Report on the Standard ML Meeting
Edinburgh, May 23–25, 1985
DRAFT

Robert Harper

1 Introduction

A three day meeting was held in Edinburgh on May 23 to May 25, 1985. The purpose of the meeting was to finalize the definition of Standard ML and to discuss any problems that have arisen since last year. The following people were present at the meeting:

Dave Berry                    Edinburgh
Rod Burstall                  Edinburgh
Luca Cardelli                 Bell Labs
Bob Harper [RH]               Edinburgh
Dave MacQueen [DBM]           Bell Labs
Dave Matthews [DCM]           Cambridge
Robin Milner                  Cambridge
Kevin Mitchell                Edinburgh
Alan Mycroft                  Cambridge
Larry Paulson                 Cambridge
Don Sanella                   Edinburgh
John Scott                    Edinburgh
Nick Sharman                  Software Sciences
Mads Tofte                    Edinburgh
Chris Wadsworth               Rutherford

Unless otherwise noted, each person is identified by his initials in the following notes. Written comments were received from David Park of Coventry.

The major topics under discussion were Dave MacQueen’s modules proposal, the I/O proposal put forth by Kevin and Robin, selection of a labelled product type (based on several proposals), and some miscellaneous syntactic issues and problem areas. Also discussed were implementation issues raised by the modules
proposal, the design of a separate compilation facility and its relation to dynamic
types, and several proposals for minor language changes.

Three subcommittees were formed:

- Revised I/O proposal: Bob Harper, Dave MacQueen, Dave Matthews, and
  Kevin Mitchell;

- Syntax: Bob Harper, Dave MacQueen, and John Scott;

- Persistent structures and separate compilation: Bob Harper, Dave Mac-
  Queen, Dave Matthews, and Kevin Mitchell.

The I/O and syntax committees will each produce a report within two weeks.
The following is a transcript of my notes translated into readable prose. Com-
ments and questions are in slant font.

2 Thursday morning, May 23: Modules

DBM: Brief overview of modules proposal. Introduced notion of a transparent
version opaque structure: manifest structure declarations are transparent in the
sense that the definitions within the structure are visible to the compiler; param-
eters to modules are opaque since the compiler is restricted to the information in
the signature.

RM: Does the inferred signature of a structure include type abbreviations? That
is, is the inferred signature of the structure struct type t=int ... end have
type t=int or just type t in it?

DM: It is not clear what to infer because there are lots of type equalities (and
other sharing) among structures.

RM: But type equalities are a natural special case.

DBM: In any case, the information is available to the compiler since manifest
structures are transparent in the above sense. Type equations need not appear
in the inferred signature of a manifest structure since the definition of the type is
available to the compiler anyway. Think of the structure as providing the witness
of the type t declaration in the signature.

RM: What is the relationship between sharing specifications in modules and type
equations? Consider type t=int vs. type t sharing t=int.

DBM: We don't really want to infer sharing constraints since they are lot all
legally expressible in signatures due to the closure rules, and in any case there is
always a great deal of incidental sharing among structures.

RM: I propose that we allow users to put type t=int into a signature but that
it not be in the inferred signature. Further, sharing constraints in signatures are
disallowed (so they are limited to module parameter lists).
DBM: But then must every structure with that signature include that type equality? Consider a signature with the type equation \( \text{type } t = \text{data} \ldots \). If every structure must contain this type equation, each occurrence of \( t \) will be a different type since unions are generative.

*I'm not sure how or if this was resolved.*

DBM: For manifest structures, all of the type equations under consideration are available to the compiler anyway, and for opaque contexts (module parameters), the sharing specifications allow you to specify the desired sharing. So it seems unnecessary to allow these equations in signatures. Notice that the compiler is, in general, aware of sharing amongst structures and components that cannot be expressed in a signature (because of the closure constraint).

RM: Summary: (1) sharing constraints may only appear in module parameter lists; (2) type equations can be viewed as a special case of sharing between structures (think of the pervasives as a standard prelude structure); (3) inferred signatures contain all internally expressible sharing constraints.

