Mistakes and Ambiguities in the Definition of Standard ML

Stefan Kahrs*
University of Edinburgh
Laboratory for Foundations of Computer Science
King's Buildings, EH9 3JZ
email: smk@dcs.ed.ac.uk

Version of April 6, 1993

Abstract

The *Definition of Standard ML* contains several errors and ambiguities. Some of them have already been published in the *Commentary on Standard ML*, but the list given there is incomplete.

The paper lists all errors known to me today, including the errors listed in the Commentary. On most of the others I came across when writing the semantics of Extended ML. Most errors are supplied with an explanation "what goes wrong" and with a suggested correction. I understand "error" in a very broad sense — ranging from typos to serious flaws in the rules. Some of the problems I mention are originated by a certain tension between formal definitions and informal explanations, e.g. overloading is informally explained though impossible in the given formal setting.

Some parts of the paper are difficult to understand without prior knowledge of the Definition and the Commentary, because I rely on the notation and jargon introduced in these books. However, the nature of this paper has it that the various sections do not depend on each other, making it possible for somebody not familiar with the entrails of SML to read the less technical sections on their own.

Structure of the Paper

I have divided the errors into the classes "Errors Listed in the Commentary" and "Further Errors". This is by no means a conceptual distinction. It simply gives a quicker answer to the question "What's new?" for the lucky owners of the Commentary. Here and in the following, the "Definition" always refers to [MTH90], the "Commentary" to [MT91].

Within the two parts, the errors are listed in the same order as they appear in the Definition. References of the form "line -n" refer to the nth line from the bottom of the corresponding page or section.

*The research reported here was partially supported by SERC grant GR/E 78463.

Part I
Errors Listed in the Commentary

All errors described in this part are mentioned in the Commentary. Not all corrections proposed in the Commentary are entirely satisfactory, this is also discussed. Some of the quotations from the Commentary refer to certain "Sections" — these are sections of the Commentary itself.

1 Syntax

1.1 Comments

The Commentary adds the following clarification for SML comments on page 4 of the Definition:

> No space is allowed between the two characters which make up a comment bracket (`(*` or `*)`). Even an unmatched `*)` should be detected by the compiler. Thus the expression `(op *)` is illegal. But `(op * )` is legal; so is `op*` .

The example suggests that "should be detected" cannot be interpreted as "may only be accepted with a warning".

It is not obvious whether `(op **)` is permitted or not. It should be, according to the principle of longest match, and it should not be, according to the principle of open comment brackets. Notice that the Definition does not define `*)` as a lexical item. We could make it a lexical item by adding `*)` to the list of reserved words — this decides the ambiguity in favour of "should", because the longest match principle applies to all lexical items. The *use* of the reserved word `*)` in programs is then not allowed, because it is not mentioned elsewhere, i.e. no program containing it can pass the syntax check.

1.2 Identifier Status

There are nine classes of identifiers in SML. "Class" is an attribute of an identifier depending on the context of its occurrence, it does not refer to a partition of identifiers into disjoint classes. For instance, an identifier can be a structure identifier (*strid*) and a variable identifier (*var*) in the same scope. The classes Var, Con, and ExCon have to be kept disjoint in each scope, because identifiers in those classes denote values and can occur anywhere a value can occur. Therefore each scope has an associated status map that assigns to identifiers one of the values $\{\mathtt{v}, \mathtt{c}, \mathtt{e}\}$, indicating to which of the three classes it belongs in that scope.

Pages 5 and 10 of the Definition give principles to determine the status of an identifier. These principles are informally stated and incomplete. In particular, they do not explain the effect of signatures and structures on identifier status.

A more detailed description of how to keep track of identifier status is in appendix B of the Commentary, pages 132–134. One can take this appendix almost unchanged as an additional section for the appendix of the Definition. I do not repeat it here.

Two things about appendix B deserve a remark. The Commentary says about incorporating the treatment of identifier status into the Definition:

> (...) and indeed the status map could have been combined with the static environment, so that elaboration could be given the task of assigning status.

This is more an insinuation for implementors than a serious remark about the presentation of the semantics, for the dynamic semantics has no access to the results of static analysis.

Appendix B comments briefly about the identifier status in derived forms:

> Also, we ignore derived forms.

This is a pity, because the replacement of a derived form by its equivalent form is inevitably preceded by parsing the program text into the parse tree of a derived form — a process which depends on identifier status. For the derived form of function declarations there is indeed a choice: the left-hand side of `fun f x=1` could be parsed in a similar way to patterns, requiring `f` not to have status `c` or `e`, but this requirement is actually redundant. Not all SML implementations agree on this matter.

Part II discusses a couple of problems related to the treatment of identifier status in Modules (as it is suggested in the Commentary).

