Lessons from the design of a Standard ML library

DAVE BERRY

Laboratory for Foundations of Computer Science, University of Edinburgh, UK†

Abstract

We describe the Edinburgh SML Library and draw lessons from its development. These lessons are of two kinds. The first concerns how to use SML to write a library, and shows some of the design choices that have to be considered. Most of the paper concerns this topic. The second kind of lesson concerns ways in which the language hinders the construction of a library. We suggest some changes to the language as a result of this experience, the most important being the addition of higher-order functors. We also suggest that language designers should consider the needs of libraries early on in the design of a language.

Capsule Review

Berry addresses language design issues of ML pertaining to its use in the construction of libraries. This is a crucial concern for a language to be successful in the development of 'real' software. The author's remarks derive from his work on a general purpose ML library of significant size. He comments on some of the design choices necessary when building a library, and on some of the language shortcomings that inhibit the library construction. He discusses the use of structures vs. functors, universal polymorphism vs. existential polymorphism, treatment of failure, higher order functors, and other issues.

This paper provides pertinent information for the debate on how to evolve ML-like languages into truly general-purpose languages.

1 Introduction

We had two aims when designing the Edinburgh SML Library. We wanted to provide a range of reusable software components, so that people wouldn't have to keep re-inventing the lever. We also wanted to define a common interface to some operations that were not included in the formal definition of the language but that were provided by the compilers we were using.

Both these goals arose from the paucity of primitive operations included in The Definition of Standard ML (Milner *et al.*, 1990). That document concentrates on the semantics of the language itself, and quite intentionally defines only a few basic operations. These operations, called the *initial basis*, are not really adequate for real programs, so implementers of SML added their own features. As a result, the implementations became incompatible, and we were reduced to having to port

† This work was supported by a grant from the SERC.

code between different implementations. This undermined the work that went into producing a formal definition of the language.

In addition, there was little reuse of SML software. If you wanted a hash table, you had to write your own, unless you knew somebody who had written one already. The modules of SML were being used to structure individual applications, but not to define reusable software components. The portability problems were a hindrance here as well.

Furthermore, the implementers were extending the initial basis in a piecemeal fashion. As they saw a need for a new operator, they would add it. There was no attempt to provide a consistent naming scheme and a consistent use of types, which would have provided a coherent interface to the user. And there was even less consistency between implementations. The implementers cannot be blamed for this, but the result was unsatisfactory.

It is impossible to predict all the operations that people will require, so we decided to produce a well-designed framework that other people could extend as necessary. We began by asking several people from different organisations to provide utility functions that they had developed for their own use. We started with the most commonly used functions from those that we were given, and formed several basic library entries. Then we extended these entries to provide a consistent user interface.

This paper is structured as follows. Section 2 gives a brief overview of the Edinburgh library. Sections 3 to 9 discuss some of the choices that are faced when writing a library in SML. Some of these choices face writers of applications as well. Sections 10 to 13 discuss aspects of the design of SML that hinder the construction of libraries.

There are many choices that must be considered when writing a library in SML. In this paper we discuss the following:

- When to use structures and when to use functors to implement library entries (section 3).
- How to record dependencies between entries (section 4).
- When to use type variables for polymorphism and when to use functors instead (section 5).
- How generic signatures may be used to structure the library (section 6).
- What conventions to use for types and identifiers (section 7).
- How to treat cases in which functions may fail (section 8).
- How to use polymorphism when passing functions as arguments (section 9).

The main change to the language that we recommend is the addition of higher-order functors. Other language features discussed include constant references, the relation between infix identifiers and modules, polymorphic equality, arithmetic exceptions, and the standard I/O primitives. At a general level, we suggest that language designers should consider the needs of libraries as well as those of applications.

The discussion in this paper is necessarily informal. Although functional languages in general, and SML in particular, are based on a strong theoretical understanding, there is also a need to discuss how our languages behave in practice. This involves

matters of judgement as well as logic. The paper assumes that the reader is familiar with Standard ML.

2 An overview of the Edinburgh library

At present the Edinburgh library contains 59 entries, and consists of 15,000 lines of code. It provides implementation-specific versions for Poly/ML, Poplog ML, Harlequin MLWorks, and Standard ML of New Jersey. The ML-Yacc parser generator (Tarditi & Appel, 1991) and ML-Lex lexical analyser generator (Appel *et al.*, 1989) have been ported to it. It is popular both at Edinburgh and elsewhere, and is freely distributed by FTP. It is documented by a technical report (Berry, 1991).

Most entries declare a type and basic operations on that type. Some entries declare a group of functions that either extend the operations on a certain type or implement a given task such as memoisation. One entry provides a make system that can be used to load just those entries in the library that are needed, instead of loading them all at once, or for users to specify the dependencies between their own modules.

Most entries are defined by a signature and a structure. The signature gives the type information of each entity in the entry, and also describes each entity with a short comment. It also includes a general description of the whole entry. Thus the signature can be read as an on-line manual page for the entry. The technical report that describes the library includes all the signatures defined at the time of its publication.

The structure defines the behaviour of the entry by giving an implementation in terms of the standard initial basis. An entry may include alternative implementations, for example a readable one to define the behaviour and a more efficient one for actual use.

Some of the entries in the library are functors. The signatures used in the parameters and result of each functor are presented in the same way as the signatures for structure entries. The technical report includes these signatures, and also includes the header of each functor, to show what arguments it expects and what result it produces. These headers use the syntax of the 'functor specifications', mentioned in The Definition Of Standard ML as being a possible future extension.

Most of the signatures used in arguments to functors are *generic* signatures that may be matched by several entries. For example, the `MonoSet` functor uses the `EQUALITY` signature in its argument, which means that it can be applied to any structure that defines a type and an equality function on that type.

The appendix describes the current contents of the library in more detail. The make system will be described in section 4. Some other specific entries are mentioned where appropriate.

The following sections discuss some of the choices that are faced when writing a library in SML. Some of these involve the use of the modules system to structure the library. Others concern smaller scale but widespread decisions such as how functions

should deal with failure. The first discusses the best way to use the modules system to implement simple entries.

3 Use of the modules system

The SML modules system is powerful, and offers a range of facilities. Some of these directly support the task of specifying and implementing a library entry. For example, we use signatures to specify an entry, and leave the details of its implementation to a structure or functor. Other choices are more difficult. For example, we have found that the widely-recommended approach of building SML software entirely from functors is not appropriate to a library. This section discusses these issues.

One straightforward decision is to describe the type information of each entry with a signature. This signature can then be used to constrain the implementation as well as act as documentation. Since signatures, structures and functors are separate entities (unlike module interfaces and bodies in many other languages), the library may provide alternative implementations, each emphasising a different aspect such as readability or efficiency.

