Eliminating Spurious Error Messages Using Exceptions, Polymorphism, and Higher-Order Functions

Norman Ramsey

Department of Computer Science, University of Virginia, Charlottesville, Virginia 22903 USA
Email: nr@cs.virginia.edu

Many language processors make assumptions after detecting an error. If the assumptions are invalid, a compiler may issue a cascade of error messages in which only the first represents a true error in the input; later messages are side effects of the original error. Eliminating such spurious error messages requires keeping track of values within the compiler that are not available because of a previously detected error. Examples include symbol-table entries, types, and intermediate code.

This paper presents a discipline for tracking unavailable values and avoiding cascading error messages. The discipline extends the *error monad* of Spivey [1] and Wadler [2]. The extension is expressed formally as a type constructor and combinators written in Standard ML. The type constructor distinguishes intermediate results that are unavailable because of a previously detected error. The combinators transform ordinary functions, which assume all intermediate results are available, into functions that silently propagate the unavailability of intermediate results. In an ML implementation, the ML type rules enforce the discipline; if the compiler writer does not account for a potentially unavailable value, the source code of the compiler does not type-check.

The cost of the discipline is negligible. In an experimental compiler, the discipline adds at most 5–10% to total compile time, and about 1% in the common case in which no errors are detected.

Received 17 April 1997; revised 9 February 1999 ; accepted 10 May 1999

1. INTRODUCTION

It can be difficult to write a compiler or other language processor that issues more than one trustworthy error message per run. If an intermediate result is wrong because of an error in the input, the compiler may complain not only about the original error, but about other errors that follow from its attempt to process the faulty intermediate result. The worst offenders, like early Pascal compilers, are so bad that users learn to disregard all but the first error message. Even modern compilers written by smart people are not immune. One widely used C compiler assumes that undeclared identifiers are integers; when this assumption is unreasonable, the compiler sprays error messages. Another compiler prints messages identifying faulty inputs as "bogus," but it complains multiple times about the same bogosities. One can eliminate spurious error messages by halting after detecting one error, but this solution is acceptable only if the compiler is very fast. The ideal is for a compiler to detect every "real" error in its input, but never to issue a spurious message.

This paper describes an implementation technique that helps compiler writers approach the ideal; in principle, a compiler using this technique can detect every error not made undetectable by a previously detected error. Moreover, a compiler writer need not apply this technique from the beginning; it can be retrofitted to compilers that assume no errors occur. The technique is not needed for parsing; it helps in later phases, like static semantic analysis and generation of intermediate code. The technique need not be applied to even later phases, like optimization and code generation, because these phases are normally executed only on intermediate results obtained from valid inputs.

The technique itself is a simple programming discipline: keep track of intermediate results that are unavailable because of a previously detected error. The idea behind this discipline is old; for example, Horning [3] recommends that spurious error messages related to an identifier be suppressed by entering the identifier into the symbol table with a special "error entry" flag. Johnson and Runciman [4] extend the idea beyond symbol-table entries; when their compiler detects a semantic error, "offending parts of the internal program representation are replaced by substitutes that cannot arise from an error-free program."

To implement this discipline, we extend the *error monad*, which Spivey [1] uses to handle exceptional behaviors in lazy languages, and which Wadler [2] uses to support error termination in pure languages. The extension is suitable for use in eager languages, including languages with imperative features like exceptions. The paper presents an illustrative implementation in the modern functional language Standard ML [5]. Because ML supports exceptions, higher-order functions, parametric polymorphism, and static type checking, it provides a powerful, concise, formal, and executable notation for talking about the discipline, and the ML compiler can check that the discipline is applied correctly. The paper briefly discusses using the discipline in other programming languages. Languages like C++ and Generic Java [6] have enough features to support the full discipline, and languages like Java, Modula-3, or Ada 95 can support a weaker form of the discipline at the cost of some dynamic checking and indirection.

2. REPRESENTING FAULTY INTERMEDIATE RESULTS

The essence of the technique is to write most functions under the assumption that their arguments are always valid, then to apply combinators to handle cases in which the arguments may have been invalidated by a previously detected error. We use the exception `Error` to flag new errors that should be reported, and we use the type `'a error` to represent a value of type `'a` that may be invalid because of a previous error. These can be expressed in ML as follows:[2]

```
exception Error of string
datatype 'a error = ERROR | OK of 'a
```

The `'a` in `'a error`, pronounced "alpha," is an ML *type variable*. The `datatype` definition creates a *polymorphic* type, i.e., one that can be instantiated with any type. Instantiation means replacing the `'a` with another type. So, for example, if type `exp` represents expressions, type `exp error` represents values that are either `ERROR` or `OK e`, where `e` has type `exp`. The name `error` by itself is normally called a *type constructor*, since it is applied (postfix) to an argument to produce a type.

A function that is aware of error propagation should do one of three things:

A. Return `OK x` when the function can return a valid value `x`. For example, a symbol-table lookup function might return `OK x` when it finds something valid in the symbol table.
B. Raise `Error` (with a suitable error message) when the function detects an error that is *not* the result of a previously detected error. A symbol-table lookup function might raise `Error` if it fails to find an identifier in the symbol table. (The handler for the exception might then bind that identifier to `ERROR`.)
C. Return `ERROR` if the function cannot return a value because some necessary intermediate result is unavailable as a consequence of a previously detected error. In particular, the function should return `ERROR` if any of its arguments is `ERROR`. A symbol-table lookup function might return `ERROR` if it finds `ERROR` in the symbol table.

The paper returns to these cases below.

Using `ERROR` makes it easy for a function to report that it cannot provide a value, not because of a new error, but because of one that has already been reported. A function that uses `ERROR` can be identified by the `error` constructor in its type; for example, if `denotable` is the type of things that can be found in a symbol table, the symbol-table lookup function described in the example might have type `name -> denotable error`. (The arrow type constructor `->` is built into ML; the type `'a -> 'b` is the type of functions from values of type `'a` to values of type `'b`.)

3. PROPAGATING ERROR INFORMATION WITH COMBINATORS

Functions that don't expect values of type `'a error` can be made to handle them by using the combinators described in this section. One set of combinators manages functions' arguments and results; another set manages values of type `'a error` when they appear in data structures.

3.1. Functions' arguments and results

If every result or argument in the compiler is potentially `OK` or `ERROR`, there is a bookkeeping problem: every argument and result must be checked for validity. Users of Unix system calls will recognize this problem as the check for a returned value of -1. Higher-order functions can encapsulate the bookkeeping in a few combinators; other functions accept only valid arguments, and they may or may not have to wrap `OK` around valid results. In other words, one can write functions that are "unaware of error propagation," then use combinators to make them aware of error propagation in the sense used above. The type rules of ML guarantee that the combinators are used correctly; moreover, looking at the difference between the expected and actual types of an argument or result can suggest which combinator should be used.