1.3 Syntactic Restrictions

Add the following restriction to the end of section 3.5, page 12:

- In the *tyvarseq tycon* in any *typdesc* or *datdesc*, *tyvarseq* must not contain the same *tyvar* twice. Any *tyvar* occurring on the right side of the *datdesc* must occur in *tyvarseq*.

Similar restrictions already exist for the syntax of the Core, i.e. for *datbind*, etc.

2 Static Semantics for the Core

Three corrections on page 30, sections 4.11 and 4.12:

> Lines 9–10 of [Sec 4.11]: principal type schemes $\longmapsto$ principal environments
>
> Line -5: $E \succ E' \;\longmapsto\; \mathrm{Clos}_C E \succ E'$
>
> Line -3: delete "and imperative type variables".

There is no reference in the rules to anything called a principal type scheme, making the old remark in the Definition rather pointless.

The second correction captures a problem caused by SML's imperative type variables, more precisely by the limitations that apply to the abstraction of imperative type variables. An environment E is principal for *dec* in C if $C \vdash dec \Rightarrow E$ and for any environment E' with $C \vdash dec \Rightarrow E'$ we have $(N)(m, E') \geq S \prec (m, E)$ where N are the names in E' not in C and m is fresh[1]. The restrictions for abstracting imperative type variables occasionally exclude the environments that are principal in this stricter sense, and only in these cases the correction is significant, declaring environments with free

[1] A formula $\Sigma \geq S \prec S'$ denotes signature matching in SML, see sections 5.9, 5.11, and 5.12 in the Definition.

imperative type variables as principal. For the purposes of this paper I call them *weakly principal*. Weakly principal environments *characterise* all other derivable environments, but they do not *generalise* them.

The third correction is a consequence of the second, the phrase "and imperative type variables" is now obsolete. One could even remove the whole sentence if (in the second correction) $E \succ E'$ were replaced by the more liberal "$\varphi(\mathrm{Clos}_C E) \succ E'$, for some realisation φ with $\mathrm{Supp}\,\varphi \cap T$ of $C = \emptyset$". This would more directly reflect the above idea of principality in terms of signature matching.

3 Static Semantics for Modules

> page 31, Line -12: delete "imperative,".

There is simply no such thing as an imperative attribute for type names.

About page 35, lines -5,-4, the Commentary says

> the claim that a principal signature exists must be slightly qualified, since it may be ill-formed in a mild sense. This is discussed at the end of Section 11.3.

This correction is a rather technical point, because principal signatures are only required in an intermediate step to get equality-principal signatures and the existence of the latter cannot be claimed in general anyway.

> Page 42, rule 86: $\mathrm{tyvars}(\tau) = \emptyset \;\longmapsto\; \mathrm{tyvars}(ty) = \emptyset$. This is needed to ensure that rule 86 is a structural contraction; see Section A.1.

The new condition is slightly more restrictive, because ty might contain more type variables than τ.

A type expression ty is a syntactic object composed from type variables and type constructors. A type τ is a semantic object composed from type variables and type names. In a static context, type constructors are bound to type functions, k-ary functions mapping k types to a type. All type variables occurring in the result of a type function have to occur in one of the arguments, but the converse is not true, for example $\Lambda\alpha.\mathtt{int}$ is a type function mapping any type (possibly containing type variables) to `int`.

Therefore, all type variables occurring in a type τ also occur in any type expression ty that it denotes, but again the converse is not true. Unfortunately, tyvars is only defined for semantic objects, not for syntactic ones such as ty, such that one also has to extend the domain of tyvars to syntactic objects (page 17). The change is needed to guarantee the existence of principal signatures. Type names are not quite as expressive as type functions (they cannot ignore arguments), but in a principal signature a type name is supposed to generalise a class of type functions.

Another correction:

> Page 44, rule 99: $\mathrm{names}\, S' \setminus ((N \,\mathrm{of}\, B) \cup N) \;\longmapsto\; \mathrm{names}\, S'\langle'\rangle \setminus ((N \,\mathrm{of}\, B) \cup N)$

This is obviously a typo in the Definition.

4 Dynamic Semantics for the Core

Two mistakes of the dynamic Core semantics are reported in the Commentary: the variable environments that a datatype binding or an exception bindings generates (or rather should generate); and the treatment of the constructor `ref`, which does not coincide with the static semantics.

4.1 Variable Environments

On page 46, section 6.1, three corrections are necessary:

first bullet: exception bindings $\longmapsto$ constructor and exception bindings.

second bullet: delete "or "`datatype` *datbind*" " (see Section 2.7).

fourth bullet: delete "DatBind, Conbind,".

All three changes are about the same problem, the first and third change being necessary because of the second. Datatype definitions give rise to variable environments (the constructors) in the static semantics and so they should in the dynamic semantics. On first look it seems unnecessary, because such environments only bind constructors to themselves and constructors evaluate to themselves anyway; but the status of an identifier can change. There is a related problem in the dynamic semantics for Modules, which was not mentioned in the Commentary, see part II.

