# Defects in the Revised Definition of Standard ML

Andreas Rossberg
Universität des Saarlandes
rossberg@ps.uni-sb.de

Updated 2007/01/22

## 1 Introduction

This document[1] is intended to be a comprehensive list of all known bugs, ambiguities, problems and other 'grey areas' in the revised Definition of Standard ML [MTHM97]. For better overview of what really is important the issues are classified into several categories:

- *major*: mistakes that compromise soundness without an obvious fix, that create problems for implementers, or lead to annoying incompatibles among implementations,
- *minor*: mistakes that leave open questions as well but do not seem to produce problems in practice,
- *pedantic*: nitpicking on issues that are not "quite right" but everybody knows how to interpret them,
- *typos*: obvious slips in formal bits.

The list marks with * all issues that have been newly introduced in SML'97. Others are inherited from SML'90 [MTH90].

Note that this compilation does not try to be a general critique of the language. It does not discuss design decisions or weaknesses of particular features, but merely lists problems with the specification of the language.

## 2 Issues in Section 2 (Syntax of the Core)

Section 2.5 (Lexical analysis):

- $[minor]$ In Section 2.2 the Definition includes only space, tab, newline, and formfeed into the set of obligatory formatting characters, that are allowed in source code. However, some major platforms require use of the carriage return character in text files. In order to achieve portability of sources across platforms it should be included as well. Preferably, for consistency, all formatting characters should be included, for which there is explicit escape syntax.

[1]This document has been derived from Appendix A of the documentation to HaMLet, www.ps.uni-sb.de/hamlet

Section 2.6 (Infixed Operators):

- $[minor]$ The Definition says that “the only required use of `op` is in prefixing a non-infixed occurrence of an identifier which has infix status”. This is rather vague, since it is not clear whether occurrences in constructor and exception bindings count as non-infixed [K93].

Section 2.8 (Grammar), Figure 4 (Expressions, Matches, Declarations and Bindings):

- $[pedantic]$ The syntax rules for $dec$ are highly ambiguous. The productions for empty declarations and sequencing allow the derivation of arbitrary sequences of empty declarations for any input.
- $[pedantic]$ Another ambiguity is that a sequence of the form $dec_1\ dec_2\ dec_3$ can be reduced in two ways to $dec$: either via $dec_{12}\ dec_3$ or via $dec_1\ dec_{23}$ [K93]. See also section 3.

Section 2.9 (Syntactic Restrictions):

- $[pedantic]$ * The restriction that $valbind$s may not bind the same identifier twice (2nd bullet) is not a syntactic restriction as it depends on the identifier status of the $vid$s in the patterns of a $valbind$. Identifier status is derived by the elaboration rules.

  Ideally, all restrictions should be handled by appropriate side conditions in the rules of the static semantics instead.
- $[minor]$ * An important syntactic restriction is missing:

  > “Any $tyvar$ occurring on the right side of a $typbind$ or $datbind$ of the form $tyvarseq\ tycon = \cdots$ must occur in $tyvarseq$.”

  This restriction is analogous to the one given for $tyvar$s in type specifications (section 3.5, item 4). Without it the type system would be unsound. [2]

## 3 Issues in Section 3 (Syntax of Modules)

Section 3.4 (Grammar for Modules), Figure 6 (Structure and Signature Expressions):

- $[pedantic]$ The syntax rules for $strdec$ contain the same ambiguities with respect to sequencing and empty declarations as those for $dec$ (see section 2).
- $[minor]$ Moreover, there are two different ways to reduce a sequence $dec_1\ dec_2$ of core declarations into a $strdec$: via $strdec_1\ strdec_2$ and via $dec$ [K93]. Both parses are not equivalent since they provide different contexts for overloading resolution (Appendix E). For example, appearing on structure level, the two declarations

```
fun f x = x + x
val a = f 1.0
```

[2] Interestingly enough, in the SML’90 Definition the restriction was present, but the corresponding one for specifications was missing [MT91].

may be valid if parsed as $dec$, but do not type check if parsed as $strdec_1\ \ strdec_2$ because overloading of `+` gets defaulted to `int`. The problem does not seem to arise in practice, though, because most implementations use smaller contexts for overloading resolution (see section 12).