As an example, consider converting integer expressions from abstract syntax to some intermediate form of type `exp`. A function

```
val elaborate : abstract_syntax -> exp error
```

[2] This paper explains most ML constructs as they are introduced. For more detailed explanations, Ullman [7] presents ML at a level suitable for understanding this paper.

could return ERROR (case C above) because the abstract syntax could contain an improperly declared identifier bound to ERROR in the symbol table. Cases A or B might also apply, e.g., return OK e (A) if the abstract-syntax tree elaborates correctly, or raise Error (B) if the abstract-syntax tree breaks a static-semantic rule. (The val declaration could appear in an interface to indicate that the interface provides a value of type abstract_syntax -> exp error. ML interfaces use the val syntax to declare both functions and values.)

Application, by contrast with elaboration, cannot be invalidated by a bad identifier in the symbol table, because if an identifier is bad, we never attempt to apply an operator to it. Therefore, when we apply an operator to a list of expressions, the result is always a valid expression or a newly detected error; OK, ERROR, and the error type constructor are unneeded.

```
val apply : operator -> exp list -> exp
```

This is an example of a *curried* function; when apply is given an argument rator of type operator, it returns another function of type exp list -> exp. This anonymous function can then be applied to a list of expressions, as in apply rator exps. The result has type exp, not type exp error—if apply's arguments are valid, it can always return a valid x or raise Error.

Part of the implementation of elaborate is the application of an operator to its (elaborated) operands. We would like simply to compose elaborate and apply, but we cannot write

```
fun elaborate (APPLY (operator, operands)) =
  apply operator (map elaborate operands)
  (* incorrect -- does not type-check *)
```

because map elaborate operands returns a value of type exp error list and apply operator expects one of type exp list.

The definition of elaborate shows how fun is used to define functions. ML functions are defined by *pattern matching*; the function definition contains a list of alternatives, each with a pattern on the left and a result on the right. The definition above shows just one alternative. When the function is applied, the implementation searches the alternatives for the first pattern that matches the structure of the argument, and it then evaluates the corresponding right-hand side, the value of which is the result of the application. The right-hand side in this example is incorrect, because it does not type check.

Two combinators help. The first, liftList, has type 'a error list -> 'a list error, and it applies the obvious rule that the list is OK if and only if all its elements are OK. This and related "lifting" combinators are discussed below. The second, more interesting combinator does for error what map does for list:

```
val emap
  : ('a -> 'b) -> 'a error -> 'b error
  = fn f => (fn OK x  => OK (f x)
              | ERROR => ERROR)
```

The emap combinator takes a function f and a value. If the value is OK x, emap unwraps x, applies f to x, and wraps the result in OK. Otherwise, if the value is ERROR, emap returns ERROR without applying f. Using emap prevents f from issuing spurious error messages, because f is called only when its arguments are valid.

The definition of emap shows how val is used to define functions. ML's fn and => are the syntax used to define anonymous functions; fn f => corresponds roughly to $\lambda f.$ in the λ-calculus, except that fn can work with a list of alternatives, just as fun can. The nested fn's show the higher-order nature of emap. The advantage of the val syntax is that it is easy to include the type annotation, between : and =, which gives the type of emap explicitly. It is not necessary to include this type, because the ML implementation can infer it, but it is very useful documentation for programmers. The type variables 'a and 'b indicate that emap has a polymorphic type, i.e., it can be applied to values of more than one type. For example, the type shows that emap can be applied to any function.

The names of liftList, emap, and the other error combinators are designed to appeal to programmers' experience with list combinators (map), to appeal to programmers' intuition about sequential checking for errors, or to appeal to the way the combinators manipulate type constructors. As discussed in Section 5.1, the theory of monads suggests other names for some of the error combinators.

To return to our example, apply operator has type exp list -> exp, and so the function emap (apply operator) has type exp list error -> exp error. The function liftList o map elaborate has type exp list -> exp list error, and using the combinators we can now compose apply and elaborate:

```
fun elaborate (APPLY (operator, operands)) =
  emap (apply operator)
       (liftList (map elaborate operands))
```

A simpler example shows emap alone. To elaborate integer expressions, one must look up identifiers in the symbol table. Since identifiers can represent other things besides integers, we need a projection function. Again, we write it in a simple form, assuming that its argument is always valid:

```
val projectInt : denotable -> exp =
  fn (INTEGER e) => e
   | _ => raise Error "integer expected"
```

The _ is a "wildcard" pattern, which matches anything.

If the symbol table can contain ERROR, perhaps as a result of an invalid declaration, then a lookup function should have type name -> denotable error, and we use emap to write another case for elaborate:

```
  | elaborate (NAME n) =
      emap projectInt (lookup n)
```

Here we elaborate an identifier by looking it up and insisting that it stand for an integer.

It's probably easiest to think about the combinators not in terms of what they do at run time, but in terms of how they affect the types of the values and functions to which they are applied. If `emap` is applied to a function `f`, it applies the type constructor `error` to `f`'s argument and result types. If `f`'s result type is already an application of `error`, then the result type of `emap f` contains successive applications of `error`. For example, the abstract-syntax tree for an expression might not be available, perhaps because of an error during parsing. One could apply `emap` to `elaborate`, but `emap elaborate` has type `abstract_syntax error -> exp error error`, which is inconvenient. The `combine` combinator combines the two applications of `error` into one. It is written

```
val combine : 'a error error -> 'a error =
  fn ERROR => ERROR | OK x => x
```

The anonymous function `combine o emap elaborate` has the type we want our elaboration function to have, viz., `abstract_syntax error -> exp error`. Applying `combine` to the result of `emap f` is such a common operation that for convenience we present it as a separate combinator:

```
val sequence
  : ('a -> 'b error)->('a error -> 'b error)
  = fn f => combine o emap f
```

The intuition behind the name `sequence` is that the original `f` may be applied *after* some previous computation has produced a value of type `'a error`. If, however, that previous computation failed to produce a value, but produced `ERROR` instead, then `f` is not applied.

The combinators presented so far handle *arguments* that have type `'a error` when we prefer `'a`. To handle *results* that have type `'a error` when we prefer `'a`, we introduce the `errorEnv` combinator, which uses exceptions to abort computations in which important functions return `ERROR`. For example, in a simple compiler, we might write code generation as a function of symbol table and intermediate form:

```
val codegen
  : (name -> denotable) -> exp -> code
```

where `name -> denotable` is the type of the symbol table. Unfortunately, we cannot pass `lookup` to `codegen` because `lookup` has type `name -> denotable error`. By applying the `errorEnv` combinator, we can use `lookup`; the function `errorEnv codegen lookup` has type `exp -> code error`.

