, the term

```
{x | queen(x)}
```

reduces to

```
{[1, 3, 5, 2, 4], [1, 4, 2, 5, 3], [2, 4, 1, 3, 5],
 [2, 5, 3, 1, 4], [3, 1, 4, 2, 5], [3, 5, 2, 4, 1],
 [4, 1, 3, 5, 2], [4, 2, 5, 3, 1], [5, 2, 4, 1, 3],
 [5, 3, 1, 4, 2]}.
```

Also the term

```
card {x | queen(x)}
```

reduces to

```
10.
```

```
module Queens(queen) where {

queen :: [Int] -> Bool;
queen(x) =
    safe(x) &&
    permutation([1,2,3,4,5], x);

safe :: [Int] -> Bool;
safe([]) =
    True;
safe(x : y) =
    noDiagonal(x, 1, y) &&
    safe(y);

noDiagonal :: (Int, Int, [Int]) -> Bool;
noDiagonal(_, _, []) =
    True;
noDiagonal(x, y, z : w) =
    y /= abs(z - x) &&
    noDiagonal(x, y+1, w);

}
```

3.3 List Processing

The module `ListProcessing` contains some further list processing predicates which show how typical logic programming predicates are defined in the Escher programming style.

3.4 Conditionals

Escher has some very useful syntactic sugar for a form of conditional first made available in NU-Prolog [18]. This notation has the form

`if exists \`$x_1 \ldots x_n$ `->` *Cond* `then` *Form1* `else` *Form2*

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

`(exists \`$x_1 \ldots x_n$ `->` *Cond* `&&` *Form1*`) ||`
`(not (exists \`$x_1 \ldots x_n$ `->` *Cond*`) &&` *Form2*`).`

The main advantages of using the syntactic sugar (instead of its meaning) are that the syntactic sugar provides a compact and expressive notation for a common programming idiom and the Escher system avoids the inefficiency of computing the condition twice.

Module `AssocList` 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 module `AssocList`, the term

```
lookup(5, value, [(4, "How"), (5, "You")], list)
```

reduces to

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

and the term

```
lookup(5, value, [(4, "How"), (5, "You"), (5, "Then")], list)
```

```
module ListProcessing(member, append, permute, delete, sorted)
where {

member :: (a, [a]) -> Bool;
member(x, []) =
    False;
member(x, y : z) =
    (x == y) || member(x, z);

append :: ([a], [a], [a]) -> Bool;
append(u, v, w) =
    (u == [] && v == w) ||
    exists \r x y -> u == r : x && w == r : y && append(x, v, y);

permute :: ([a], [a]) -> Bool;
permute([], x) =
    x == [];
permute(x : y, w) =
    exists \u v z -> w == u : v && delete(u, x : y, z) &&
                     permute(z, v);

delete :: (a, [a], [a]) -> Bool;
delete(x, [], y) =
    False;
delete(x, y : z, w) =
    (x == y && w == z) ||
    exists \v -> w == y : v && delete(x, z, v);

sorted :: [Int] -> Bool;
sorted([]) =
    True;
sorted(x : y) =
    if y == []
    then True
    else exists \u v -> y == u : v && x <= u && sorted(y);
}
```

```
module AssocList(lookup) where {

lookup :: (Int, String, [(Int,String)], [(Int,String)]) -> Bool;
lookup(key, value, assoc_list, new_assoc_list) =
    if exists \v -> member((key, v), assoc_list)
    then
      value == v &&
      new_assoc_list == assoc_list
    else
      new_assoc_list == ((key, value) : assoc_list);

}
```

reduces to

```
((value == "You") &&
 (list == [(4, "How"), (5, "You"), (5, "Then")])) ||
((value == "Then") &&
 (list == [(4, "How"), (5, "You"), (5, "Then")])).
```

The Gödel book [8] contains many programs which use conditionals.

3.5 Higher-Order Facilities

The module `Relational` is the Escher version of a λProlog program [16]. The function `mappred` is a relational version of the usual `map` function. 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 module `Relational`, the term

```
mappred(age, [Bob, Sue], x)
```

reduces to

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

```
module Relational(Person(..), parent, age, mappred, forevery)
where {

data Person =  Bob | John | Mary | Sue | Dick | Kate | Ned;

parent :: (Person, Person) -> Bool;
parent =
    {(Bob, John), (Bob, Dick), (John, Mary), (Sue, Dick),
     (Dick, Kate)};

age :: (Person, Int) -> Bool;
age =
    {(Bob, 24), (John, 7), (Mary, 13), (Sue, 23),
     (Dick, 53), (Kate, 11), (Ned, 23)};

mappred :: (((a, b) -> Bool), [a], [b]) -> Bool;
mappred(p, [], z) =
    z == [];
mappred(p, x : xs, z) =
    exists \y ys -> p(x, y) && mappred(p, xs, ys) &&
                    z == y : ys;

forevery :: ((a -> Bool), [a]) -> Bool;
forevery(p, []) =
    True;
forevery(p, x : y) =
    p(x) &&
    forevery(p, y);

}
```