- $[minor]$ Similarly, it is possible to parse a structure-level `local` declaration containing only core declarations in two ways: as a $dec$ or as a $strdec$ [K93]. This produces the same semantic ambiguity.

Section 3.4 (Grammar for Modules), Figure 7 (Specifications):

- $[pedantic]$ Similar as for $dec$ and $strdec$, there exist ambiguities in parsing empty and sequenced $spec$s.
- $[minor]$ The ambiguity extends to sharing specifications. Consider:

  ```
  type t
  type u
  sharing type t = u
  ```

  This snippet can be parsed in at least three ways, with the sharing constraint taking scope over either both, or only one, or neither type specification. Since only the first alternative can be elaborated successfully, the validity of the program depends on how the ambiguity is resolved.

  The proper and most permissive disambiguation rule is to make sequential specifications and sharing specifications both left associative at the same precedence level. It could be expressed as a syntactic restriction stating that "The $spec_2$ in a sequential specification may not contain a sharing specification."

Section 3.4 (Grammar for Modules), Figure 8 (Functors and Top-level Declarations):

- $[minor]$ * Finally, another ambiguity exists for reducing a sequence $strdec_1\ \ strdec_2$ to a $topdec$: it can be done either by first reducing to $strdec$, or to $strdec_1\ \ topdec_2$. The latter is more restrictive with respect to free type variables (but see section 13 with regard to this).

Altogether, ignoring the infinite number of derivations involving empty declarations, the grammar in the Definition allows three ambiguous ways to reduce a sequence of two $dec$s to a $topdec$, as shown by the following diagram. All imply different semantics. A further ambiguity arises at the program level (see section 8).

$$\begin{tikzcd}
 & dec_1\ dec_2 \arrow[dl, dash] \arrow[dr, dash] & \\
dec \arrow[d, dash] & & strdec_1\ strdec_2 \arrow[d, dash] \arrow[dll, dash] \\
strdec \arrow[dr, dash] & & strdec_1\ topdec_2 \arrow[dl, dash] \\
 & topdec &
\end{tikzcd}$$

## 4 Issues in Section 4 (Static Semantics for the Core)

Section 4.8 (Non-expansive Expressions):

- $[minor]$ * The definition of non-expansiveness is purely syntactic and does only consider the right hand side of a binding. However, an exception may result from matching against a non-exhaustive pattern on the left hand side. It is rather inconsistent to disallow `raise` expressions in non-expansive bindings but allow implicit exceptions in the disguise of pattern match failure. More seriously, the possibility of exceptions stemming from polymorphic bindings is incompatible with type passing implementations.

Section 4.9 (Type Structures and Type Environments):

- $[pedantic]$ The definition of the Abs operator demands introduction of "new distinct" type names. However, type names can only be new relative to a context. To be precise, Abs would thus need an additional argument $C$ [K96].

- $[minor]$ Values in `abstype` declarations that are potentially polymorphic but require equality types have no principal type [K96]. For example, in the declaration

  ```
  abstype t = T with
      fun eq(x,y) = x = y
  end
  ```

  the principal type of `eq` *inside* the scope of `abstype` clearly is `''a * ''a -> bool`. However, outside the scope this type is not principal because `''a` cannot be instantiated by `t`. Neither would `t * t -> bool` be principal, of course. Although not strictly a bug (there is nothing which enforces the presence of principal typings in the revised Definition), this semantics is very hard to implement faithfully, since type inference had to deal with unresolved type schemes and to cascadingly defer decisions about instantiation and generalisation until the correct choice is determined. Most if not all SML implementations assign `eq` the type `''a * ''a -> bool`.

