Debugging for a Declarative Programming Language

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

Abstract

This paper investigates debugging in declarative programming languages, concentrating specifically on the integrated functional and logic programming language Escher. The Escher language has types and modules, higher-order and meta-programming facilities, and declarative input/output. It also has a collection of system modules, providing numerous operations on standard data types such as integers, lists, characters, strings, sets, and programs.

After a brief introduction to the Escher language, a framework for declaratively debugging Escher programs is presented and an implementation of this framework is illustrated by an example. The paper concludes with a discussion of the practicalities of declarative debugging and some open problems.

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) in (Shapiro, 1983) and was studied by a number of authors in subsequent years. (A fairly complete bibliography up to 1987 is given in (Lloyd, 1987).) 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 (Lloyd, 1987) 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 certain 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, its 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!

The declarative, general-purpose programming language Escher is such a language. Escher 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 (Hill and Lloyd, 1994), Haskell (Fasel et al., 1992), and λProlog (Nadathur and Miller, 1988). 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.

For general background on integrated functional and logic programming languages, the reader should consult the book (DeGroot and Lindstrom, 1986) or the recent survey (Hanus, 1994). Also this volume contains a paper on the VESPER language (Robinson and Barklund, 1995) which has some remarkable similarities to Escher. A much more comprehensive and detailed discussion of the facilities of Escher, including its module system, its system modules, and its declarative and procedural semantics, than can be given in this paper is contained in (Lloyd, 1995a). A general discussion of the advantages of declarative programming is also

given there. Two other papers on Escher are (Lloyd, 1994) and (Lloyd, 1995b)

This paper provides a framework for declarative debugging in Escher. As well as presenting the key concepts and debugging algorithm, I give an example 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 (Lloyd, 1987), 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.

The next section gives a brief account of the logic underlying Escher. The third section introduces the key ideas of the Escher language. The fourth section contains a framework for declarative debugging in languages with an Escher-like computational model. The fifth section contains an example session using the debugger given in the fourth section. In the last section, I discuss the practicalities of declarative debugging and give some open problems.

2 Type Theory

In this section I outline (an extension of) Church's simple theory of types (Church, 1940), which is the logic underlying Escher. 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 (Church, 1940), a more comprehensive account of higher-order logic in (Andrews, 1986), a more recent account, including a discussion of higher-order unification, in (Wolfram, 1993), or useful summaries in the many papers ((Nadathur, 1987) and (Nadathur and Miller, 1994), for example) of Miller, Nadathur, and their colleagues, on λProlog. A detailed account of (the extension of) type theory underlying Escher is presented in (Lloyd, 1995a). For the purposes of this paper, I simply outline now the main concepts of 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 $\mathcal{C}$ of constructors of various arities. Included in $\mathcal{C}$ are the 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 constructor o is the type of propositions. The *types* of the logic are built up in the standard way from the set of constructors and a set of parameters (that is, 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 func-

tions having types of the form $\alpha \to \beta$ 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 $o \times o \to o$ and the (generalized) existential quantifier, Σ, and universal quantifier, Π, have type $(\alpha \to o) \to o$. (The $\to$ is right associative.) Terms of the form $\Sigma(\lambda x B)$ are written as $\exists x B$ and terms of the form $\Pi(\lambda x B)$ are written as $\forall x B$. In addition, if B is of type o, the abstraction $\lambda x B$ is written $\{x : B\}$ to emphasize its intended meaning as a set. A set abstraction of the form $\{x : (x = t_1) \vee \ldots \vee (x = t_n)\}$ is abbreviated to $\{t_1, \ldots, t_n\}$. 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 **1** have their argument omitted.

Type theory has an elegant and useful model theory. The key idea, introduced by Henkin in his paper (Henkin, 1950) 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 constructor in $\mathcal{C}$ is some set, the domain for a type of the form $\alpha \to \beta$ 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, (Andrews, 1986), (Henkin, 1950), (Lloyd, 1995a), or (Wolfram, 1993) 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.

3 Elements of Escher

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 in the left column and their equivalent in the notation of Escher in the right column.