*Was this agreed on?*

DBM: Let's use the term “full signature” to refer to the present notion of signature, together with all internally expressible sharing constraints (i.e., all those that can be written without violating the closure rule). All of the information in the full signature is available to the compiler, together with non-expressible sharing information. Even within the full signature there is a lot of irrelevant sharing information, say among the types declared in logically unrelated substructures. In a sense the inferred signature is the full signature, but what prints on the user's screen will not reflect all of this information. Perhaps we should print sharing specifications to a limited substructure depth. It remains to be decided what to print to reflect the propagation of sharing information from the parameters to the result signature. For instance, in `LexOrd(0:ORDSET):ORDSET`, the compiler knows that \( S = 0.S \cdot \text{list} \) in the result.

Free references to structures that are not parameters is desirable for efficiency and ease of use. So type \( t = S.t \cdot \text{list} \) is legal in a structure that does not contain \( S \) as a substructure.

Structures are evaluated when the compiler sees them, just like any other expression in SML. To defer evaluation, use a parameterless module (analogous to an empty lambda).

Preservation of sharing amongst components of persistent structures is a problem. Perhaps only internal sharing ought to be guaranteed, leaving it to the user to package things so as to preserve required sharing.

*At this point there was a general discussion of what separate compilation means and how it relates to persistent structures.*

LC: Whatever you type ought to get executed (compiled and evaluated).
KM: Writing out structures and modules ought not be different from writing out any other value or function.

3 Thursday afternoon

DBM: Summary of his note on labelled fields.
RM: These are the issues:

- Overloading of labels;
- Significance of ordering and coercion between labelled and unlabelled tuples;
- Selector functions;
- Tying labelled tuples to constructors;
- Using special brackets or label to distinguish labelled tuple expressions;
- Using labelled tuples in constructions and patterns;
- Call by keyword;
- Visibility of labels in type expression;
- Derived form or new primitive form;
- Flexible records and subtyping;
- Puns.

RM: Tying labelled tuples to constructors eliminates the need for explicit type specifications since the constructor identifies a unique labelled tuple type. Independent labelled tuples afford greater flexibility and are more natural. *It was generally agreed that independent labelled tuples are to be adopted.*
RM: Given independent labelled tuples, we must settle the remaining issues. We do wish to support overloading of labels, and we do wish to support some abbreviation mechanism in patterns, but we do not wish to support flexible records and subtyping because the type checking problem for this case has not been solved. *It was decided that the abbreviation mechanism in Luca’s ML was satisfactory since type annotations do not often seem necessary. There was disagreement about whether and how to indicate that a given tuple pattern is or is not complete. It was suggested that there be ellipsis syntax to indicate a partial pattern.*
RM: Do we want selector functions or some selection notation, or do we just stick with patterns?
CW: Selector functions do not fit smoothly with the pattern-matching mechanisms that were introduced to eliminate destructors.

*It was agreed that patterns will suffice for the core language since selector functions are definable at will. AM raised the question of efficient compilation of selectors, but I'm not sure whether people felt that there was a problem with omitting selectors.*

RM: Do we wish to allow punning: \((1 \text{ c } 1)\) for \((1 \text{ c=} 1)\)?

LC: I introduced this notation and I no longer like it.

*It was agreed that we do not want this.*

AM: What about order of evaluation? Any sort of normalization of expressions is incompatible with any intended side effects or exceptions.

*It was agreed that labelled tuple expressions are to be evaluated left-to-right as written.*

RM: So, the order of label bindings in expressions is significant, but in patterns and type declarations it is not.

*Agreed.*

RM: As for the lexical issues, ought labels be decorated with a special mark (e.g., \#foo)? Ought we use decorated brackets?

*It was agreed that no lexical distinction for labels is needed or desirable and that Luca's decorated brackets \([1 \ldots 1]\) are acceptable.*

Break.

RM: There are several core language issues that have been raised, and some syntactic changes that I'd like to propose:

- Streamlining keywords [RM1]: banning rec, having fun;

- Minor syntax changes [RM3]: optional parentheses, semicolons, comment brackets;

- Response to David Park [RM5];

- Scope of type variables [RH's note].

RM: I propose the following changes to the syntax in order to eliminate rec. It is especially repugnant to have three keywords for a recursive type binding: type rec t = data ....