- $[minor]$ A related problem is the fact that the rules for `abstype` may infer type structures that do not respect equality [K96]:

  ```
  abstype t = T with
      datatype u = U of t
  end
  ```

  Outside the scope of this `abstype` declaration type `u` will still be an equality type. Values of type `t` can thus be compared through the backdoor:

  ```
  fun eqT(x,y) = U x = U y
  ```

Section 4.10 (Inference Rules):

- $[minor]$ * The comment to rule 26 states that a declaration like

  ```
  datatype t = T
  val rec T = fn x => x
  ```

is legal since $C + VE$ overwrites identifier status. However, this comment omits an important point: in the corresponding rule 126 of the dynamic semantics recursion is handled differently so that the identifier status is *not* overwritten. Consequently, the second declaration will raise a `Bind` exception. It arguably is a serious ill-design to infer inconsistent identifier status in the static and dynamic semantics, but fortunately it does not violate soundness in this case. Most implementations do not implement the ‘correct’ dynamic semantics, though.

- $[typo]$ There is an unmatched left parenthesis in the consequent of rule 28.

Section 4.11 (Further Restrictions):

- $[minor]$ Under item 1 the Definition states that “the program context” must determine the exact type of flexible records, but it does not specify any bounds on the size of this context. Unlimited context is clearly infeasible since it is incompatible with separate compilation and with `let` polymorphism: at the point of generalisation the structure of a type must be determined precisely enough to know what we have to quantify over.[3] The context should thus be restricted to at least the innermost value declaration surrounding the flexible record pattern.

- $[minor]$ Under item 2 the Definition demands that a compiler must give warnings whenever a pattern is redundant or a match is non-exhaustive. However, this requirement is inconsistent for two reasons:

  1. * There is no requirement for consistency of datatype constructors in sharing specifications or type realisations. For example,

     ```
     datatype t = A | B
     datatype u = C
     sharing type t = u
     ```

     is a legal specification. Likewise,

     ```
     sig datatype t = A | B end where type t = bool
     ```

     is valid. Actually, this may be considered a serious bug on its own, although the Definition argues that inconsistent signatures are “not very significant in practice” (section G.9). If such an inconsistent signature is used to specify a functor argument it allows a mix of constructors to appear in matches in the functor’s body, rendering the terms of irredundancy and exhaustiveness completely meaningless.
  2. It is difficult in general to check equality of exception constructors – they may or may not be aliased. Inside a functor, constructor equality might depend on the actual argument structure the functor is applied to. It is possible to check all this by performing abstract interpretation (such that redundant matches are detected at functor application), but this is clearly infeasible weighed against the benefits, in particular in conjunction with separate compilation. The Definition should point out that it only requires considering syntactic equivalence in the case of exception constructors.

## 5 Issues in Section 5 (Static Semantics for Modules)

Section 5.7 (Inference Rules):

[3] Polymorphic approaches to record typing are clearly not supported by the Definition.

- $[pedantic]$ * The rules 64 and 78 use the notation $\{t_1 \mapsto \theta_1, \cdots, t_n \mapsto \theta_n\}$ to specify realisations. However, this notation is not defined anywhere in the Definition for infinite maps like realisations – section 4.2 only introduces it for finite maps.

- $[minor]$ * More seriously, both rules lack side conditions to ensure consistent arities for domain and range of the constructed realisation. Because $\varphi$ can hence fail to be well-formed (section 5.2), the application $\varphi(E)$ is not well-defined. The necessary side conditions are:
$$t \in \text{TyName}^{(k)} \tag{64}$$
$$t_i \in \text{TyName}^{(k)}, i = 1..n \tag{78}$$

- $[minor]$ * The presence of functors provides a form of explicit polymorphism which interferes with principal typing in the core language. Consider the following example [DB07]:

```
functor F(type t) =
    struct val id = (fn x => x) (fn x => x) end
structure A = F(type t = int)
structure B = F(type t = bool)
val a = A.id 3
val b = B.id true
```

  The declaration of `id` cannot be polymorphic, due to the value restriction. Nevertheless, assigning it type `t -> t` would make the program valid. However, finding this type would require the type inference algorithm to skolemize all undetermined types in a functor body's result signature over the types appearing in its argument signature, and then perform a form of higher-order unification. Consequently, almost all existing implementations reject the program.[4]

- $[minor]$ * The side conditions on free type variables in rules 87 and 89 do not have the effect that obviously was intended, see section 13.

