J. Functional Programming 3 (1): 77–92, January 1993

The interactive Lazy ML system

LENNART AUGUSTSSON
Department of Computer Sciences, Chalmers University of Technology, S-412 96 Goteborg, Sweden
(*e-mail:* augustss@cs.chalmers.se)

Abstract

In this paper we describe an implementation of an interactive version of the purely functional programming language Lazy ML (LML). The most remarkable fact about the interactive system is that it is written in a pure functional style using LML, yet the efficiency still compares favourably to other conventional interpretative systems.

We describe how the system is designed, and also the exception mechanism that was added to facilitate the handling of errors in the system.

1 Introduction

Program development and testing benefits from the advantages of an interactive system. New ideas can be tried out quickly and the response to syntax and type errors is immediate. There is a long tradition of interactive functional language implementations, from LISP (Steele, 1984) to Standard ML (Milner *et al.*, 1990, Appel and MacQueen, 1987). Interactive systems are often implemented by interpreting the source language (or some fairly high level abstract code), since this usually gives less overhead before execution can start. On the other hand, batch compilers are often slow, but the resulting programs are much faster than their interpreted counterparts.

The interactive LML system tries to combine the advantages of interpreted and compiled code. It is possible to load compiled code into the interactive system and use it with full speed. This facility is available in many LISP systems, in contrast to SML of NJ which is an interactive system that always compiles.

The most remarkable thing about the interactive LML is not its external behaviour (a small sample session is shown in Fig. 1), it is like a LISP or SML system, but rather its internals. It is written almost entirely in LML, which is a side-effect free lazy language, thus showing that a pure language is indeed powerful enough to accomplish this. The only important part which is not written in LML (apart from the parser which is written with YACC (Johnson, 1975) for historical reasons) is the function that loads compiled code into the system, but even this function is pure in the way it is interfaced to LML; it is just the innards of it that are gory.

We have had a batch compiler for LML for many years. The compiler was in no way designed to be used together with an interactive environment. The interactive

```
Welcome to interactive Haskell B. version 0.998.1!
Loading prelude...982 values, 70 types found.
Type "help;" to get help.
> let square x = x * x;;
square::(Num a) ⇒ a → a
> square 5;
25
> 91 - square 7;
42
>
```

Fig 1. A sample session with interactive Haskell B.

LML was a much later invention, but the original compiler did not have to be changed in any way. The interactive LML was also produced with a fairly small effort (it consists of 1200 lines added to the 15,000 of the LML compiler),[1] most of the code in it is shared with the compiler.

The LML compiler has a Haskell (Hudak *et al.*, 1992) front end;[2] this means that there is also an interactive Haskell which is just the interactive LML with the Haskell front end. The rest of the paper will always talk about the interactive LML, but the same things are true for the interactive Haskell.

This paper is organized as follows: section 2 shows how a reasonably efficient interpreter can be written for a functional language in a functional language and some problems with strong typing in the interpreter; section 3 discusses mixing compiled and interpreted code; section 4 describes the simple exception handling mechanism used in LML; and section 5 contains implementation issues.

2 The interpreter

At the top level, the user can enter a normal LML binding defining types and/or values which are added to the global environment. The user can also evaluate an expression in the global environment by simply entering one (like ***square 5*** in Fig. 1). The expression is evaluated and printed (in a lazy fashion). It is also possible to load files with source code or compiled code.

Figure 2[3] is a sketch of the top loop of the system. All error handling is ignored in this sketch. After a command has been parsed it is executed and then the next command is processed. To add a new binding the expression is first transformed (by ***transform***) in various ways (described further in section 5) into a simple form. The ***eval*** function is then applied to get the value of it. This value is then added to the environment. Note that because of lazy evaluation the value is not actually computed until it is first needed, and that will be the only evaluation of it. In an expression

[1] It is hard to say how much time development took compared to the compiler, since both of them have evolved over time.

[2] Actually, it is more than a front end; there is also a new type checker and a few other things, but they are all integrated into the LML compiler.