The term

```
mappred(parent, [Bob, Dick], x)
```

reduces to

```
(x == [John, Kate]) ||
(x == [Dick, Kate]).
```

The term

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

reduces to

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

The term

```
mappred(\z -> exists \x y -> z == (x, y) && age(y, x),
[24, 23], w)
```

reduces to

```
(w == [Bob, Sue]) ||
(w == [Bob, Ned]).
```

The term

```
(\x -> age(x, 24))(Bob)
```

reduces to

```
True.
```

The term

```
forevery(\x -> age(x, y), [Ned, Bob, Sue])
```

reduces to

```
False.
```

The term

```
forevery(\x -> age(x, y), [Ned, Sue])
```

reduces to

```
y == 23.
```

The term

```
forevery(\x -> exists \y -> age(x, y), [Ned, Bob, Sue])
```

reduces to

```
True.
```

3.6 Databases

The next example illustrates typical database querying in Escher. The module `England` contains an (incomplete) database about cities and counties in England.

```
module England(County(..), City(..), neighbours, distance, isin)
where {

data County = Avon | Bedfordshire | Berkshire |
              Buckinghamshire | Cambridgeshire | Cornwall |
              Devon | Dorset | Essex | Gloucestershire |
              Hampshire | Herefordshire | Hertfordshire |
              Kent | London | Northamptonshire | Oxfordshire |
              Somerset | Surrey | Sussex | Warwickshire |
              Wiltshire | Worcestershire;

data City = Bath | Bournemouth | Bristol | Cheltenham |
            Cirencester | Dorchester | Exeter | Gloucester |
            Penzance | Plymouth | Salisbury | Shaftesbury |
            Sherbourne | Taunton | Torquay | Truro |
            Winchester;


neighbours :: (County, County) -> Bool;

neighbours =
    {(Devon, Cornwall),
     (Devon, Dorset),
     (Devon, Somerset),
     (Avon, Somerset),
     (Avon, Wiltshire),
     (Avon, Gloucestershire),
     (Dorset, Wiltshire),
     (Somerset, Wiltshire),
     (Gloucestershire, Wiltshire),
     (Dorset, Somerset),
     (Dorset, Hampshire),
     (Hampshire, Wiltshire),
     (Hampshire, Berkshire),
     (Hampshire, Sussex),
     (Hampshire, Surrey),
     (Sussex, Surrey),
     (Sussex, Kent),
     (London, Surrey),
     (London, Kent),
     (London, Essex),
     (London, Hertfordshire),
     (London, Buckinghamshire),
     (Surrey, Buckinghamshire),
     (Surrey, Kent),
     (Surrey, Berkshire),
     (Oxfordshire, Berkshire),
     (Oxfordshire, Wiltshire),
     (Oxfordshire, Gloucestershire),
     (Oxfordshire, Warwickshire),
     (Oxfordshire, Northamptonshire),
     (Oxfordshire, Buckinghamshire),
     (Berkshire, Wiltshire),
     (Berkshire, Buckinghamshire),
     (Gloucestershire, Worcestershire),
     (Worcestershire, Herefordshire),
     (Worcestershire, Warwickshire),
     (Bedfordshire, Buckinghamshire),
     (Bedfordshire, Northamptonshire),
     (Bedfordshire, Cambridgeshire),
     (Bedfordshire, Hertfordshire),
     (Hertfordshire, Essex),
     (Hertfordshire, Cambridgeshire),
     (Hertfordshire, Buckinghamshire),
     (Buckinghamshire, Northamptonshire)};


distance :: (City, City, Int) -> Bool;

distance =
    {(Plymouth, Exeter, 42),
     (Exeter, Bournemouth, 82),
     (Bristol, Taunton, 43),
     (Bristol, Gloucester, 35),
     (Torquay, Exeter, 23),
     (Plymouth, Torquay, 24),
     (Bristol, Bath, 13),
     (Exeter, Taunton, 34),
     (Penzance, Plymouth, 78),
     (Taunton, Bournemouth, 70),
     (Bournemouth, Salisbury, 28),
     (Taunton, Salisbury, 64),
     (Salisbury, Bath, 40),
     (Bath, Gloucester, 39),
     (Bournemouth, Bath, 65),
     (Truro, Penzance, 26),
     (Plymouth, Truro, 52),
     (Shaftesbury, Salisbury, 20),
     (Sherbourne, Shaftesbury, 16),
     (Dorchester, Bournemouth, 28),
     (Salisbury, Winchester, 24),
     (Exeter, Sherbourne, 53),
     (Sherbourne, Taunton, 29),
     (Bath, Cirencester, 32),
     (Cirencester, Cheltenham, 16),
     (Cheltenham, Gloucester, 9),
     (Dorchester, Sherbourne, 19),
     (Bath, Shaftesbury, 33),
     (Winchester, Bournemouth, 41),
     (Exeter, Dorchester, 53)};


isin :: (City, County) -> Bool;

isin =
    {(Bristol, Avon),
     (Taunton, Somerset),
     (Salisbury, Wiltshire),
     (Bath, Avon),
     (Bournemouth, Dorset),
     (Gloucester, Gloucestershire),
     (Torquay, Devon),
     (Penzance, Cornwall),
     (Plymouth, Devon),
     (Exeter, Devon),
     (Winchester, Hampshire),
     (Dorchester, Dorset),
     (Cirencester, Gloucestershire),
     (Truro, Cornwall),
     (Cheltenham, Gloucestershire),
     (Shaftesbury, Dorset),
     (Sherbourne, Dorset)};

}
```

