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 Löf'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 Löf’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`.

Another kind of specification that may appear in signatures is the *sharing constraint*, the purpose of which is to insure a kind of type coherence among functor parameters. The program sketch in Figure 2 illustrates the use of sharing constraints.

```
signature SYMBOL = sig type symbol ... end

signature LEX =
  sig
    structure Symbol : SYMBOL
    val next : unit -> Symbol.symbol
    ...
  end

signature SYMBOLTABLE =
  sig
    structure Symbol : SYMBOL
    type var
    val bind : Symbol.symbol * var -> unit
    ...
  end

signature PARSE_ARGS =
  sig
    structure Lex : LEX
    structure SymTab : SYMBOLTABLE
    sharing Lex.Symbol = SymTab.Symbol
  end

functor Parse(A: PARSE_ARGS) =
  struct ... A.SymTab.bind(A.Lex.next(), v) ... end
```

Figure 2

The functor `Parse` essentially takes two structure arguments, `Lex` (implementing a lexical analyzer) and `SymTab` (implementing a symbol table), which are bundled as components of a single parameter structure. The sharing specification in `PARSE_ARGS` requires that the same `Symbol` structure be used in both `Lex` and `SymTab`. This insures that `Lex` and `SymTab` can consistently interact, as in the expression `A.Symtab.bind(A.Lex.next(),v)`, which is well-typed only if `A.Lex.Symbol.symbol` and `A.Symtab.Symbol.symbol` are the same type.

An important point about datatype and structure declarations is that they are *generative*, meaning that each time they are elaborated (*e.g.* in a functor body as a result of functor applications) a new, distinct structure or type is created. For example, in

```
functor F () =
  struct
    datatype t = A | B of t
  end

structure S1 = F()
structure S2 = F()
```

`S1` and `S2` are distinct structures and `S1.t` and `S2.t` are distinct types, so `S1.B(S2.A)` is an ill-typed expression.

On the other hand, simple type definitions (whether occurring inside or outside of structures) are *transparent* rather than generative. For instance, in

```
structure IntOrd =
  struct
    type t = int
    fun le(x,y) = x <= y
  end
```

the type `S.t` is identical to `int`. In other words, there is no information hiding or abstraction inherent in the formation of structures. This applies even to the results of functor applications; type information is propagated through functor applications, so that after the declaration

```
structure IntLexOrd = LexOrd (IntOrd)
```

`IntLexOrd.le` has type `int list * int list -> bool`. This reflects the dependent product nature of functor signatures, and the fact that structures represent a form of *strong* dependent sum (see [9,11] for discussion of the relation between ML modules and dependent types).

3. Implementation of modules

The principal tasks that an implementation must deal with are as follows:

(1) representation of signatures, structures, and functors.

(2) signature matching, including instantiation of the signature template and possible thinning of the matched structure.

(3) functor application, including

  (a) matching formal signature to actual parameter, with possible thinning of the parameter.

  (b) creation of the result structure, including propagation of type information from parameter to result and generation of new instances of datatypes and structures.

(4) representation and checking of sharing constraints.

Most of these tasks have two parts, the *static* or compile-time task and the *dynamic* or run-time task. The run-time problems are straightforward and are discussed in the next subsection. Our main focus will be on the static aspects of the module language, for which our principal implementation goals are:

(1) compact representation of structures having a given signature

(2) efficient signature matching and functor application, with minimal duplication of static (*i.e.* symbol table) information

(3) efficient representation and checking of sharing constraints.

3.1. Dynamic representations and processes.

The run-time representations of modules are remarkably simple [1]. Signatures and types have no run-time representation — they exist only at the static level. A structure is represented as a record whose components represent the dynamic structure components (*i.e.* substructures, values, and exceptions) in a canonical order. A functor is represented as an ordinary function closure, and functor application corresponds to the normal application of this function to a record representing the argument structure. The thinning coercions associated with signature matching give rise to in-line code to construct the thinned record.

In the middle-end of the compiler, all module constructs are reduced to the same simple lambda-calculus based intermediate language that is used for the core ML constructs of value declarations and expressions. In effect, the back-end of the compiler is unaware of the existence of the module constructs — they have been reduced to common notions of records and functions.

3.2. Static Representations

A naive representation of signatures and structures can be modeled more or less directly on the semantic constructs used in the operational static semantics [5,6]. There a structure is modeled by an environment *E* that maps component identifiers to the appropriate sort of static binding (type, structure, variable, etc.), and a *stamp*,* *n*, that uniquely identifies the structure: $str = (n,E)$. We can view a structure as a tree or dag with nodes labeled by stamps and edges labeled by component names. A signature is then a structure together with a designated set of *bound* or *schematic* stamps occurring within the structure: $sig = (N)(n,E)$.