[3] All examples are in Haskell notation, even though the system is written in LML.

https://doi.org/10.1017/S0956796800000617 Published online by Cambridge University Press

```
--parseCmd:: String → (Command, String)
--   parses one command and returns it and the rest of the input
--add Env:: Env → Symbol → VALUE → Env
--   adds a binding to an environment
--eval : Env → Expr → VALUE
--   evaluate an expression
--transform: Expr → Expr
--   various transformations
--showValue: VALUE → String
--   show a value

interp:: Env → String → String
interp env input =
  let (cmd, restOfInput) = parseCmd input in
  case cmd of
    Cbind var expr → interp (addEnv env var (trans env expr)) restOfInput
    Cexpr expr     → showValue (trans env expr) ++ interp env restOfInput
    Cquit          → " "
trans:: Env → Expr → Value
trans env e = eval env (transform e)
```

Fig. 2. Simplified interactive top level.

command the expression is transformed, evaluated, converted to a string (by ***showValue***), and then printed.

An odd consequence of having the interactive system written in a lazy language is that when definition is made (like ***square***), the translation will not be completely finished; only as much as is needed to ensure that there were no statically detectable errors (such as unbound variables or type errors) in the definition is actually performed. It is only when a definition is first used that it is completed, and even then only the used parts! The consequence of this is that when a definition is entered the response is quite rapid, but the first use of the definition will take some time; further usage of the definition will take no extra time since the translation is then completed.

2.1 The heart of the translator

Most interpreters written in conventional languages work on a representation of the program to evaluate and the result of the interpretation is some representation of the result. We use a different approach, more akin to the EVAL function in LISP. The basic idea is that to evaluate an expression you first translate the syntactic representation of it (e.g. the string *"5"* or *"λx.x"*) into the object that it represents (i.e. *5* or the identity function). This requires a language with functions as first class citizens, as the result may well be of function type. This approach was also used in Augustsson (1985).

The first passes of the interpreter (the ***transform*** function in Fig. 2), i.e. the scope checking, pattern matching transformation, type checking, etc., are the same as for

https://doi.org/10.1017/S0956796800000617 Published online by Cambridge University Press

the LML compiler. They do many transformations, like turning complex patterns into simple patterns. When the program has been suitably transformed by a number of passes it is essentially a number of definitions of new global variables. Each definition has a right hand side that is expressed in λ-calculus extended with constructors and a case expression (it may still contain λ-expressions since there has been no lambda-lifting).

The first step of the translation which is not in the compiler is to remove all constructors and case expressions. This operation is not quite trivial since we wish the underlying representation of values used in the interactive system to be the same as for compiled code (otherwise mixing interpreted and compiled code would be difficult). The difficulty arises from the fact that it is possible to make new type definitions interactively. This is not really possible in LML, since types cannot be created dynamically; when the compiler compiles a program it decides on the actual representation for each of the constructors in the type at compile time. In the interactive system, on the other hand, we want to be able to make new type definitions and have the elements of the new type be represented exactly as the compiler would have represented them if the definition had been compiled. To accommodate this in an efficient way, there are special generic constructor functions that cannot easily be coded in LML, instead they are coded in M-code (the lowest level of intermediate code). There is a special pass which changes constructors into calls of these generic constructor functions, and correspondingly with case expression (which take the constructed values apart). This pass of the compiler and the load mechanism are the only parts which contain code that depends on the actual implementation of LML. After this transformation an expression is a pure λ-term with constants.

2.2 The evaluator

One of the objectives in the design of the interactive LML system was to write as much as possible of it in LML to minimise the need for inter-language working. An easy way to do this is to write an interpreter that interprets some intermediate code, or even the abstract syntax tree directly. Writing an interpreter for the syntax tree is more or less like writing a denotational semantics for the language. A simplified form of the interpreter is given in Fig. 3. The interpreter takes an environment and an

```
--lookup :: Symbol -> Env -> VALUE
--   finds the value of an identifier in the environment
--addEnv :: Env -> Symbol -> VALUE -> Env
--   adds a new binding to the environment, as before
data Expr  =  Id Symbol | Lambda Symbol Expr | Apply Expr Expr

eval :: Env -> Expr -> VALUE
eval env (d s)          = lookup s env
eval env (Lambda s e) = λx -> eval (addEnv env s x) e
eval env (Apply f a)    = (eval env f) (eval env a)
```

Fig. 3. An interpreter for the abstract syntax of LML.

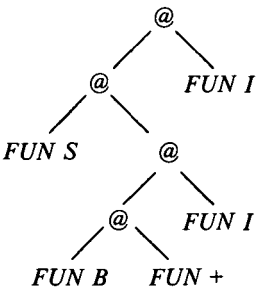
https://doi.org/10.1017/S0956796800000617 Published online by Cambridge University Press

```
-- addEnvl::[(Symbol, VALUE])→Env→Env
--   adds a number of definitions to an environment.
eval'::Env→Expr→VALUE
eval' env (Apply e1 e2)   =   (eval' env e1) (eval' env e2)
eval' env (d i)           =   lookup env i

eval::Env→Expr→VALUE
eval env e   =   eval' (addEnvl combEnv env) e
combEnv   =   [
  ("S", mkValue (λf→λg→λx→(f x) (g x)));
  ("K", mkValue (λx→λy→x));
  ("I", mkValue (λx→x));
  ...
]
```

Fig. 4. A combinator version of the interpreter and a small part of the global environment.