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 life-cycle in some detail by first giving the intended interpretation of the application and then writing down

$\mathbf{1}$	`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`

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`.) 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.

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 beginning with the keyword `MODULE` simply gives the name of the module. 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 earlier 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

$F(t_1, \ldots, t_n)$

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

```
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)).
```

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).

Mode declarations begin with the keyword `MODE`. In general, mode declarations restrict the possible calls that can be made to a function. For the `Perm` function, the `NONVAR` in the first argument of the mode declaration means that a call can only proceed if its first argument is a pattern. (A *pattern* 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. 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. Arguments in the head of a statement corresponding to underscores in the mode declaration must be variables.

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

3. Statements must be pairwise non-overlapping.

The first restriction comes about because the head of a statement should be at least as general as a redex and an argument in a redex 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 second 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 (standardized apart) statements, $h_1 \Rightarrow b_1$ and $h_2 \Rightarrow b_2$, are *non-overlapping* if, whenever there exists a substitution θ such that $h_1\theta$ and $h_2\theta$ are identical, we have that $b_1\theta$ and $b_2\theta$ are identical (where in both cases the (syntactic) identity is modulo renaming of bound variables). The restriction that statements must be pairwise non-overlapping means that if two or more statements match a redex 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.

Now I turn to a discussion of function calls (which I simplify somewhat for the sake of the exposition – the details are in (Lloyd, 1995a)). A redex is a subterm

of the form $F(t_1, \ldots, t_n)$, for some defined function F, which satisfies the mode declaration for F. Escher selects the leftmost redex in the current term on which to make the call. In a function call, a statement is viewed as a rewrite which behaves as follows. A statement $h \Rightarrow b$ *matches* a redex r if there is a substitution θ such that $h\theta$ is identical to r (modulo renaming of bound variables). In this case, the redex r in the current term is replaced by $b\theta$ to give the next term in the computation and this defines the rewrite given by the statement.

The Escher mode system provides several important facilities. First, for functions appearing in the export parts of modules, mode declarations show explicitly how the functions are meant to be called. Second, if a subterm satisfies the mode declaration for a function (and is therefore a redex), then either a statement in the definition of the function matches the redex or else a control error is generated. The compiler can do much to help avoid this kind of control error. For example, if an argument in a mode declaration is `NONVAR`, then the compiler can check there is a statement in the definition for each possible choice of free function at the top level in that argument.

Here are some typical goals and their answers for the program consisting of the modules `Permute` and `Booleans`. (Every module implicitly imports the system module `Booleans`.) 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 `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 1 below. 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 `Concat`, while the fourth term, which is the answer, is obtained by using the first statement in the definition of `Concat`. In each term, the redex is underlined. The other three goals for the module `Permute` require the use of statements for the functions `=`, `~`, and `\/` in the module `Booleans`.

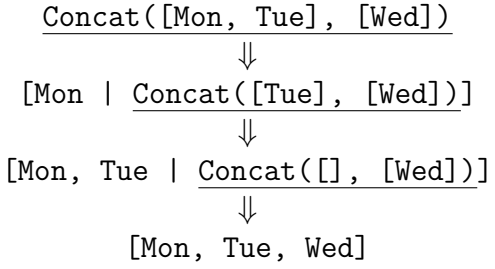

Figure 1: An Escher computation

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 term in the intended interpretation. However, given a term, 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 `Concat([Mon, Tue], [Wed])` is simplified to `[Mon, Tue, Wed]` which can be easily evaluated. 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.

The reduction process terminates when a term is reached which contains no redexes. This final term, 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 (potential) redex containing a user-declared defined function at the top level 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 modules `Permute` and `Booleans`, the goal

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

reduces to the answer

```
False.
```

4 Principles of Declarative Debugging

In this section, I give the theoretical underpinnings of declarative debugging for languages with an Escher-like computational model. Throughout the remainder of this paper, 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 in (Shapiro, 1983) and is called *divide-and-query*. To understand

how divide-and-query works, consider a typical computation in Figure 2. In this computation, t_1 is the goal, t_n is the answer, and $h_i \Rightarrow b_i$ is a statement used in a function call in the term t_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.