## 6 Issues in Section 6 (Dynamic Semantics for the Core)

Section 6.4 (Basic Values):

- $[pedantic]$ The APPLY function has no access to program state. This suggests that library primitives may not be stateful, implying that a lot of interesting primitives could not be added to the language without extending the Definition itself [K93].

  On the other hand, any non-trivial library type (e.g. arrays or I/O streams) requires extension of the definition of values or state anyway (and equality types – consider `array`). The Definition should probably contain a comment in this regard.

[4]Interestingly, MLton accepts the program, thanks to its defunctorization approach. However, it likewise accepts similar programs that are *not* valid Standard ML, e.g.:

```
functor F() = struct val id = (fn x => x) (fn x => x) end
structure A = F()
structure B = F()
val a = A.id 3
val b = B.id true
```

## 7 Issues in Section 7 (Dynamic Semantics for Modules)

Section 7.2 (Compound Objects):

- $[typo]$ * In the definition of the operator ↓: Env×Int → Env, the triple "$(SI, TE, VI)$" should read "$(SI, TI, VI)$".

Section 7.3 (Inference Rules):

- $[typo]$ * Rule 182 contains a typo: both occurrences of $IB$ have to be replaced by $B$. The rule should actually read:

$$\frac{\text{Inter}B \vdash \mathit{sigexp} \Rightarrow I \qquad \langle B \vdash \mathit{funbind} \Rightarrow F\rangle}{\begin{array}{r}B \vdash \mathit{funid}\ (\ \mathit{strid} : \mathit{sigexp}\ ) = \mathit{strexp}\ \langle \mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \\ \{\mathit{funid} \mapsto (\mathit{strid} : I, \mathit{strexp}, B)\}\langle + F\rangle\end{array}} \tag{182}$$

- $[typo]$ * The rules for toplevel declarations are wrong: in the conclusions, the result right of the arrow must be $B'\langle +B''\rangle$ instead of $B'\langle'\rangle$ in all three rules:

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \qquad B' = E \text{ in Basis} \qquad \langle B + B' \vdash \mathit{topdec} \Rightarrow B''\rangle}{B \vdash \mathit{strdec}\ \langle \mathit{topdec}\rangle \Rightarrow B'\langle +B''\rangle} \tag{184}$$

$$\frac{\text{Inter}B \vdash \mathit{sigdec} \Rightarrow G \qquad B' = G \text{ in Basis} \qquad \langle B + B' \vdash \mathit{topdec} \Rightarrow B''\rangle}{B \vdash \mathit{sigdec}\ \langle \mathit{topdec}\rangle \Rightarrow B'\langle +B''\rangle} \tag{185}$$

$$\frac{B \vdash \mathit{fundec} \Rightarrow F \qquad B' = F \text{ in Basis} \qquad \langle B + B' \vdash \mathit{topdec} \Rightarrow B''\rangle}{B \vdash \mathit{fundec}\ \langle \mathit{topdec}\rangle \Rightarrow B'\langle +B''\rangle} \tag{186}$$

## 8 Issues in Section 8 (Programs)

- $[minor]$ The comment to rule 187 states that a failing elaboration has no effect. However, it is not clear what infix status is in scope after a failing elaboration of a program that contains top-level infix directives.

- $[minor]$ * There is another syntactic ambiguity for programs. A note in section 3.4, Figure 8 restricts the parsing of $topdec$s:

  > "No $topdec$ may contain, as an initial segment, a $strdec$ followed by a semicolon."

  The intention obviously is to make parsing of toplevel semicolons unambiguous so that they always terminate a program. As a consequence of the parsing ambiguities for declaration sequences (see section 3) the rule is not sufficient, however: a sequence $dec_1;\ dec_2;$ of core level declarations with a terminating semicolon can be first reduced to $dec$;, then to $strdec$;, and finally $program$. This derivation does not exhibit an "initial $strdec$ followed by a semicolon." Consequently, this is a valid parse, which results in quite different behaviour with respect to program execution.