$\lambda x.x+x$	Input
Lambda "x" (*Apply* (*Apply* "+" "x") "x")	Syntax tree
Apply (*Apply* "S" (*Apply* (*Apply* "B" "+") "I")) "I"	Combinator version



Internal representation using the G-machine

Fig. 5. An example of translation.

expression and returns the actual value – not a representation of the value – of that expression in the given environment. (The *eval* function is not type correct, more about that in section 2.3.) A problem with this solution is the environment. It is something which is not static during the evaluation (since *Lambda* adds a new binding), so the lookup has to be performed every time an expression involving a variable is evaluated. This problem can be alleviated by separating the static part of the environment (the part which contains globally known functions, etc.) and the dynamic part which changes during execution. We can take this idea even further by eliminating the dynamic part completely, using Turner style combinator abstraction (Turner, 1979) to avoid the use of λ-expressions.

After the abstraction the only identifiers left in an expression are globally defined functions (combinators). We can now use the evaluator shown in Fig. 4. Here the environment is static, which means that lookup can be performed once and for all.

https://doi.org/10.1017/S0956796800000617 Published online by Cambridge University Press

The translation yields a representation of the value of the given expression, the precise details of which depend on the underlying implementation technique, but you could say that what happens when it is evaluated is that we run a combinator interpreter where the individual combinators have been programmed in the underlying functional language (here this is also LML). We exemplify this in Fig. 5. An advantage of this method is that we do not have to code the individual combinators nor the interpretative loop itself; instead, we rely on the implementation mechanism of the language in which we write the evaluator. If this is a reasonably efficient implementation we will get good code for the individual combinators, and for the dispatch mechanism.

It is worth noting that after the combinator translation the expression represented in memory is the same as the compiler would have produced for the (combinator abstracted) expression. The G-code for the example in Fig. 5 is shown in Fig. 6. But

```
PUSHGLOBAL I; PUSHGLOBAL I;
PUSHGLOBAL +; PUSHGLOBAL B;
MKAP; PUSHGLOBAL S; MKAP; MKAP; MKAP
```

Fig. 6. G-code (from the LML compiler) for the expression $S\ (B+)\ I\ I$.

the compiler does not do combinator abstraction. This is, of course, the big difference between the compiled and interpreted code. The compiler produces new functions, represented by actual machine code, whereas the interpreter uses a fixed set of functions to do the same thing.

2.3 Type safety

The *eval* function in Fig. 4 is not type correct (in a Hindley/Milner type system), since the result of it is used both as the function and argument part of an application (first equation of *eval'*). In all the examples we have used the type *VALUE* to indicate these values. It is only in compiling the *eval'* function and the *combEnv* list of values that we get a type problem. This is a rare example where strong typing makes it impossible (at least if efficiency is of concern) to write a function.

We are now faced with a small problem, how this function can be compiled, and a big problem, how to guarantee type safety despite it being untypable.

The first of these problems, how to compile it, is easily solved, since it is possible to compile programs without type checking with the LML compiler. If such a program is type correct in a semantic sense (a wider sense than the type system allows), everything will work when running it.

The second problem, type safety, is solved by type checking the representation of the expression with the usual algorithm before it is evaluated. Only if this succeeds does the evaluation take place. In this way we know that it will not 'go wrong' (in the usual sense) during runtime.

On the top level (when printing the result) we want to have a string, but an evaluated expression can be of any type. We solve this problem by using the type of

the expression to tailor a special function to convert the value of the expression to a string. If the expression to be converted E has type T (as determined by type checking the syntax tree E) a special purpose function *show_T*, of type $T \rightarrow String$ is made. The expression E is then transformed to *show_T E* before evaluation. This means that the evaluator effectively has the type $Expr \rightarrow String$, and we can guarantee type safety since it has been checked by the usual type checker.