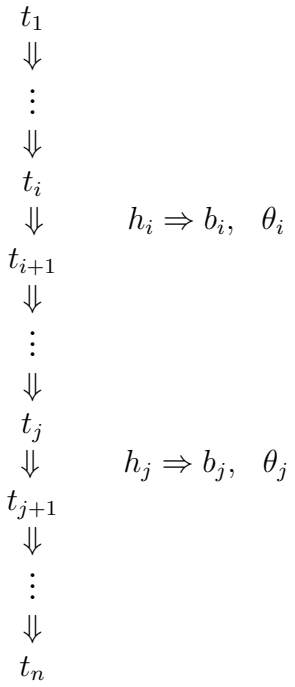

Figure 2: A computation

Definition Let C be a computation with goal t_1 and answer t_n. Then C is *incorrect* if $t_1 = t_n$ is not valid in the intended interpretation.

Informally, divide-and-query proceeds as follows. Consider the computation with goal t_1 and answer t_n. Having confirmed with the oracle that the computation is indeed incorrect, the debugging algorithm then chooses the term $t_{\lfloor (n+1)/2 \rfloor}$ at the midpoint of the computation and asks the oracle whether $t_1 = t_{\lfloor (n+1)/2 \rfloor}$ 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 t_1 to $t_{\lfloor (n+1)/2 \rfloor}$. If the answer is yes, then the algorithm discards the top half of the computation and continues with the segment from $t_{\lfloor (n+1)/2 \rfloor}$ to t_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 t_1. The *debugging trace* is the subsequence of the computation formed from t_1 together with all terms t_i in the computation which resulted from the (direct) use of a statement from a user module.

In the debugging trace in Figure 3 below, t_1 is the goal and t'_{i+1} is the term which resulted from the function call using the (user) statement $h'_i \Rightarrow b'_i$ with associated substitution θ'_i.

$$\begin{array}{cc} t_1 & \\ t'_2 & (h'_1 \Rightarrow b'_1, \quad \theta'_1) \\ t'_3 & (h'_2 \Rightarrow b'_2, \quad \theta'_2) \\ \vdots & \vdots \\ t'_{m+1} & (h'_m \Rightarrow b'_m, \quad \theta'_m) \end{array}$$

Figure 3: A debugging trace

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 $t_1, \ldots, t_m$ from an incorrect computation.
Output: An incorrect statement.

begin
  $i := 1$;
  $j := m$;
  while $i + 1 < j$ **do**
  begin
    if $t_i = t_{\lfloor (i+j)/2 \rfloor}$ 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 t_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 terms 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 t_1 and that it is possible to replace the query about $t_i = t_{\lfloor (i+j)/2 \rfloor}$ with one about $t_1 = t_{\lfloor (i+j)/2 \rfloor}$. This idea cuts the complexity of oracle queries dramatically. The last optimization is to simplify the term $t_{\lfloor (i+j)/2 \rfloor}$ 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 $t_1, \ldots, t_m$ from an incorrect computation.
Output: An incorrect statement.

begin
  $i := 1$;
  $j := m$;
  while $i + 1 < j$ **do**
  begin
    $F :=$ simplified form of $t_{\lfloor (i+j)/2 \rfloor}$;
    if $t_1 = F$ 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 t_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 terms in the debugging trace of the computation, then the number of oracle queries required is bounded by $\lceil log_2 m \rceil$.

It is worth noting that the context in which the divide-and-query algorithm is applied in this paper is different to, and rather more general than, the context in

(Shapiro, 1983). Essentially, in (Shapiro, 1983), the algorithm is applied to the computation tree of a refutation of a definite goal for a definite program where the refutation gives a wrong answer. The algorithm picks a node (an atom) in this tree which has about half the weight of the entire tree and, depending on the oracle answer, either discards the subtree rooted at this node or else enters it and applies the algorithm recursively. Furthermore, for missing answers, a different algorithm is used which involves the oracle giving values for variables to make atoms valid. The presence of negative literals and SLDNF-resolution introduce further complications which are accounted for in (Lloyd, 1987).