The predicates `neighbours`, `distance`, and `isin` have been written in the most general form possible to allow flexible querying of the database. In some circumstances, more efficient versions of these predicates could be used. For example, if it was known that queries to the database would only ever involve asking about distances between two given cities, then it would be possible to replace the predicate `distance` in the module `England` by a (more efficient) function
`distance :: (City, City) -> Int`.

Note the use of underscores in some queries. An underscore (`_`) stands for a unique variable which is (implicitly) existentially quantified at the front of the smallest subterm of the form $f(t_1, \ldots, t_n)$ which has type `Bool` and contains the underscore. So, for example, the subterm `isin(_, x)` in query 8 is an abbreviation for `exists \w -> isin(w, x)`, where `w` is a new variable. This convention can simplify database queries considerably.

In each of the following queries, I first give the query in English and then follow this with the Escher form of the query.

1. *Find all cities which are less than 40 miles from Bristol.*

```
{x | exists \y -> (distance(Bristol, x, y) ||
                   distance(x, Bristol, y)) &&
                  y < 40}
```

reduces to

```
{Gloucester, Bath}.
```

2. *Find all pairs of cities which are less than 20 miles apart.*

```
{(x, y) | exists \z -> distance(x, y, z) && z < 20}
```

reduces to

```
{(Bristol, Bath), (Sherbourne, Shaftesbury),
 (Cirencester, Cheltenham), (Cheltenham, Gloucester),
 (Dorchester, Sherbourne)}.
```

3. *Find all counties which neighbour Oxfordshire.*

```
{x | neighbours(Oxfordshire, x) || neighbours(x, Oxfordshire)}
```

reduces to

```
{Berkshire, Wiltshire, Gloucestershire, Warwickshire,
 Northamptonshire, Buckinghamshire}.
```

4. *Find all cities which are not in Wiltshire.*

```
{x | exists \y -> isin(x, y) && y /= Wiltshire}
```

reduces to

```
{Bristol, Taunton, Bath, Bournemouth, Gloucester, Torquay,
 Penzance, Plymouth, Exeter, Winchester, Dorchester,
 Cirencester, Truro, Cheltenham, Shaftesbury, Sherbourne}.
```

5. *Find all cities which are in the counties neighbouring Oxfordshire.*

```
{x | exists \y -> (neighbours(Oxfordshire, y) ||
                   neighbours(y, Oxfordshire)) &&
                  isin(x, y)}
```

reduces to

```
{Salisbury, Gloucester, Cirencester, Cheltenham}.
```

6. *Find all cities which are in the West Country (that is, in Devon, Cornwall, Somerset, or Avon).*

```
{x | exists \y -> y `in` {Devon, Cornwall, Somerset, Avon} &&
                  isin(x, y)}
```

reduces to

```
{Torquay, Plymouth, Exeter, Penzance, Truro, Taunton, Bristol,
 Bath}.
```

