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

(Short Title: Eliminating Spurious Messages)

Norman Ramsey
Dept of Computer Science, University of Virginia
Charlottesville, Virginia 22903 USA
Phone: +1 804 982 2227 Fax: +1 804 982 2214
Email: nr@cs.virginia.edu

April 3, 1997

Abstract

Many language processors make assumptions after detecting an error. If the assumptions are invalid, processors 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 itself is unsurprising, but it is both formalized and implemented in terms of a type constructor and combinators expressed in Standard ML, and the ML type rules enforce the discipline. 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. ML's type system guides the application of the combinators; if the compiler writer does not account for a potentially unavailable value, the source code of the compiler does not type-check. The techniques presented exploit several features of Standard ML, including exceptions, higher-order functions, polymorphism, and static type checking. Using these features enables the ML type system to ensure that the error-tracking discipline is applied consistently, relieving the programmer of that burden.

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 the first Pascal compiler I ever used, are so bad that users learn to disregard all but the first error message. Even modern compilers written by smart people are not immune. My favorite 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. The technique is not needed for parsing; it helps in later phases, like static semantic analysis and generation of intermediate code. We don't need to worry about detecting errors in 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 [1] 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 [2] 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." The contribution of this paper is to show how to exploit features of a modern functional programming language to enforce proper use of this discipline. The features used are exceptions, higher-order functions, polymorphism, and static type checking, and the functional language is Standard ML [3]. By exploiting this combination, we not only reduce the effort required to implement the discipline, but we get the ML compiler, not the programmer, to check that the discipline is applied correctly. Ullman [4] presents ML at a level suitable for understanding this paper.

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.

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

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.

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`.

3 Propagating error information with combinators

If every result or argument in the compiler is potentially `OK` or `ERROR`, we have 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 enable us to 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, we 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 combinators are easy to use because the type rules of ML show us where to apply them.

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
```

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 AST elaborates correctly, or raise `Error` (B) if the AST breaks a static-semantic rule. On the other hand, 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
```

`apply` does not return `exp error`—if its 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) (* bogus *)
```

because `map elaborate operands` returns type `exp error list` and `apply operator` expects type `exp list`.

Two combinators come to the rescue. The first combinator, `listError`, 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`.

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

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)
```

`emap` takes a function `f` and a value. If the value is `OK x`, it unwraps it, 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.

To return to our example, `apply operator` has type `exp list -> exp`, and so `emap (apply operator)` has type `exp list error -> exp error`. The function `listError 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) (listError (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"
```

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 we apply `emap` 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. We could apply `emap` to `elaborate`, but `emap elaborate` has type `abstract_syntax error -> exp error error`, which is inconvenient. The `strip` combinator removes the extra `error`. It is written

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

The function `strip o emap elaborate` has the type we want:

```
abstract_syntax error -> exp error.
```

We have given combinators to 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 `emapFun` 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 `emapFun` combinator, we can use `lookup`; the function `emapFun codegen lookup` has type `exp -> code error`.

`emapFun` 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`. `rho'` applies

rho, then strips off OK. If rho ever returns ERROR, rho' raises the local exception RhoFailed, and emapFun aborts execution of f, returning ERROR.

```
val emapFun : (('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 emapFun, 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 emapFun, OK is applied to f rho' x to make sure it, too has type 'd error. If I want to use emapFun in a context where 'd is already an error type, I apply strip to the result.

Returning to the example, lookup has type name -> denotable error, emapFun codegen lookup has type exp -> code error, and applying strip and emap gets to type exp error -> code error. By using all the combinators, we can easily propagate the error information through stages of parsing, elaboration, and code generation:[1]

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

Propagating the error information becomes a matter of applying the correct combinators. We don't need to worry that we might get them wrong, because the type checker will prevent us from using the combinators incorrectly. The summary in Table 1 can help a programmer figure out which combinators to apply where in order to get the code to type-check.