- $[pedantic]$ The negative premise in rule 187 has unfortunate implications: interpreted strictly it precludes any conforming implementation from providing any sort of conservative semantic extension to the language. Any extension that allows declarations to elaborate that would be illegal according to the Definition (e.g. consider polymorphic records) can be observed through this rule and change the behaviour of consecutive declarations. Consider for example:

  ```
  val s = "no";
  strdec
  val s = "yes";
  print s;
  ```

  where the *strdec* only elaborates if some extension is supported. In that case the program will print `yes`, otherwise `no`.

  This probably indicates that formalising an interactive toplevel is not worth the trouble.

## 9 Issues in Appendix A (Derived Forms)

Text:

- $[pedantic]$ The paragraph explaining rewriting of the *fvalbind* form rules out mixtures of *fvalbind*s and ordinary *valbind*s. However, the way it is formulated it does not rule out all combinations. It should rather say that all value bindings of the form $pat = exp$ `and` *fvalbind* or `rec` *fvalbind* are disallowed.

Figure 15 (Derived forms of Expressions):

- $[pedantic]$ The Definition is somewhat inaccurate about several of the derived forms of expressions and patterns. It does not make a proper distinction between atomic and non-atomic phrases. Some of the equivalent forms are not in the same syntactic class [MT91, K93].

Figure 17 (Derived forms of Function-value Bindings and Declarations):

- $[minor]$ The syntax of *fvalbind*s as given in the Definition enforces that all type annotations are syntactically equal, if given. This is unnecessarily restrictive and almost impossible to implement [K93]. The obvious solution is the more permissive syntax:

$$\begin{array}{lllll} & \langle\texttt{op}\rangle vid\ atpat_{11} & \cdots\ atpat_{1n} & \langle :ty_1\rangle & = exp_1 \\ | & \langle\texttt{op}\rangle vid\ atpat_{21} & \cdots\ atpat_{2n} & \langle :ty_2\rangle & = exp_2 \\ | & \qquad \cdots & \cdots & & \\ | & \langle\texttt{op}\rangle vid\ atpat_{m1} & \cdots\ atpat_{mn} & \langle :ty_m\rangle & = exp_m \\ & & \langle\texttt{and}\ fvalbind\rangle & & \end{array}$$

  This probably was the original intention of the authors anyway.

Figure 19 (Derived forms of Specifications and Signature Expressions):

- $[minor]$ * The derived form that allows several definitional type specifications to be connected via `and` is defined in a way that makes its scoping rules inconsistent with all other occurences of `and` in the language. In the example

```
type t = int
signature S =
sig
    type t = bool
    and  u = t
end
```

type `u` will be equal to `bool`, not `int` like in equivalent declarations. It would have been more consistent to rewrite the derived form to

```
include
  sig type tyvarseq_1 tycon_1
       and ···
       ···
       and tyvarseq_n tycon_n
  end where type tyvarseq_1 tycon_1 = ty_1
      ···
      where type tyvarseq_n tycon_n = ty_n
```

and delete the separate derived form for single definitional specifications.

- $[pedantic]$ * The Definition defines the phrase

  *spec* `sharing` *longstrid*$_1$ `=` $\cdots$ `=` *longstrid*$_n$

  as a derived form. However, this form technically is not a derived form, since it cannot be rewritten in a purely syntactic manner – its expansion depends on the static environment.

- $[major]$ * The derived form for type realisations connected by `and` is not only completely redundant and alien to the rest of the language (`and` is nowhere else followed by a second reserved word), it also is tedious to parse, since this part of the grammar is LALR(2) as it stands. It can be turned into LALR(1) only by a bunch of really ugly transformations. Consequently, almost no SML system seems to be implementing it correctly. Even worse, several systems implement it in a way that leads to rejection of programs *not* using the form.

## 10 Issues in Appendix B (Full Grammar)

Text:

- $[pedantic]$ The first sentence is not true since there is a derived form for programs (Figure 18). Moreover, it is not obvious why the appendix refrains from also providing a full version of the module and program grammar. It contains quite a lot of derived forms as well, and the section title leads the reader to expect it.