7. *Find all counties which have at least one city in them less than 50 miles from Bristol.*

```
{x | exists \y z -> (distance(Bristol, y, z) ||
                     distance(y, Bristol, z)) &&
                    z < 50 && isin(y, x)}
```

reduces to

```
{Somerset, Gloucestershire, Avon}.
```

8. *Do all counties neighbouring Avon have cities in them?*

```
forall \x -> (neighbours(Avon, x) || neighbours(x, Avon)) ==>
             isin(_, x)
```

reduces to

```
True.
```

9. *Are all cities which are less than 40 miles from Bristol in the same county as Bristol?*

```
exists \x -> isin(Bristol, x) &&
   forall \z ->
     exists \y -> (distance(Bristol, z, y) ||
                   distance(z, Bristol, y)) &&
                  y < 40
                  ==> isin(z, x)
```

reduces to

```
False.
```

10. *How many counties neighbour Oxfordshire?*

```
card({x | neighbours(Oxfordshire, x) ||
          neighbours(x, Oxfordshire)})
```

reduces to

```
6.
```

4 Putting It All Together

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 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 existential quantifiers and predicate calls containing variables are not normally allowed in functional languages, the search capability required in some applications has to be captured by various programming tricks which can obscure what is really going on. Thus Escher provides greater expressive power than conventional functional or logic languages because exactly the right kinds of functions can be employed to accurately model the application.

The overall structure of a typical Escher program is similar to a Haskell program. (In fact, the most interesting Escher programs are *concurrent* ones, but I leave the discussion of this to [11].) At the top level there is monadic IO and most functions in the program are conventional Haskell functions. (Monadic IO is discussed in [17].) However, typically there are also a few crucial predicates programmed in the logic programming style. Calls to these predicates, such as `permute` and `likes`, are made at a lower level in the program and are usually encapsulated in a set. As a simple example of this, consider the `Queens` module again and suppose we want to print out the number of solutions to the queens problem. A suitable top-level function `main` to implement this is as follows.

```
main :: IO ();
main =
    print (card {x | queen(x)});
```

Here the answers to `queen(x)` are returned in a set, `card` computes the number of elements in this set, and then `print` prints out this number.

This example illustrates the point that set processing is an important component of the proposal in this paper. In essence, existential quantifiers allow computations in the logic programming style, while sets encapsulate for further processing the answers produced by this style of computation.

As a second example, consider the task of returning a single solution to the queens problem. For this, one can use the (system) function `setToList` with signature

```
setToList :: (a -> Bool) -> IO [a];
```

which takes a set as its argument and gives an IO action as a result. This action returns a list containing distinct elements in the set (in some order). If the set is finite, the list contains all elements in the set. If the set is infinite, then so is the list. The idea is that the world argument, which is hidden, encodes the information for choosing a particular list. Thus `setToList` doesn't suffer the same semantic problems as the function `choice` discussed earlier. Then an appropriate top-level function `main` is as follows.

```
main =
    setToList {x | queen(x)} >>= \y ->
    print (head y);
```

The same idea can be used to print a set: first convert it to a list with `setToList` and then print the list in a suitable form. More generally, a computation in the logic programming style returns a set which can then be further processed by set functions or can be converted to a list for further processing by conventional Haskell functions.

The low-level view of an Escher computation has already been discussed. The top-level view is as follows. Given an initial state of the world, the overall aim of the programmer is to write a program which takes the world through a sequence of transitions to new states, each obtained by executing some action. This sequence of transitions of the world state, which is often called a *trace*, may be finite or, in principle, infinite. It is this trace which is of primary interest since it is through the world that the results of computations are displayed, communicated, and so on. The low-level computations discussed earlier are subsidiary to producing the required trace.

For example, the trace for the first example of this section is simply the one where the initial world becomes, after the execution of one action, the new world which contains on standard output the value of the expression `card {x | queen(x)}`. The evaluation of the set `{x | queen(x)}` and its cardinality are subsidiary to this top-level computation. This example is not typical for real-life applications. Usually there is a dialogue between a user and a program (or, more generally, communication and synchronisation between processes in a concurrent program) which results in a much longer (possibly, potentially infinite) trace. Examples of this kind of behaviour are given in [11].

5 The `Booleans` Module

In this section, I discuss in more detail the definitions in the `Booleans` module.