4 Converting new errors into ERROR

By propagating ERROR, and by avoiding applying functions to ERROR, we avoid executing code that might issue spurious error messages. At some point, however, we have to issue real error messages. We do this 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

[1] 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 strip     : 'a error error -> 'a error
val listError : 'a error list -> 'a list error
val emap      : ('a -> 'b) -> 'a error -> 'b error
val emapFun   : (('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
```

Table 1: Table of combinators

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.

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
```

takes an error-printing function and applies another function within a handler. To show its use, I return to the expression-elaboration example. In my latest compiler, I use a special AST node to mark the source region from which the AST was derived.[2] `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`.

`ERROR` is not always the right thing to return when a computation is aborted because of a new error. For example, when processing a declaration, my compiler returns a new symbol table containing a binding for the identifier declared. 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 this way:[3]

```
fun catch' default printMsg f x =
  f x handle Error msg => (printMsg msg; default)
val catch = fn p => catch' ERROR p
```

[2] I thank John Reppy and Greg Morrisett for teaching me this trick.

[3] The definition of `catch` is η-expanded to avoid the "value restriction" on polymorphism. This restriction was added to Standard ML in 1996.

I use `catch'` to update a symbol table. I assume a polymorphic, functional table, indexed by strings, and offering these functions for update and search:[4]

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

In my compiler, I want not to redefine identifiers, and I want identifiers I look up to be defined. `bindUnbound` and `get` do these jobs; both can raise `Error`.

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

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

Now I can process a declaration to add a binding to the symbol table, where I instantiate `'a` with `denotable error`.

```
fun processDeclaration table (INTDECL(name, ast)) =
  let val rhs = catch print (elaborateFrom (get table)) ast : exp error
      val denoted = 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. 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 Application and limitations

The combinators in this paper should work best in 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. I characterize such places 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

[4] `'a option` is a predefined type defined by

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

It is conventionally used for optional values; in this case, `lookup` should return `NONE` when there is no entry in the table, and `SOME x` otherwise.

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 `emapFun`. The combinators `listError`, `strip`, 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.

I have used the combinators in this paper to help build a new compiler for the SLED specification language for encoding and decoding of machine instructions [5]. In SLED, one defines assembly-language and binary representations for such elements as opcodes, addressing modes, and instructions. SLED therefore has declarations and expressions, but no statements. I use the error combinators mostly in the symbol table, in the elaboration of expressions, and in the elaboration of higher-level specifications. SLED expressions can be checked bottom-up, so the compiler uses `catch` at each expression node, which makes it possible to check sub-expressions independently.

Analysis of SLED declarations gives another example of fine-grain error detection. Such declarations may contain several different kinds of elements, each of which must be correct for the declaration to be meaningful. For example, a "constructor specification" declares a group of instructions. The elements of a constructor specification are "branches," which give binary representations, opcodes, operands, and an optional type. Opcodes, operands, and type are checked for errors independently. Branches are checked only if opcodes and operands are free of errors. Finally, the constructors are added to the symbol table only if all four parts are free of errors. Because the four parts have different semantics, and therefore are represented internally using different types, I cannot simply use `errorList` and `emap`. Instead, I check for `ERROR` using a simple ML pattern match; the new symbol table, as changed by the elaboration

of the constructor specification, is