The behaviour of the entries must also be specified. Ideally, this would be done using a formal specification language such as Extended ML (Sannella, 1991). However, this seems to be beyond the current state of the art, of users and contributors as much as of programming environments and specification languages themselves. A less ambitious solution is to use a combination of natural language and a reference implementation.

The remaining question is whether to use structures or functors to implement the entries. Several authors recommend that application programs should be written entirely in terms of functors and signatures (Harper, 1986; Tofte, 1989; Paulson, 1991). However, this approach doesn't work so well for libraries. The rest of this section explains why.

When programs are built with functors and signatures, structures are only used in the final build sequence, in which the application is constructed by applying the functors to their arguments. The advocates of this approach claim that it allows modules to be reused and makes it easier to compile modules separately. It also ensures that all dependencies between modules are explicit.

For example, the following code builds a Vector module from functors:

```
functor MkList(): LIST = ...
```

The MkList functor doesn't depend on other modules, and produces a structure that matches the LIST signature.

```
functor MkVector (structure L: LIST ...) = ...
```

The MkVector module depends on the specification of the List entry, but not on the implementation.

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

```
local
  structure List = MkList();
  structure ...
in
  structure Vector = MkVector (structure L = List; ...)
end
```

The final structure is built from functor applications. The dependency between the Vector and List structures is explicit in the arguments and application of the MkVector functor. The List structure is hidden from the user.

If this approach is not followed, then someone developing a new module could accidentally refer to an already loaded structure without documenting this dependency. For example, if the `List` and `Vector` structures were declared directly, the programmer might fail to document the dependency between them. Later attempts to build the system could then fail because the modules weren't loaded in the same order. If the writer uses only functors, there will be no existing structures to refer to in this way, and all links to other modules must be made using the parameters of the functor. This produces more reliable software.

Unfortunately, this approach is not suitable for building a library. An application program typically presents the user with a single structure, with all the other components of the system hidden, as in the above example. This means that the dependencies between the other components are also hidden. By contrast, a library consists of many entries, each of which may depend on several others. All the entries are visible, even though the dependencies between them should still be hidden. If the above example were part of a library instead of an application program, then the `List` structure would probably be made visible to the user. But a user's program should not need to know whether the Vector entry is implemented in terms of lists or of arrays, especially as the implementation may change in a later version of the entry.

This implies that even if entries are implemented using functors in preference to structures, the functors should be applied to their arguments before users see them. In other words, users should see each entry as a structure. Otherwise the arguments to the functor will make the dependencies visible. (Of course, if a functor has some arguments that the user should see, this reasoning does not hold. An example of such a case is a functor used to implement a generic type.)

The Edinburgh Library implements entries directly as the user will see them, which means that most entries are structures. Two alternative approaches have been suggested. The first is to implement all entries as functors, and to put an application of each dependent functor inside the body of the parent functor. In our example, the `Vector` functor would contain an application of the `List` functor:

```
functor MkVector () =
struct
  structure L = MkList ();
  ... L.apply ...
end;
```

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

This hides the dependency between the two modules from users, since the `List` entry is not mentioned in the arguments of the `Vector` entry. But it has the disadvantage that it creates a new copy of the `List` module in every entry that uses it. With current implementations of SML, this uses too much space to be practical.

The second alternative is to implement every entry as a functor, and to use the make system to apply the functor to generate the required structure when the user loads the entry. This would allow separate compilation under the same conditions as an application written entirely in functors. It could also partially protect writers from introducing accidental dependencies on other modules. However, the possibility is still there. This is especially true when writing an entry that the user sees as a functor.

In this case references to other entries that are only used to implement the functor cannot appear as parameters to the functor, because we don't want the user to see them. For example, if the `MonoVector` functor is parameterised on a structure that matches the `EQ_PRINT` signature, and is implemented in terms of lists, we can't add the `List` entry to the arguments without making this dependency visible to the user. So in this case the `List` entry must be loaded when developing the `MonoVector` functor, setting up the possibility of a later definition referring to the `List` entry by mistake.

It could be argued that careful use of a make system would prevent such mistakes. But this argument also applies to the approach of implementing entries directly as structures.

In the interests of uniformity and simplicity, the current implementation of the Edinburgh library defines all entries directly as the user sees them. This has the added advantage that some compilers compile structures into smaller, more efficient code than they do functors.

4 Loading dependent entries

The discussion on the previous section shows that most dependencies between library entries should be hidden from users. Therefore when a user loads an entry, the library must automatically load all entries that the requested entry itself requires. Also, entries should not be re-loaded if they have been loaded earlier. Lastly, since the SML modules system allows alternative implementations of the same entry, we would like the dependency system to support this.

Some SML compilers provide their own dependency management tools. But we wanted a portable system, which ruled out the use of features specific to a particular compiler. We were happy to settle for a simple system that did the basic task, because we couldn't afford the time to implement a more sophisticated tool. Even so, there were design choices to be made.

The system is based on information about which files depend on which others. Two common approaches are to put this information in a separate file, as in the UNIX make system (Feldman, 1979), or to include it in each file.

An advantage of the UNIX approach is that all the dependency information is in

one place, where it can be analysed. However, it should be possible to generate this information from the second system when it is required.

Our make system puts all dependency information in the appropriate files. It actually allows more than one compilation unit per file, but the library itself follows the convention of one compilation unit per file. This means that if the ability to check the modification time of each file is added to SML in the future, the library can easily make use of it.

Dependencies between files are specified in special comments, called *tag declarations*, at the start of the relevant piece of code. A tag declaration begins with the string (*$. Since comments begin with the string (*, the compiler itself ignores tag declarations.

The body of the tag declaration consists of an alphanumeric tag, usually followed by a colon and a list of other tags. The first tag names the code that follows the tag declaration. The optional list specifies the code that this piece of code depends on. The make system scans the tag declarations in the files specified by the user (or in our case, the library build file), and builds a dependency graph. When a piece of code is requested, all the necessary code is extracted and compiled. If a piece of code has been loaded earlier, it is not loaded again.

Another possible approach to the question of dependency information is for the system to discover the information by analysing the code of each file before loading it. This approach has the advantage that there is no separate dependency information that could get out of step with the code. It is more complicated to write than the other systems, and it is possible that some features of SML (such as the ability to open structures) will increase this complexity.

Any of these dependency systems are good enough for the basic task.

It should be possible to adapt existing more sophisticated systems to SML as well. The important thing is that a dependency system is essential to support an SML Library. The system is also useful to users, and so increases the usefulness of the library.