It is always possible to construct a function that converts values of a particular type to a string if we do not guarantee that the string can be converted back. The user may supply her own *show_T* function to be used for printing a value of type T; if none has been supplied, a default function that just displays the type itself is provided by the system, e.g. without a *show_Tree* function any value of type *Tree* would be displayed as "⟨⟨*Tree*⟩⟩". If the type is polymorphic, the *show_T* function has as many arguments as there are type variables in T; each argument is a function to display the corresponding values.

3 Mixing compiled and interpreted code

In an interpreted system there is always a speed penalty compared with compiled code. It is therefore desirable to be able to mix interpreted and compiled code. Interpreted code is used during program development and debugging, because of speed of translation and easier debugging. When a program fragment works correctly it can be compiled (which may take some time) and then somehow incorporated ('loaded') into the interactive, interpreted system. The ability to mix interpreted and compiled code has been available in LISP systems for a long time.

The interactive LML also allows compiled code to be used. Compiling a file with the LML compiler produces an interface file that contains the type information and an object file (a standard UNIX '.o'-file). Using the compiled file requires information from both of them. The type information from the interface file is easy to use; it is an ordinary text file and it can just be read, and the information contained in it can be incorporated into suitable tables in the interactive system. The object file is much more complicated; it requires the code contained in the file to be added to a running program. This part of interactive LML is written in C. This part is basically a function

$$load :: String \rightarrow [(Symbol, VALUE)]$$

load takes a file name and returns an association list with the names and the objects that were contained in this object file.[4] The type *VALUE* does not really capture the type of the loaded objects, since they are not of a single type, but each has a specific type. This is the same kind of problem as with the *eval* function, and is handled in the same way. Here the actual type of each object is found in the interface file.

There is nothing strange about the *load* function from a puritanical point of view; it simply takes a string and returns an association list, and gives the same result when applied to the same argument.[5] This means that the *load* function does not destroy

[4] In reality it is, of course, more complex, since it has to deal with errors as well.

[5] Assuming that the file system is handled in a proper way, i.e. that rereading a file gives the same result. This is in fact not true in LML, but could easily be made true by buffering the file or by using system tokens (Augustsson, 1989).

referential transparency. The only strange thing about *load* is the way it is implemented.

The implementation of *load* is rather straightforward, but tedious on some machines. When an object file is loaded, memory is allocated for the new code and data parts of the file, the file is loaded into memory, and the relocation information is used to patch all address references to make them point to the right place. After relocation, external references have to be handled; the loaded file may contain references to previously loaded files and other routines in the runtime system. The *load* function has to keep a symbol table internally to facilitate this resolution of external references. Once a file has been loaded the objects in it will be usable, and will have exactly the same status and work in the same way as predefined values in the global environment (such as the combinators). This means that in effect they become new 'combinators'.

Loading of object files into a running program is fairly easy on ordinary UNIX machines where object files have a simple format. A condition for it to work is that code can be loaded into dynamically allocated memory and then executed; this is usually possible. Some care has to be exercised on some modern RISC machines, though, since some of them have separate caches for code and data, and the code is loaded (and cached) in data space, but later read (but not cached) from code space.

In the current implementation it is possible to call compiled code from interpreted code, but not vice versa. Calling interpreted code from compiled code would pose new problems. When the compiler has enough information it uses different (more efficient) calling conventions than the most general one which the interpreted code uses all the time. It would be possible to implement calling compiled code, but the need has not motivated the effort yet.

4 Exceptions

An exception occurs when a 'catastrophic' event, such as division by zero, occurs. In the original LML, as in several other lazy functional languages such as Haskell (Hudak *et al.*, 1992) and Miranda (Turner, 1985), the problem of exception handling is glossed over by stating that exceptions are semantically equivalent to $\perp$. But an implementation is free to handle the exception in a more sensible way by giving an error message instead of non-termination when an exception occurs. This is acceptable because the extra information is only available outside the program, and can never be inspected from within. So with a suitable abstraction, the output from the program is exactly what the semantics says it should be.

In LML exceptions arise from hardware traps, e.g. division of zero or interrupts, and exceptions generated in the program by the *fail* function. *Fail* takes a string argument and produces an exception. Normally an evaluation of an application of *fail* will terminate the program with the given error message.

The evaluation of an expression that the user types into the interactive system can produce an exception. The entire session with the interactive system is just the evaluation of a single expression, the top level of the system. Since the evaluation of a user expression is just a part of evaluating the bigger expression that is the whole

session with the user, the usual way of handling the exception would terminate the interactive system. This is, of course, unacceptable; exceptions are probably going to be quite frequent when testing new programs, and terminating on any kind of error would be a disaster. It must also be possible to interrupt an evaluation if the output is unwanted (or perhaps infinite).