We need three additional rules to cope with this properly, also rule 130 has to be changed for a similar reason, as an exception environment gives rise to a variable environment in the static semantics, compare rule 21.

After rule 129 we add the following rule for datatype declarations:

$$\frac{\vdash \mathit{datbind} \Rightarrow \mathit{VE}}{E \vdash \texttt{datatype}\ \mathit{datbind} \Rightarrow \mathit{VE}\ \text{in Env}} \tag{129.1}$$

Rule 130 changes to the following version:

$$\frac{E \vdash \mathit{exbind} \Rightarrow \mathit{EE} \qquad \mathit{VE} = \mathit{EE}}{E \vdash \texttt{exception}\ \mathit{exbind} \Rightarrow (\mathit{VE}, \mathit{EE})\ \text{in Env}} \tag{130}$$

After rule 137 we insert two extra rule sections and add two rules.

Data Type Bindings $\boxed{\vdash \mathit{datbind} \Rightarrow \mathit{VE}}$

$$\frac{\vdash \mathit{conbind} \Rightarrow \mathit{VE} \qquad \langle \vdash \mathit{datbind} \Rightarrow \mathit{VE}' \rangle}{\vdash \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind} \rangle \Rightarrow \mathit{VE}\ \langle + \mathit{VE}' \rangle} \tag{137.1}$$

Constructor Bindings $\boxed{\vdash conbind \Rightarrow VE}$

$$\frac{\langle \vdash conbind \Rightarrow VE \rangle}{\vdash con \ \langle \ \mathtt{|} \ conbind \rangle \Rightarrow \{con \mapsto con\} \ \langle + \ VE \rangle} \qquad (137.2)$$

The Commentary is not quite as explicit about the insertion (the suggested version for rule 130 is the same, for rule 129.1 almost — see page 21 of the Commentary), in particular it avoids these two extra sections. Similar context-free rules exist in the dynamic semantics for Modules anyway, i.e. for consistency of style the insertion should be as explicit as described.

Remark: these corrections are not sufficient. Abstract types have the same problem with variable environments, see section 11.2 in part II.

4.2 The Constructor `ref`

The identifier `ref` is not a reserved word, it can be hidden or redefined just as any other identifier. But, as the Commentary states:

> Rule 158 deals incorrectly with the case in which a program redeclares `ref` as a value constructor, since it will always interpret `ref` as a memory reference. Rule 114 [p51] is similarly at fault in this case. For this reason, compilers may wish to issue a warning if `ref` is redeclared or specified as a value constructor.

This is not quite the whole story, because rules 112, 154, and 155 make exactly the same mistake. The suggestion to "issue a warning" is too tame, because it urges the compiler to make the programs still behave as defined by the incorrect rule. This obviously loses type-safety[2] and it would be difficult and very unsatisfactory to implement. The correction is moreover incomplete, since the redeclaration of `ref` as an exception constructor would not work well with rule 158.

Looking up the type of `ref` at all those places seems to be a simple correction, because its result type distinguishes it from any other value constructor or exception constructor. Unfortunately, the dynamic semantics has no access to the results of the static analysis. The problem goes a bit deeper, because there are similar problems with the constructors `true`, `false`, `nil`, and :: in appendix A (Derived Forms), figures 15 and 16, page 67.[3]

All these problems are caused by the fact that value constructors are syntactic *and* semantic objects, that there is no distinction between the syntactic item *con* and the semantic value *con*. Such a distinction could be introduced analogously to the treatment of exception names:

1. Extend the Simple Semantic Objects (Figure 12, page 46) by a class of constructor names ConName, $vcon \in$ ConName.

[2] "Type-safety" is the soundness of the static semantics: if a program elaborates (it successfully passes the static analysis), then the evaluation of the program "does not go wrong", i.e. the case analysis in the dynamic semantics is exhaustive.

[3] A simple solution to this problem is to make these identifiers reserved words that cannot be (re-)bound, similarly as `=`.

2. Add for ConName semantic classes ConNameSet and ConVal to the Compound Semantic Objects (Figure 13, page 47), analogous to ExName.

3. Replace "Con $\cup$ (Con $\times$ Val)" in the definition of Val by "ConVal".

4. Extend the class State by a third component ConName, $vcon \in$ ConName.

5. Change rule 105 to:

$$\frac{E(longcon) = vcon}{E \vdash longcon \Rightarrow vcon} \tag{105}$$

6. Replace con by $vcon$ everywhere in rule 112.

7. Change rule 137.2 as follows:

$$\frac{vcon \notin vcons \text{ of } s \qquad s' = s + \{vcon\} \qquad \langle s' \vdash conbind \Rightarrow VE, s'' \rangle}{s \vdash con \ \langle \ | \ conbind \rangle \Rightarrow \{con \mapsto vcon\} \ \langle + \ VE \rangle, s' \langle ' \rangle} \tag{137.2}$$

8. Change in rules 144, 145, 154, and 155 the premise (side-condition) $longcon = strid_1.\cdots.strid_k.con$ to $E(longcon) = vcon$ and replace con in the other premises by $vcon$.