5 Polymorphism

The ML family of languages have always supported polymorphic datatypes and functions using type variables. This is called universal polymorphism. With the addition of the modules system, Standard ML also supports polymorphism using functors. This is called existential polymorphism, and is similar to the generic packages of ADA (Ada, 1989) or the generic classes of Eiffel (Meyer, 1991). This raises the question of which approach to use when. This depends on the needs of the programmer, so the Edinburgh library supports both approaches.

A related question concerns the use of equality types, another new feature of Standard ML. Equality types can make some code simpler, but are not truly general, and their use is not encouraged in the Edinburgh library.

A final question concerns the possibility of using existential polymorphism to provide versions of polymorphic types that are optimised for particular instances. The Edinburgh library provides a framework for such entries.

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

An example of universal polymorphism is the `Vector` type. This can be defined in terms of lists. (In most compilers it is implemented directly, for greater efficiency, but its behaviour can still be defined in terms of lists.) Here is one possible definition of the type, with a `map` function for this type:

```
datatype 'a Vector = V of 'a list

fun map f (V l) = V (List.map f l)
```

For simple datatypes like lists and vectors, universal polymorphism is usually more flexible. The same type may be instantiated to several different instances without having to apply a functor each time, and the same function can be applied to each instance.

However, some polymorphic types need operations on their elements. For example, the implementations of sets in the Edinburgh library require an equality function on the elements. Other types, or alternative implementations of sets, might require an ordering function. This changes the balance between the two types of polymorphism.

If we use universal polymorphism, then we have to give the set operations access to the equality function on the elements. One way to do this is to pass the function as a parameter to the operation. This makes the operations cumbersome to use. It also fails to prevent different functions being used in different places. For an implementation of sets based on an ordering function, this would give erroneous results. For example, someone could write:

```
val s = Set.fromList Int.lt [7, 3, 9];
(* Now s is implemented as the list [3, 7, 9] *)

val s' = Set.insert Int.gt 5 s;
(* s' is implemented as the list [5, 3, 7, 9] *)

Set.member Int.gt 7 s';
(* This call incorrectly returns false. *)
```

An alternative is to make the ordering part of the value, by passing it to the creation function. Then operations will always use the correct function, and only the creation operations will have the encumbrance of taking the function as a parameter. The drawback to this approach is that it doesn't prevent operations from combining two values with different ordering functions, as both functions will have the same type. As in the previous case, this could lead to errors.

However, the drawback is not so severe with this approach. Often, access to the ordering functions will be limited to the person writing the library entry. If that person can code the entry correctly, then users of the entry will not see any problems. For example, the set union function might be defined as follows:

```
fun union (Set (f, l)) (Set (_, l')) =
  Set (f, remove_duplicates f (l @ l'))
```

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

This definition picks one of the ordering functions to use in the result set, but ensures that this function is used to order the elements in the set. Thus the result of the union operation is consistent. This approach still isn't completely safe, and there may be functions that can't be written safely at all with this approach, but it is better than the previous approach.

The effect aimed for by passing operations to the set creation function can be achieved easily using a functor. For example, the `MonoSet` functor of the Edinburgh library takes two arguments, a type and an equality operation on that type. Applying the functor produces a new type, so any attempts to combine sets with different equality operations will give a type error. Also, the result signature of the functor doesn't mention the equality function at all, and so is simpler than either of the alternatives for universal polymorphism. So existential polymorphism is generally better when a polymorphic type requires some operations on its types. For example,

```
functor MonoSet (
  type T
  val eq: T -> T -> bool
): sig
  type Set
  type Element
  val fromList: Element list -> Set
...
end = ...
```

The drawback of this approach is that each application of the functor creates a new type, and new functions on that type. Any functions on sets written outside the set entry itself will have to be parameterised on the result signature of the set entry if they are to be polymorphic. This adds more functor calls to the program.

The choice of which approach to use depends on the program. If several instantiations of a type are needed in a module, then the universal approach is probably easier. If only one instantiation is required, the existential approach is often easiest. The existential approach also has the advantage that type safety is guaranteed, while the universal approach sometimes requires care on the part of the library implementer. The Edinburgh library provides both sorts of entries for sets, so that programmers can select the version most appropriate to their needs.

When one of the operations required by a type is equality, another option is to use equality types. This approach seems to combine the advantages of the first two approaches. It does not require an equality function to be passed as an argument, and so the result signature is simpler than for ordinary universal polymorphism. If equality is the only operation required, then the implementation is fully type-safe without having to use functors.

However, a set entry that uses equality types can only be used with elements that support the built-in definition of equality. So a general-purpose library must provide a version with explicit equality as well, and a truly polymorphic application must use that entry. Polymorphic equality only solves some of the problems some of the

time. The Edinburgh library does provide an implementation of sets with equality types, but its not clear how useful this will be.

Existential polymorphism can also be used to provide versions of polymorphic types that are specialised to their argument type. For example, the Edinburgh library includes a `MonoVector` functor that takes a type and produces a vector of elements of that type. In the generic case, these vectors will usually be implemented as a sequence of machine words, either integers or pointers to more complex values. However, we can provide structures that implement a boolean vector as a bitset, or a real vector as a sequence of double-word values, in such a way that these structures match the same result signature as the functor. Provided the vector type is abstract, the specialised cases will be indistinguishable from structures generated by applying the functor, except that they will be more efficient.

A sophisticated compiler might be able to optimise the representation of the type automatically, by delaying the compilation of the functor until it is applied. In this scenario, we would not need to supply the specialised implementations.

This idea could be extended to support the implementation of vectors on special hardware. For example, Blelloch's vector processing system requires that the elements of a vector are stored sequentially (Blelloch, 1990). A hardware-specific implementation of the `MonoVector` functor could implement the desired representation for any datatype.

6 Generic signatures

The use of functors to implement existential polymorphism can be aided by conventions about the names of types and functions. For example, if the `MonoVector` functor requires a type `T` and functions `eq`, `string` and `print`, we can ensure that all entries that define a type give the type the name `T` in addition to the usual name, and we can also ensure that we always use the name `eq` for the equality function, `string` for the string conversion function, and `print` for the print function. For example, in the Edinburgh library the `Int` structure contains, inter alia, entities that match the following specifications:

```
eqtype int
eqtype T
sharing type T = int
val eq: T -> T -> bool
val print: outstream -> T -> unit
val string: T -> string
```

With this convention we can apply the `MonoVector` functor to any such entry. For example, we can apply it to the `Int` structure directly. We can also apply other functors that follow this convention.