Exception handling and lazy evaluation do not go very well together. To find out if the evaluation of an expression can ever cause an exception it must be fully evaluated. This is in contrast with lazy evaluation, which normally only evaluates to WHNF. There have been several suggestions for adding exception handling in a lazy language (e.g. Reeves *et al.*, 1989).

A very simple kind of exception handling has been added to LML to support writing interactive systems (among other things). The exception mechanism has the advantage that it is easy to explain semantically, and it is quite cheap[6] and easy to implement. We did not seriously consider implementing any other mechanism since we wanted the simplicity and cheapness. It is a mechanism with a simple semantics, but it is somewhat cumbersome to use.

To add the exceptions the value domain has to be augmented with a single extra element (or if you prefer to view each type as a separate domain, each domain has been augmented), called Error. This element is just above $\perp$ and is incomparable with all other elements (see Fig. 7).

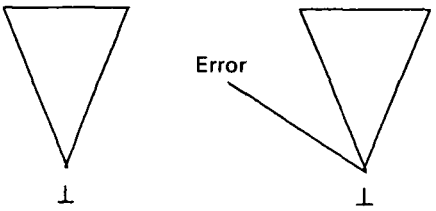


Fig. 7. The original and augmented value domain.

The denotational semantics of LML (given in Augustsson, 1987) has to be changed in a few places:

- All strict operations have to propagate Error. Function application, for instance, has to check if the function is Error. If it is, the value of the application has to be Error to propagate the exception. The only strict operations in LML (apart from strict predefined functions) are function application (strict in the function) and case expressions (strict in the scrutinized expression). Thus, the modification of the semantics is quite small (an example is shown in Fig. 8). By letting all strict operations preserve the exception it will propagate up until it is caught. The propagation of Error is quite similar to the propagation of $\perp$.
- All primitive functions that previously gave the result $\perp$ on failure (e.g. division by zero) have to return Error instead.

[6] The cost when no exception occurs is only in the call to *catch*, and that is about 35 machine instructions. If an exception occurs about 20 additional instructions are executed.

https://doi.org/10.1017/S0956796800000617 Published online by Cambridge University Press

$$\mathscr{E}[\![e_1\, e_2]\!]\,\rho = \varepsilon \in F \to (\varepsilon \mid F)\,(\mathscr{E}[\![e_2]\!]\,\rho);\ \bot \quad \text{Original}$$
$$\textbf{where}\ \varepsilon = \mathscr{E}[\![e_1]\!]\,\rho$$

$$\mathscr{E}[\![e_1\, e_2]\!]\,\rho = \varepsilon \in F \to (\varepsilon \mid F)\,(\mathscr{E}[\![e_2]\!]\,\rho);\ \varepsilon = \mathit{Error} \to \mathit{Error};\ \bot \quad \text{Modified}$$
$$\textbf{where}\ \varepsilon = \mathscr{E}[\![e_1]\!]\,\rho$$

$$\mathscr{E}[\![\mathit{catch}\, e_1\, e_2]\!]\,\rho = \varepsilon = \mathit{Error} \to \mathscr{E}[\![e_2]\!]\,\rho;\ \varepsilon \quad \text{New equation}$$
$$\textbf{where}\ \varepsilon = \mathscr{E}[\![e_1]\!]\,\rho$$

Fig. 8. Modified semantic equation for application, new equation for *catch*.

- A mechanism testing for Error has to be introduced to catch exceptions. It is called *catch*, and is described further below.

The explanation above is somewhat simplified. The real implementation uses a family of exception values, each of which carries a string which is used to give more information about the error. This complicates things somewhat, but the same basic idea still holds.

Note that there is no way to pattern match on Error; the only way to check for it is through a new function, *catch*, which is the only strict function that does not propagate the error value. It has type

$$\mathit{catch} :: a \to (\mathit{String} \to a) \to a$$

catch takes two arguments. The first argument is evaluated to weak head normal form (WHNF): if it is not an error value this is the result of the call to *catch*; if it is an error the second argument gets applied to the error string, and this is the result of the call to catch.