- $[minor]$ The Definition gives precedence rules for disambiguating expressions, stating that "the use of precedence does not increase the class of admissible phrases". However, the rules are not sufficient to disambiguate all possible phrases. Moreover, for some phrases they actually rule out *any* possible parse, e.g.

  ```
  a andalso if b then c else d orelse e
  ```

  has no valid parse according to these rules. So the above statement is rather inconsistent [K93]. The common way to deal with this probably is to just use Yacc precedence declarations for expression keywords that correspond to the precedence hierarchy given in the Definition. This seems to be the best way to approximate the intention of the Definition's rules.

- $[\mathit{major}]$ There is no comment on how to deal with the most annoying problem in the full grammar, the infinite look-ahead required to parse combinations of function clauses and `case` expressions, like in:

  ```
  fun f x = case e1 of z => e2
    | f y = e3
  ```

  According to the grammar this ought to be legal. However, parsing this would either require horrendous grammar transformations, backtracking, or some nasty and expensive lexer hack [K93]. Consequently, there is no SML implementation being able to parse the above fragment. To legalise the behaviour of implementations, an informal restriction of the form

  > The expressions $\mathit{exp} = 1, \ldots, \mathit{exp}_{m-1}$ in a $\mathit{fvalbind}$ may not terminate with a $\mathit{match}$.

  could be added.

Figure 21 (Grammar: Declarations and Bindings):

- $[\mathit{minor}]$ The syntax given for $\mathit{fvalbind}$ is incomplete as pointed out by the corresponding note. This is not really a bug but sloppy enough to cause some divergence among implementations.

Figure 22 (Grammar: Patterns):

- $[\mathit{minor}]$ While there are additional non-terminals $\mathit{infexp}$ and $\mathit{appexp}$ to disambiguate parsing of infix expressions, there is no such disambiguation for patterns. This implies that a pattern like `x:t ++ y` can be parsed if `++` is an appropriate infix constructor [K96]. Of course, this would result in heavy grammar conflicts.

  This appears to be an oversight. The full grammar obviously implemented by all SML systems is something like:

$$\begin{array}{lll}
\mathit{atpat} & ::= & \text{...like before...} \\
\mathit{apppat} & ::= & \mathit{atpat} \\
 & & \langle \texttt{op} \rangle \mathit{longvid}\ \mathit{atpat} \\
\mathit{infpat} & ::= & \mathit{apppat} \\
 & & \mathit{infpat}_1\ \mathit{vid}\ \mathit{infpat}_2 \\
\mathit{pat} & ::= & \mathit{infpat} \\
 & & \mathit{pat} : \mathit{ty} \\
 & & \langle \texttt{op} \rangle \mathit{vid}\ \langle : \mathit{ty} \rangle\ \texttt{as}\ \mathit{pat}
\end{array}$$

## 11 Issues in Appendix D (The Initial Dynamic Basis)

- $[\mathit{minor}]$ The Definition does specify the minimal initial basis but it does not specify what the initial state has to contain. Of course, it should at least contain the exception names `Match` and `Bind`. The obvious definition thus is:

$$s_0 = (\{\}, \{\texttt{Match}, \texttt{Bind}\})$$

- $[pedantic]$ The Definition does nowhere demand that the basis a library provides has to be consistent in any way. Nor does it require consistency between intial basis and initial state.

## 12 Issues in Appendix E (Overloading)

- $[major]$ Overloading is the most hand-waving part of the otherwise pleasantly accurate Definition. Due to the lack of formalism and specific rules, overloading resolution does not work consistently among SML systems. For example, type-checking of the following declaration does not succeed on all systems:

  ```
  fun f(x,y) = (x + y)/y
  ```

- $[minor]$ The Definition defines the overloading mechanism by enumerating all overloaded entities the library provides. This is rather unfortunate. It would be desirable if the rules would be a bit more generic, avoiding hardcoding overloading classes and the set of overloaded library identifiers on one hand, and allowing libraries to extend it in systematic ways on the other. More generic rules could also serve as a better guidance for implementing overloading.