As a further step, the argument signatures of these functors can be given names. For example, in the Edinburgh library the argument signature of the `MonoVector` functor is called `EQ_PRINT`. We call these *generic signatures*. Other examples define orderings, print functions, parse functions and equality types. For example:

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

```
signature EQUALITY =
sig
  type T
  val eq: T -> T -> bool
end

signature PRINT =
sig
  type T
  val print: outstream -> T -> unit
  val string: T -> string
end
```

These signatures can be combined using the `include` feature of SML. This includes the components of a previously declared signature in the current signature. The general way to combine these signatures is shown by this example:

```
signature EQ_PRINT =
sig
  local type T' in
    include EQUALITY
    sharing type T = T'
    include PRINT
    sharing type T = T'
  end
end
```

We have to ensure that the type `T` from the second inclusion doesn't hide the type `T` from the first inclusion. This is done by using the `sharing` specifications to make them both be the same as the local type `T'`. Since they are both the same as `T'`, the two types must be identical. Since `T'` is local, it doesn't appear in the resulting signature value.

Unfortunately, Standard ML of New Jersey doesn't support local specifications or the `include` construct. (The implementers seem to be omitting these constructs largely on ideological grounds). Since this compiler is widely used, the Edinburgh library implements signatures such as `EQ_PRINT` directly.

We also give a name to the result signature of the functor, which we use to document the entry in the same way as signatures of other entries are used as documentation. In the Edinburgh library we also use the convention that all arguments to functors are written using the expanded form `structure S : SIG ...`. The final definition of the `MonoVector` functor looks like this:

```
functor MonoVector (
  structure Element: EQ_PRINT
): MONO_VECTOR =
...
```


https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

7 Name and type conventions

The naming conventions introduced in the previous section are useful for more than just making functors easier to use. They also provide users with a consistent naming scheme, so that they don't have to remember different names for similar functions in different entries. In short, they are part of the user interface of a library. This section considers related aspects of a consistent naming scheme, such as the choice of case for identifiers, and whether functions should be curried. Although these aspects of library design seem almost trivial, there are some constraints on sensible choices, and some useful guidelines that can be followed.

The choice of which case to use for identifiers is largely arbitrary, but the language does impose some constraints. SML has seven types of identifiers: record labels, variables, constructors, types, structures, signatures and functors. Except for variables and constructors, these groups have distinct name spaces. Therefore most identifiers can use the same conventions without fear of clashes. However, variables and constructors should have different conventions to help users distinguish between them. Also, when pattern matching with records, a commonly-used abbreviation allows the same identifier to be used for both a record label and a variable. Therefore variables and record labels should follow the same conventions.

The Edinburgh library uses the convention that most identifiers capitalise the first letter of each word. Variables and record labels differ in that they begin with a lower-case letter. Thus the library inherits the long-standing SML tradition of distinguishing variables and constructors by the case of their first letter. The library also follows the tradition of using all upper-case letters for signature identifiers. Although it isn't strictly necessary, it does have the advantage that, on most operating systems, structures and signatures with the 'same' name (e.g. `Vector` and `VECTOR`) can be stored in files of the same name without causing name clashes.

The choice of which names to use is not restricted by the language; any conventions are up to the library designer to suggest. The Edinburgh library follows the approach taken by the Eiffel and Smalltalk libraries (Meyer, 1990; Goldberg & Robson, 1983) and uses consistent names for the same operations in different entries. For example, the function to create a compound value from its components is always called `create`. Functions that convert values from one type to another are given names of the form `Array.list` or `Array.fromList`, depending on where they are defined and the direction of the conversion. An alternative to the latter convention, used before the modules system was added to ML, is to use identifiers such as `arrayFromList`. But when the modules system is used, the extra occurrence of the string `array` in `Array.arrayFromList` is redundant.

Another decision to be made in a higher-order language such as SML is whether functions should be curried or whether they should take a single tuple argument. Providing both would greatly increase the size of the library for little gain. Curried functions have the advantage of flexibility, and allow the use of many functional programming techniques. This is the approach that we took in the Edinburgh library. On the other hand, curried functions can be less efficient, since the implementation must build a new closure for each curried argument.

Most compilers try to optimise the case when a curried function is applied to all its arguments at once. If this can be done, it makes the curried version as efficient as the the tupled version. Unfortunately, this optimisation isn't always safe. Consider the following expression:

```
f x (g y)
```

The optimised version evaluates both arguments before calling the function. But the curried version applies `f` to `x` before evaluating `(g y)`. If both these applications produce a side-effect, then the optimised version will perform the side-effects in the wrong order. Often it isn't possible to tell whether an expression will produce a side-effect. This is especially true in the body of a functor, when the function to be applied may be specified only by the argument signature. It may be possible to develop this optimisation further, but at present curried functions do seem to be less efficient than their tupled equivalents.

8 Failure and exceptions

Another design decision concerns functions that can fail. One example is a function that searches for a symbol in a dictionary. Another is a function that parses a value from a string. A third is a function for extracting an element of an array. SML provides two ways of indicating failure. Functions can either encode the success or failure in the type of the result, or they can represent failure by raising an exception. The Edinburgh library uses both approaches, depending on the situation.

The first of these approaches uses a type to encode the success status. Here is an example from the Edinburgh library:

```
datatype 'a Option =
  None | Some of 'a

val lookup: ('a, 'b) Table -> 'a -> 'b Option
```

The returned value can be extracted with a case expression:

```
case lookup t x of
  Some y => y
| None => code for fail case
```

This approach has the advantage that the type of the function shows that the function may fail. It also ensures that programmers are warned if they don't test the success status, because the result can only be obtained using a case expression, and the compiler will report that the case expression isn't exhaustive. (Note: Standard ML of New Jersey provides a built-in type similar to this, but other compilers do not. Therefore the library defines its own version.)

The second approach signals a failure by raising an exception, and just returns the desired value in the successful case. The following example is equivalent to the previous one:

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

```
exception Lookup of unit

val lookup: ('a, 'b) Table -> 'a -> 'b
```

The failure case is dealt with by trapping the exception:

```
lookup t x
handle
  Lookup () => code for fail case
```

This approach may be slightly more efficient in some cases, because the success status doesn't have to be built in to the result by the function and then decoded by the caller. This advantage will be most noticeable when the error occurs deep in a nested function call and is signalled back to the original caller without being modified along the way. In such cases this approach may yield a neater program as well. However, the possibility of failure is not indicated by the type of the function, and the compiler won't warn users if they forget to add the exception handler or if they put it in the wrong place.

For cases like this, the Edinburgh library uses the first approach. This is mainly for the extra safety, but also because we expect that the success status of library calls will usually be tested by the immediately enclosing caller, either because the caller is the user's own function, or because the caller will be adding information to the error value. An example of the second case is the function for parsing lists, which returns the elements of the list that were successfully parsed before the error occurred. We believe that any loss of efficiency won't be important in real programs. If necessary, a more efficient version of these functions could be provided for those cases where the extra efficiency is needed.

