Principles and a Preliminary Design for ML2000

The ML2000 Working Group*

March 10, 1999

Abstract

We describe the methodology and current features for ML2000, a new-generation design of ML. ML2000 adds a number of features to Standard ML and Caml, providing better support for extensibility and code reuse, while also fixing latent problems. Although none of the features is particularly novel on its own, the combination of features and design methodology are.

1 Introduction

In 1992, an ad hoc group of researchers started to investigate ways to improve the two main dialects of ML, Standard ML (SML) [33] and Caml [9]. ML already provides a strong basis for developing programs in the small and in the large: higher-order functions, references, datatypes and polymorphism are useful for programming in the small, and first-order modules with separate notions of interfaces and implementations are useful for programming in the large. No language is perfect, though: a number of new implementation techniques and advances in programming language design had been made since the development of free-standing ML implementations in the early 1980s. In order to take advantage of these advances, the group decided to begin a clean design, not necessarily upwardly compatible with any ML dialect but remaining true to the spirit of ML.

It is now 1999 and time for a status report. This paper gives a snapshot of the design of ML2000 as well as the principles used in arriving at the design. Part of the purpose here is to consolidate decisions and get feedback from the community; the rationale and principles for the design may also be useful in other contexts.

This paper is not the first to discuss language features in ML2000. Many possible design alternatives have been explored elsewhere by various members of the group, e.g., higher-order modules [20, 25, 26, 30], classes and objects [14, 15, 43, 44], and type-theoretic foundations [10, 22, 45]. The 1997 revision of SML [34] is one of the more visible outgrowths of the discussions. Simplifications and improvements made in this revision, including type abbreviations in signatures, value polymorphism, lack of structure sharing, restrictions to local datatype declarations, and others, are a part of ML2000.

The main goal of ML2000 is to better support programming in the large, and in particular, to make systems easier to extend, maintain, and modify. To achieve this goal, ML2000 enhances ML with support for object-oriented programming, subtyping, and higher-order modules. ML2000 also adds concurrency mechanisms to ML, which facilitates some programming paradigms and makes parallel implementations of the language easier. A number of other smaller features are also included: enhancements to polymorphism and type inference, changes in equality among datatypes, and support for hierarchical exceptions. Section 3 describes the new features of ML2000.

The design of ML2000 is, however, incomplete, and the group is still considering alternatives for some parts of the language. The discussion in Sections 3 and 4 include descriptions of some of the main questions, together with some possible answers. For instance, although much of the basic design of objects and classes have been settled (e.g., that classes are second-class entities separate from modules, providing for object construction and some sort of visibility control) the specifics has not been finalized. The ongoing development of the class systems in MOBY [14, 15] and Objective Caml [39] provide possible approaches.

*Participants in the ML2000 Working Group include Andrew W. Appel (Princeton University), Luca Cardelli (Microsoft Research), Karl Crary (Carnegie Mellon University), Kathleen Fisher (AT&T Labs-Research), Carl Gunter (University of Pennsylvania), Robert Harper (Carnegie Mellon University), Xavier Leroy (INRIA Rocquencourt), Mark Lillibridge (Compaq Systems Research Center), David B. MacQueen (Bell Laboratories, Lucent Technologies), John Mitchell (Stanford University), Greg Morrisett (Cornell University), John H. Reppy (Bell Laboratories, Lucent Technologies), Jon G. Riecke (Bell Laboratories, Lucent Technologies), Zhong Shao (Yale University), and Christopher A. Stone (Carnegie Mellon University). This summary was prepared and edited by Jon G. Riecke and Christopher A. Stone, with significant guidance from the working group.

2 Methodology

In building a new general purpose language, the design methodology can be as important as the technical design points. The starting point for ML2000 is, of course, ML. ML2000 shares some of the same characteristics as ML: it is a typed, call-by-value language with polymorphism, higher-order functions, recursive datatypes and pattern-matching, side effects in the form of exceptions, input/output operations, and mutable reference cells, and modules. The main design principles of ML—types as an organizing principle, focus on predictable semantics, and foundations in call-by-value λ-calculus—carry over to the design of ML2000.