9. Move derived forms which make explicit use of constructors to the bare language[4]. This concerns `if-then-else` and lists in square bracket notation, expressions and patterns. The reason is that in a particular context there might not *exist* an equivalent form in the bare language.

10. Expressions containing `andalso` and `orelse` can still be expressed as derived forms, but in a different way:

exp_1 `orelse` exp_2	`let`
	    `val` var_1 `=` exp_1
	    `and` var_2 `= fn() =>` exp_2
	`in if` var_1 `then` var_1 `else` var_2`() end`
		$var_1 \neq var_2$

exp_1 `andalso` exp_2	`let`
	    `val` var_1 `=` exp_1
	    `and` var_2 `= fn() =>` exp_2
	`in if` var_1 `then` var_2`()` `else` var_1 `end`
		$var_1 \neq var_2$

11. The $vcons$ component of the initial state becomes $\{\texttt{true}, \texttt{false}, ::, \texttt{nil}, \texttt{ref}\}$.

These changes respect the original SML semantics, in the sense that programs produce the same values on observable types (`bool`, `int`). However, there is one exception:

```
fun f _ = let datatype a = B in B end
```

[4]Remark: moving `case`-expressions to the bare language would allow to make their bound identifiers polymorphic.

In the SML semantics, `(f 1=f 2)` is a well-typed expression and evaluates to `true`. After the changes indicated above, the expression would instead evaluate to `false`, because each evaluation of the datatype declaration gives a fresh set of constructor values. But there is a type-safety problem with rule 6 (let expressions, static semantics) anyway, see page 17 in part II, and the correction given there would eliminate these differences between SML semantics and the suggested treatment of constructors. The example does not elaborate then, and the phenomenon that each evaluation of a datatype declaration gives a fresh set of constructor values would be unobservable and hence of no concern for implementations.

5 A Appendix: Derived Forms

The Commentary makes two comments about derived forms, referring to page 67 in the Definition:

> Some of the derived forms of expressions [Fig 15], such as `()`, must be parsed as atomic expressions; they can be found under *atexp* in the full grammar [App B, Fig 19, p 71]. Similarly, the derived forms of patterns [Fig 16] must be parsed as atomic patterns; they all appear under *atpat* in [Fig 21, p 73].

I understand that this means to split figures 15 and 16 into figures 15a (for *exp*) and 15b (for *atexp*), similarly for patterns. The derived forms which happen to be atomic go to 15b or 16b, the others to figures 15a or 16a, respectively. We have to do a little bit more: if the derived form is atomic but the equivalent form is not, then the equivalent form has to be enclosed in parentheses.

These corrections are necessary but not sufficient. Several equivalent forms do not respect parsing, because the equivalent forms have not always the same precedence as the corresponding derived forms. For example, when a nested `if-then-else` (nested in its `then`-part) is rewritten into its equivalent form, then the "parse tree" could be reshaped, because the resulting nested `case`-expression does not parse in the intended way. Or rather, as `case`-expressions are derived forms themselves, the result depends on the order in which the rewrite rules are applied — they do not form a confluent string rewriting system. This can be seen in Figure 1.

Therefore, we have to add parentheses at the appropriate places to prevent the parse tree to be reshaped — this is the price one has to pay for omitting an abstract syntax and for the absence of parse trees as semantic objects. The most convenient way to add those parentheses is to have a convention for inserting parentheses:

> All occurrences of the syntactic variables *exp* and *pat* (with or without index) in the equivalent forms are abbreviations of the corresponding atomic expressions or patterns (*exp*) and (*pat*) (with their old index, if any). Similarly, the equivalent forms of expressions and patterns have to be considered as atomic, i.e. if figure 15 or 16 defines an equivalent form *phrase* of the syntactic classes Exp or Pat, then this is an abbreviation for (*phrase*).

This convention emulates parse trees on the level of strings.

The second comment in the Commentary about derived forms addresses a problem we have already considered:

```
if A then
    if B then C
    else D
else E
```

```
if A then
    case B of true => C
           | false => D
else E
```

```
case A of true =>
    case B of true => C
           | false => D
       | false => E
```

```
case A of true =>
    (fn true => C |
       false => D |
       false => E)(B)
```

```
(fn true =>
    (fn true => C |
       false => D |
       false => E) (B))(A)
```

```
if A then
  (fn true => C | false => D)(B)
else E
```

```
case A of true =>
  (fn true => C | false => D)(B)
       | false => E
```

```
(fn true => (fn true => C
             | false => D)(B)
 | false => E)(A)
```

Figure 1: Derived Forms

> Note that the meanings of certain derived forms [Fig 15 and 16] change if certain parts of the initial basis are overwritten. For example, the meaning of an `if`···`then`···`else` expression is affected by a rebinding of `true` or `false`; similarly, giving `it` constructor status changes the meaning of the derived form of expressions at top-level [Fig 18]. For this reason, compilers may wish to issue a warning if `true`, `false`, `nil` or `::` is redeclared or specified as a value constructor, exception constructor or variable, or if `it` is declared at top-level as a value constructor or exception constructor.