The library takes a different approach when a function is called with an argument that falls outside the range that it can handle. An example is a function for accessing an array element called with an index greater than the size of the array. We hold this problem to be a genuinely exceptional case; in a correct program a function should not be called with incorrect arguments. Therefore the Edinburgh library raises an exception in such cases. The exception is given arguments that suggest the cause of the error. For example, if a subscript is out of range, the subscript is returned as part of the exception. This gives programmers some indication of what caused the problem.

Another possible cause of failure is a function that is specified but not implemented. This can occur when the function relies on the compiler providing an operation that isn't defined in the standard initial basis. For example, most compilers provide a function to change the current directory, but this is not part of the definition of the language. Rather than omit the function altogether, the Edinburgh library specifies the type of the function in the signature of the entry, and uses the

appropriate call to the compiler. If the compiler doesn't provide this function, or if the user compiles the portable version of the library (which only uses features defined in the initial standard basis), then the implementation of the function raises the `NotImplemented` exception instead.

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

9 Higher-order functions and polymorphism

SML provides both higher-order functions and (universally) polymorphic datatypes. The interaction of these features constrains the definition of certain types, such as the `Option` type defined in the the previous section. To be practical, these types must be defined centrally, rather than in each module.

An example of this is provided by the functions that the library provides to parse values. Parsing values of simple types such as integers, booleans and strings is straightforward. But parsing polymorphic sequences such as lists and vectors is more problematic, because the parameters of these functions must include a function to parse the elements of the sequence.

The complication arises with the result type of these function parameters. As explained in the previous section, we prefer to encode the success status of a function in its result value. The parse functions encode this status using an `Option` type, as defined in the previous section, and a `Result` type, which is defined as follows:

```
datatype ('a,'b) Result =
 OK of 'a | Fail of 'b
```

One approach would have been to define a `Result` type and an `Option` type in each library entry. This would have been the most modular solution. For example, with this approach the parse function for integers could have the following type:

```
val parse: string ->
             (int * string,
             int Int.Option * string) Int.Result
```

This function takes a string, and returns one of the following values:

`OK (`*i*`, `*s*`)` A successful parse of the integer *i*, with the rest of the string being *s*.

`Fail (None, `*s*`)` An unsuccessful parse. The first argument provides the possibility of returning a partially read value, but in this case there is nothing to return. As before, the rest of the string is returned is *s*.

If there was a partial value to return in the second case, the value `None` would be replaced with a value of the form `Some (i)`.

This function would be fine if it were only used on its own. (Indeed, if that were the case it need not use the `Option` type.) However, if it were passed to the function for parsing lists, the latter function wouldn't be able to refer to the `Int.Option` and `Int.Result` types directly, because it is polymorphic. The best it could do would be to treat the result type as an abstract type, and take extra arguments for extracting the components of the type. This would be cumbersome.

Instead, the Edinburgh library defines a single `Result` type and a single `Option` type. These are defined in a structure called `General`. The types in this structure are shared by all the other library entries. In particular, all parse functions return values of this type. With this approach, the type of the parse function for integers becomes:

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

```
val parse: string ->
             (int * string,
             int General.Option * string) General.Result
```

With this convention, we can give the type for the function to parse lists. Its first argument is a function to parse the elements of the list, such as the above function for parsing integers. The other arguments are the same as for the simple function. If this function encounters an error during the parse, it returns a list of the elements parsed so far.

```
val parse: (string -> ('a * string, 'a Option * string) Result) ->
      string -> ('a list * string, 'a list Option * string) Result
```

Similar considerations apply to the definitions of exceptions. If failure were signalled by exceptions instead of using the `Result` type, it would be most practical to define these exceptions centrally.

This concludes our discussion of library design issues. Clearly there are many more detailed decisions to be made when writing a library, from general conventions such as how to specify field widths in print functions, to application specific aspects such as which operations to include and which to leave out. All software design includes such decisions, and although they are important they are too detailed to discuss in this paper.

10 Higher-order functors

The remainder of the paper discusses the second kind of lesson that we have learnt from the design of the Edinburgh library; those aspects of SML itself that obstructed a clean library design. The first such issue that we discuss, and the main call for change in this paper, is the issue of higher-order functors.

Section 3 raised the problem of hiding the dependencies of a functor on other entries in the library. Ideally, we would like these dependencies to be made via parameters of the functor, but we don't want the users to know about them. We can't do this with the language as it stands, because we can't separate the implementation dependencies from the arguments that the user sees. Research by Tofte and others on higher-order functors may be able to help.

What we need is the possibility of defining curried functors, just as we can define curried functions in the core language. The idea is that a functor could be applied to one set of arguments, used to implement the functor, and return another functor that could later be applied to the user's arguments. For example, the implementer could define the curried functor `MkMonoVector`:

```
functor MkMonoVector (structure L: LIST)
                     (structure Element: EQ_PRINT) = ...
```

When the user loaded this entry, the make system would automatically apply it to the first argument:

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

```
functor MonoVector (structure Element: EQ_PRINT) =
  MkMonoVector (structure L = List);
```

The user would then see the desired functor, and the dependency on the `List` entry would be hidden. The user could apply the functor in the normal way:

```
structure IntVector = MonoVector (structure Element = Int);
```

Curried functors are a particular form of higher-order functors, and would be allowed if higher-order functors were added to the language.

A related question is why the library entries aren't contained in a single `Library` structure. The advantage of such a scheme would be that the library could be manipulated as a whole, and name clashes would be avoided when another package used the same name as a library entry.

This would indeed be desirable. The problem is that the library includes signatures and functors as well as structures, and SML does not allow these to be contained in structures. Structures are the only constructs of the modules system that can be contained in each other. This limitation would be removed by admitting higher-order functors and signatures.

Higher-order functors are also useful for structuring applications. Tofte gives the following example in his paper (Tofte, 1992). We define a signature for monoids, and a functor `Prod` that takes two monoids and produces their product (another monoid). Then we want to define a functor `Square` that uses `Prod` to produce the product of a monoid with itself, which we can use to define a plane. Without higher-order functors, we can only define `Square` if we have already defined `Prod`, as shown here:

```
signature MONOID = sig
  type t
  val e: t
  val plus: t * t -> t
end;
functor Prod (structure M: MONOID
              structure N: MONOID): MONOID =
struct
  type t = M.t * N.t
  val e = (M.e, N.e)
  fun plus ((x1, x2), (y1, y2)) =
    (M.plus (x1, y1), N.plus (x2, y2))
end;

functor Square (X: MONOID): MONOID =
  Prod (structure M = X; structure N = X);
```

With higher-order functors we can parameterise `Square` on a functor signature `PROD` that is matched by `Prod`. Now `Square` is more modular and can be compiled without reference to the `Prod` functor:

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

```
signature PROD =
  (structure M: MONOID
  structure N: MONOID): MONOID;

functor Square (structure X: MONOID
                functor Prod: PROD): MONOID =
  Prod (structure M = X; structure N = X);
```

The Definition of Standard ML did not include higher-order functors because they presented significant theoretical problems. Tofte shows how most of these problems can be resolved. The few remaining problems seem to have been resolved since the publication of Tofte's paper, and there exists a trial implementation. In my opinion, it would be desirable to add higher-order functors to the language. Their utility outweighs the disadvantages involved in changing the published definition.