First and foremost, ML2000 is a statically typed programming language. Every well-formed expression has a type, and Milner's theorem that "well-typed expressions do not go wrong" must hold. Types help the programmer, aiding in structuring both large and small programs, and provide a layer of abstraction above the machine. Moreover, types help the compiler writer: implementations of ML2000, unlike C, can choose different representations for data, even mixing different representations in a single compiled program. The types of ML2000 are chosen with orthogonality in mind: there should be at most one easily programmable means of coding up an algorithm or data structure with the same level of efficiency. Moreover, the design process for ML2000 relies on types: types provide a useful way to classify and compare language features. This idea, also used in the design of ML, goes back as far as Strachey's original work on denotational semantics [46]: the inherent type structure of a language lends insight into its design. Other designs of programming languages, e.g., Haskell [24], also use types to structure the language.

Types facilitate a second principle of the design of ML2000, namely predictability of semantics. We intend to have a formal description of the language specifying, in as complete a way as possible, the type system and the operational semantics of the language. Formal descriptions help to guide the development and ensure the consistency of independent implementations; moreover, they give a basis for comparing implementations when they stray from the precise definition. Note that having a precise definition and predictable semantics is not the same as a deterministic semantics; with concurrency the semantics can be non-deterministic, but the semantics can still specify an allowable range of behaviors.

The third main design principle of ML2000 is the "mostly functional" or "value oriented" paradigm. In a value-oriented language, one programs with expressions, not commands. Reference cells, exceptions, and input/output primitives are part of the language but used infrequently. The experience of programming in ML shows that many algorithms can be implemented almost as efficiently as in conventional, imperative languages, even without extensive use of the imperative features.

Two other distinguishing features of ML—type inference and the top-level loop—have less influence on the design of ML2000, although nothing precludes them. There are good reasons for deemphasizing type inference and the top-level loop, even though doing so might be seen as a radical break with ML's roots in interactive theorem proving. ML2000 is meant to be a standalone language, compiled and linked via standard tools or a system like the Compilation Manager (CM) of SML of New Jersey [5, 6]. Type inference, as convenient as it is in ML, is less important in such a language, and it allows us to consider a syntax with more type annotations. The inclusion of a top-level loop is relegated to the design of programming environments. Also, support for a top-level loop biases the design and complicates features such as concurrent input/output; the lack of an explicit interactive loop makes building standalone executables slightly simpler and gives a more standard basis for programming environments.

In designing features for ML2000, we evaluate choices based on three criteria [29]: semantics (what do the constructs do?), implementability (can the constructs be translated to efficient machine code?), and pragmatics (can the constructs express common and useful idioms?). Pragmatics, one of the more difficult criteria to apply, also includes issues of programming methodology (e.g., avoiding constructs like `typecase` which can break data abstraction). The criteria are interdependent. For instance, a single-minded emphasis on implementability can force one to rule out constructs that make the language substantially more usable; to ignore implementability can be disastrous for the practicality of a language.

One criterion not in this list is syntax. In fact, the concrete syntax still has not been designed; for the purposes of discussion, examples in this paper use a SML-style syntax. The syntax, once designed, should be without ambiguities and easy to parse (preferably LALR(1), unlike SML [1]). The decision to lay aside syntactic discussions was made early, and was, in retrospect, one of the most important. This is not to say that syntax does not matter, but one can still make decisions about features without constructing a concrete syntax. In order to avoid syntactic discussions, and to focus on the semantics of the language, the design of ML2000 falls into two parts (and both parts use abstract, not concrete, syntax). One part of the design, called the "external language," is the language that the programmer uses.

The external language may use type inference to fill in types left out by the programmer and may have constructs that can easily be translated as syntactic sugar. The second part, called the "internal language," serves to focus on the most essential features of the language. The internal language is explicitly typed, and all external language features can be translated into it. It may be suitable for use as the intermediate language of a compiler.

We are also committed to experimentation and working with implementations as a way to resolve open questions. Many mistakes in language designs are made in premature attempts at standardization and are only revealed during implementation or use of the language. We hope to avoid as many of these problems as possible. The ongoing work on MOBY and Objective Caml provides good opportunities for testing many of the ideas in ML2000.

3 New features

Most of the new features of ML2000 are designed to support reuse and the construction of open systems that can be easily adapted and extended. ML already provides support for reuse in the form of polymorphism, modules (structures), and parameterized modules (functors). ML2000's support for reuse, in the form of a module system, object-oriented programming, and subtyping, significantly extends ML at both the module and the core levels. The other main addition standardizes a set of concurrency primitives.