The first function in `Booleans` is the equality function `==`, whose statements have a close connection with unification. Now Escher doesn't have unification explicitly present in the computational model. Instead, matching is carried out during a function call and the remainder of unification is handled by explicit equalities which appear in the bodies of statements. The first three statements in the definition of `==` correspond exactly to three of the steps in the equational version of the unification algorithm given in [10]. The step in the unification algorithm in which an equation of the form `x == x`, where `x` is a variable, is deleted from the set of equations isn't included in Escher as it doesn't seem useful for programming. The remaining step in the unification algorithm where the existence of an equation of the form `x == t` leads to `x` being replaced by `t` throughout the equations, where `x` is not free in `t`, appears in statements in various other definitions in `Booleans`. The fourth statement in the definition of `==` gives a standard property of tuples. The final statement simply states that two sets are equal iff each is a subset of the other. The definition of subset appears later.

The definition of disequality appears in the Haskell Prelude. Two of the first four statements in the definition of `&&` also appear there. Next follow the laws for distributing `&&` over `||`. These laws are crucial since they force disjunctions to the top level of boolean expressions. They are needed in Escher as, in contrast to Haskell, Escher allows variables in expressions. The next rule is used to widen the scope of an existential quantifier and the last statement is part of a step in the unification algorithm referred to above.

Two of the first four statements in the definition of `||` appear in the Haskell Prelude. The rewrites for conditionals have been added to this definition since a conditional is a boolean expression with a disjunction at the top level. The definition of `not` is unremarkable.

Next follows the definition of the function `exists`, which is Church's generalised existential quantifier. The first three statements of this definition are straightforward. The fourth one is very important for the whole proposal. Definitions of predicates written in the logic programming style typically have existentially quantified variables in their bodies. What happens in a computation is that reduction is done inside the scope of an existential quantifier until a term of the form `x == u` appears. Then, provided the side conditions are met, the last statement in the definition of `exists` is used to eliminate the local variable `x`.

The definition of the generalised universal quantifier `forall` exploits the fact that the most common use of a universal quantifier is in an expression of the form `forall \x -> u ==> v`, for some terms `u` and `v`. There are 3 cases in the definition corresponding to when `u` is `False`, a conjunction containing a term of the form `x == t`, or a disjunction. Note the close parallel here with the form of set-processing functions. This comes about because an expression of the form `forall \x -> u ==> v` can be rewritten as `{x | u} ‘subset‘ {x | v}`.

The definitions of `union`, `inters`, and `minus` are straightforward. The definitions of the functions `subset`, `superset`, `power`, and `mapset` are written in the style discussed earlier in which there is a statement corresponding to each of the cases, `False`, `==`, and `||` at the top level in the body. Set membership, `in`, is just function application. The functions `linearise` and so on, associated with the extensional representation of sets, were discussed earlier.

6 Conclusion

This paper has provided an introduction to the Escher language, concentrating on the issue of programming style in an integrated functional and logic language. The paper has also provided an answer to the question: what has to be added to Haskell so that it can provide the facilities made available by typical logic programming languages? I have argued that this question is important because Haskell could become the *lingua franca* of both functional and logic programming if these facilities could be successfully added.

Taking for granted that it is worthwhile adding these facilities to Haskell, the question then is whether these extra facilities can be efficiently implemented. This question is currently under investigation at Bristol, but it will be some time before a definitive answer can be given. An ideal solution would be an Escher system which ran (standard) Haskell programs with only a very modest overhead compared with current Haskell systems and also ran programs in the logic programming style with nearly the efficiency of typical Prolog systems.

I believe that the current divide between the fields of functional programming and logic programming is highly artificial and that the future lies in declarative programming, an integration of the best concepts in the two fields. Research which addresses implementation issues for integrated languages is one way of bridging this divide. I hope researchers in both functional and logic programming take up this challenge.

Acknowledgements

Many people have contributed ideas for the design of Escher over the last three years. Of these, I would particularly like to mention Antony Bowers, Kerstin Eder, and Michael Hanus whose suggestions and constructive criticisms have been influential. I would also like to thank Elwood Wilkins for some observations in [19] which improved the way set processing is handled.

References

[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] K. Eder. Implementing the rewriting computational model of functional logic languages. In preparation.

[4] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[5] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[6] M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`.

[7] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.

[8] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994. Logic Programming Series.

[9] I. Holyer and E. Spiliopoulou. The Brisk machine: A simplified STG machine. In *Workshop on Implementation of Functional Languages, IFL'97*, September 1997. To appear in Lecture Notes in Computer Science, Springer-Verlag, 1998.

[10] J.-L. Lassez, M.J. Maher, and K. Marriot. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[11] J.W. Lloyd. Interaction and concurrency in a declarative programming language. In preparation.

[12] J.W. Lloyd. Combining functional and logic programming languages. In *Proceedings of the 1994 International Logic Programming Symposium, ILPS'94*, pages 43–57. MIT Press, 1994.