11 Identifier attributes

In SML, value identifiers can have several attributes. They may be infix or nonfix, constructors or variables, overloaded or not. Each of these options causes problems.

The problem with infix status is that it isn't exported from a structure when the structure is opened. So a library can design a certain identifier to be infix, but if users want to use it this way they have to type the infix declaration themselves.

The problem with constructor status is that it is lost whenever the constructor is bound to a new identifier. This means that it isn't possible to include, for example, the constructors of the `list` datatype in the `List` structure. This would be desirable for several reasons, including the ability to access these constructors even if the identifiers were rebound at top level, and for completeness.

The problem with overloaded status is that it is lost if the identifier is rebound, and it can't be reintroduced. This has the same effect as the limitation on constructor status; overloaded identifiers can't be included in library entries without losing their overloaded status. (Users can't define overloaded operators – they exist only in the initial basis.)

This is a known limitation, but the designers of Standard ML thought that the advantages of overloading the arithmetic operations outweighed the disadvantages. Generalising the ability to overload functions in the context of polymorphic type inference is a hard problem, as the development of Haskell illustrates (Hudak *et al.*, 1992; Wadler & Blott, 1989). More work yet would be required to integrate the Haskell scheme with the modules system of Standard ML, and it is unclear whether the extra complexity would be worth the gains.

These problems have been noted before, for example by Appel (Appel, 1992). They affect application programs as well as libraries. However, their effect on the design of libraries is particularly noticeable.

12 Equality

The SML notion of polymorphic equality has been widely criticised. We have already noted in section 5 that library entries should use an explicit equality function, so that they are as general as possible. Polymorphic equality offers little to the library implementer. Even at best it is little more than a distraction.

Unfortunately it can be worse than just a distraction. Given that the language includes a definition of equality, we would like it to apply correctly to types that we define. However, it doesn't always give us the behaviour that we desire. During the development of the Edinburgh library this became apparent in the cases of abstract types and of constant references.

An example of the first case is the `Set` entry. We can implement a set as an unordered list. (There are other, better, representations, but this simple one might be used for reference purposes, and shows the problem well.) If we don't use the `abstype` construct to implement the type, but rely on hiding the constructors with a signature to make the type abstract, then the built-in equality applies to this type. Of course, the built-in equality compares the representations of two sets as lists, which is not the desired behaviour.

On the other hand, if we use the `abstype` construct, then the built-in equality doesn't apply to the new type. While this is better than the alternative, it reduces the utility of the built-in equality. There is no way to integrate a user-written equality function with the built-in version. This is another advantage of the Haskell type system, which was mentioned in the previous section with respect to overloading.

The second case arises when defining constant references. A SML reference value is an imperative construct: it can be assigned new values of the appropriate type. Two expressions with a reference type are only equal if they evaluate to the exact same reference. Sometimes it is useful to have values with the same definition of equality as references but which cannot be assigned to. We call these constant references.

At first sight it seems that we could implement constant references in terms of ordinary references, restricting the operations available on the type with a signature, so that only creation and equality were permitted. Ideally, we would like this type to have a visible constructor `const`, analogous to the constructor `ref` for ordinary references. Even if this is not possible, we would like it to have the same behaviour with regards to equality as ordinary references.

Unfortunately neither of these are possible. If we made the constructor visible, then users could get at the underlying implementation. If constant references were implemented in terms of ordinary references, this would let them assign new values to a supposedly constant reference.

We can define a type for constant references that admits equality, but it will only do so when its argument does. There is no way to tell the type system that the equality status of the argument is unimportant. The best we can do is to provide an equality function that does the right thing, for the cases when the built-in polymorphic equality function doesn't apply.

When polymorphic equality was introduced, it seemed to simplify some examples,

even though it was never expected to solve all the problems of defining equality on user-defined types. Experience has shown that its use is limited, and that it greatly complicates the semantics and implementation of the language. It also complicates the design of a library. The evidence suggests that it causes more problems than it solves.

13 The initial basis

The types, functions and exceptions defined to be part of SML itself are collectively called the initial basis. The existing definition has several shortcomings. Compiler writers have already agreed on extensions for one-dimensional arrays and one-dimensional constant vectors. The development of the Edinburgh library has encountered several more problems with the existing definition. This section briefly mentions several of these limitations.

The numeric types have a couple of problems. One that is widely recognised concerns the exceptions raised by the arithmetic operators. The Definition of Standard ML defines a separate exception for each operation. This doesn't match the hardware exceptions provided by most processors, and typically reduces the efficiency of implementations. It also doesn't match the proposed Language Compatible Arithmetic Standard (Wichmann *et al.*, 1990) or the IEEE standard for floating point numbers (IEEE, 1985), and would be unlikely to match any similar standard.

Therefore the Edinburgh library uses a different scheme, also used by some compilers, in which any arithmetic operation that results in overflow raises the `Overflow` exception. This can be defined in terms of the official exceptions, and can be implemented efficiently. (This is similar to the way the `Vector` and `Array` types are defined in terms of lists but implemented directly.)

The numeric type `real` has other problems as well. Although basic arithmetic operations are provided on reals, there are no functions to extract the mantissa and exponent, or to create a real number given these components. This makes printing and parsing real numbers difficult, at best. At present, the Edinburgh library only specifies these operations in a signature. It only implements them in the structure if the compiler provides them; otherwise calls to these functions will raise the `NotImplemented` exception as described in section 8.

A different problem arises with the `Array` type. The other sequence types that the Edinburgh library provides each have an associated `empty` constant value that contains the empty sequence. Unfortunately it isn't possible to provide this constant for arrays. The problem is that arrays are imperative data objects. This means that the form of polymorphism that they can support is limited (Milner & Tofte, 1991). However, the empty array can never contain any values, so it could have a fully polymorphic type without causing any problems. The type system can't detect this fact; the best that could be done would be to define the type of the empty array in the definition of the language (as was done for the dereference operator).

The last problem considered in this section concerns the I/O primitives. The Definition of Standard ML defines what happens when these primitives are called on streams that are closed, but it doesn't define what 'closed' means. In particular, it

doesn't say what happens when the user types an end-of-file indication on a terminal and the program then attempts to read from that stream again.