The `errorEnv` combinator takes arguments `f rho x`, where `rho` has type `'a -> 'b error`. It works by creating a new `rho'` of type `'a -> 'b` to use within `f`. The internal function `rho'` applies `rho`, then strips off `OK`. If `rho` ever returns `ERROR`, `rho'` raises the local exception `RhoFailed`, and `errorEnv` aborts execution of `f`, returning `ERROR`.

```
val errorEnv
  : (('a -> 'b) -> 'c -> 'd)
    -> ('a -> 'b error) -> 'c -> 'd error
  = fn f => fn rho => fn x =>
      let exception RhoFailed
          fun rho' n =
            case rho n
              of OK x => x
               | ERROR => raise RhoFailed
      in  OK (f rho' x)
          handle RhoFailed => ERROR
      end
```

The sudden termination of `f`'s execution keeps `f` from issuing spurious messages. This trick is defined only once, in `errorEnv`, but thanks to higher-order functions it works at every call site of `rho`, no matter how deeply the call is hidden within `f`. In the body of `errorEnv`, `OK` is applied to `f rho' x` to make sure it, too has type `'d error`. To use `errorEnv` in a context where `'d` is already an `error` type, apply `combine` to the result.

Returning to the example, the symbol-table function `lookup` has type `name -> denotable error`, so the anonymous function `errorEnv codegen lookup` has type `exp -> code error`, and the result of applying `sequence` has type `exp error -> code error`. By using all the combinators, we can propagate the error information through stages of parsing, elaboration, and code generation:[3]

```
val elaborateAndGenerate
  : (name -> denotable error)
    -> abstract_syntax error -> code error
  = fn lookup =>
      let fun fromAst ast =
            sequence
              (errorEnv codegen lookup)
              (elaborateFrom lookup ast)
      in  sequence fromAst
      end
```

Propagating the error information becomes a matter of applying the correct combinators. Programmers don't need to worry about getting them wrong, because the type checker prevents the combinators from being used incorrectly. The summary in Table 1 can help programmers figure out which combinators to apply when code does not type-check.

3.2. "Lifting" `ERROR` out of data structures

Not all values are passed from one function to another; some are put into data structures. An example above shows that elaborating a list of expressions produces a list of results, with type `exp error list`. It is useful to be able to elaborate each expression separately, so that each expression can be checked for errors independently—*producing* `exp error list`

[3]The function `elaborateFrom`, the definition of which is not shown, accepts a `lookup` function and returns `elaborate`.

```
exception Error of string
val ERROR    : 'a error
val OK       : 'a -> 'a error
val combine  : 'a error error -> 'a error
val emap     : ('a -> 'b      ) -> 'a error -> 'b error
val sequence : ('a -> 'b error) -> 'a error -> 'b error
val errorEnv : (('a -> 'b) -> 'c -> 'd) -> ('a -> 'b error) -> 'c -> 'd error
val catch    :       (string->unit) -> ('a -> 'b error) -> 'a -> 'b error
val catch'   : 'b -> (string->unit) -> ('a -> 'b)       -> 'a -> 'b

val liftList   : 'a error list -> 'a list error
val liftOption : 'a error option -> 'a option error
val liftPair   : 'a error * 'b error -> ('a * 'b) error
val liftLeft   : 'a error * 'b       -> ('a * 'b) error
val liftRight  : 'a       * 'b error -> ('a * 'b) error
```

TABLE 1. Table of combinators

is a good idea. Unfortunately, such a list is awkward to *consume*—a value of type `exp list error` is much easier to consume. For example, the function `apply operator` expects a value of type `exp list`, and as shown above, the function `emap (apply operator)` expects a value of type `exp list error`. To get such a value, we use the combinator `liftList`, which transforms `'a error list` into `'a list error`. The name "lift" suggests not only the transformation on the type, in that the `error` type constructor is lifted to a higher position, but also the action of the combinator in searching for the `ERROR` value and "lifting" it out of the list to make it the result.

```
val liftList
  : 'a error list -> 'a list error
  = fn l =>
      let fun lift (ERROR::_, _) = ERROR
            | lift (OK h::t, prev') =
                lift(t, h::prev')
            | lift ([], prev') =
                OK (rev prev')
      in  lift(l, [])
      end
```

The auxiliary, nested function `lift` walks the list until it encounters `ERROR` or the end of the list. It uses an "accumulating parameter," `prev'`, to hold the list elements already seen and determined to be `OK`. This list is stored in reverse order. (The prime in `prev'` suggests list reversal.) The accumulating parameter is a standard idiom in functional programming; it makes `lift` tail recursive, which makes it possible for an ML compiler to translate `lift` into a tight machine-code loop.

ML provides built-in tuple types, along with special notation for tuples containing any number of values. For example, if `x` has type `a`, `y` has type `b`, and `z` has type `c`, then `(x, y, z)` is a 3-tuple whose type is written `a * b * c`. Since tuples can contain error values, we provide combinators that manipulate such tuples. We show only combinators for pairs, which are the most commonly used tuples. These three combinators lift pairs in which the left element, the right element, or both elements can be `ERROR`.

```
val liftLeft
  : 'a error * 'b       -> ('a * 'b) error
  = fn (OK x,    y) => OK (x, y)
     | _            => ERROR

val liftRight
  : 'a       * 'b error -> ('a * 'b) error
  = fn (   x, OK y) => OK (x, y)
     | _            => ERROR

val liftPair
  : 'a error * 'b error -> ('a * 'b) error
  = fn (OK x, OK y) => OK (x, y)
     | _            => ERROR
```

Pairs may be used in compilers when elaborating declarations, e.g., to produce bindings. A compiler designed without error propagation in mind might create a binding using the expression `(id, elaborate ast)`, which has type `string * exp error`. The "lifted" expression `liftRight (id, elaborate ast)` has the more convenient type `(string * exp) error`. Lifting combinators for triples, quadruples, and so on may also be useful.

In addition to structures like lists and tuples, compilers also deal with symbol tables. Since symbol tables are finite structures, a table-lookup function must have way to indicate that an identifier is not present in the table. In ML, the conventional method is to return a value of type `'a option`, which is a standard type defined by

```
datatype 'a option = NONE | SOME of 'a
```

A lookup function should return `NONE` when there is no entry in the table, and `SOME x` otherwise. Since it is also possible for an identifier to be present in the table, but for it to have no value because of a previously reported error, the lookup function might return a value of type `'a error option`. The `liftOption` combinator turns this into a value of type `'a option error`.

```
val liftOption
  : 'a error option -> 'a option error
  = fn (NONE)          => OK NONE
     | (SOME (OK x)) => OK (SOME x)
     | (SOME ERROR)  => ERROR
```

Although `'a error` and `'a option` are isomorphic, it would be a mistake to identify them. Perhaps more than in other programs, types in ML programs are valued as documentation, and the `error` and `option` type constructors have quite different interpretations. For example, if `NONE` appears where a value is expected, the program being compiled is wrong, and the compiler should issue an error message. If `ERROR` appears where a value is expected, the program is also wrong, but an error message has already been issued, and another message would be spurious.