3.1 Subtyping and subsumption

While some features can be added to ML by extending the language, one that cannot is structural subtyping and subsumption. This is a change in the underlying type system and not a modular feature that is orthogonal to the other extensions. The primitive rule for subsumption, namely

$$\frac{C \vdash e \Rightarrow s \qquad C \vdash s \mathrel{<:} t}{C \vdash e \Rightarrow t}$$

implies that subtyping permeates the entire type system: there must be rules for every primitive type constructor defining the `<:` relation. For instance, the rule for subtyping on function types has the usual contravariant/covariant definition.

To make subtyping more useful, a simple declaration mechanism can be added to the module system. We have a way to declare partially abstract types in signatures similar to the "REVEAL" keyword of Modula-3 [18]:

```
type t <: T
```

Any value of type `t` can be used in a context expecting `T`, but the exact type is hidden. Partially abstract types are useful in providing object interfaces for abstract types.

3.2 Object-oriented programming

One of the goals of object-oriented programming is to support open systems and extensibility. In class-based languages like C++ [47] or Java [17], for instance, one can add to the methods of objects using subclasses; parts of the program that manipulate objects of the superclass still work for objects of the subclass. ML2000 includes objects and classes also to make systems more extensible, but objects and classes in ML2000 have some rather distinctive characteristics.

Object types exist independently from classes in ML2000; this gives additional flexibility to pieces of code that depend only on the signature, and not the particular class from which an object is generated. Object types are declared through definitions like

```
objtype window = {
   field position : int * int
   meth verticalSplit : int -> window * window }
```

where the form of the declaration resembles that found in MOBY [15]. This defines a recursive object type called `window` with one field and one method; the order of fields and methods is unimportant. A subtype of windows with borders can be defined by

```
objtype borderwin = {
   field position : int * int
   field borderSize : int
   meth verticalSplit : int -> borderwin * borderwin }
```

Object types follow a structural subtyping discipline. Both width subtyping (longer object types are subtypes of shorter object types) and depth subtyping (the types of corresponding methods and immutable fields of object types are in the subtype relation) are permitted. So, for example, the above type `borderwin` is a subtype of `window`; it is a width subtype because it has the additional `bordersize` field, and it is a depth subtype because the return type of the `verticalSplit` method has been specialized. The use of structural subtyping differs from Java, which uses a subtyping order fixed by class implementors. In Java, interfaces are the closest analogue of object types, but an interface `A` denotes a subtype of `B` in Java only if it has been declared to extend `B` [17].

The class system of ML2000, in contrast, has not been worked out as fully as object types; classes are discussed below in Section 4.

3.3 Modules

ML2000 includes a module system based on SML's with two main improvements. The first addresses the shortcomings of SML with respect to separate compilation. In SML, the typing information known about a structure is not expressible as a source-language signature; precise module interfaces can only be derived from their implementations. In ML2000, the principal signature (the one from which all other signatures may be derived) of every module is expressible within the source language; some restriction on the source language (such as requiring signature annotations on module bindings) seems to be required.

A second improvement is the addition of higher-order modules. In a higher-order module system, functors can serve as functor arguments, functor results, and structure components. The precise semantics for such higher-order modules has been, however, a subject of much debate. The semantics of MacQueen and Tofte [30] naturally extends SML behavior and the SML Definition, but it does not have a type-theoretically defined equational theory on types. Conversely, the systems of manifest types [25] and translucent sums [20, 27] have the desired type-theoretic flavor, but in the higher-order case may propagate fewer type equalities than one might expect. The extension of manifest types to applicative functors [26] appears to recover these equations but makes even fewer distinctions that SML or the MacQueen-Tofte semantics. Although versions of higher-order modules have been implemented in the SML of New Jersey and Objective Caml compilers, it is still not clear in practice that there are many compelling examples for higher-order modules. Indeed, some uses of higher-order modules can be replaced by uses of a separate compilation facility such as CM [5]. For these reasons, the exact flavor of higher-order modules has not been determined yet.

3.4 Concurrency

While existing ML implementations have one or more libraries for concurrent programming [8, 11, 41], none of these tightly integrates concurrency into the syntax, semantics, and implementation of the language. ML2000, in contrast, integrates support for concurrency into all aspects of the language.