<table>
<thead>
<tr>
<th>Current syntax</th>
<th>Proposed syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>val rec</td>
<td>fun</td>
</tr>
<tr>
<td>fun</td>
<td>fn</td>
</tr>
<tr>
<td>type rec</td>
<td>data</td>
</tr>
<tr>
<td>abstype t = data ...</td>
<td>absdata t = ...</td>
</tr>
<tr>
<td>abstype t = ...</td>
<td>unchanged</td>
</tr>
</tbody>
</table>
General discontent and disagreement.
RB: Leave val and val rec, but provide fun as sugar for val rec.
I think that he meant to accept Robin’s changes to type and abtype syntax.
LC: Why not reserve val for values of non-functional type (according to the type checker) and use fun and fun rec for values of functional type.
AM: But then the syntax depends on type checking.
At this point the discussion became unproductive.
RM: David Park has offered an example of three syntactically very similar declarations that are interpreted quite differently by the compiler.
See David Park’s note and Robin’s response. There was general discontent over Robin’s proposal, and it was agreed not to make a decision on any of Robin’s suggestions. A syntax committee (DBM,RH,JS) was formed to consider the issues and make a proposal.
RH: There is a problem in the interpretation of explicit type variables. The issues are the scope of a type variable (where does it become generic?) and the meaning of a type annotation (should type variables in an annotation be instantiable?). RH presented his note on this issue. It was agreed that the proposed interpretation was the right one. Changes to the type checker are required.

4 Friday morning, May 24

DBM entertained questions on the modules proposal.
KM: There is a problem about the order of declarations in structures (see his note)
If redeclaration of identifiers is allowed in a structure, then DBM’s remark that order is insignificant is not correct.
DBM: I think that an identifier ought to be required to be mentioned before it is used [so order has some significance?], and that no redeclarations ought to be admitted in a structure. In fact, no redeclaration ought to be admitted anywhere, but it should be a “soft” error to do so because of the top level.
Generally agreed.
DBM: If a type identifier is bound to a data type, and that type identifier is subsequently rebound, then the constructors for the data type ought to be hidden as well.
Generally agreed, though there are implementation difficulties and no compiler in fact hides the constructors.
DBM: The restriction of structure and signature declarations to the top level is to avoid having to deal with higher-order modules and structures with embedded signature declarations.
Agreed.
DBM: There has been some discontent with my nomenclature. The issues seem to be the terminology used in the paper and the keywords used in the language. The main proposals are:

- Eliminate structure and struct in favor of a single keyword, using the presence of a parameter list to distinguish the two cases.
- Replacing the keywords structure and module by some others such as package, cluster, algebra, generic ..., or parametric ....
- Replacing signature with interface.

RB: I object to the parametric and generic terminology on the grounds that it is too syntactic and incompatible with ordinary function definition terminology. It was agreed, after much argument, that we would stick to the current terminology.

RH presented a simplification of Kevin and Robin's I/O proposal. The essential change was to eliminate the notion of a channel and to add primitives for closing a channel. It was agreed that his model was acceptable for the basic I/O component of SML. The details were delegated to a committee (RH,DBM,KM,DCM).

5 Friday afternoon, May 24

Discussion on separate compilation and persistent structures. The discussion focussed on the preservation of sharing across persistent structures.

LC: The import primitive (to load a persistent structure from storage) ought to reload all dependent structures so that there is not problem about preservation of sharing (this is the method used in his ML and in Amber).

RM: Sharing ought to be preserved within a structure but not across structures.

RH: Can this lead to problems? If reloading a structure causes generative type bindings to be evaluated, then type equations that were true before the structures were written out will no longer hold after they have been reloaded.

DBM: There is not problem with that. You end up with values that cannot be denoted by any expression in the reloaded context because the constructors have been hidden.

It was generally agreed that separate compilation consists of writing out structures. Libraries are built using the substructure mechanism. There is an issue of what to name the entity written out and how to relate this name to the name of the structure. A subcommittee (DBM,DCM,KM,RH) was formed to come up with a proposal.

LP or DCM (?): It seems necessary to make a precise specification of the standard library in order to avoid incompatibilities across implementations.
Generally agreed, though no one wanted to actually write or document such thing. It was not clear what to do about extensions. The remainder of the afternoon was a free-for-all.