4. CONVERTING NEW ERRORS INTO ERROR

Propagating `ERROR` makes it possible to avoid applying functions to bad values, thereby not executing code that might issue spurious error messages. At some point, however, the compiler must issue real error messages. It does so by catching the `Error` exception, printing a message, and turning the exception into `ERROR`. Handling exceptions is better than checking return values or using special flags; by controlling the placement of handlers, the compiler writer controls the granularity of error detection. At one extreme, the compiler writer can put a single handler at top level, resulting in a compiler that detects one error, then halts. A more aggressive placement might protect the analysis of every sub-expression with a handler, so independent sub-expressions could be checked independently. Declarations and statements could be treated similarly. Most importantly, a compiler writer can improve the error-detecting ability of his compiler by gradually inserting more and more handlers, maintaining a working compiler at each step.

Handling the `Error` exception is highly stylized—it usually involves printing a message and returning the `ERROR` value. We could write such handlers directly, but it is easy to encapsulate the handling in another combinator.

```
val catch
  : (string -> unit)
    -> ('a -> 'b error) -> 'a -> 'b error
```

The `catch` combinator takes an error-printing function and applies another function within a handler. The type of the first argument, `string -> unit`, uses the built-in type `unit`, which refers to the empty tuple. The `unit` type has a single value, which is written `()` and also pronounced "unit." An ML function that returns `unit` is like a C function that returns `void`; it is executed for its side effect—in this case, printing an error message.

To show the use of `catch`, I return to the expression-elaboration example. A compiler can profitably use a special abstract-syntax tree node to mark the source region from which the abstract-syntax tree was derived. The function `printMsgAt` uses the region to print error messages that include line numbers:

```
  | elaborate (MARK (region, exp)) =
      catch (printMsgAt region) elaborate exp
```

The parser puts `MARK` nodes in appropriate places, so this single case handles all of the error reporting for `elaborate` and `apply`.

The `ERROR` value is not always the right thing to return when a computation is aborted because of a new error. For example, when processing a declaration, a compiler written in a functional language should return a new symbol table containing a binding for the identifier declared, and if something goes wrong during the elaboration of the binding, the correct value to return is a table in which the offending identifier is bound to `ERROR`. For that reason, `catch` is defined as a special case of `catch'`, which can return non-`ERROR` values:

```
fun catch' default print f x =
  f x
  handle Error msg => (print msg; default)
fun catch print f x = catch' ERROR print f x
```

A compiler can use `catch'` to update a symbol table. This example assumes a polymorphic, functional table, indexed by strings, and offering these functions for update and search:

```
val bind  : 'a table * name * 'a -> 'a table
val lookup: 'a table * name -> 'a option
```

The example compiler does not redefine identifiers, and it requires identifiers to be defined before use. The function `bindUnbound` defines a new identifier, raising `Error` if the identifier is defined already.

```
fun bindUnbound(table, name, value) =
  case lookup(table, name)
    of NONE   => bind(table, name, value)
     | SOME _ =>
         raise Error ("Identifier " ^ name
                      ^ " already defined")
```

The function `get` looks up an identifier, raising `Error` if the identifier has not been defined.

```
fun get table name =
  case lookup(table, name)
    of SOME x => x
     | NONE   =>
        raise Error ("Identifier " ^ name
                     ^ " is undefined")
```

A compiler can use these functions when processing a declaration. For example, the following fragment of function `processDecl` adds a binding to the symbol table. The annotations on `rhs` and `denoted` document their types.

```
fun processDecl table (INTDECL(name, ast))
  = let val rhs : exp error =
          catch print
                (elaborateFrom (get table))
                ast
        val denoted : denotable error =
          emap INTEGER rhs
    in  catch' (bind(table, name, ERROR))
               print
               bindUnbound
               (table, name, denoted)
    end
```

If the elaboration fails, `rhs` is `ERROR`, and `name` is bound to `ERROR` in the symbol table.[4] Whether the elaboration fails or not, if `name` already appears in `table`, `bindUnbound` raises `Error`, and `catch'` prints a message and returns a table in which `name` is bound to `ERROR`.

5. RELATED WORK

The `error` type and accompanying combinators extend a mathematical construct called a *monad*. This section of the paper discusses how the error combinators relate to monads in general, and in particular how they relate to Spivey's [1] work on the error monad. This section also discusses the relationship of this work to other techniques for error recovery.

5.1. Monads

Monads have recently drawn attention for their useful properties in specifying the semantics of programming languages [8], in structuring compilers and interpreters [9], and in reasoning about the equivalence of functional programs [2]. The most parsimonious formulation of a monad uses a type constructor M, together with *unit* and *bind* operations that satisfy certain mathematical laws. In ML, the type constructor and operations can be declared using `signature` and `sig`, as follows:

```
signature MONAD = sig
  type 'a M
  val unit : 'a -> 'a M
  val bind : 'a M -> ('a -> 'b M) -> 'b M
end
```

The `unit` and `bind` operations must satisfy three laws: left and right unit laws and an associative law. Wadler [2] uses this formulation to show that several impure language features—error handling, profiling and tracing, and I/O—can be introduced into a pure language by using monads. Wadler's paper emphasizes equational reasoning about programs, for which his formulation of monads is well suited.

There is an alternative formulation of monads that is more cumbersome mathematically, but it may be easier to program with.

```
signature MONAD' = sig
  type 'a M
  val unit : 'a -> 'a M
  val map  : ('a -> 'b) -> 'a M -> 'b M
  val join : 'a M M -> 'a M
end
```

The two operations `map` and `join` replace the single operation `bind`, and the operations must obey seven laws, not three. Spivey [1] uses the second formulation in his exploration of the error monad, which he uses to add exceptions to a pure, lazy language. The two formulations are equivalent; `bind` can be defined in terms of `map` and `join`, and `map` and `join` can be defined in terms of `unit` and `bind` [2].

Steele [9] uses monads and approximations thereto as a way of structuring interpreters. Steele's *pseudomonads* are related to monads, but unlike arbitrary monads, they can always be composed. Steele also demonstrates that monad-like structures can be useful even when their operations don't obey the monad laws.

The preceding papers use monads in the context of a pure, lazy language. In several cases, monads are used to provide features that are built into impure languages. Many of the combinators in this paper are monadic operations (Table 2), but this paper shows that monads can be useful even in an impure, strict language. In fact, this paper shows that the error monad can be useful in a language with exceptions—the very feature the error monad is sometimes used to replace. The technique presented here uses the error monad to support a disciplined, typechecked way of propagating error information. The problem with using *only* the error monad is that the entire compiler would have to be rewritten in monadic style; *every* function of type `t -> u` would have to be rewritten to have type `t -> u error`. While a number of our example functions naturally fit this scheme (e.g., `abstract_syntax -> exp error`), not all do. For example, it is often more convenient to leave a function of type `t -> u` untouched and to use `emap` to lift its type to `t error -> u error`. This strategy enables us to write two kinds of functions that cannot be written in the monadic style:

- A function that is oblivious to the possibility of error.
- A function that is oblivious to the possibility of its inputs being erroneous, and which returns only correct results, but which can signal a new error by raising the `Error` exception.