ML2000 has the notion of independent threads of control and support for first-class synchronous operations. First-class synchronous operations were first introduced in Pegasus ML [40] and were refined in Concurrent ML (CML) [41]. For the most part, ML2000 follows the design of CML. It provides an event type constructor for representing synchronous values and the same set of combinators for constructing new event values from existing ones. It does, however, provide a different set of base event constructors: unlike CML, ML2000 does not provide output event constructors for channel communication (i.e., rendezvous communication). Instead, channels are buffered and send operations are non-blocking, making implementations on multiprocessors possible.

In addition to channels, ML2000 also provides synchronous memory in the form of I-structures [2] and M-structures [3]. The semantics of ML2000 must also specify the behavior of unsynchronized uses of shared memory. Such behavior is not easy to specify: an interleaving semantics, such as those for CML [4, 42], requires sequential consistency, but sequential consistency imposes a significant implementation cost on modern multiprocessors. Therefore, we need a weaker model, such as release consistency [16], for ML2000, and must specify the memory consistency model as a global property of program traces.

3.5 Other enhancements

3.5.1 Extensible type

The ML2000 extensible type is a new feature useful for creating user-controlled, hierarchical tagging schemes. These tags may be used for exception values, as in SML, as well as for tagging objects to support run-time dispatch and safe downcasting.

The declaration form

```
exttype io = IO of string
exttype read extends io = Read of string
exttype write extends io = Write of string
```

declares a new constructor `IO` of type `io` carrying values of type `string`, and subtypes `read` and `write`. Other type-constructor pairs can be defined as subtypes of `io`, `read`, or `write`, yielding a tree of exceptions rooted at `io`.

The pattern matching succeeds if the raised exception is a sub-exception of the pattern in the handle. This behavior is similar to exceptions in Java or Common LISP, except that there is no requirement that exceptions carry objects. For example, values of type `read` or a subtype match the pattern `(Read s)`; values of type `io` or a subtype (including values of type `write`) match the pattern `(IO s)`, and all other exceptions will match neither. In this way, the programmer can handle many different kinds of exceptions (e.g., all arithmetic exceptions) without having to exhaustively list the known possibilities or catch every exception, and allows a program to be extended with new variants of exceptions while preserving old code.

The extensible type can also be used to tag values and provide a notion of named subtyping instead of structural subtyping. Suppose, for instance, that `borderwin` is a subtype of `window` and both `window` and `borderwin` are object types. If we want, say, to distinguish objects of these types in heterogeneous collections, we can create extensible types

```
exttype win = Win of window
exttype bwin extends win = BWin of borderwin
```

and create our heterogeneous collection to be a group of tagged objects of type `textwin`. One can then safely downcast using pattern matching:

```
case obj of
     BWin (w : borderwin) => ...
   | Win (w : window) => ...
end
```

In effect, the extensible type gives the programmer the ability to define a typecase or classcase construct, much like that provided in Java [17] or other object-oriented languages.

The formal type rules of the extensible type follow those of Object ML [44], without the close association with objects. For soundness, the value carried by sub-constructors must be a subtype of the value carried by the parent constructor.

3.5.2 Datatypes

Datatypes and pattern matching are some of the most useful features in ML, and they are a part of ML2000 as well. There are two main differences, though, between datatypes in ML and ML2000.

The first difference is familiar from Prolog: there is an enforceable, syntactic distinction between constructors and variables. This distinction prevents certain programming errors and makes implementations simpler [1]. It already occurs in Objective Caml, Haskell [24], and MOBY [15].

The second difference is that datatypes no longer define abstract types (that is, they are no longer "generative"). Type-theoretically, a datatype declaration is simply a definition of a recursive sum type, with the associated injections and pattern-matching mechanism. In contrast to SML, two definitions of the same datatype bind equal types, just as two equal `objtype` definitions bind the same type, or two equal `type` definitions bind the same type. Abstract types then are created by signature ascription in the module system (i.e., by `:>` in SML'97 [34]).

3.5.3 Laziness

ML2000 is a call-by-value language. Nevertheless, there are times when lazy datatypes, the generalization of streams, can prove useful in programs. There are three known methods of incorporating lazy datatypes and functions on those datatypes in a strict language, one found in CAML [31, 50], one due to Okasaki, and one to Wadler, Taha, and MacQueen [49]. The last variant has been implemented in experimental versions of SML of New Jersey. One of these, or possibly a clean-slate design, will be included in ML2000.