[13] J.W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, 1995. Available at `http://www.cs.bris.ac.uk/`.

[14] J.W. Lloyd. Debugging for a declarative programming language. In K. Furukawa, D. Michie, and S. Muggleton, editors, *Machine Intelligence 15.* Oxford University Press, 1998. In press.

[15] G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD thesis, University of Pennsylvania, 1987.

[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] J. Peterson and K. Hammond (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.4). Available at `http://haskell.org/`.

[18] 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.

[19] E. Wilkins. Programming in sets: An exploration of Escher. Master's thesis, Department of Computer Science, University of Bristol, 1997.

[20] D.A. Wolfram. *The Clausal Theory of Types.* Cambridge University Press, 1993.

A Type Theory

The basic logic of Escher is an extension of Church's simple theory of types [2]. In the following, I shall refer to Church's logic as *type theory.* 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 [20], or the summaries in [15] and [16]. A more detailed account of (the extension of) type theory underlying Escher is contained in [13]. For the purposes of this paper, I simply outline the main concepts of (extended) type theory in a few paragraphs, leaving the reader to consult the above accounts if more detail is needed.

First, I assume there is given a set of type constructors $\mathcal{C}$ of various arities. Included in $\mathcal{C}$ are the type constructors $\mathbf{1}$ and o both of arity 0. 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.* The main purpose of having $\mathbf{1}$ is so that constants can be given types in a uniform way as for functions. The type o is the type of propositions. The *types* of the logic are built up in the standard way from the set of type constructors and a set of type variables, using the symbol $\rightarrow$ (for function types) and $\times$ (for product types). Note that the logic is polymorphic, an extension not considered by Church.

The *terms* of type theory are the terms of the typed λ-calculus, which are formed in the usual way by abstraction and application from a given set of functions having types of the form $\alpha \rightarrow \beta$ and a set of variables. A term of type o is called a *formula.* In type theory, one can introduce the usual connectives and quantifiers as functions of appropriate types. Thus the connectives conjunction, $\wedge$, and disjunction, $\vee$, are functions of type $o \rightarrow o \rightarrow o$ and the (generalized) existential quantifier, Σ, and universal quantifier, Π, have type $(\alpha \rightarrow o) \rightarrow o$. (The $\rightarrow$ is right associative.) 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 $\{x \mid t\}$ to emphasize its intended meaning as a set. The notation $\{\}$ means $\{x \mid False\}$. A set abstraction of the form $\{x \mid (x = t_1) \vee \ldots \vee (x = t_n)\}$ is abbreviated to $\{t_1, \ldots, t_n\}$, where x is not free in any t_i. There is also a tuple-forming notation $< \ldots >$. Thus, if $t_1, \ldots, t_n$ are terms of type $\tau_1, \ldots, \tau_n$, respectively, then $< t_1, \ldots, t_n >$ is a term of type $\tau_1 \times \ldots \times \tau_n$. 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 mapping from the domain of type $\mathbf{1}$ have their argument omitted.

Type theory has an elegant and useful model theory. The key idea, introduced by Henkin in his paper [7] which proved the completeness of type theory, is that of a *general model*. General models are a natural generalization of first-order interpretations. Very sketchily, leaving aside the extension to handle polymorphism, the model theory of type theory is as follows. The domain for a nullary type constructor in $\mathcal{C}$ is some set, the domain for a type of the form $\alpha \rightarrow \beta$ is a set of functions mapping from the domain of type α to the domain of type β, and the domain for a type of the form $\alpha \times \beta$ is the cartesian product of the domains of type α and β. A function of type τ is assigned some element of the domain of type τ and the meaning of the connectives and quantifiers is what one would expect. From this, the notions of general model, satisfaction, validity, model of a set of formulas, and so on, can be given in a rather straightforward way. (See, for example, [1], [7], [13], or [20] for the details.) I propose that Henkin's concept of a general model be the appropriate one for capturing the *intended interpretation* of an application.

Finally, here is a table which shows the correspondence between various symbols and expressions of type theory in the left column and their equivalent in the syntax of Haskell/Escher in the right column.

$\mathbf{1}$	`()`
o	`Bool`
$\sigma \to \tau$	`Sigma -> Tau`
$\sigma \times \tau$	`(Sigma, Tau)`
$=$	`==`
$\neg$	`not`
$\wedge$	`&&`
$\vee$	`||`
$\to$	`==>`
$\lambda x.t$	`\x -> t`
Σ	`exists`
Π	`forall`
$\exists x.t$	`exists \x -> t`
$\forall x.t$	`forall \x -> t`
$\{x \mid t\}$	`{x | t}`
$\in$	`in`
$< s, t >$	`(s, t)`

B Operational Semantics

This appendix contains an outline of the operational semantics of Escher.