Using exceptions makes it possible to support a disciplined approach to error messages *without* requiring the entire compiler to be rewritten in monadic style.

[4] The astute reader will have noticed that, because ML is a strict language, the "default" value `bind(table, name, ERROR)` is computed whether it is needed or not. This computation might be expensive, and it would be more efficient to delay the computation by using the value `fn () => bind(table, name, ERROR)`, and by making the appropriate change in the definition and type of `catch'`.

Monads	Spivey	Ramsey
$\alpha\, M$	$\alpha\ maybe$	`'a error`
unit	*Just*	`OK`
$map\ f\ x$	$f \bullet x$	`emap f x`
join	*prop*	`combine`
$\lambda x.bind\,(f\ x)\ g$	$g \circ f$	`sequence g o f`
$bind\ x\ f$	$(f \circ id)\ x$	`sequence f x`
—	*Nothing*	`ERROR`
—	$x\ ?\ y$	—
—	$x\ ??\ y$	—

N.B. Functional composition $f \cdot g$ is written `f o g` in ML.

TABLE 2. Table of equivalents

In a language like Haskell [10], which provides special notation for working with monads [11], the burden of programming in monadic style is greatly reduced.

5.2. Spivey's exception combinators

Spivey [1] uses the error monad as a *replacement* for a built-in exception mechanism. His technique requires that any function that could discover an error be written in monadic style, rather than raise an exception. Spivey's primary motivation is to enable equational reasoning, even in the presence of exceptions. His technique makes it possible to use equational techniques to derive programs, transform them, and prove their properties.

Spivey presents combinators that are mostly equivalent to those used in this paper (see Table 2), and he gives algebraic laws that can be used with those combinators. Spivey's ? combinator has no direct equivalent in this paper; it is used to replace exception handlers. The expression $x\ ?\ y$ produces x unless x is an error value, in which case it produces y. This construction is useful only in a non-strict (lazy) language, in which y is evaluated only if x is determined to have failed. In languages like ML, C++, Java, and Ada, it is possible to implement this construction by simulating lazy computation, but it is more idiomatic to have x raise an exception if it fails, and to delay the evaluation of y by putting it in a handler.

Spivey's paper does not include an equivalent to `errorEnv`, which uses built-in exceptions to avoid having to rewrite the compiler in monadic style. The unusual type of `errorEnv` suggests that it may not be possible to write such a function in a pure setting.[5] In this case, a compiler writer has an alternative strategy, as advised by Wadler [2]: write a compiler that is oblivious of errors, but write it in monadic form—then use the error monad. If monadic form seems natural, this may be a good strategy.

Spivey uses the error monad for different purposes than this paper. Spivey uses it to develop a pure version of exceptions, whereas this paper uses it to suppress spurious error messages. To use the error-suppression technique in a pure setting, one could combine two instances of the error monad, or perhaps instead define

```
datatype 'a error_or_exn
  = OK of 'a
  | OLD_ERROR
  | FRESH_ERROR of string (* error message *)
```

Given this extended monad, one could define a pure combinator that does the same job as `catch`, i.e., convert `FRESH_ERROR` to `OLD_ERROR`, probably issuing a message in the process.

5.3. Recovery from syntactic errors

There are many published papers about recovery from syntactic errors, so that parsing can continue [12, 13]. Chapter 17 of Fischer and LeBlanc [14] surveys the field. By contrast, semantic errors and their processing are rarely discussed—perhaps because, as David Gries suggested twenty years ago, they haven't been formalized [15]. A handful of researchers have used attribute grammars to formalize semantic errors.

Adorni, Boccalatte, and Di Manzo [16, 17] extend LL(1) techniques for syntactic error recovery. Their algorithm, which combines parsing and attribute evaluation, can detect a semantic error at the first incorrect symbol; that is, the symbols parsed are a prefix of some program that is both syntactically and semantically correct. Their algorithm can also use semantic information to guide recovery, so that the edited stream of symbols will be both syntactically and semantically acceptable. Their paper notes that recovery may cause later (spurious) error messages, and that it is impossible to choose a recovery set that minimizes the probability of causing further errors. By contrast, the techniques in this paper do not attempt to recover from errors; instead, they help us create compilers that can digest incorrect constructs without causing further errors later on.

It would be possible to use the error monad to express the results of recovering from syntactic errors, e.g., by using `ERROR` nodes in the abstract syntax tree. This technique would make it possible to run the elaboration phrase after an incorrect parse, possibly detecting semantic errors in the correct parts of the syntax tree. (It would not help reduce spurious error messages in the parser itself, because the technique is useful only for checking different subtrees independently, and the input to a parser is a list.) To apply this idea, one would have to change the definition of the `abstract_syntax` type, replacing recursive occurrences of `abstract_syntax` with `abstract_syntax error`. One would also have to develop techniques to

[5] Phil Wadler, private communication, 25 August 1998.

figure out which subtree of the abstract-syntax tree to replace with `ERROR` in the event of a syntax error. The potential benefit seems marginal in today's computing environment. It is much easier to avoid spurious error messages by simply not running the elaboration phase in case of syntax error. Parsing is so fast, and experienced programmers make so few syntactic errors, that it is reasonable simply to run the compiler again.

5.4. Recovery from semantic errors

Koskimies [18] describes Lisa, a specification language based on attribute grammars, in which "check clauses" give conditions on attributes that must hold if the program is correct. If the attributes at a node violate a check clause, the synthesized attributes at that node are given a special `UNDEFINED` value, which corresponds to `ERROR` as used in this paper. This strategy does suppress spurious error messages, but it does not give the compiler writer sufficiently fine control over what information is marked erroneous. For example, because the symbol table is computed as an attribute, an incorrect declaration marks the entire symbol table as `UNDEFINED`, not just one entry. Lisa therefore also permits the compiler writer to designate certain (internal) types as *insecure*, which means that attributes of those types are never forced to `UNDEFINED`, even when an error occurs. When working with values of insecure types, the compiler writer is left to his own devices, and mistakes can lead to spurious error messages. Using the `error` type constructor gives the compiler writer superior control, since it can be applied selectively not only to types, but to components of structured types, or to different values of the same type.