3.5.4 First-class Polymorphism

ML provides polymorphic values via a `let` construct, where polymorphic quantification is inferred. For example, in the SML expression

```
let fun f x = x
in  (f true, f 0)
end
```

the type inferred for `f` is `['a] 'a -> 'a`, and it can therefore be applied to values of type `bool` and `int`. The quantification implicitly appears at the outermost layer of the type (the prenex restriction). ML2000, in contrast, requires the use of an explicit binding form for polymorphic functions, via a declaration like

```
fun f ['a] (x:'a) = x
```

This form of declaration is allowed, although not required, in SML'97 [34]. The explicit type abstraction in ML2000 also permits the use of nonregular recursive types, which has uses in the construction of purely functional, persistent data structures [36].

Once explicit type abstraction is part of the language, it seems reasonable to allow non-prenex quantified types, e.g., `(['a] 'a -> 'a) -> int`. Such types are useful, for instance, in Haskell [24], and in building records of polymorphic functions, and have been added in clever ways to languages with type inference [35]. No decision has been made, however, whether to include such types.

3.5.5 Tuples and records

SML records have a fixed number of named fields, whose values may be specified in any order. Tuples are a special case of records. Both tuples and records are used in functions taking multiple arguments or returning multiple results. Record arguments have a dual use in simulating call-by-keyword. For these two purposes, it seems desirable to allow record components to be specified in any order, and to allow very efficient implementations. In fact, both tuples and records in SML may be represented in a flat data structure with constant-time access to the components.

The design choices for records and tuples become wider in a language with structural subtyping. One possibility is to simply mimic SML, and have no structural subtyping on records or tuples. This possibility, however, limits potential interesting uses of records or tuples. For example, one might want to extend the above example of input-output exceptions with

```
exttype io = IO of {error:string}
exttype read extends io = Read of {error:string,file:string}
```

with the values carried by the exceptions in the subtype relation, so that more specific exceptions carry more information.

Another possibility for ML2000 is to have three different forms, in order of implementation cost and decreased efficiency of access: ordinary tuples or records with no width subtyping, records with prefix subtyping (where no reordering of fields is permitted), and records with width subtyping. Any of these forms might support depth subtyping, mutable fields, or pointer equality. For the sake of reducing the complexity of the language, it is probably best to choose two or three of the possible designs, and make the implementations as efficient as possible. The choice is a difficult issue of pragmatics, and experimentation is needed to see which choices are worthwhile.

3.5.6 Arithmetic

The 1990 definition of SML severely restricts the implementation of arithmetic: it defines a number of different exceptions that can be raised by various operations, each signifying "overflow" or "divide-by-zero" errors. For instance, an overflow during the computation of `(1 + x)` raises the `Sum` exception, whereas an overflow during `(2 * x)` raises the `Prod` exception. Interesting reorderings of computations are prohibited, and most implementations did not implement the standard. To avoid these problems, the 1997 definition of SML has only one exception for overflow and one for divide-by-zero.

ML2000 makes further steps in simplifying arithmetic. Integer computations in ML2000 are performed in two's-complement form with "wraparound" (no overflow exceptions occur), and moreover return values that match standard sizes of integers in languages like C and Java. This allows even more aggressive optimizations, and ensures better interoperability with operating systems and other languages like C. Of course, the programmer must be careful to ensure that computations do not overflow, or that overflows do not affect the correctness of computations. ML2000 will also provide arbitrary precision integers, and possibly other integer types or operations with overflow detection, in its library.

3.6 Features not found in ML2000

Every language must limit the number of primitives. ML2000 will not contain the following features—even though they can be useful in other contexts—because they do not interact well with the rest of the language, violate some of the design principles, or simply do not increase the expressivity of the language that much. Some of these features are

Generic equality Equality tests for values of base type is indispensable, and pointer equality of mutable structures like arrays and references is useful. The generic equality mechanisms in SML, however, are difficult to understand and are often the wrong equality on data structures.