We illustrate this with the definitions in Figure 3 and the corresponding graphs in Figure 4 (adapted from [6]), in which *(a)* represents the structure `C` and *(b)* represents the signature `SIGC`. Our convention for distinguishing between constant and bound stamps is that metavariables k_i range over constant stamps, while metavariables x_i range over bound stamps in a signature. This is a more concise alternative to the separate specification of the graph and the set of bound stamps We emphasize the distinction by using solid circles for nodes with constant stamps and open circles for nodes with bound stamps. A structure will always contain only constant stamps, while a signature will typically contain only bound stamps. The graphs are simplified by showing only structure components, but type components are dealt with similarly.

```
structure A =                       signature SIGA =
  struct                              sig
    type t = int                        type t
    fun f n = 2 * n                     val f : t -> t
  end                                 end

structure B =                       signature SIGB  =
  struct                              sig
    structure BA = A                    structure BA: SIGA
    fun g x = BA.f(x) + 1               val g: BA.t -> BA.t
  end                                 end

structure C =                       signature SIGC =
  struct                              sig
    structure CA = A                    structure CA : SIGA
    structure CB = B                    structure CB : SIGB
    fun h x = CB.g(CA.f x)              val h: CA.t  -> CA.t
  end                                 end
```

Figure 3

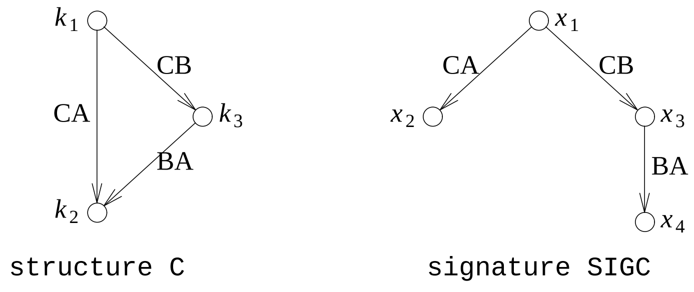

Figure 4

The purpose of a signature matching `S: SIG` is to produce a structure `S'` that has exactly the form specified by `SIG` and yet shares the identity (*i.e.* the stamps) of `S`. In some cases, `S` and `S'` are identical, as when `S` had already matched the signature `SIG`. In other cases `S'` is a thinned version of `S` having fewer components or components whose types are generic instances of their types in `S`. Another product of matching is the realization map, whose use in functor applications is explained in Section 3.5.

*We prefer the term ‘‘stamp’’ for this purpose, rather than the term ‘‘name’’ used in [6,7], since ‘‘name’’ could also refer to the identifier to which a structure is bound or an identifier bound within a structure.

We can think of `SIG` as a scheme analogous to a generic type scheme or polytype in the core ML type system [4], with the bound stamps playing the role of generic type variables in a type scheme. The product of matching is then an instance of this scheme under the substitution represented by the realization map. The details of this analogy have been worked out by Mads Tofte, including a version of the principal typing theorem of [4].

A naive implementation of matching would make a copy of the signature `SIG`, in the process replacing each bound stamp by the corresponding constant stamp from `S`. This would involve copying most of the environment part of the signature, since we have to instantiate the type specifications of values and exceptions as well as instantiating the types and substructures themselves. However, the environment or symbol table part of the signature can be regarded as a template relative to its type and substructure components, which are the only parts that need to change during signature matching. For instance, a type specification like `f: t->t` can remain fixed if it is interpreted relative to the type component `t`. We can abstract out the type and structure components carrying the bound stamps and use the rest of the information in a signature as an unchanging template that can be shared by all instances of the signature. This is the familiar *structure sharing* idea first proposed by Boyer and Moore in the context of resolution theorem proving [3] and later exploited in the implementation of Prolog [13]. The use of structure sharing in the basic ML type system has been considered, but in that context it does not appear to have a clear advantage over the simpler approach of instantiation by copying. In the case of signature matching, however, the shared information in the template is typically of considerable volume, so structure sharing is quite effective in saving space relative to copying.