In practice, we want the ability to clear an EOF condition on a stream. Several compilers did this automatically, so that the first attempt to read from the terminal when the user typed the EOF character encountered the end-of-file indication, and later attempts read the next characters typed. They also made the `lookahead` function (and hence the `end_of_stream` function, which is defined in terms of `lookahead`) clear the end-of-file status when they encountered it.

However, this is a problem when reading a sequence of elements that is delimited by an end-of-file indication, such as a vector of integers. The obvious way to define the read functions is simply to return the read value when `end_of_stream` returns true. However, if the call of `end_of_stream` from the function that reads the elements also clears the end-of-file indication, a subsequent call from the function that reads the sequence won't detect the end-of-file. Therefore it will try to read another element. This behaviour is not what we desire, so the implementation of `lookahead` should be corrected.

In any case, the Definition of Standard ML and the implementations differ on the behaviour of the `input` function. The Definition says nothing about clearing an end-of-file indication. It should be changed, either to agree with the approach taken by the compilers or to include a specific function to clear the end-of-file status. The latter approach would be similar to that taken by the ANSI C standard.

One aim of the Edinburgh library was to specify extensions to the initial basis that would be portable between different compilers. Some of these extensions were implemented in compiler-specific implementations of the library, when compilers provided suitable functionality. Some features remain unimplemented because they require extra compiler support. For a few features, such as constant references, we faced a choice between a completely desirable implementation that we couldn't implement without further compiler support, and a less desirable specification that we could implement. In these cases we chose the version that we could implement.

This ends our discussion of the language design lessons learned from the implementation of the Edinburgh library. This has not been a complete discussion of the shortcomings of the language. It has concentrated on those aspects of the language that particularly affected the design of the Edinburgh library. Appel's critique mentions several problems that didn't affect the development of the library (Appel, 1992). Some other shortcomings of the language exist because the supporting theory for certain topics was not well developed when the language was designed. Research into these topics has advanced in recent years. Two particularly interesting areas are dynamic types (Abadi *et al.*, 1989) and local polymorphic references (Leroy & Weis, 1991; Wright, 1992).

The most important issue for the development of libraries is the lack of higher-order functors. Work is progressing on this topic, and we may see them added to the language sometime in the next couple of years. This extension should not cause problems to existing code. Work is also progressing on a revision of the initial basis, and this should be finished in 1993. We are probably stuck with polymorphic equality, because removing it would invalidate existing code.

14 Future developments

Since the release of the Edinburgh library, most of the ML team at Edinburgh have moved on. Cuts in funding mean that Edinburgh can no longer support the library. I have provided some stop-gap cover, but I can not develop it further.

On the other hand, the library has a liberal copyright. Users are free to extend or modify the library as they see fit, and are encouraged to add new entries. Thus new revisions can be added as particular groups of users see fit. The text of the technical report is distributed with the library. Indeed, it is partly generated from the library entries themselves, so users can get up-to-date documentation easily. We hope that the flexibility of this copyright will enable someone to take this work and develop it further.

Since the library is in active use at several sites, it faces the familiar tension of any published software between stability and evolution. Users don't want the framework or existing entries library to change, because they don't want to rewrite their code. On the other hand, some parts of the library could do with being improved. The library may also have to change to keep up with developments in the language.

One way that the library could be improved without changing the current specification would be to provide support tools. One useful tool would be a browser with the capability of searching for operations by name and by type. Research by Runciman and Toyn (Runciman & Toyn, 1991) and by Rittri (Rittri, 1991) has shown that functions can be located given only partial type information; this would be more useful than requiring users to know the exact types of the functions they are trying to find. Simpler tools would be useful too. One possibility is a program to check that entries are in the correct format, which would simplify the tasks of writing and checking in new entries. Another is a program to list the entries in each signature by name alone.

The library may also have to adapt to changes in the language. The language itself faces tension between stability and evolution, and although the tendency is (correctly) to resist change for the sake of change, there are two areas where changes could occur that would affect the library. The first is the addition of higher-order functors. As described above, these would be very useful. However, the library would have to be restructured to take advantage of them.

The other change to the language that would affect the library would be a revision of the initial basis. As mentioned in the previous section, a working party is currently looking at this. If the new basis adds new types, such as the types of characters or files, then the library should also support these types. Also, one aim of the library was to specify extensions to the standard initial basis. If the revised basis adds similar functionality, it will make sense to change the library to match in order to minimise confusion.

15 Conclusion

Designing a good, consistent library turned out to be much harder than we expected. However, the Edinburgh SML Library is popular at several sites. The library is freely distributed with all SML compilers and by FTP, and has a liberal copyright.

The initial choice of entries for the library was based on utility functions that programmers had already written for themselves. The interesting work was to design a framework for these functions, that would provide a consistent user interface and scope for extension.

This design was strongly influenced by the design of the language. The SML modules system allows the specification of each library entry to be separated from its implementation, and allows multiple implementations of an entry. The modules system extends the support for polymorphism in the core language, so that types can be parameterised on operations as well as other types. It also supports the use of generic signatures to structure both the library and users' code.

The presence of powerful constructs in the language raises questions about when and how to use them. We had to decide when it was suitable to use exceptions to signal failure and when it was best to encode the success status in the result. The presence of two forms of polymorphism meant that we had to decide which form to use when. We also had to consider the general structure of the library, and came to slightly surprising conclusion that we should use structures instead of functors.

The design of the library has also highlighted some shortcomings in the language. This suggests that language designers should attempt to write some libraries in their languages, as well as applications, so that problems can be caught early on.

The most important of these shortcomings is the lack of higher-order functors. The SML modules system was explicitly designed to support programming in the large, yet it is now clear that without higher-order functors it does not fully achieve that goal.

One aim of the library was to specify extensions to the standard initial basis. Sites that use the library have found that this does indeed help programmers to write portable code. We also hoped that widespread acceptance of the library would result in the adoption of these extensions by compiler writers. Instead, compiler writers are working on a comprehensive revision of the initial basis, which will be independent of the library.

The other aim of the library was to provide a framework for reusable software components. We believe that we have succeeded. The test of our success will be how widely the library is used, and how many people extend it with their own components. We hope that it will stimulate people to develop libraries for their particular application areas.

Acknowledgements

Kevin Mitchell, Mike Fourman, Nick Rothwell and Diana Bental gave useful comments on a draft of this paper, as did anonymous referees.