The main difficult in understanding comes from equality type variables. For instance, in SML, one can declare a function

```
fun equalToBit (x,y) = if x=y then 0 else 1
```

with the inferred type `(''a * ''a) -> int`. The double quotes are used to flag the fact that any instantiation of the type variable `''a` must be a type that permits equality: integers, strings, characters, or lists or datatypes built from equality types alone. Unfortunately, types with equality type variables must often be propagated into signatures, often in places where equality type variables might not be necessary [1]. They also noticeably complicate the semantics of the language.

Generic equality is also rarely useful on complex data structures. For instance, the equality operation on sets of integers implemented as lists is not the same as the structural equality on lists. ML2000 thus does not have a generic equality function, equality type variables, or `eqtype` declarations.

Imperative type variables Like SML'97 [34], imperative type variables are not a part of ML2000. The value restriction of SML'97—the only polymorphic entities are those that cause no obvious side effects—may or may not be a part of ML2000. If type abstraction yields expressions with no side effects—that is, suspends computation [19]—there is no need for the value restriction.

Infix declarations ML2000 does not include support for infix identifiers. Infix identifiers become too complicated in a module-based language, when one must declare fixity in modules; moreover, infix identifiers have dubious utility in building large programs [29].

Abstype In SML, the "abstype" mechanism gives a way to construct abstract data types. Since such abstract types can be built in the module system, we omit this feature from ML2000.

4 Decisions left to be made

4.1 Type inference

Support for subtyping forces a radical change in the type system, and makes aggressive type inference less palatable. Type inference as well becomes much more complicated, both algorithmically—type inference without `let` is PSPACE-hard [23, 28]—and in the information given back to programmers [12, 38]. At the moment, Pierce and Turner's local type inference [37], itself based partly on Cardelli's ideas from Quest [7], appears to be the best hope for incorporating type inference in ML2000. Local type inference appears to be understandable and predictable. Both aspects are essential in explaining the language to beginning programmers: it would be best to avoid, if at all possible, complicated algorithms producing types with many quantified bounds.

One possibility is to make type inference into an implementation or programming environment issue rather than specifying a particular inference technique. For portability between different implementations, an *exchange format* must be defined (presumably mostly or completely annotated source code). A good programming environment could then elide or show these types as desired.

4.2 Subtyping and type constructors

Since ML2000 has subtyping, it could also have more complex type constructors (functions mapping types to types). In parallel with term-level bounded quantification the language could allow bounded quantification for the arguments of type constructors:

```
type ['a <: T] t = ...
```

Similarly, one might define an abstract type `'a bag` representing a (functional) collection type and want covariant subtyping behavior, where ($\mathtt{t}_1$ `bag`) is a subtype of $\mathtt{t}_2$ `bag` whenever $\mathtt{t}_1$ is a subtype of $\mathtt{t}_2$. This declaration is only typesafe for implementations of `'a bag` which use the type variable `'a` covariantly. Therefore one might want to annotate abstract type constructor specifications with polarity information to specify the variance of abstract type constructors.

This may complicate the metatheory considerably, and it is not clear yet how important this is in practice. If this is not included in ML2000 then abstract type constructors would all be considered invariant. Type constructors with known definitions (including datatypes and parameterized object types) would subtype according to the variance of their definitions.

4.3 Other object facilities

There are two points of view on cloning and functional update of objects (creating copies of objects with updated field values). In a value-oriented programming language based on ML, one might expect these operations, and they can be useful in programs. Indeed, ML2000 object methods can return objects, e.g., by calling a constructor of the enclosing class. However, generic operations for cloning or functional update of objects, like those supplied in Objective Caml, have the potential to break global invariants determined by an object's creator. For example, we might want to maintain a list of all objects created from a certain class or a certain function (e.g., so a message can be broadcast to all such objects); object copying operations would allow this invariant to be violated. If there is no functional update or cloning in the language, then `selftype` or `mytype` (a type variable which in object types represents the type of the enclosing object) become less useful. Eliminating cloning and functional update also simplifies the semantics of the language. We have yet to determine whether or not to include functional update or cloning in ML2000.

ML2000 provides no mechanism for downcasting or typecase on arbitrary objects, as this would allow clients of modules to break type abstraction. The hierarchical tagging mechanism described in Section 3.5.1 allows objects to be explicitly tagged in a fashion supporting safe downcasting.

4.4 Classes

Classes in conventional object-oriented languages provide a number of facilities combined into a single construct. They provide repositories for code, a mechanism for inheritance, a explicitly declared subtyping hierarchy on object types, access control (e.g., `public`, `private`, and `protected` members), and constructors to create objects. Classes may also enforce invariants about the objects they create.