This is again about the distinction between *con* as a syntactic item and as a value. In the derived form for `if-then-else`, one needs a syntactic gadget that accesses the values `true` and `false`. In general, this might not exist[5]. In a strict sense, this means that `if-then-else` in its usual meaning cannot be defined as a derived form.

On the other hand, in a similarly strict sense, one could take the corresponding rewriting rule literally as the definition of if-then-else and leave it to the programmer to possibly (but recommendably not) redefine the meaning of this construct by redefining

[5] Not quite true: `()=()` always denotes `true`, because `=` cannot be rebound. But it would be bad style to make the definition of derived forms dependent on such a trick.

`true` and `false`. Notice that giving the identifier `true` value status would make any `if-then-else` expression evaluate to its `then`-part.

The same problem exists for lists in square-bracket notation, i.e. the equivalent form for lists relies on the access to list-constructors.

The problem with `it` might be regarded as less serious, because value variables are not values, in contrast to value constructors. In particular, type-safety is not affected. It is an inconvenience that after introducing `it` as a value constructor or exception constructor evaluation of expressions on top level (not within a declaration) is not longer possible, unless the identifier `it` regains value status.

6 C Appendix: The Initial Static Basis

Page 74, line -7: $(\texttt{true}, \texttt{false}, \texttt{nil}, ::) \longmapsto (\texttt{true}, \texttt{false}, \texttt{nil}, ::, \texttt{ref})$

7 D Appendix: The Initial Dynamic Basis

According to the Commentary, the fifth bullet on page 77 of the Definition should read:

$$\begin{aligned} VE_0' \;=\; & \{id \mapsto id\ ;\ id \in \text{BasVal}\} \;\cup\; \{:= \;\mapsto\; :=\} \;\cup\; EE_0' \\ & \cup \{\texttt{true} \mapsto \texttt{true},\ \texttt{false} \mapsto \texttt{false},\ \texttt{nil} \mapsto \texttt{nil},\ :: \mapsto ::,\ \texttt{ref} \mapsto \texttt{ref}\} \end{aligned}$$

A similar obvious oversight; actually, all the "$\cup$" should be "+".

After making the indicated corrections for exception environments, the EE component of E seems to be redundant, at least in the dynamic semantics. Each time an EE enters an environment, a copy of it goes to the variable environment. Similarly redundant is the *excons* component of an interface. Exception environments are only significant for structure/signature matching in the static semantics. There is a problem with the way the Definition treats exception environments separately from variable environments, see section 10.3 in part II.

Page 80, line 13: after "initially empty." add

> Any existing contents of the file s are lost. The exception packet
>
> [(`Io`,`"Cannot open` s`"`)]
>
> is returned if write access to the file s is not provided.

Raising an exception is the only sensible thing `open_out` can do in such a case. The addition is consequent in the sense that `open_in` already allowed an analogous way of failure.

Part II
Further Errors

Most of the other errors have been found during the development of the semantics of Extended ML. Some of them are probably known for some time, because they (should) become apparent when one writes a compiler for SML.

8 Syntax

A couple of typos:

> Page 9, Fig 4: There is a '::=' missing, after *pat*.
>
> Page 11, line -6: In "call call" there is one "call" too many.

Most problems related to the syntax are caused by a lack of formalism in the definition of the SML syntax. On the one hand, this leaves room for interpretation and the reader is urged to fill the gaps, on the other some ambiguities remain unresolved, affecting the meaning of the semantic rules.

8.1 Reserved Words

The reserved words of SML are presented in two parts, "reserved words used in the Core" and "reserved words used in Modules". The Definition does not say what this partition into two sets is supposed to mean. Is for instance `struct` a reserved word in a Core declaration, can it be used as a record label? It is certainly unusual to have context-dependent reserved words, but it is not unknown. One could clarify this by replacing the first two sentences of section 2.1 (page 3) by the following:

> The reserved words of Standard ML can be divided into two groups, namely (1) those that are necessary for presenting the grammar of the Core and (2) those additional reserved words that are needed for presenting the grammar of Modules. Below we list reserved words of the first group (the rest are listed in Section 3.1). Reserved words may not (except =) be used as identifiers.

This insertion makes clear that "reserved" means "reserved everywhere".