Appendix: The current contents of the library

`General.`	Types, exceptions and functions that are widely used. In particular, the types `Result` and `Option` mentioned in the main text.
`Bool, Instream, Int, List, Outstream, Real, Ref, String.`	Basic operations on the pervasive (built-in) types. The `Ref` entry includes a random number generator.
`BoolParse, IntParse, ListParse, StringParse.`	Functions to parse the basic types from strings or instreams. The `ListParse` structure matches the `SEQ_PARSE` generic signature; the others match the `PARSE` generic signature.
`Ascii, StringType, StringListOps.`	`Ascii` defines constants for the non-printing Ascii characters. `StringType` defines functions for testing whether the first character in a string is a letter, digit, control character, etc. `StringListOps` defines functions on strings that mimic those in the `List` entry.
`ListSort.`	Functions to sort and permute lists.
`AsciiOrdString, LexOrdString, LexOrdList.`	Orderings on sequences. `AsciiOrdString` compares strings when case difference is significant; by contrast `LexOrdString` ignores case.
`Array, ArrayParse, Byte, ByteParse, Vector, VectorParse.`	Types and functions for bytes, arrays and constant vectors.
`Pair, PairParse, ListPair, StreamPair.`	Operations on pairs of objects. `StreamPair` also includes suggested functions for interacting with the host file system using pairs of one instream and one outstream.
`EqSet, Set.`	Polymorphic sets. `EqSet` defines sets over equality types. `Set` defines sets over arbitrary types.
`Hash, EqFinMap.`	Hash tables and finite maps. The keys to finite maps must be equality types.
`User.`	Functions to prompt the user for input.
`Make.`	The `Make` system.
`Const.`	Creates unique copies of objects.

https://doi.org/10.1017/S0956796800000873 Published online by Cambridge University Press

`System.`	Some suggested functions for interacting with the host file system and the SML compiler. Many of these functions are not implemented in the portable version of the library.
`Combinator.`	Simple combinator functions.
`Memo.`	Memoising functions.

The library also provides the following functors:

`MonoArray`, `MonoArrayParse`, `MonoList`, `MonoSet`, `MonoVector`, `MonoVectorParse`.	These return monomorphic equivalents of the `Array`, `ArrayParse` structures, etc.

The library provides the following generic signatures:

`PARSE`.	A type with functions for parsing values of the type from strings and instreams.
`ARRAY_PARSE`, `SEQ_PARSE`.	Variants of the `PARSE` signature for arrays and types with one type variable.
`MONO_SEQ_PARSE`.	The result signature for the `MonoVectorParse` and `MonoListParse` functors.
`EQUALITY`, `ORDERING`, `PRINT`.	A type with a function for testing equality, testing ordering or for printing values of the type.
`EQTYPE_ORD`, `EQTYPE_PRINT`.	An equality type with a function for testing ordering or for printing.
`EQ_ORD`, `EQ_PRINT`, `ORD_PRINT`, `OBJECT`.	Various combinations of `EQUALITY`, `ORDERING` and `PRINT`.
`SEQUENCE`, `SEQ_ORD`.	Variants of the above signatures for types with one type variable.

References

Abadi, M., Cardelli, L., Pierce, B. C., & Plotkin, G. D. (1989) (June) *Dynamic typing in a statically typed language.* Tech. rept. DEC SRC.

Ada. (1989) *The programming language Ada reference manual.* LNCS 155, Springer-Verlag.

Appel, A. W. (1992) (Feb) *A critique of Standard ML.* Tech. rept. CS-TR-364-92. Princeton University.

Appel, A. W., Mattson, J. S., & Tarditi, D. R. (1989) (Dec) *A lexical analyzer generator for Standard ML.* Tech. rept. Princeton University.

Berry, D. (1991) *The Edinburgh SML library.* LFCS Report Series ECS-LFCS-91-148. Laboratory for Foundations of Computer Science, University of Edinburgh.

Blelloch, G. (1990) *Vector models for data-parallel programming.* MIT Press.

Feldman, S. I. (1979) Make – a program for maintaining computer programs. *Software – practice and experience*, **9**, 255–65.

Goldberg, A., & Robson, D. (1983) *Smalltalk-80: The language and its implementation.* Addison-Wesley.

Harper, R. (1986) *Introduction to Standard ML.* LFCS Report Series ECS-LFCS-86-14. Laboratory for Foundations of Computer Science, University of Edinburgh. Revised 1989 by Nick Rothwell and Kevin Mitchell.

Hudak, P., Peyton Jones, S., & Wadler (editors), P. (1992) Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, **27**(5).

IEEE. (1985) *IEEE standard for binary floating point arithmetic, ANSI/IEEE Std 754-1985.*

Leroy, X., & Weis, P. (1991) (Jan) Polymorphic type inference and assignment. *Pages 291–302 of: Proceedings of the eighteenth ACM symposium on principles of programming languages.*

Meyer, B. (1990) Lessons from the design of the Eiffel libraries. *Communications of the ACM*, **33**(9), 68–88.

Meyer, B. (1991) *Eiffel: The language.* Prentice-Hall.

Milner, R., & Tofte, M. (1991) *Commentary on Standard ML.* MIT.

Milner, R., Tofte, M., & Harper, R. (1990) *The definition of Standard ML.* MIT.

Paulson, L. C. (1991) *ML for the working programmer.* Cambridge University Press.

Rittri, M. (1991) Using types as search keys in function libraries. *Journal of functional programming*, **1**(1), 71–90.

Runciman, C., & Toyn, I. (1991) Retrieving reusable software components by polymorphic type. *Journal of functional programming*, **1**(2), 191–212.

Sannella, D. (1991) Formal program development in Extended ML for the working programmer. *Pages 99–130 of: Proc. 3rd bcs/facs workshop on refinement, hursley park.* Springer Workshops in Computing.

Tarditi, D. R., & Appel, A. W. (1991) (Mar) *ML-Yacc.* Tech. rept. Princeton University.

Tofte, M. (1989) (March) *Four lectures on Standard ML.* LFCS Report Series ECS-LFCS-89-73. Laboratory for Foundations of Computer Science, University of Edinburgh.

Tofte, M. (1992) (Jan) Principal signatures for higher-order ML functors. *Pages 189–199 of: Proceedings of the nineteenth ACM symposium on principles of programming languages.*

Wadler, P., & Blott, S. (1989) (Jan) How to make *ad-hoc* polymorphism less *ad-hoc. Pages 60–76 of: Proceedings of the sixteenth ACM symposium on principles of programming languages.*

Wichmann, B., Schaffert, C., Payne, M., & Jaffe, M. (1990) (Dec) *Language compatible arithmetic standard, version 3.0.* Dfraft ISO/ANSI Standard. ISO Doc. No. ISO/IEC JTC1/SC22/WG11/N212. Ansi Doc. No. X3T2/91-006.

Wright, A. K. (1992) Typing references by effect inference. *In: European symposium on programming.* Springer-Verlag Lecture Notes in Computer Science.