An obvious question is whether classes are necessary in a language like ML2000. Namespace management, code encapsulation, and access control, for example, are already provided for by the module system. ML2000 has structural rather than by-name subtyping. Object constructors could simply be functions which return objects.

The model of user-programmed classes was examined in the context of Object ML [43]. This approach is possible, but inconvenient; it involves substantial programmer overhead and may be difficult to compile efficiently. Another approach explored by Vouillon [48] extends the module system to incorporate classes; that approach makes functors into more dynamic entities, which changes the character of the module system.

For convenience, therefore, ML2000 will have a modest class mechanism to support convenient object-oriented programming while minimizing the amount of new mechanism involved. The MOBY [15] and Objective Caml [39] class mechanisms provide example of this approach. Both provide class mechanisms for code inheritance and object creation. Both take advantage of the module system to provide functionality. For example, parameterized classes can be implemented using functors. The semantics of MOBY also uses standard notions of modules and subsumption to model component access restrictions (e.g., `private` and `friend`), using ideas from the study of extensible objects [13, 45]. Java or C++-style overloading of methods (multiple methods of the same name with different types) will not be a part of the class mechanism.

One of the more interesting uses of classes and modules together is a simulation of final classes. In Java, a final class is one that may not have any subclasses. In other words, final classes permit only object construction, not inheritance. We can achieve the same effect using a module and partial type abstraction, e.g.,

```
module Final =
struct
   objtype S extends T with { ... }
   class Foo { ... (* implements objects of type S *) ... }
   fun sConstruct (...) { ... (* returns object of class Foo *) ... }
end
```

where we restrict the module to a signature of the form

```
signature FINAL =
sig
   type S <: T
   val sConstruct : (...) -> S
end
```

Thus, the signature hides the class so that no inheritance is possible. Due to the partial abstraction, no other class can create objects of type `S` either, so the concept of final classes can be faithfully and simply modelled.

5 Conclusions

We have outlined the methodology, potential design space, and current decisions and rationale for ML2000, and a number of yet unresolved problems. Parts of the proposal may change, but we have confidence in the broad outline.

Once some tentative decisions have been made for the unresolved areas, the design will enter a phase of concrete syntax design, construction of a formal definition (type system and operational semantics), and experimentation with implementations and programs. We expect these pieces to affect one other: for instance, concrete syntax design influences, and must be influenced by, the formal definition and experience in programming with ML2000. Fortunately, the tools and techniques for these pieces already exist: formal definition techniques exist in the definition of SML [34] and in type-theoretic semantics [10, 22], and concrete syntax and implementations can borrow ideas from SML, Objective Caml, and MOBY.

Much of what we have learned goes beyond the design of the language. In the original ML2000 Manifesto from 1992, the original members of the working group wrote that

> The ML2000 project is largely a consolidation effort, incorporating recent research results and insights.
> This is not an open-ended research program.

More precisely, at that time it was thought that the state of research in type theory and language implementation was sufficiently advanced to improve ML. It has taken the group longer than anticipated, partly because we pursued a more aggressive design involving object-oriented features, and partly because a full consideration of the interactions among language features requires more work than just consolidation. Nevertheless, the effort has sparked a number of research projects (despite the original goal) and a substantial revision of Standard ML[34]. We hope that the eventual design will be as successful.

Acknowledgements

We thank Mads Tofte for being an early and valuable contributor to the working group. Thanks to Lars Birkedal, Kim Bruce, Elsa Gunter, Atsushi Ohori, Amit Patel, Jens Palsberg, Benjamin Pierce, Riccardo Pucella, Vitaly Shmatikov, Andrew Tolmach, and Andrew Wright for their contributions to the discussions.

References

[1] A. W. Appel. A critique of Standard ML. *Journal of Functional Programming*, 3(4):391–430, 1993.

[2] Arvind, R. S. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Programming Languages and Systems*, 11(4):598–632, October 1989.

[3] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In J. Hughes, editor, *Conference Proceedings of the Fifth International Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1991.

[4] D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 119–129. ACM, 1992.

[5] M. Blume. *Hierarchical Modularity and Intermodule Optimization*. PhD thesis, Princeton University, 1997.