The first line of Section 3.1, page 10, becomes:

> In addition to the reserved words listed in Section 2.1, Standard ML reserves the following words, which are used in the grammar for Modules:

The rôle of the reserved word `=` as an identifier is not entirely clear. The only restriction about its use is on page 4:

> The identifier `=` may not be re-bound;

This could be interpreted as that it is allowed to define a type (structure, signature, functor, label) named `=`, but that it is not allowed to overwrite such a binding once it is established. This is surely not the intended meaning. Most implementations do not allow the use of `=` for any of these purposes. New Jersey ML allows `=` to be used for new types, structures, signatures, and functors, but not for labels; it also allows to overwrite these bindings. However, the following does not pass the syntax check in New Jersey ML:

```
signature = = sig end;
functor = (= : =) = =
```

So it seems advisable to exclude `=` from such applications. The simplest clarification is to replace on page 4 "re-bound" by "bound".

8.2 Infixed Operators

The meaning of fixity directives is given in section 2.6 of the Definition. One of the principles formulated there is rather counter-intuitive and can be regarded as a mistake, it is:

> association is always to the left for different operators of the same precedence.

As Andrew Appel pointed out in [App92], two right-associative operators of the same precedence should associate to the right, i.e. the passage should read: "... for operators of the same precedence but opposite associativity." It is a matter of taste whether those operators associate to the left, to the right, or mix at all without parentheses. The proposed change keeps the SML meaning when there is such a choice. But if `++` and `**` are two right-associative operators of the same precedence, then `a++b**c` should parse as `a++(b**c)` — this is well-established folklore in operator precedence parsing.

The SML design was probably influenced by a sloppy passage in a standard textbook on compiler construction [ASU86], page 31:

> Consider the expression `9+5*2`. [...] The associativity of `+` and `*` do not resolve this ambiguity.

This is not directly wrong (apart from the grammatical error), because the operators `+` and `*` have different precedence anyway, but it leaves the wrong impression that the associativity of an operator does not help to resolve ambiguity against another operator. The authors of [ASU86] are a bit more precise about this matter in a later section, page 207ff. There it is also implicitly suggested (page 210, the point under "1.") that mixing operators of the same precedence and different associativity is an error, which was also proposed by Appel as the best solution.

Related to infixed identifiers is the question when the keyword `op` is required in constructor bindings and exception bindings. The Definition says about `op` on page 6:

> The only required use of `op` is in prefixing a non-infixed occurrence of an identifier *id* which has infix status;

$$\begin{tikzcd}[column sep=small]
 & \texttt{dec ; dec} \arrow[dl] \arrow[dr] & \\
\texttt{strdec ; strdec} \arrow[dr] & & \texttt{dec} \arrow[dl] \\
 & \texttt{strdec} &
\end{tikzcd}$$

Figure 2: Ambiguity of Declaration Sequences

But is the occurrence of *con* in a constructor binding an occurrence for which this principle applies? The intended answer seems to be "yes", but this is not quite obvious, because not all non-infix occurrences of an infixed identifier are even allowed to be preceded by `op`, for example occurrences in constructor descriptions are not; also, constructor bindings cannot contain infix-occurrences of an infixed identifier anyway. The analogy between constructor (exception) bindings and descriptions suggests that the syntax of these constructs is badly designed; it would probably be better to entirely remove `op` from constructor and exception bindings.

8.3 Resolving Ambiguity

The context-free grammar of SML is highly ambiguous. The Definition gives several more or less informally stated principles how to resolve ambiguity in many cases. These principles are not sufficient to overcome all syntactic ambiguities. The remaining ambiguities could be considered harmless, as long as the semantics is not affected. They are annoying anyway, because even a harmless ambiguity requires a proof of its harmlessness and a formalisation what this "not affected" actually means.

Let us first look at a rather harmless case: a Core declaration *dec* can be of the form $dec_1\ \langle ; \rangle\ dec_2$. This syntax rule "overlaps" (in the sense used for rewrite systems) with itself, i.e. a declaration $dec_1;dec_2;dec_3$ can reduce in two ways to *dec*. The semantic rules are defined on the syntactic structure, which (for *dec*) implicitly requires that the semantic functions that replace $\langle ; \rangle$ in static and dynamic semantics are associative. Inspecting rules 25 and 134, we find the straightforward way to prove that is to show that "+" is associative on (static and dynamic) environments and that (static semantics) $C \oplus (E_1 + E_2) = (C \oplus E_1) \oplus E_2$. Unfortunately, the latter is not true: the static context on the right-hand side of the equation can contain more type names than the one on the left. This means that $C \oplus (E_1 + E_2) \vdash dec \Rightarrow E$ does not imply $(C \oplus E_1) \oplus E_2 \vdash dec \Rightarrow E$. Therefore, the associativity of $\langle ; \rangle$ is at least not obvious and it would be better to avoid the problem in the first place.

There are more serious ambiguities in the syntax. A sequence of two Core declarations can be parsed as a structure declaration in two different ways, see figure 2. A similar ambiguity (with similar consequences) exists for `local`, as there are local (Core) declarations and local structure declarations.