The definitions of the basic datatypes used to represent type constructors, structures and signatures are given in Figure 5. The representations of structures and signatures both use the `Structure` datatype, and differ only in the value of the `kind` field. The `stamp` field contains the identifying stamp, the `table` is the environment component represented as a hash table mapping symbols to the various sorts of bindings, and the `env` field contains a pair of instance vectors for type and structure components. The `sign` field in a signature to identify the signature (the `stamp` field will have a formal value representing a bound stamp); in a structure it identifies the signature the structure is an instantiation of, if any. Bound and constant stamps are both represented as integers; stamps greater than some base value are constant stamps, while stamps less than that value are bound. Within a given signature, bound stamps are canonically numbered starting from 0.

```
datatype tycon
  = TYCON of {stamp : stamp, ...}
  | INDtyc of int list

datatype Structure
  = STRstr of
      {stamp : stamp,
       sign  : stamp option,
       table : symtable,
       env   : strenv,
       kind  : strkind}
  | INDstr of int

and strkind
  = STRkind
  | SIGkind of
      {share : sharespec,
       bindings : binding list,
       stampcounts : {s : int, t : int}}
```

Figure 5

The `INDtyc` and `INDstr` forms of type constructors and structures are used within the symbol table to refer indirectly to components stored in the instance vectors. The term `INDtyc[`*i*`]` refers to the ith element of the type instance vector, while INDtyc[i,j] refers to the jth element of the type vector of the ith element of the structure vector. The type specifications

```
f: t -> t
h: CA.t -> CA.t
```

from Figure 3 are represented internally as

```
f: INDtyc[0] -> INDtyc[0]
h: INDtyc[0,0] -> INDtyc[0,0]
```

The representation of the entire signature SIGC from Figure 3 can be summarized as follows:

```
SIGC:
  stamp:  0
  table:  CA => INDstr 0
          CB => INDstr 1
          h  => VAR: INDtyc[0,0] -> INDtyc[0,0]
  strenv: structures = <SIGA′,SIGB′>
          types      = <>

SIGA′:
  stamp:  1
  table:  t  => INDtyc[0]
          f  => VAR: INDtyc[0] -> INDtyc[0]
  strenv: structures = <>
          types      = <DUMMY 0>


SIGB′:
  stamp:  2
  table:  BA => INDstr 0
          g  => VAR: BA.t -> BA.t
  strenv: structures = <SIGA′′>
          types      = <>

SIGA′′:
  stamp:  3
  table:  t  => INDtyc[0]
          f  => VAR: INDtyc[0] -> INDtyc[0]
  strenv: structures = <>
          types      = <DUMMY 1>
```

Note that there are two copies of the signature SIGA, identified as SIGA′ and SIGA′′, each with its own stamp. This duplication is required to get the canonical numbering of stamps for each component of SIGC, but each of these copies shares the original symbol table component from SIGA. DUMMY 0 and DUMMY 1 are dummy type constructor components, which have their own separate numbering within the context of SIGC.

3.3. Signature Matching

We now describe the process of signature matching in terms of the representation described above. Given a signature sig and structure str represented as

```
sig = STRstr{stamp = x, sign = n, table = sigtab, env = sigenv,
             kind = SIGkind{bindings,stampcounts,sharing}}

str = STRstr{stamp = k, sign = s, table = strtab, env = strenv,
             kind=STRkind}
```

we first check whether `s=n` , and if so return `str`, because `str` is already an instance of `sig`. Otherwise we attempt to construct a new instance of `sig`. We start by allocating a new pair of instance vectors, `newenv={s=sNew,t=tNew}`, based on the size information in the `stampcounts` field. Then we iterate through the list of all of `sig`'s bindings (*i.e.* specifications), which is available in the field `bindings`. For each structure binding `(id,INDstr i)` in *sig*, we look up a structure named *id* in `strtab`. If it does not exist, matching fails. If it does exist, we recursively match it against the substructure signature bound to `id` in `sig` (obtained as the ith element of the structure vector in `sigenv`), and if successful use the result to define the ith element of `sNew`. Similarly for type bindings, where we check that the type constructor bound in `str` agrees with the specification in sig (*e.g.* they must have the same arity). For value specifications like `x: ty`, we interpret indirect type constructors in `ty` with respect to `newenv` and check that we have a generic instance of the type of the corresponding component of `str`. Checking value components has no effect on the instance vectors, but if necessary (*i.e.* if `sig` has fewer value components than `str`) we calculate the translation between the old and new runtime positions of the components and collect this information in a thinning specification to be applied at runtime.

When we have successfully matched all the bindings in `sig` we build the result structure `str´` using `sig´`s table and `sign`, `str´`s stamp , and `newenv`

```
str′ = STRstr {stamp=n, sign=SOME k, table=sigtab,
               env=newenv, kind=STRkind}
```

Finally any sharing constraints (from sharing) are checked as described in Section 3.5, and we return `str′` and a thinning specification as the result of the match. Note that the bulk of the information in the signature is in `sigtab`, and this is directly shared with the instantiation `str´`.