Definition A *redex* of a term t is a subterm of t which is identical to the head of some instance of a statement schema.

Definition Let $\mathcal{L}$ be the set of terms constructed from the alphabet of a program and let $\mathcal{DS}_{\mathcal{L}}$ be the set of subterms of terms in $\mathcal{L}$ distinguished by their position. A *selection rule* S is a function from $\mathcal{L}$ to the power set of $\mathcal{DS}_{\mathcal{L}}$ satisfying the following condition: if t is a term in $\mathcal{L}$, then $S(t)$ is a subset of the set of outermost subterms of t each of which is a redex.

Typical selection rules are the parallel-outermost selection rule for which all outermost redexes are selected and the leftmost selection rule in which the leftmost outermost redex is selected. An implementation of Escher can employ any appropriate selection rule. For example, an implementation of Escher which aims to encompass Haskell must employ the Haskell selection rule on the Haskell subset of the language.

Definition A term s is obtained from a term t by a *computation step* if the following conditions are satisfied:

1. $S(t)$ is a non-empty set, $\{r_\alpha\}$, say.
2. For each α, the redex r_α is identical to the head h_α of some instance $h_\alpha = b_\alpha$ of a statement schema.
3. s is the term obtained from t by replacing, for each α, the redex r_α by b_α.

Definition A *computation* from a term t is a sequence $\{t_i\}_{i=1}^n$ of terms such that the following conditions are satisfied.

1. $t = t_1$.
2. t_{i+1} is obtained from t_i by a computation step, for $i = 1, \ldots, n-1$.

The term t_1 is called the *goal* of the computation and t_n is called the *answer*.