[6] M. Blume and A. W. Appel. Hierarchical modularity: Compilation management for Standard ML. Technical report, Princeton Univ., 1997. Available from `http://www.cs.Princeton.edu/~appel/papers/`.

[7] L. Cardelli. Typeful programming. In E. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991.

[8] E. Cooper and J. G. Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, CMU School of Computer Science, December 1990.

[9] G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.

[10] K. Crary, D. Walker, and G. Morrisett. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, August 1998.

[11] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 113–123. ACM, 1993.

[12] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–184. ACM, 1995.

[13] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* 1993, 26–38.

[14] K. Fisher and J. H. Reppy. Foundations for MOBY classes. Technical report, Bell Laboratories, 1998. Available from `http://www.cs.bell-labs.com/~jhr/moby/index.html`.

[15] K. Fisher and J. H. Reppy. The design of a class mechanism for MOBY. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1999. To appear.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 15–26, May 1990.

[17] J. Gosling, B. Joy, and G. Steele. *The Java(tm) Language Specification*. Addison Wesley, 1996.

[18] S. P. Harbison. *Modula-3*. Prentice Hall, 1992.

[19] R. Harper and M. Lillibridge. Explicit polymorphism and cps conversion. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 206–219. ACM, 1993.

[20] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules and sharing. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 123–137. ACM, 1994.

[21] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. Programming Languages and Systems*, 15:211–252, 1993.

[22] R. Harper and C. Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, CMU School of Computer Science, 1997.

[23] M. Hoang and J. C. Mitchell. Lower bounds on type inference with subtypes. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 176–185. ACM, 1995.

[24] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[25] X. Leroy. Manifest types, modules, and separate compilation. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 109–122. ACM, 1994.

[26] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 142–153. ACM, 1995.

[27] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. Available as technical report CMS-CS-97-122.

[28] P. D. Lincoln and J. C. Mitchell. Algorithmic aspects of type inference with subtypes. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–304. ACM, 1992.

[29] D. B. MacQueen. Reflections on Standard ML. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693, pages 32–46. Springer Verlag, 1994.

[30] D. B. MacQueen and M. Tofte. Semantics for higher order functors. In D. Sannella, editor, *Programming Languages and Systems—ESOP '94*, number 788 in Lect. Notes in Computer Sci., pages 409–423. Springer-Verlag, 1994.

[31] M. Mauny. Integrating lazy evaluation in strict ML. Technical Report 137, INRIA, 1991.

[32] R. Milner. A proposal for Standard ML. *Polymorphism*, 1(3), December 1983.

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

[34] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[35] M. Odersky and K. Läufer. Putting type annotations to work. In *Conference Record of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 65–67. ACM, 1996.

[36] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[37] B. C. Pierce and D. N. Turner. Local type inference. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 252–265. ACM, 1998.

[38] F. Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 122–133. ACM, 1996.

[39] D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Conference Record of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 40–53. ACM, 1997.

[40] J. H. Reppy. Synchronous operations as first-class values. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259. ACM, 1988.

[41] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305. ACM, 1991.

[42] J. H. Reppy. *Higher-order concurrency*. PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, January 1992. Available as Cornell Technical Report 92-1285.

[43] J. H. Reppy and J. G. Riecke. Classes in Object ML via modules. In *Third Workshop on Foundations of Object-Oriented Languages*, July 1996.

[44] J. H. Reppy and J. G. Riecke. Simple objects for Standard ML. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–180. ACM, 1996.

[45] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999. To appear.

[46] C. Strachey. The varieties of programming language. In *Proceedings of the International Computing Symposium*, pages 222–233. Cini Foundation, Venice, 1972. Available as Technical Monograph PRG-10, Programming Research Group, University of Oxford, Oxford, March 1973. Reprinted in Peter O'Hearn and Robert Tennent, eds., *Algol-like Languages*. Birkhäuser, 1997.

[47] B. Stroustrup. *The C++ Programming Language (Second Edition)*. Addison Wesley, 1991.

[48] J. Vouillon. Using modules as classes. In *Informal Proceedings of the FOOL'5 Workshop*, January 1998.

[49] P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language, without even being odd. In *Workshop on Standard ML*, Baltimore, September 1998.

[50] P. Weis. The CAML reference manual, version 2.6.1. Technical Report 121, INRIA, 1990.