```
case (opcodes, operands, constype, branches)
  of (OK interps, OK operands, OK ty, OK branches) =>
          ⟨symbol table with new constructors added⟩
   | _ => ⟨symbol table with names of new constructors bound to ERROR⟩
```

The combinators support a model of translation in which information about the program is created at "sources" and flows to "sinks." For example, a declaration of an identifier is a source, and information about that identifier flows through a symbol table to multiple sinks: one sink for each use of the identifier. If the information from the source is valid, errors can be detected independently at independent sinks. Thus, for example, statements in a sequence can be checked independently, because no static-semantic information flows between statements. When an error prevents a source from creating valid information, the compiler uses `ERROR` as a place-holder, and it ignores everything reachable from that source. For example, an error in a declaration prevents full checking of sub-expressions and statements that depend on that declaration.

This source-to-sink model fits many languages, but it does not work well when static-semantic analysis requires global information. This paper is about using ML to build compilers, not about compiling ML itself, but it is nevertheless ironic that ML type inference is such an analysis. ML type inference requires global knowledge about all uses of identifiers anywhere in a nest of mutually recursive functions. ML compilers avoid spurious error messages in type inference by aborting type inference as soon as they detect an error. Some halt completely; some skip to the next declaration and resume type inference.

Compiling ML would provide other opportunities to exploit the combinators. For example, if a declaration of a `let`-bound variable x (by `val` or `fun`) contains a type error, the writer of an ML compiler might wish to enter x into the symbol table not with `ERROR` but with the most general type, $\forall\alpha.\alpha$. Any use of the variable that would be correct under the "real" type will also be correct under $x : \forall\alpha.\alpha$, so it will be possible to check expressions that depend on x instead of ignoring them. This trick can be implemented by applying "`catch'` $\forall\alpha.\alpha$ `print`" to the function that computes the types of `let`-bound variables.

These combinators 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 the degenerate case in which the correctness of the whole program depends on the erroneous part. In this case, the combinators will propagate `ERROR` throughout the compiler's data structures, and the compiler will wind up issuing the single error message, then halting. For many practical cases, however, a compiler can continue looking for errors, using the combinators to ignore anything that is "tainted" by earlier errors.

The combinators do not necessarily make error messages less confusing. For example, errors in ML 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 [6, 7, 8].

Even when semantic analyses follow the source-sink model, a compiler may issue "too many" error messages when many sinks are inconsistent with their source. (I write "too many" in quotation marks, because although such error messages are annoying, they are correct in terms of the source program.) For example, a variable x may be declared to be an integer (the source) and may be used in n places (the sinks). 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, we would like to 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 the source-sink 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, but the mutation of the symbol table would be awkward, since it is not in the style of ML programming to mutate the symbol table after building it.

6 Related work

There are many published papers about recovery from syntactic errors, so that parsing can continue [9, 10]. Chapter 17 of Fischer and LeBlanc [11] surveys the field. By contrast, semantic errors and their processing are rarely discussed—perhaps, as David Gries suggested twenty years ago, because they haven't been formalized [12]. A handful of researchers have used attribute grammars to formalize semantic errors.

Adorni, Boccalatte, and Di Manzo [13, 14] 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. Moreover, their algorithm can 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.

Koskimies [15] 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 [2] 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 the combinators themselves might be small.

7 Conclusion

Avoidance of spurious error messages comes naturally from using the combinators in this paper. Compilers written using these combinators can easily check

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.

This discipline for propagating error information helps compiler writers produce trustworthy error messages. Many compiler writers have implemented similar disciplines in other languages, including languages like C and Ada, but using ML has several advantages:

- Exceptions mean I can use `catch` to check for errors exactly where I want to, instead of having to check the return value of every function.
- Polymorphism means there is a single `'a error` type constructor, instead of requiring each type to carry a value indicating `ERROR`.
- Static type inference means I can use types to document exactly where `ERROR` may (or must not) occur, and this "documentation" is checked by the compiler, which guarantees that `ERROR` can never occur unexpectedly.
- Higher-order functions and currying mean I can define combinators to propagate `ERROR` as needed, and by using these combinators, I almost never have to check for `ERROR` explicitly.

The idea of using special values to mark bad intermediate results has been used in many compilers. By writing a compiler in ML, I can get the ML implementation to check that these special values are used correctly.

Acknowledgements

I developed these techniques during a summer visit to Bell Labs. I 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 grant ASC-9612756 and by the US Department of Defense under contract MDA904-97-C-0247 and DARPA order number E381. The views expressed are those of the author and should not be taken to be those of the DoD.

Appendix

This paper was prepared with the `noweb` tool for literate programming [16]. The examples, together with supporting code, have been extracted from the paper and compiled with Standard ML of New Jersey, version 109.17. 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] 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.

[2] 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.

[3] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[4] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[5] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 1997. To appear.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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.

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