A few examples showing *catch* in action:

$$\begin{aligned} &\mathit{catch}\ (1+2)\ (\lambda x \to 0) && \Rightarrow\ 3\\ &\mathit{catch}\ (\mathit{fail}\ \text{“}\mathit{no}\text{”})\ (\lambda x \to x +\!\!+\ \text{“!”}) && \Rightarrow\ \text{“}\mathit{no}!\text{”}\\ &\mathit{catch}\ (1+\mathit{fail}\ \text{“}\mathit{no}\text{”})\ (\lambda x \to 0) && \Rightarrow\ 0 \end{aligned}$$

Using this error handling mechanism is somewhat tedious, since *catch* only evaluates its first argument to WHNF, thus only an error during this computation will be caught, and no deeper exceptions in the first argument. This means that, for instance, *catch* (1/0, 1) ($\lambda m.(0, 0)$) evaluates to (1/0, 1), since the pair is in WHNF; no evaluation takes place inside the pair so there is no error value. An exception can thus 'escape' from its handler in a perhaps surprising way. One way to avoid this is to call *catch* in such a way that the first argument forces complete evaluation of itself before returning the WHNF, but this is not lazy enough for applications like the top loop in the interactive system.

With regard to referential transparency, there are two problems with the error handling: evaluation order and interrupts. In the simple model where there is only a single error value, the evaluation order of, for example + does not matter, but when the error messages are introduced it suddenly does. The expression *catch* (*itos* (*fail* "*left*" + *fail* "*right*")) ($\lambda s.s$) will produce different results depending on the

https://doi.org/10.1017/S0956796800000617 Published online by Cambridge University Press

evaluation order (this problem was present before exception handling was introduced, but then all exceptions were identified internally, and only an external observer could see the difference). The problem is that the compiler makes the assumption that evaluation order does not matter, since this simplifies program transformation. It may thus rearrange expressions in such a way that the result is not correct, but this only happens if there are exceptions caught. If exceptions are used as they are intended in LML, as a way to signal catastrophic failure, this is not of any practical consequence, since one catastrophe is as bad as another.

The other problem is interrupts. An interrupt is treated as yet another exception, but it is really of a quite different nature. It emanates from a source that is external to the program, and you cannot rely on it to be generated at the same point when evaluating two expressions with the same denotation. If the evaluation of *catch* e_1 ($\lambda x.e_2$) is interrupted the value will be v_2 (where v_2 is the WHNF of e_2); another evaluation of the same expression will, if it is uninterrupted, result in v_1 (where v_1 is the WHNF of e_1). This means that two occurrences of the same expression can have different values.

We have simply accepted these two problems without trying to give any solution. Since the exception handling is (so far) isolated to a few programs and otherwise discouraged, this uncleanness is almost acceptable.

A simplified version of the top loop of the interactive system is presented in Fig. 9. The *topPrint* function takes a string and evaluates it fully (all conses and all

```
topPrint  ::  String → String
topPrint s  =
  case catch (strictHead s) (λmsg → "Failure:" ++ msg) of
    []    →  []
    c:cs  →  c:topPrint cs

strictHead []                  =  []
strictHead (c:cs)|c  ==  c  =  c:cs --force evaluation of c
```

Fig. 9. The top loop.

characters). If there is no exception this string is the result; otherwise, the string up to the point where the error occurred, concatenated with an error message is the result. The *strictHead* function ensures that both the cons and the character are evaluated before the cons is returned. This ensures that a failure in the evaluation of *"c"* cannot escape the handler. A new handler has to be inserted for each character; it is not possible to force *"s"* completely in the *catch*, since this would not handle lazy lists properly.

4.1 Exception implementation

The implementation of *catch* is quite easy, and uses standard techniques in exception handling. Before the evaluation of the first argument of a *catch* the current context (i.e. a suitable program counter and stack pointer) are stored on an auxiliary stack.

https://doi.org/10.1017/S0956796800000617 Published online by Cambridge University Press

If an exception occurs the context on top of the stack is popped and reinstalled, and the exception handler is thus entered. If evaluation of the first argument returns in a normal way the current context is popped off the stack and thrown away.

This implements the semantics faithfully and efficiently. There is no testing for error values going on as is indicated by the semantics. Instead the catch routine is entered at once when an exception occurs. This is possible because as soon as an exception has occurred it has to propagate upwards, and it will be caught by the last called catcher.

The only problematic part is the interrupt exception, since this may occur at any time. It is not possible to just enter the exception handler as soon as the interrupt occurs, since the runtime machinery may then end up in an inconsistent state (the interrupt could, for example, occur in the middle of a garbage collection). The interrupt handler must instead set a flag that is polled from time to time when the runtime machinery can handle an exception. Fortunately, this can be handled without any additional cost. There is already a check that is made quite frequently, namely the check to determine if there is any memory left in the heap. The interrupt handler fakes an out of heap condition (by changing a global pointer that points to the end of the heap), and when this is checked (soon afterwards) the garbage collector will be entered, but only to raise the interrupt exception.