C Booleans Module

```
module Booleans where {

data Bool = True | False;

data (a -> Bool) = Inc a (a -> Bool);
--
--  Inc is used in the extensional representation of sets.
--  Inc x s = {x} union s.


(==) :: a -> a -> Bool;

f x1 ... xn == f y1 ... yn  =  (x1 == y1) && ... && (xn == yn);
--
--  where n >= 0; and f is a data constructor.
--  (If n=0, then the RHS is True.)

f x1 ... xn == g y1 ... ym  =  False;
--
--  where n and m >= 0; f and g are data constructors;
--  and f is distinct from g.

y == x  =  x == y;
--
--  where x is a variable; and y is not a variable.

(x1,...,xn) == (y1,...,yn)  =  (x1 == y1) && ... && (xn == yn);

{x | u} == {y | v}  =
    ({x | u} `subset` {y | v}) && ({y | v} `subset` {x | u});


(/=) :: a -> a -> Bool;

x /= y  =  not (x == y);
```

```
(&&) :: Bool -> Bool -> Bool;

True && x  =  x;

x && True  =  x;

False && x  =  False;

x && False  =  False;

(x || y) && z  =  (x && z) || (y && z);

x && (y || z)  =  (x && y) || (x && z);

x && (exists \x1 ... xn -> v) && y  =
    exists \x1 ... xn -> x && v && y;
--
--  where no xi is free in x or y; and x or y may be absent.

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; u is free for x in y and z;
--  and y or z may be absent.


(||) :: Bool -> Bool -> Bool;

True || x  =  True;

x || True  =  True;

False || x  =  x;

x || False  =  x;

--  Rewrites for the conditional syntactic sugar.
```

```
if exists \x1 ... xn -> True then x else y  =
    exists \x1 ... xn -> x;

if exists \x1 ... xn -> False then x else y  =  y;

if exists \x1 ... xn -> x && (xi == u) && y then z else v  =
    if exists \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; u is free for xi in x, y and z;
--  and x or y (or both) may be absent.

if exists \x1 ... xn -> x || (xi == u) || y then z else v  =
    exists \x1 ... xn -> (x || (xi == u) || y) && z;
--
--  where x or y (or both) may be absent.


not :: Bool -> Bool;

not False  =  True;

not True  =  False;

not (not x)  =  x;

not (x || y)  =  (not x) && (not y);

not (x && y)  =  (not x) || (not y);


exists :: (a -> Bool) -> Bool;

exists \x1 ... xn -> True  =  True;

exists \x1 ... xn -> False  =  False;

exists \x1 ... xn -> x || y  =
    (exists \x1 ... xn -> x) || (exists \x1 ... xn -> y);

exists \x1 ... xn -> x && (xi == u) && y  =
    exists \x1 ... xi-1 xi+1 ... xn -> x{xi/u} && y{xi/u};
--
--  where xi is not free in u; u is free for xi in x and y;
--  and x or y (or both) may be absent.
--  If n=1, then the RHS is x{x1/u} && y{x1/u}.
--  If both x and y are absent, then the RHS is True.


forall :: (a -> Bool) -> Bool;

forall \x1 ... xn -> False ==> u  =  True;

forall \x1 ... xn -> x && (xi == u) && y ==> v  =
    forall \x1 ... xi-1 xi+1 ... xn ->
                      x{xi/u} && y{xi/u} ==> v{xi/u};
--
--  where xi is not free in u; u is free for xi in x, y and v;
--  x or y (or both) may be absent; and, if n=1, then both x
--  and y are absent.
--  If n>1 and both x and y are absent, then the RHS is
--  forall \x1 ... xi-1 xi+1 ... xn -> True ==> v{xi/u}.
--  If n=1, then the RHS is v{x1/u}.

forall \x1 ... xn -> (u || v) ==> w  =
    (forall \x1 ... xn -> u ==> w) &&
    (forall \x1 ... xn -> v ==> w);


union :: (a -> Bool) -> (a -> Bool) -> (a -> Bool);

s `union` t  =  {x | (x `in` s) || (x `in` t)};


inters :: (a -> Bool) -> (a -> Bool) -> (a -> Bool);

s `inters` t  =  {x | (x `in` s) && (x `in` t)};
```

```
minus :: (a -> Bool) -> (a -> Bool) -> (a -> Bool);

s `minus` t  =  {x | (x `in` s) && (not (x `in` t))};


subset :: (a -> Bool) -> (a -> Bool) -> Bool;

{} `subset` s  =  True;

{u} `subset` s  =  u `in` s;

{x | u || v} `subset` s  =
    ({x | u} `subset` s) && ({x | v} `subset` s);


superset :: (a -> Bool) -> (a -> Bool) -> Bool;

s `superset` {}  =  True;

s `superset` {u}  =  u `in` s;

s `superset` {x | u || v}  =
    (s `superset` {x | u}) && (s `superset` {x | v});


power :: (a -> Bool) -> (a -> Bool) -> Bool;
--
--  Power set function.

power {}  =  {{}};

power {u}  =  {{}, {u}};

power {x | u || v}  =
    {s | exists \l r -> l `in` (power {x | u}) &&
                        r `in` (power {x | v}) &&
                        s == l `union` r};
```

```
mapset :: (a -> b) -> (a -> Bool) -> (b -> Bool);
--
--  Analogue of map for set processing.

mapset f {}  =  {};

mapset f {u}  =  {f u};

mapset f {x | u || v}  =
    (mapset f {x | u}) ‘union‘ (mapset f {x | v});

in :: a -> (a -> Bool) -> Bool;
--
--  Set membership.

y ‘in‘ {x | u}  =  u{x/y};
--
--  where y is free for x in u.

linearise :: (a -> Bool) -> (a -> Bool);
--
--  Convert from standard to extensional representation
--  of a set.

linearise {}  =  {};

linearise {x}  =  Inc x {};

linearise {x | u || v}  =
    combine (linearise {x | u}) (linearise {x | v});

combine :: (a -> Bool) -> (a -> Bool) -> (a -> Bool);
--
--  Union of sets (in extensional representation).

combine {} s  =  s;

combine (Inc x s) t  =  Inc x (combine s t);
```

```
delinearise :: (a -> Bool) -> (a -> Bool);
--
--  Convert from extensional to standard representation
--  of a set.

delinearise {}  =  {};

delinearise (Inc x s)  =  {x} `union` (delinearise s);


remove :: a -> (a -> Bool) -> (a -> Bool);
--
--  Delete an element from a set.

remove x {}  =  {};

remove x (Inc y s)  =
    if x == y then remove x s else Inc y (remove x s);


deletedup :: (a -> Bool) -> (a -> Bool);
--
--  Delete duplicates from an extensional representation
--  of a set.

deletedup {}  =  {};

deletedup (Inc x s)  =  Inc x (deletedup (remove x s));

}
```

@Marcin Fortunately, are several functional logic programming languages that combine these paradigms in a more convenient way. â€“ Anderson Green Jun 7 '16 at 1:07. | show 1 more comment.Â Some say that logic programming is a superset of functional programming since each function could be expressed as a predicate: foo(x,y) -> x+y. could be written as.