In the Escher context, the divide-and-query algorithm is applied uniformly to an entire computation irrespective of whether it has a "wrong" or "missing" answer and irrespective of the presence of negation or quantifiers. Furthermore, the oracle only ever has to answer *yes/no* queries and never needs to supply values for variables. As can be seen from this section, the theoretical framework which results from applying the divide-and-query algorithm to the Escher computational model is considerably simpler than the corresponding frameworks for SLDNF-resolution in (Shapiro, 1983) and (Lloyd, 1987). On the other hand, the simplicity and uniformity of the Escher approach can lead to rather large and complex oracle queries being posed which may be difficult to answer. I return to this issue in the last section.

5 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.

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

Primes(5, x)
 =
```

```
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]
```

```
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))))
```

6 Discussion

Unfortunately, to build a practical debugging system, much more needs to be done than simply implement the algorithm of section 4. 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 a term 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 terms in the query may be very large. If the query concerns the equation $s = t$, where s is the goal, then typically it will be t that is large. This is probably the most troublesome aspect of the presentation problem. 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 can attempt to break the query up into smaller subqueries, the answers to which can answer the whole query. A typical case where this can be useful is when t is a disjunction of a large number of subformulas. This occurs in Escher programs which correspond to logic programs involving a great deal of search.

To see what can be done, suppose the query concerns validity of the equation

$$s = t_1 \vee t_2 \vee \ldots \vee t_n$$

where s is the goal. The debugger can break this up into a series of queries concerning the validity of the implications

$$\begin{array}{c} s \leftarrow t_1 \\ s \leftarrow t_2 \\ \vdots \\ s \leftarrow t_n \end{array}$$

If any one of these implications is not valid, then the original equation is not valid either. If all these implications are valid, then the validity of the original equation reduces to that of the validity of the implication

$$s \rightarrow t_1 \vee t_2 \vee \ldots \vee t_n$$

which can then be put to the oracle.

As a simple example of this, consider the equation

$$P(x) = ((x = A) \vee (x = B) \vee (x = C))$$

Then the series of queries concerning the validity of

$$P(x) \leftarrow (x = A)$$
$$P(x) \leftarrow (x = B)$$
$$P(x) \leftarrow (x = C)$$

is checking that the "answers" $x = A$, $x = B$, and $x = B$ are all correct. The query concerning the validity of

$$P(x) \rightarrow (x = A) \vee (x = B) \vee (x = C)$$

is checking that the "answers" given are complete.

The next issue concerns the use of abstract data types (ADT's). The problem here is that a subterm 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 a term, 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 terms 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 term 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.

References

Andrews, P. (1986). *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press.

Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68.

DeGroot, D. and Lindstrom, G., editors (1986). *Logic Programming: Relations, Functions and Equations*. Prentice-Hall.

Fasel, J., Hudak, P., Peyton Jones, S., and Wadler, P. (1992). Special issue on the functional programming language Haskell. *ACM SIGPLAN Notices*, 27(5).

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

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

Hill, P. and Lloyd, J. (1994). *The Gödel Programming Language*. MIT Press. Logic Programming Series.

Lloyd, J. (1987). *Foundations of Logic Programming*. Springer-Verlag, second edition.

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

Lloyd, J. (1995a). Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol. Also available at http://www.cs.bris.ac.uk/.

Lloyd, J. (1995b). Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1. To appear.

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

Nadathur, G. and Miller, D. (1988). An overview of λProlog. In Kowalski, R. and Bowen, K., editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming,* Seattle, pages 810–827. MIT Press.

Nadathur, G. and Miller, D. (1994). Higher-order logic programming. Technical Report CS-1994-38, Department of Computer Science, Duke University. 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.

Robinson, J. and Barklund, J. (1995). VESPER. In Furukawa, K., Michie, D., and Muggleton, S., editors, *Machine Intelligence 15*. Oxford University Press.

Shapiro, E. Y. (1983). *Algorithmic Program Debugging*. MIT Press.

Wolfram, D. (1993). *The Clausal Theory of Types*. Cambridge University Press.