Johnson and Runciman [4] do not use attribute grammars. Their York Ada Workbench compiler uses an internal representation based on trees, and when it detects a semantic error, it issues a message and replaces the offending tree node with a "plastic" node. Procedures that are passed plastic nodes issue no error messages, unless the non-plastic arguments by themselves indicate that an error is present. Moreover, procedures that receive plastic arguments must return plastic results. The York compiler is implemented in Ada, and since Ada does not support polymorphic type constructors like `error`, there must be different types of plastic nodes to be used in different data structures. Because there are no polymorphic, higher-order functions to hide plastic values, the internal procedures that process nodes must recognize plastic parameters or must call procedures to test for plasticity. Thus, using plastic nodes offers many of the same benefits as using our combinators, but it requires more implementation effort. In the York compiler, that extra effort has been taken further; each different type of plastic node carries extra information that supports an innovative error-diagnosis scheme. One could accommodate such extra information in an ML implementation by defining a new type constructor

```
datatype ('a, 'b) error' = ERROR of 'b
                         | OK of 'a
```

in place of the type constructor `error`. The type `'a error` would be isomorphic to `('a, unit) error'`. The combinators could easily be changed to work with this new type, but it's not clear how useful they would be. One might have to supply a different diagnostic function for each different type parameter `'b`, in which case the gain from using the combinators might be small.

6. APPLICATION AND EVALUATION

The combinators in this paper are most relevant to compiler phases that check static semantics or generate intermediate code. Such phases typically produce their results by a top-down, bottom-up pass over abstract-syntax trees. (Sometimes, as with parsers generated by `yacc`, the trees are implicit.) A compiler writer can apply the combinators selectively by identifying places in the tree walk where it would be profitable to attempt to recover from errors and to continue checking. Such places may be characterized as nodes of which two or more children could be checked independently.

Combinators can be added gradually to a new or existing compiler. A compiler writer can begin with a single exception handler, at top level, producing a compiler that halts after the first error message. Then, as the compiler writer identifies opportunities to detect multiple errors, he or she can use `catch` to inform the user about those errors and to turn them into `ERROR`. Instead of rewriting an existing function to handle the `ERROR` case, the compiler writer can apply `emap` to it. When it becomes useful to allow `ERROR` to appear in the symbol table, functions that expect the symbol table to contain only non-`ERROR` values can by altered by applying `errorEnv`. The combinators `liftList`, `combine`, and `catch` help handle most of the other cases that arise in improving a compiler's ability to detect errors.

Traditional, bottom-up type checking is a good target for aggressive error checking. Most expression nodes have children that are also expression nodes and that can be checked independently. Using the combinators, the checker can assign a value of type `ty error` to each sub-expression, and it can use `errorList` and `emap` to check only those expressions whose sub-expressions are error-free. If, for example, the C expression

```
digits(n)[i] = d > base ? d - base : d
```

appears in a context in which `digits(n)[i]` does not type check, the compiler can check the right-hand side even though the type of `digits(n)[i]` is `ERROR`. Moreover, because the type of the left-hand side is `ERROR`, the compiler will not try to check the assignment, avoiding what would be a spurious message if it guessed the wrong type for that left-hand side.

6.1. Case study

I have used the combinators in this paper to help build compilers for SLED, the Specification Language for Encoding and Decoding of machine instructions [19] and for λ-RTL, a language for describing instructions' semantics [20]. The λ-RTL compiler is still under development, but the SLED compiler is mature enough to use to evaluate the effectiveness of the error combinators.

To evaluate how this technique changes a compiler's source code, I examined the uses of the error combinators in different parts of the SLED compiler. This compiler reads a specification of the binary representations of instructions and emits code that encodes or decodes the instructions [19]. The compiler is implemented in about 16,000 lines of Standard ML, which may be divided into several parts.

- The encoding and decoding back ends, plus support for generating C code and ML code, are about 10,000 lines, and they are oblivious of errors.
- Lexing and parsing are about 1,000 lines. The parser generator uses its own scheme for error detection, and it raises its own exceptions to signal syntax errors. A single wrapper function maps these exceptions to the `Error` exception.
- The elaboration phases are about 3,000 lines. About 40% of this code is elaboration of abstract syntax, akin to `elaborate` above. This code is written in monadic style and makes heavy use of the error combinators, as described further below. The remaining 60% is semantic functions, which do not use the error *combinators*, but which do raise the error *exception* upon detecting an error. The exception is then caught in the elaboration modules. The ability to write the semantic functions in non-monadic style is the primary benefit of using exceptions in this way. The semantic functions need only detect errors, without worrying about what later results may be invalidated by them.
- The remaining 2,000 lines is miscellaneous support code, as well as a driver that links together the various phases of the compiler. About 60% of this code uses the error combinators, primarily for linking.

The error combinators are used most heavily in the elaboration phase of the compiler. In SLED, one defines assembly-language and binary representations for such elements as opcodes, addressing modes, and instructions. The elaboration phase turns the abstract syntax of these descriptions into a normal form designed to simplify the construction of encoders and decoders. The elaboration of expressions is written in monadic style, much as suggested in Section 3. Elaboration of declarations is similar, except declarations may have more elements that can be checked independently. For example, a "constructor specification," which declares an instruction, includes an opcode, operands, an optional type, and a binary representation. Constructors are added to the symbol table only if all four parts are free of errors. The compiler implements the check by elaborating all four parts, then using `liftQuadruple` (an analog of `liftPair`) on the result.

Overall, using the combinators has a modest impact on the compiler. About 15% of the code uses the combinators heavily, and another 15% is aware of the error type or exception but makes minimal or no use of the combinators. Error propagation is irrelevant to the remaining 70% of the compiler.

6.2. Compile-time cost

The preceding section presents the impact of error propagation on the compiler's source code, but it is also useful to know how much the error-propagation technique adds to compile time. An ideal measurement would show how a compiler with error propagation would compare with a similar compiler that halted after one error message, but such a direct comparison would require writing two versions of a compiler—an onerous chore. A somewhat different comparison can give a reasonable estimate of the answer, without much re-implementation.

To estimate the cost of error propagation, I modified the λ-RTL compiler to use the trivial monad instead of the error monad, and I measured the effect on compile time. As input data, I used λ-RTL descriptions of the SPARC and Pentium architectures. I used four versions of the SPARC description and two versions of the Pentium descriptions; different versions varied in the number of instructions described or in the level of detail in the descriptions, or both. Because the overhead of the error-propagation technique is so small, I measured the changes in the times required for the elaboration phase only. These times ranged from about 200–500 msec. For comparison, total compile times ranged from 2-10 sec, not counting the time required to write output. Elaboration times were computed by averaging 50 runs each on a 200MHz Pentium Pro with 128M RAM, running Linux 2.0.30. Experimental error in the measurements was 0.4–1.5%. The λ-RTL compiler was compiled with Standard ML of New Jersey, version 110.0.2.