5 Implementation

The interactive LML system is based on the LML compiler, described in Johnsson (1987), Augustsson (1987) and Augustsson and Johnsson (1989). The LML compiler is a batch oriented compiler which reads a source file and (ultimately) produces an executable program. The compiler is written almost exclusively in LML. It has code generators for several different target architectures, and runs on many platforms.

The compiler is divided into many passes, most of which are shared by interactive LML. The most important passes are (described in more detail in Johnsson, 1987, and Augustsson 1987):

Parsing

The parser is not written in LML but with YACC, and runs as a separate program which parses and then outputs the syntax tree for further processing by the real compiler. The reason for this is purely historical, and the current parser will soon be replaced by one written in LML.

Renaming

The renaming pass checks that all scope rules are obeyed, and also gives unique names to all identifiers in the program. This makes program transformations easier.

Program transformations

There are many passes that do program transformations, e.g. transforming case expressions to a simpler form, elimination of complex declarators, constant folding. These transformations constitute a large part of the compiler (Augustsson, 1987).

Type checking

The type checker is an ordinary Hindley/Milner type checker (Milner, 1978). The information obtained by the type checker is used to improve code generation.

Table 1. Execution time for some simple benchmark programs. Normalized with respect to compiled LML

	Program		
	NFib	Mirror	Queens
Compiled LML 0.997.5	4·5 (1·0)	4·6 (1·0)	1·2 (1·0)
Interpreted LML 0.997.5	86 (19)	65 (14)	62 (51)
Miranda† 2.014	142 (31)	104 (23)	38 (32)

† Miranda is a trademark of Research Software Ltd.

Lambda lifting

The lambda lifting turns the program into a set of supercombinators (Johnsson, 1985).

G-code generation

G-code generation produces a fairly high level intermediate code that is used for some further transformations (Johnsson, 1984).

M-code generation

The M-code is a fairly low level intermediate code on the level of ordinary machine instructions. The M-code generated depends on the target machine (Johnsson, 1986).

Target code generation

Finally, the M-code is turned into assembly code for the target machine, this is almost 'pretty printing'.

The interactive LML reuses the compiler passes (see above) from parsing to type checking without change. It is only the top level which calls the different passes that is slightly changed:

Parsing

The parser is essentially the same as for the compiler (it is in fact the same program, but with an extra flag). It has just been adapted for interactive use. This requires error recovery, since an error must not end the parsing process. Interactive LML also has a different top level syntax.

Renaming

Unchanged.

Program transformations

Unchanged.

Type checking

Unchanged, but see section 2.3 on type safety.

Constructor/case elimination

New pass. This pass eliminates all use of constructors and case expressions. They are replaced by special functions which ensure that all data types used in interpreted code will get the same representation as used in compiled code.

Combinator abstraction

New pass. This is an ordinary combinator abstraction using SKI etc.

https://doi.org/10.1017/S0956796800000617 Published online by Cambridge University Press

These passes constitute the *transform* function as described in previous sections. The passes shared with the compiler comprise by far the largest parts of the system (more than 90%).

The interactive system consists of a large part of the old LML compiler, plus about 1200 lines of LML, 500 lines of C (the C code implements the *load* function), and 500 lines of M-code (for the generic constructor/destructor functions). This must be considered to be a quite economical way of making an interactive system.

The performance of the interactive system is not as good as for compiled code; a typical slowdown for interpreted code is 15–50 (see Table 1). A space comparison is very difficult to make, since the size of the interpreted code is not available, but practical use indicates that it is not a problem. If speed is of importance, parts of the program may be compiled and loaded without loss of compiled efficiency.

5.1 Problems

The interactive LML system suffers from a number of problems, some of which are shared by other interactive systems, and some new ones.

We have already touched upon the problem of the loss of referential transparency when the ability to catch failures is introduced. This is not of concern in systems for LISP or SML which do not have the referential transparency property in the first place.

Another problem which is also present in SML systems but not in LISP systems is the static binding at the top level. Since both LML and SML have static binding it is natural to extend this to the top level. This makes program development more difficult, since when a bug is found in a previously defined function it is impossible to make a new definition of that function and have all functions defined in terms of its benefit from that. Instead, the first definition has to be redefined, and then all those definitions depending on that (directly or through other functions) have to be redefined (but with definitions identical to the old ones!). This is annoying, but could be handled by keeping track of dependencies and doing the redefinitions automatically, or by binding identifiers just before each evaluation. This is similar to what the *make* program does.