  A more generic description of overloading classes might be to formalise them as a pair of a type name set and the type name being the designated default:

  $$(T,t) \in \text{OverloadingClass} = \text{Fin}(\text{TyName}^{(0)}) \times \text{TyName}^{(0)}$$

  An overloading class is *well-formed* iff the following properties hold:

  $$t \in T \tag{1}$$

  $$\text{Eq}(T) = \emptyset \quad \vee \quad t \text{ admits equality} \tag{2}$$

  where $\text{Eq}(T) = \{t \in T \mid t \text{ admits equality}\}$. A set $\{(T_1,t_1),\cdots,(T_n,t_n)\}$ of overloading classes is *consistent* iff

  $$\text{for all } i \in \{1,..,n\}, \quad (T_i,t_i) \text{ well-formed} \tag{3}$$

  $$\text{for all } i,j \in \{1,..,n\}, \quad T_i \cap T_j = \emptyset \quad \vee \quad |\{t_i,t_j\} \cap T_i \cap T_j| = 1 \tag{4}$$

  The set of all overloading classes used in an initial basis must be consistent. A library could provide arbitrary overloading classes, as long as they adhere to these restrictions. The restrictions guarantee that intersection of overloading classes is reflexive, associative and commutative with respect to the default and that there always is a unique default. We claim that this is necessary to make defaulting unambiguous and enable a feasible type inference algorithm. Note that these properties hold for the minimal initial basis given in section E.1, although the Definition forgets to demand that any extension of the basic overloading classes must be consistent with respect to equality (well-formedness property (2)).

- $[minor]$ * That the Definition specifies an *upper* bound on the context a compiler may consider to resolve overloading is quite odd – of course, implementations cannot be prohibited to conservatively extend the language by making more programs elaborate. On the other hand, much more important would have been to specify a *lower* bound on what implementations *have to* support – it is clearly not feasible to force the programmer to annotate every individual occurence of an overloaded identifier or special constant.

  A natural and sensible lower bound seems to be the smallest enclosing core declaration the overloaded identifier or constant appears in, consistent with the treatment of flexible records (see section 4). Preferably, this would also be the upper bound as far as the standard is concerned, in order to achieve consistent behaviour among implementations.

Figure 27 (Overloaded Identifiers):

- $[typo]$ * The types for the comparison operators `<`, `>`, `<=`, and `>=` must correctly be $\texttt{numtxt} \times \texttt{numtxt} \rightarrow \texttt{bool}$.

## 13 Issues in Appendix G (What's New?)

Section G.8 (Principal Environments):

$[minor]$ * At the end of the section the authors explain that the intent of the restrictions on free type variables at the toplevel (side-conditions in rules 87 and 89) is to avoid reporting free type variables to the user. However, judging from the rest of the paragraph, this reasoning confuses two notions of type variable: type variables as semantic objects, as appearing in the formal rules of the Definition, and the yet undetermined types during Hindley/Milner type inference, which are also represented by type variables. However, both kinds are variables on completely different levels: the former are part of the formal framework of the Definition, while the latter are an 'implementation aspect' that lies outside the scope of the Definition's formalism. Let us distinguish both by referring to the former as *semantic type variables* and to the latter as *undetermined types*.

The primary purpose of the aforementioned restrictions obviously is to avoid reporting *undetermined types* to the user. However, they fail to achieve that. In fact, it is impossible to enforce such behaviour within the formal framework of the Definition, since it essentially would require formalising type inference (the current formalism has no notion of undetermined type). Consequently, the comment in section G.8 about the possibility of relaxing the restrictions by substituting arbitrary monotypes misses the point as well.