AM: I have a means of handling overloading cleanly. As it stands there are at least 3 overloaded identifiers (+, =, and >) which are dealt with on an ad-hoc basis in the compiler. It seems useful to allow users to declare that an identifier is overloaded, and there is a need to decide whether or not functions in the standard library (whatever it may turn out to be) can overload any names.

The proposal is to introduce a declaration of the form overloaded foo:t where t is a type expression, perhaps optional. The idea is that all types for foo must match t (e.g., + matches 'a*'a->'a). This type expression is taken to be the type of foo everywhere that it is used. Overloading can be resolved in two passes, thereby avoiding exponential behavior as in HOPE. [Did I get this right?]

DCM: Equality is a very special case, quite unlike +. I propose that we do away with all overloaded identifiers except for equality. I missed an example that he offered.

RB: I propose that there be a prescribed set of identifiers that the system and the user may overload, and no others.

It was put to a vote. AM’s proposal received 4 votes, DCM’s 4, status quo 6, RB’s 3. There were 7 votes in favor of allowing the set of overloaded identifiers to grow without allowing user’s to further overload those.

KM: I have a simple way to implement eval. The idea is analagous to printing. Eval takes a string argument, calls the type checker to obtain its type (failing if type checking fails), ensures that the inferred type matches the type required by the context of the occurrence of eval, and calls the compiler to compile and execute the expression if this succeeds. For instance, 1+eval("2+3") has value 6:int since the type of the expression 2+3 is int, as required by +. Free identifiers can be resolved at top level.

DBM: I would prefer to work with abstract syntax rather than strings.

KM: Yes, but we’re talking about a quick and dirty experiment.

It was agreed that this was worth playing with, but ought not be part of the standard.

CW: What about polymorphic ref’s. They have been removed from the standard. Why not use the old ML scheme?

RM: Because we now have free-standing ref’s.

DBM: I have been looking at this problem and still have not come up with a satisfactory solution. Luis Damas’s algorithm is too restrictive to use.

It was agreed to leave monotype ref’s in the standard.

Someone raised the question of introducing Cardelli–style unions. LC was against and no one was for, so it was dropped.
LC: In Amber I eliminate the infix directive by adopting the convention that all alphanumeric identifiers are prefix and all others are infix. This means that or else and and then must be replaced by symbolic identifiers, and and ! must be replaced by alphanumeric identifiers.

_The simplicity of the proposal was appreciated, but no one wanted to adopt it._

RB: The precedence levels cause problems for a YACC–like parser. I propose that we define a small, fixed set of precedences, each tied to a built-in operator with that precedence, and then change the syntax to infix ++ like +, for instance. _I think that this may have been adopted._

KM: I think that the semicolon between declarations ought to be mandatory since it is mandatory at the top level. Leaving it out leads to problems with error messages coming out too late.

_No one liked this and it was suggested that one can adopt the convention for oneself._

JS: We can use fun to resolve the problem that David Park raised: its use implies a clausal function definition, whereas val implies a pattern. _This proposal was adopted._

JS: There are serious problems associated with deleting the constructors associated with a hidden (rebound) data type. Consider

```latex
    \text{type } t = \text{data } a | b | c \\
    \text{local} \\
    \text{type } t = \text{data } u | v \\
    \text{in} \\
    \text{val } a \ x = x \ \text{end;}
```

_No clear solution, so the problem is tabled._

LC: I propose that local type declarations be eliminated (so that type declarations are allowed only at top level and in structures).

_There was no consensus, though everyone admitted that they never use the feature._ RB thought that local type declarations might be useful in structures to avoid “clipping” by signatures. LC noted that a side benefit of removing local type declarations is that val can be eliminated inside let and local.

RM proposed relaxing the restriction on the right–hand side of recursive val bindings as per his note. _No time to discuss this._

6 Saturday morning, May 25

_This session was devoted to summarizing what is to be done._

- Documentation of the language definition.
- Ammended core language proposal [RM].
- I/O proposal [I/O subcommittee].
- Modules proposal [DBM].

• Formal semantics.
  - Operational semantics of the core language [RM].
  - Denotational semantics of the modules proposal [DTS].
  - Mads Tofte may have a semantics for ML.

• User documentation.
  - Reference manual. Organization?
  - Book.
  - Tutorial.

• Course notes [LP].

• Libraries (lists, arrays, strings, ...).