The λ-RTL compiler was originally written to use the error monad directly. In particular, it had direct access to the representations of `ERROR` and `OK`. The first stage of modification was to place the error monad behind an abstraction boundary, and in particular, to remove all direct references to `ERROR`. This stage *increased* elaboration times by 0–6%, because the compiler had to manipulate values through abstract functions instead of manipulating the representation directly. The second stage of modification was to replace the error monad with the trivial monad, i.e.,

```
type 'a error = 'a
fun OK x = x
  ...
```

Pass	1	2	3	4	5	6	7
# of lines flagged with errors	50	26	37	26	32	32	0
# of fixes required to repair errors	33	3	33	15	32	32	0
fraction of total errors detected	.22	.02	.22	.10	.21	.21	.00
# of errors remaining	150	117	114	81	66	34	2
fraction of remaining errors detected	.22	.02	.28	.18	.48	.94	.00

TABLE 3. Detection of artificially seeded errors

In this modified version, almost all of the combinators are identity functions, `catch` does not actually catch exceptions, and the first time `Error` is raised, it halts the compiler. This modification *reduced* elaboration times by 1–10%.

If we believe the cost of calling the identity function in another module is negligible, we might conclude that error propagation adds 1–10% to elaboration time. A more conservative estimate is to assume that the cost of using the trivial monad is approximately the cost of adding an abstraction barrier to the error monad. In that case, we estimate the overhead of error propagation at 1–16%. In any case the fraction of total compile time is small. If no errors are detected, error propagation adds about 1% to total compile times. If errors are detected, error propagation may add as much as 5–10%, but the benefit is that multiple errors can be detected on a single run of the compiler, so error propagation pays for itself if among every 10 runs that detect an error, at least one detects more than one error.

6.3. Error seeding

To evaluate the effectiveness of error propagation in helping compilers detect multiple errors in a single run, I "seeded" a correct SLED specification with 150 errors. I began with a 260-line SLED specification of the SPARC instruction set, which I modified by changing the final identifier in every declaration to `xxx`. (SLED is a purely declarative language, so the description consists solely of declarations.) I then ran the faulty specification through multiple compile-diagnose-repair passes until the SLED compiler detected no more errors. Table 3 shows the results.

In the first four passes, more lines are flagged with errors than are required to be repaired. SLED typically defines many identifiers in a declaration, and two of the errors that were introduced caused many definitions to be missing. The first such error was caught on pass 1, and the second on pass 2. Repairing both of these errors made it possible to detect many new errors on passes 3 and 4. The errors reported in passes 1 through 5 all involve at least one definition. The errors reported in pass 6 involve only uses, and when those uses are repaired, the compiler detects no more errors. Of the two "errors" that remain, one is in fact incorrect, but it appears in a directive that is ignored by the version of the compiler under study. The other does not make the specification incorrect; it simply results in a specification that, although correct, is different from the original specification.

While no general conclusions can be drawn from a study with a sample size of one, Table 3 does show that multiple errors can be detected effectively on a single pass. Even the least effective pass, pass 2, results in the ability to repair three errors. A single fault in the source can cause the compiler to issue multiple error messages; the most probable cause is a missing definition, which causes multiple uses to be flagged as unknown. None of these messages is spurious, however, and the ratio of 203 total messages to 150 faults approaches the ideal of one message per fault.

7. DISCUSSION

The combinators presented in this paper solve only part of the problem of issuing good error messages—when an error occurs, they limit the damage to the compiler's internal invariants, ensuring that later error messages are issued only if they relate to parts of the program that don't depend on earlier errors. Some readers may think that no compiler can always give a correct second error message, because the compiler cannot know what was intended in place of the first error. This claim is true in principle, because the correctness of the whole program could depend on the erroneous part. In such case, the combinators would propagate `ERROR` throughout the compiler's data structures, and the compiler would issue one error message, then halt. In many practical cases, however, a compiler can continue looking for errors, using the combinators to ignore anything that is "tainted" by earlier errors.

Even when using the combinators, a compiler may issue "too many" error messages when many uses are inconsistent with a declaration. (I write "too many" in quotation marks, because although such error messages are annoying, they are not spurious—they make sense in terms of the source program.) For example, a variable x may be declared to be an integer and may be used in n places. Suppose that k of the uses are in locations where an integer is incorrect. If $k = 0$, the program is correct, at least with respect to x. If k is small, the compiler should issue k error messages, one for each incorrect use. If $k = n$, it seems likely that the declaration is wrong, and the best error-reporting strategy might be to issue a single error message at the declaration. Deciding which error-reporting strategy to use requires global knowledge, and the combinators described in this paper won't help, because they are tuned to a local model of error detection and reporting. Some real compilers, like `gcc`, limit the number of error messages by marking x's symbol-table entry as `ERROR` as soon as $k > 0$. The combinators could help support this reporting strategy.

The combinators do not necessarily make error messages less confusing. For example, errors in type inference are notoriously confusing, perhaps because of the global nature of type inference. The reported location of the error may be far away from the code that causes the error, and the error message itself may confuse those who don't understand the underlying unification algorithm. Elaborate methods have been proposed to help users understand errors in type inference [21, 22, 23].

This particular collection of combinators arose as part of an effort to improve the error-reporting ability of a compiler designed to halt after a single error. The `ERROR` and `OK` combinators provided the basic ability of adjoining an error value to any type. The next additions were `emap`, which enables the use of functions that are oblivious of errors, and `catch`, which gives control over the detection and reporting of errors. The least obvious combinator, and the only one that makes no sense in terms of general monad operations, is `errorEnv`, which was introduced as a general way of handling "`ERROR` in the symbol table." Using `errorEnv` makes it possible to use, without modification, code that expects only valid entries in the symbol table. The last main combinator to be added was `sequence`; it is the analog of the monad operation *bind*, which would otherwise have been missing. Replacing `combine o emap f` with `sequence f` made several parts of the λ-RTL compiler easier to read and understand.

The `liftList` combinator filled a clear need in elaborating function applications, but it was some time before it became clear that `liftList` was an instance of a more general idea. There is no end to the number of lifting combinators possible, since for any polymorphic type constructor $\alpha\ T$ one can wish for a lifting combinator of type α `error` T `->` $\alpha\ T$ `error`. (Here, α is a "type variable" that can stand for any type.) Still, the lifting combinators presented in this paper, together with `liftTriple` and `liftQuad`, suffice in most practical situations.

Since Spivey's combinators are used explicitly to replace exceptions, the curious reader might wonder if exceptions are really necessary to the error-propagation technique presented here. They are, in fact, not strictly necessary, since their use can be replaced by Spivey's combinators, which would be completely appropriate were one to use this error-propagation technique in a pure, lazy language like Haskell or Miranda. When using impure languages, exceptions provide two significant advantages, both grounded in separation of concerns:

- The entire compiler need not be written in monadic style. In particular, "semantic" functions that detect but do not handle errors can be written to raise exceptions instead of using monadic style. Such functions comprise about 40% of the error-aware code in the SLED compiler, so in one case at least the freedom to avoid monadic style simplified the adaptation of an existing compiler.
- Through suitable use of `catch`, a compiler writer can incrementally improve the ability of a compiler to detect and report errors.

The combinators in this paper lead naturally to compilers that avoid spurious error messages. Such compilers can safely detect multiple errors by checking only intermediate results that are independent of erroneous values. For example, a compiler elaborating a bad expression can check all subexpressions that are not bad. Once a subexpression is found to be bad, the compiler can still check independent subexpressions, but by propagating `ERROR`, it automatically skips over subexpressions that depend on the bad one.