As a shortcut, when elaborating a declaration like

`structure S: SIG = struct` *declarations* `end`

we do not build the structure on the right-hand side before doing the signature match. Instead we elaborate the body declarations in the top-level environment and then do the signature matching using the top-level environment in the place of the target structure.

3.4. Functors and functor application

In the static semantics a functor F is modeled by a pair of structures representing the parameter and body of the functor, and two sets of bound stamps.

$$F = (N)(S_p, (N')S_b)$$

N is the set of bound stamps in the parameter structure, which may also occur in the body S_b, while N' is the set of stamps associated with generative elements of the functor body. To apply F to an argument structure A we perform the following steps

$$r = match((N)S_p, A)$$

$$g = generate(N')$$

$$F(A) = g(r(S_b))$$

That is, we match the parameter signature and the argument to produce a realization map r, then we generate a realization map g that maps each bound stamp in N' to a unique, new constant stamp, and finally we produce the result structure by using r and g to instantiate the body structure.

The implementation of functors follows this scheme closely. The datatype used to represent functors is defined by

```
datatype Functor = FUNCTOR of
                     {param : Structure,
                      body  : Structure,
                      tycCount : int}
```

The bound stamps in the `param` structure are numbered from 0 to n and these may also occur in the `body` structure. Generative stamps in the body are numbered from $n+1$ to $n+m$, which is the value of `tycCount`.

If the functor declaration provides an explicit result signature, as in

```
functor F(X : SIGP) : SIGR = struct ... end
```

the body will naturally be schematic (i.e. the parts with bound stamps will be isolated in instance vectors) as a result of the signature matching between the body and the result signature. However, if there is no result signature, we explicitly abstract these ‘‘volatile’’ parts of the body structure to get an instantiable scheme so that the body’s symbol table may be shared by all structures produced by the functor.

To apply the functor, signature matching is performed between the parameter signature and the argument to build a realization map for the bound stamps in the parameter. Then the body is instantiated using this realization map and introducing new constant stamps to replace generative bound stamps as required. The actual algorithm is more complicated than this because functor application can occur within the body of functor declaration, as in

```
functor F(X : SIGP) =
  struct
    structure A = G(X)
    structure B = H(A)
    ...
  end
```

In cases like the applications of `G` and `H` in this example, the actual parameter may contain parameter bound and even generatively bound stamps, and the realization of the generative stamps in the body of `G` and `H` will themselves be generatively bound stamps.

3.5. Sharing

The purpose of sharing constraints is to insure a kind of compatibility between several parameter structures of a functor, as illustrated in Figure 2. The sharing constraints are expressed as sets of equations between paths designating structures or types (there are two kinds of sharing specifications: structure sharing and type sharing) and they determine an equivalence relation amongst the components of the signature. The strategy for incorporating sharing constraints in the representation of a signature is to force all components of an equivalence class to have the same stamp.

Two components may be required to share either because they are directly equated in a sharing specification, or because they are corresponding components of structures that are required to share. Thus if

```
X : sig
      structure C1 : SIGC
      structure C2 : SIGC
      sharing C1.CB = C2.CB
    end
```

then `X.C1.CB = X.C2.CB` is directly specified, and `X.C1.CB.BA = X.C2.CB.BA` is an inferred consequence. This simply says that the complete sharing relation must be a congruence with respect to the operation of selecting a named substructure or type.

Under what circumstances may two structures be constrained to share? They must be *consistent*, in the sense that it is possible to find a structure that could simultaneously match both of them. In particular, the structures that are forced to share do not necessarily have to share the same signature, and the fact that they share does not have any effect on their signature. The idea is that various thinned versions of a given structure may have different signatures, but they can still share because they are actually restricted views of their common ancestor structure. This approach is supported by the fact that in signature matching, the stamps of the matched structure are inherited by the resultant structure. Note that this is not the approach described in [6], where signatures of sharing structures are forced to agree by formation of a kind of union signature. We do not actually verify that signatures that are specified to share are consistent, but if they are not, the signature containing the sharing specification can never be successfully matched.

The processing of the sharing constraints is performed in two stages. First, a union-find algorithm is used to determine all sharing relations, direct and inferred, and to construct the equivalence classes for the sharing relation. At this stage it is also possible to detect certain pathological sharing specifications, such as trying to identify a structure with one of its substructures. Second, the signature is copied and each element of a given equivalence class is given the same representative stamp.

