

An Implementation of Standard ML Modules

David MacQueen

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

Standard ML includes a set of module constructs that support programming in the large. These constructs extend ML's basic polymorphic type system by introducing the dependent types of Martin L f's Intuitionistic Type Theory. This paper discusses the problems involved in implementing Standard ML's modules and describes a practical, efficient solution to these problems. The representations and algorithms of this implementation were inspired by a detailed formal semantics of Standard ML developed by Milner, Tofte, and Harper. The implementation is part of a new Standard ML compiler that is written in Standard ML using the module system.

March 11, 1988

An Implementation of Standard ML Modules

David MacQueen

AT&T Bell Laboratories
Murray Hill, NJ 07974

1. Introduction

An important part of the revision of ML that led to the Standard ML language was the inclusion of module facilities for the support of “programming in the large.” The design of these facilities went through several versions [8] and was supported by concurrent investigations of the type theory of ML and related systems [9,11]. The central idea behind the design was to support modularity by introducing a stratified system of dependent types as suggested by Martin L  f’s Intuitionistic Type Theory [10]. In late 1985 Bob Harper added a prototype implementation of most of the module facilities to the Edinburgh ML compiler, which was serving as a test-bed for the evolving Standard ML design.

Starting in the spring of 1986, Andrew Appel, Trevor Jim, and I have implemented a new Standard ML compiler, written in Standard ML, and initially bootstrapped from the Edinburgh compiler. An overview of this new compiler, known as Standard ML of New Jersey, is given in [1]. The implementation of modules in this new compiler went through two generations. A first version was done in the fall of 1986, but it was completely rewritten in the summer of 1987 following discussions of the operational static semantics of modules with Robin Milner, Mads Tofte, and Bob Harper [6,7,12]. Like Harper’s prototype implementation, the new modules implementation was inspired by the static semantics, but it uses a *structure sharing* strategy [3,13] to avoid serious performance problems associated with a naive implementation of the static semantics. Although precise comparative measurements are not yet available, our experience shows that the symbol table size for a large ML program such as the ML compiler is several times smaller with the new compiler than with the old compiler.

The objective of this paper is to describe our implementation of the Standard ML module facilities, with particular emphasis on the techniques used to minimize the space consumed by static representations of modules (*i.e.* symbol table structures). We begin by reviewing the elements of the module language.

2. Summary of the module constructs

Before describing the basic issues concerning implementation of Standard ML modules, we need to review the main elements of the module language. There are three principal notions:

- (1) *signature* – interface specification or “type” of modules.
- (2) *structure* – an environment; a collection of type, structure, value, and exception bindings; corresponds to the conventional idea of a module.
- (3) *functor* – function mapping structures to structures; a form of parametric module.

Figure 1 below contains simple examples of each of these constructs.

```
signature ORD =
  sig
    type t
    val le : t*t -> bool
  end

structure S =
  struct
    datatype t = A | B of t
    val x = A
    fun le(A,_) = true
      | le(_,A) = false
      | le(B x, B y) = le(x,y)
  end

signature LEXORD =
  sig
    structure A : ORD
    val lexord : A.t list * A.t list -> bool
  end

functor LexOrd(O: ORD) : LEXORD =
  struct
    structure A = O
    fun lexord([],_) = true
      | lexord(_,[]) = false
      | lexord(x::l,y::m) = ... O.le(x,y) ...
  end

structure LS = LexOrd(S)
```

Figure 1

This example contains declarations of two signatures, ORD and LEXORD, two structures, S and LS, and one functor, LexOrd, mapping a structure of signature ORD to a new structure of signature LEXORD. The structure LS is defined as the result of applying LexOrd to S. We refer to components of a structure using qualified names or paths formed with the usual “dot” notation: *e.g.* S.t, S.x, LS.A.le.

A signature can be regarded as a form of “type” for structures, or as a schematic representation of a class of structures, and a structure *matches* a signature if it satisfies the specifications given in the signature. A structure does not have to agree exactly with a signature in order for it to match the signature; in this example the structure S matches the signature ORD, even though S has an additional value component x not specified in ORD. In such cases signature matching has a coercive effect, producing a “thinned” structure that exactly agrees with the signature in terms of number of components and their types.

Signature matching is performed in two contexts: (1) when a signature constraint is given in a structure declaration, as in:

```
structure R : ORD = S
```

and (2) when a functor is applied to an argument structure, which must match the signature specified for the formal parameter, as in

```
structure LS = LexOrd(S)
```

where S must match ORD. Actually, these two contexts are closely related under Landin’s principle of correspondence. In the first case, R is bound to a thinned version of S that does not contain an x component, and in the second case, the formal parameter O, and hence the substructure LS.A, is also bound to a thinned version of S.