The command language used in the interactive system is very poor, and it is not extensible. In SML and LISP systems the command language is easily extensible, since new commands can be functions with implicit side-effects, something that is impossible in LML. It would be possible to introduce an extensible command language, but we have not done this since the preferred way of using interactive LML is inside a powerful editor like EMACS, where it is possible to write suitable code to make 'extensions' of the command language. This way of working is also used heavily for LISP and SML.

There is no support for traditional debugging. It is possible to have more elaborate debugging mechanisms, but the exact implementation of these is still under discussion. The problem is that we still want to stay within the pure framework and also be efficient.

6 Conclusions

The interactive LML system was produced from an existing compiler with comparatively little effort. Using largely the same parts in the compiler and the interactive system means that new language features and changes will track each other perfectly. The interactive system is primitive, but it provides a programming environment that is easier to use during program development because it gives rapid responses. Compiling and running a trivial expression like "2+2" takes 4–5 seconds with the compiler, whereas the response is instantaneous with the interactive system. This kind of responsiveness is psychologically very important. The practical experience of the system, although limited, has been very positive. Several people who were reluctant to use LML when there was only a batch compiler became LML users when the interactive system appeared.

Acknowledgements

I would like to thank Thomas Johnsson who was the co-implementor of the LML compiler, the Programming Methodology Group for a stimulating environment, the anonymous referees for many good comments and suggestions, and all the LML users for their bug reports.

References

Appel, W. and MacQueen, D. B. 1987. A standard ML compiler. In *Proc. Functional Programming Languages and computer Architecture*, pp. 301–324. Volume 274 of *Lecture Notes in Computer Science*. Springer-Verlag.

Augustsson, L. 1985. SMALL, a small interactive functional system. PMG Report 28, Department of Computer Sciences, Chalmers University of Technology (Sept).

Augustsson, L. 1987. *Compiling Lazy functional Languages, Part II.* PhD thesis, Department of Computer Science, Chalmers University of Technology (Nov.).

Augustsson, L. 1989. Functional I/O Using System Tokens. PMG Memo 72, Department of Computer Sciences, Chalmers University of Technology.

Augustsson, L. and Johnsson, T. 1989. The Chalmers Lazy-ML compiler. *The Computer Journal*, **32** (2): 127–141.

Hudak, P. *et al.* 1992 *Report on the Functional Programming Language Haskell*, Version 1.2 (Mar.).

Johnson, S. C. 1975. Yacc – Yet Another Compiler Compiler. Technical Report 32, Bell Labs. (Also in UNIX Programmer's Manual, Volume 2B.)

Johnsson, T. 1984. Efficient compilation of lazy evaluation. In *Proc. SIGPLAN Symp. on Compiler Construction*, pp. 58–69, Montreal.

Johnsson, T. 1985. Lambda lifting: transforming programs to recursive equations. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*. Vol 201 of *Lecture Notes in Computer Science*. Springer-Verlag.

Johnsson, T. 1986. Code generation from G-machine code. In *Proc. Workshop on Graph Reduction*. vol. 279 of *Lecture Notes in Computer Science*. Springer-Verlag.

Johnsson, T. 1987. *Compiling Lazy Functional Languages.* Ph.D. thesis, Department of Computer Sciences, Chalmers University of Technology (Feb.).

Milner, R. 1978. A theory of type polymorphism in programming. *J. Computer and Systems Sciences*, **17**: 348–375.

Milner, R., Tofte, M. and Harper, R. 1990. *The Definition of Standard ML.* MIT Press.

Reeves, A. C., Harrison, D. A., Sinclair, A. F. and Williamson, P. 1989. Gerald: an exceptional lazy functional programming language. In K. Davis and J. Hughes (editors), *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Fraserburgh, Scotland (Aug.).

Steele, G. L. 1984. *Common LISP – the language*. Digital Press.

Turner, D. A. 1979. A new implementation technique for applicative languages. *Software – Practice and Experience*, **9**: 31–49.

Turner, D. A. 1985. Miranda: A non-strict language with polymorphic types. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*, pp. 1–16.

Availability

The LML and Haskell compiler and the interactive system is available with anonymous ftp from ***ftp.cs.chalmers.se*** in ***pub/haskell/chalmers.***

https://doi.org/10.1017/S0956796800000617 Published online by Cambridge University Press