There are two ways in which sharing information is used. (1) When a signature with sharing constraints is used as a functor parameter, the identification of stamps in the signature will automatically insure that the sharing has the desired effect during type checking, *i.e.* types that are specified to share will be seen to be identical by the type checker. (2) During signature matching, any sharing relations specified in the signature must also hold in the matched structure. One way to check this would be to make sure that the realization map was well defined, because a failure of sharing in the target structure would cause a single bound stamp to be mapped to more than one target stamp. However, the realization map is only explicitly constructed in the matching of functor parameters, where it is needed to help instantiate the functor body. Hence it is more convenient to simply save the original sharing constraints as equations in the signature and check them explicitly in the target structure as part of signature matching.

3.6. Relation with type checking

What is the relationship between the structures and signatures and the underlying ML type checking mechanism? Obviously signatures and structures are carriers of type information — that is one of their principle purposes. When we look up a value component of a structure we get the same sort of bindings as in the top-level environment, except that in some circumstances the type has been relativized to the structure's instance vectors and it contains `INDtyc` type constructors. The basic variable lookup functions have been defined to eliminate these indirections by replacing them with the referenced type constructors from the instance vectors, at the expense of partially copying the type. This would appear to undo some of the savings achieved by the structure sharing representation of structures, but these copies tend to be ephemeral, and they are quickly and efficiently garbage-collected. The type information in structures and signatures is, on the other hand, long-lived, so it is more critical to minimize their space requirements.

4. Conclusions

The challenge of implementing the Standard ML module features is to perform the compile-time matchings and instantiations necessary to propagate and check type information with a minimum of duplication of that information. Experience has shown that a naive approach leads to an explosion in the size of the static representations.

The implementation strategy described in this paper uses a structure sharing instantiation technique instead of instantiation by copying, and has proved to be reasonably modest in its space requirements. It also has the advantage that it remains quite close in spirit to the formal static semantics.

Work on the Standard ML module facilities continues, and current topics of interest include explicit functor signatures and the relation of the module constructs to separate compilation.

Acknowledgements

This implementation is inspired directly by the work done on the operational semantics of Standard ML by Robin Milner, Mads Tofte, and Bob Harper, and I have benefited from many discussions with them. Bob Harper passed on useful ideas from his prototype implementation of modules. Nick Rothwell helped me explore the use of structure sharing in type checking. Andrew Appel provided valuable advice throughout and had a direct hand in the implementation of sharing specifications, and he has been an equal

partner in the creation of Standard ML of New Jersey.

References

1. A. Appel and D. MacQueen, *A Standard ML compiler*, Proceedings of the Conference on Functional Programming and Computer Architecture, Portland, September 1987, G. Kahn, ed., LNCS Vol. 274, Springer-Verlag, 1987.
2. H.-J. Boehm and A. Demers, *Implementing Russell*, Proceedings of SIGPLAN 86 Symposium on Compiler Construction, Palo Alto, 1986, 186-195.
3. R. S. Boyer and J Moore, *The sharing of structure in theorem-proving programs*, Machine Intelligence 7, B. Meltzer and D. Michie, eds., Edinburgh University Press, 1972, 101-116.
4. L. Damas and R. Milner, *Principal type schemes for functional programs*, Proceedings of 9th ACM Symposium on Principles of Programming Languages, Albuquerque, 1982, 207-212.
5. R. Harper, D. MacQueen, and R. Milner, *Standard ML*, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, ECS-LFCS-86-2, 1986. (Also Polymorphism II, 2, October 1985.)
6. R. Harper, R. Milner, and M. Tofte, *A type discipline for program modules*, Proceedings TAPSOFT 87, LNCS Vol. 250, Springer-Verlag, New York, 1987, 308-319.
7. R. Harper, R. Milner, and M. Tofte, *The semantics of Standard ML, Version I*, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, ECS-LFCS-87-36, 1986.
8. D. MacQueen, *Modules for Standard ML*, in [5]. (An earlier version appeared in Proceedings ACM Symposium on Lisp and Functional Programming, Austin, 1984.)
9. D. MacQueen, *Using dependent types to express modular structure*, Proceedings 13th ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, 1986, 277-286.
10. P. Martin-Lőf, *Constructive mathematics and computer programming*, Sixth International Congress for Logic, Methodology, and Philosophy of Science, North Holland, Amsterdam, 1982, 153-175.
11. J. C. Mitchell and R. Harper, *The essence of ML*, Proceedings 15th ACM Symposium on Principles of Programming Languages, San Diego, 1988, 28-46.
12. M. Tofte, Operational Semantics and Polymorphic Type Inference, Ph.D. Dissertation, Dept. of Computer Science, University of Edinburgh, 1987.
13. D. H. D. Warren, *Implementing PROLOG - Compiling Predicate Logic Programs, Vol. I*, Dept. of Artificial Intelligence Report No. 39, University of Edinburgh, 1977.