Code that discovers new errors raises `Error`, and compilers can use `catch` or `catch'` to print messages about those errors and to convert them to `ERROR` values. No other code prints error messages, so spurious messages are avoided automatically. Placement of `catch` determines the granularity with which the compiler can detect different errors in a single run.

In exchange for the easy handling of error messages, a compiler writer must adapt his compiler so it can proceed even when an internal value is `ERROR` instead of what was originally expected. The type `'a error` represents such values, and the placement of the `error` constructor determines how much information is kept about places where errors occurred. For example, `name error * code error` can be more informative than `(name * code) error`, because it tells which went wrong, the `name` or the `code`. The combinators summarized in Table 1 make it easy to adapt compilers. One can keep most functions simple by writing them without regard for the `error` type constructor, then use the combinators to reconcile different assumptions about `error` in different parts of the compiler. One of the best things about this style is that ML's type inference finds mistakes—placing the combinators amounts to programming with type constructors at compile time.

Although this paper uses ML to present the combinators, not all of ML's features are needed to implement the underlying discipline of error propagation.

- Almost any language can define a type that adjoins `ERROR` to other values. In languages without polymorphism, using such a type may entail a loss of static type checking. For example, in C, it would be most convenient to define a type denoting "error or pointer."
- Exceptions make it possible to use `catch` to check for errors exactly where desired, instead of having

to check the return value of every function or to write the compiler in monadic style. Most modern imperative languages, including C++, Ada, Java, and Modula-3, support a sufficiently expressive form of exceptions.

- ML's polymorphic functions and types make it possible to use a single `'a error` type constructor, instead of requiring each type to provide a different value to indicate `ERROR`, as in the York Ada Workbench compiler [4]. Generic Java [6] and C++ templates are sufficiently expressive to define and use error combinators just as described here. Ada and Modula-3 provide a strictly weaker form of polymorphism; their "generics" make it possible to specialize the error combinators to many different types, but each specialization must have a unique name. As an alternative to the awkward namespace management required by "generic polymorphism," programmers using these languages can instead use "subtype polymorphism," i.e., they can define a single type meaning "error or object," then use dynamic type checking to make sure the "object" expected is of the right type.
- The benefits of static type checking are well known. In its application to error propagation, static type checking helps document exactly where `ERROR` may (or must not) occur, and this "documentation" is checked by the compiler, which can guarantee that `ERROR` can never occur unexpectedly. As noted above, in some languages it may be preferable, or even necessary, to trade static type checking for easier use of the combinators.
- ML's first-class functions and currying make it possible to propagate `ERROR` almost exclusively by means of the combinators—it is very seldom useful to check for `ERROR` explicitly. In more mainstream languages, top-level functions are typically first class, but nested functions are either not provided or are not first class. In such cases, it is sometimes necessary to simulate nested or curried functions, e.g., by using objects. It is not clear how much such simulation might be needed.

The idea of using special values to mark bad intermediate results has been used in many compilers. Writing a compiler in ML makes it possible to use the ML implementation to check that these special values are used correctly.

ACKNOWLEDGEMENTS

Lal George and Dave MacQueen invited me to spend a summer at Bell Labs, where I developed these techniques. John Reppy and Greg Morrisett taught me the trick of using special abstract-syntax tree nodes to mark positions. The anonymous referees asked many pointed questions, directed me to Spivey [1], and made helpful suggestions about how to evaluate my techniques. I also thank Jerry Leichter and other readers of `comp.compilers` for stimulating discussions about error messages, Mary Fernández for her encouragement, and Nevin Heintze for his skepticism.

This work was supported in part by NSF grants ASC-9612756 and CCR-9733974 and by the US Department of Defense under contract MDA904-97-C-0247 and DARPA order number E381. The views expressed are my own and should not be taken to be those of the DoD.

Appendix

This paper was prepared with the `noweb` tool for literate programming [24]. The examples, together with supporting code, have been extracted from the paper and compiled with Standard ML of New Jersey, version 110.8. This exercise ensures that the code in the examples type-checks, and that the types match those given in declarations and in Table 1. The resulting code is available as an electronic appendix, which can be found at `http://www.cs.virginia.edu/~nr/pubs/error.sml`.

REFERENCES

[1] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, June 1990.

[2] Philip Wadler. The essence of functional programming (invited talk). In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, New York, NY, USA, January 1992.

[3] James J. Horning. What the compiler should tell the user. In F. L. Bauer and J. Eickel, editors, *Compiler Construction: An Advanced Course*, chapter 5.D, pages 525–548. Springer-Verlag, New York, 1976. Originally published as LNCS Vol. 21.

[4] C. W. Johnson and C. Runciman. Semantic errors – diagnosis and repair. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 88–97. ACM, ACM, 1982.

[5] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.

[6] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA 98*, Vancouver, October 1998.

[7] Jeffrey D. Ullman. *Elements of ML Programming, ML97 Edition.* Prentice-Hall, Englewood Cliffs, NJ, 1997.

[8] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[9] Guy L. Steele Jr. Building interpreters by composing monads. In ACM, editor, *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 472–492, New York, NY, USA, 1994. ACM Press.

[10] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin

Runciman, and Philip Wadler. Haskell 98: A non-strict, purely functional language. Available from `www.haskell.org`, February 1999.

[11] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[12] Michael G. Burke and Gerald A. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems*, 9(2):164–167, April 1987.

[13] Bruce J. McKenzie, Corey Yeatman, and Lorraine De Vere. Error repair in shift-reduce parsers. *ACM Transactions on Programming Languages and Systems*, 17(4):672–689, July 1995.

[14] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, CA, 1988.

[15] David Gries. Error recovery and correction — an introduction to the literature. In F. L. Bauer and J. Eickel, editors, *Compiler Construction: An Advanced Course*, chapter 6.C, pages 627–638. Springer-Verlag, New York, 1976. Originally published as LNCS Vol. 21.

[16] A. Boccalatte, M. Di Manzo, and D. Sciarra. Error recovery with attribute grammars. *The Computer Journal*, 25(3):331–337, August 1982.

[17] G. Adorni, A. Boccalatte, and M. Di Manzo. Top-down semantic analysis. *The Computer Journal*, 27(3):233–237, August 1984.

[18] Kai Koskimies. A specification language for one-pass semantic analysis. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19(6) of *ACM SIGPLAN Notices*, pages 179–189, June 1984.

[19] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[20] Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474 of *LNCS*, pages 172–188. Springer Verlag, June 1998.

[21] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1–4):17–30, March–December 1993.

[22] Karen L. Bernstein and Eugene W. Stark. Debugging type errors. Technical report, State University of New York at Stony Brook, Computer Science Department, October 1995.

[23] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 7(1), June 1996.

[24] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.