Again, one would expect that the different parsings do not affect the result. Unfortunately they do. Rule 57, which interprets a *dec* as a *strdec* in the static semantics, enforces principality of the environment obtained from *dec*. Thus, on the left-hand side of the picture principality is enforced twice, on the right-hand side only once. This would not make a difference if the existence claim for principal environments could be extended to the stricter notion of principality I mentioned in section 2. The syntactic ambiguity causes problems if the principal environment of the first declaration is only weakly principal.

```
val x = ref [];
val y = x:=[1]
```

This declaration sequence elaborates as a *strdec* if and only if it is parsed as in the right-hand side of the picture. The other parse tree fails to elaborate, because the principal environment of the first declaration binds `x` to `'_a list ref` — the type variable `'_a` occurs free in this type and cannot be replaced later by `int`.

On page 70, the Definition states:

> Note particularly that the use of precedence does not decrease the class of admissible phrases; it merely rejects alternative ways of parsing certain phrases.

This idea of disallowing disambiguation principles to decrease the language defined by the grammar sounds nice, but it introduces further ambiguities. Example

```
false andalso if x then x else x orelse true
```

The order of precedence is: `andalso` > `orelse` > `if`. There are two ways to parse this phrase (as an `andalso`-expression and as an `orelse`-expression), but both violate the precedence `andalso` > `if`. Thus, it is not clear which one is to be preferred. Notice that the value of the expression differs for the two parsings.

All these problems suggest that one should not have an ambiguous syntax to begin with. The best fix would probably be to distinguish between abstract and concrete syntax, the concrete syntax being non-ambiguously expressed in some formalism, e.g. as an LALR(1) grammar. As this may require a complete redesign of the SML syntax, I have chosen the second best fix, which is to add some further principles that resolve the remaining ambiguities. On page 7, we can replace the last bullet by the following point:

- Longest match: Suppose F_1F_2 is an alternative form of a phrase class. A natural number i is called a *split index* w.r.t. F_1F_2 for a lexical sequence $L_1 \cdots L_k$, if $0 \leq i \leq k$ and $L_1 \cdots L_i$ reduces to F_1 and $L_{i+1} \cdots L_k$ reduces to F_2. If for a given lexical sequence $L = L_1 \cdots L_k$ there are different split indices w.r.t. F_1F_2, then L reduces to F_1F_2 by reducing $L_1 \cdots L_j$ to F_1, where j is the maximal split index.

The F_j are regular expressions as they occur in the SML grammar, with terminals and non-terminals as primitives, and concatenation and optional brackets as connectives. A lexical sequence is a sequence of terminals.

The longest match principle for parsing is a generalisation of the "extends as far right as possible" bit, it also resolves a few further ambiguities[6]. The same replacement

[6] Remark: the longest match principle stated here is not general enough for disambiguating arbitrary context-free grammars, because a word of regular expressions of length n can be split in $n+1$ different ways into F_1F_2. For the SML grammar, this does not seem to be a problem.

has to be done in appendix B. There it is also explained what "precedence" is supposed to mean — we can generalise the third bullet there as follows:

- Alternative forms for each phrase class are in order of decreasing precedence. This precedence resolves ambiguity in parsing in the following way. Suppose that a phrase class *phrase* has several alternative forms $F_1 \cdots F_n$. If a lexical sequence $L_1 \cdots L_k$ reduces to more than one of the F_i, then it reduces to *phrase* via the F_i with lowest precedence. Example: The parsing of the sequence

  $$\texttt{if } exp_1 \texttt{ then } exp_2 \texttt{ else } exp_3 \texttt{ handle } match$$

  is determined by the above principle. Because `if`-expressions have lower precedence than `handle`-expressions, the sequence parses as

  $$\texttt{if } exp_1 \texttt{ then } exp_2 \texttt{ else } (exp_3 \texttt{ handle } match)$$

  Note particularly that the use of precedence does not decrease the class of admissible phrases; it merely rejects alternative ways of parsing certain phrases. In particular, the purpose is not to prevent a phrase, which is an instance of a form with higher precedence, having a constituent which is an instance of a form with lower precedence. Thus for example

  $$\texttt{if} \cdots \texttt{then while} \cdots \texttt{do} \cdots \texttt{else while} \cdots \texttt{do} \cdots$$

  is quite admissible, and will be parsed as

  $$\texttt{if} \cdots \texttt{then (while} \cdots \texttt{do} \cdots\texttt{) else (while} \cdots \texttt{do} \cdots\texttt{)}$$

This principle is a proper generalisation, because it resolves the syntactic ambiguity of reducing *dec* $\langle ; \rangle$ *dec* to *strdec*: the alternative form with lowest precedence for *strdec* is *strdec* $\langle ; \rangle$ *strdec*, so it is parsed this way.

These two additional principles seem to resolve all ambiguities, as the first resolves the overlaps of a form with itself and the second the overlaps with other forms.