In fact, the formal rules of the Definition actually imply the exact opposite, namely that an implementation may *never* reject a program that results in undetermined types at the toplevel, and is thus compelled to report them. The reason is explicitly given in the same section: "implementations should not reject programs for which successful elaboration is possible". Consider the following program:

```
val r = ref nil;
r := [true];
```

Rule 2 has to non-deterministically choose some type $\tau$ `list` for the occurrence of `nil`. The choice of $\tau$ is not determined by the declaration itself: it is not used, nor can it be generalised, due to the value restriction. However, `bool` is a perfectly valid choice for $\tau$, and this choice will allow the entire program to elaborate. So according to the quote above, an implementation has to make exactly that choice. Now, if both declarations are entered separately into an interactive toplevel the implementation obviously has to defer commitment to that choice until it has actually seen the second declaration. Consequently, it can do nothing else but reporting an undetermined type for the first declaration. The only effect the side conditions in rules 87 and 89 have on this is that the types committed to later may not contain free semantic type variables – but considering the way such variables are introduced during type inference (mainly by generalisation), the only possibility for this is through a toplevel exception declaration containing a type variable.[5]

[5]Note that this observation gives rise to the question whether the claim about the existence of principal envi-

There are two possibilities of dealing with this matter: (1) take the formal rules as they are and ignore the comment in the appendix, or (2) view the comment as an informal "further restriction" and fix its actual formulation to match the obvious intent. Since the comments in Appendix G are not supposed to be a normative part of the Definition but merely explanatory, and moreover are somewhat inconsistent, strict reading should give the formal rules priority and choose option (1). Unfortunately, this interpretation is incompatible with implementation strategies relying on type passing, where all types must be determined prior to execution.

## Acknowledgements

Thanks go to the following people who helped with or without knowing in compiling this list: Stefan Kahrs, Claudio Russo, Matthias Blume, Stephen Weeks, John Reppy, and people on the sml-implementers list that recently came to life. And of course thanks go to the designers of ML and authors of the Definition for the magnificent programming language – it seems to be the only serious language in existence, for which even a pedantic list of all defects fits on a couple of pages. `;-)`

ronments in section 4.12 of the SML'90 Definition [MTH90] was valid in the first place. It most likely was not: a declaration like the one of `r` has no principal environment that would be expressible within the formalism of the Definition, despite allowing different choices of free imperative type variables. The reasoning that this relaxation was sufficient to regain principality is based on the same mix-up of semantic type variables and undetermined types as above. The relaxation does not solve the problem with expansive declarations, since semantic type variables are rather unrelated to it – choosing a semantic type variable for an undetermined type is no more principal than choosing any particular monotype.

## A History

- 2001/10/11: Added minor issue: carriage return is not included as supported control character in source code (Section 2.5). Some small clarifications.
- 2004/04/13: Reconsiderd and removed scoping subtleties for type names as an issue (Section 4.10).
- 2004/06/22: Added minor issue: parsing ambiguities with sequential specifications make scoping of sharing constraints ambiguous (Section 3.4).
- 2005/01/13: Added minor issue: missing side conditions ensuring consistent arities in rules 64 and 78 (Section 5.7).
- 2005/01/26: Added typo in rule 28.
- 2006/07/18: Added principality issue with functors.
- 2007/01/22: Added typo in definition of $\downarrow$ operator (Section 7.2).

## References

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, David MacQueen
*The Definition of Standard ML* (Revised)
The MIT Press, 1997

[MTH90] Robin Milner, Mads Tofte, Robert Harper
*The Definition of Standard ML*
The MIT Press, 1990

[MT91] Robin Milner, Mads Tofte
*Commentary on Standard ML*
The MIT Press, 1991

[K93] Stefan Kahrs
*Mistakes and Ambiguities in the Definition of Standard ML*
University of Edinburgh, 1993
`http://www.cs.ukc.ac.uk/pubs/1993/569/`

[K96] Stefan Kahrs
*Mistakes and Ambiguities in the Definition of Standard ML – Addenda*
University of Edinburgh, 1996
`ftp://ftp.dcs.ed.ac.uk/pub/smk/SML/errors-new.ps.Z`

[DB07] Derek Dreyer, Matthias Blume
*Principal Type Schemes for Modular Programs*
in: Proc. of the 2007 European Symposium on Programming
Springer-Verlag, 2007