Remark: the disambiguation principle suggested here does in a few cases not coincide with several existing SML compilers. For example, the mentioned expression

```
false andalso if x then x else x orelse true
```

is regarded here as an `orelse`-expression (making it `true`), while several implementations treat it as an `andalso`-expression (making it `false`). The implementations seem to use bottom-up parsing methods while the method described here is essentially a top-down disambiguation. Both methods delay the use of precedence violating grammar rules as long as possible, which means that they appear in the parse tree as high as possible for bottom-up and as low as possible for top-down.

8.4 Parsing Problems

Parsers are often generated by compiler compilers for a particular class of context-free grammars, e.g. LALR(1). The SML syntax is not described in such a formalism, which unfortunately tempts implementors to slightly redefine it. I mention here two such problems:

- The syntax allows layered patterns to have a type assertion, i.e. a proper pattern would be `x: int list as y::ys`. For LR-parsing, this is quite problematic, because *var* : *ty* can be reduced to *pat*, but it can also be the initial part of a layered pattern — we have a shift/reduce conflict at the "`:`" that cannot be resolved by (finite) lookahead. Solving shift/reduce conflicts in favour of shift (usually the default in LR parsers) is here clearly undesirable, because it would exclude type assertions for variable patterns. There is an easy way to realise layered patterns with type assertions in an LR grammar: extend the class of syntactically accepted phrases for layered patterns to

  $$\langle \texttt{op} \rangle \; pat_1 \; \langle : ty \rangle \; \texttt{as} \; pat_2$$

  and exclude after parsing those layered patterns in which pat_1 is not a variable. We can observe several implementations using this trick:

  ```
  val x = fn (y) as z => z
  ```

  Implementations that (illegally) allow the above declaration probably use the mentioned trick; they produce the same abstract syntax for the patterns `y` and `(y)`, such that they fail to find the syntax error.

- A similar lookahead problem exists for parsing *fvalbind* if the expression on the right-hand side is either a `case`-expression or an `fn`-abstraction, see the following example:

  ```
  fun f x 0 = case x of [] => ""
   |  f x n = "foo"
  ```

  The grammar (together with its disambiguation rules) permits this example, because *fvalbind* and *match* use different delimiters to separate left-hand and right-hand side, *fvalbind* uses `=` and *match* uses `=>`. Therefore, the disambiguation principle "extends as far right as possible" does not apply here, the `|` belongs to the *fvalbind*. But this is very difficult to express as an LALR(1) grammar and most (if not all) implementations reject the example.

  A comparatively simple way to solve this problem seems to let the scanner (or a preprocessor) distinguish between a `|` that belongs to a *match* and one that belongs to an *fvalbind*; thus, if the scanner finds a `|` in an expression it continues to read the input until it finds the corresponding delimiter, either `=` or `=>`. This task excesses the expressive power of finite automata, but it should be expressible in *lex*-generated scanners.

8.5 Others

In section 2.9, page 9, the Definition restricts the body of `val rec` declarations:

> For each value binding *pat* `=` *exp* within `rec`, *exp* must be of the form `fn` *match*, possibly constrained by one or more type expressions.

This is an inconsistent requirement (pointed out by Nick Rothwell in [Rot92]), because an expression of the form `fn` *match* cannot directly be type-constrained, that is: without parentheses.

Another inconsistency in the grammar description arises as a consequence from the following restriction on page 14:

> *Note:* No *topdec* may contain, as an initial segment, a shorter top-level declaration followed by a semicolon.

A program can contain functor declarations only as top-level declarations. Thus, the syntax rule for a sequential functor declaration $fundec\langle;\rangle fundec$ is equivalent to *fundec fundec*, because the semicolon is forbidden by the mentioned restriction. This also makes the empty functor declaration redundant (analogously for signature declarations).

9 Static Semantics for the Core

Each datatype definition (abstypes are similar) does not only introduce several constructors, it also attaches to the introduced type a so-called *type name* which is a kind of personal identification number for types. Its purpose is to compare types on the semantic level, for example to distinguish two types which happen to have been defined with the same type identifier. The static semantics always has to keep track of these type names to make sure that any newly introduced type gets a fresh type name. This can be seen in the rules when the $\oplus$ is used.

There are two places in the static semantics where this keeping track of type names is not done properly.

9.1 Too few type names are different

The first place is the rule for `let`-expressions in the static semantics.

$$\frac{C \vdash dec \Rightarrow E \qquad C \oplus E \vdash exp \Rightarrow \tau}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau} \tag{6}$$

The type τ may contain a type (name) which has been introduced in *dec*, i.e. a local type. There is no principal problem with having non-accessible types, but there is a related problem caused by the required uniqueness of type names. Rule 6 does *not* fully keep track of the type names introduced in *dec*. It does so for the elaboration of *exp* (this is hidden in the $\oplus$), but it does not for the elaboration of the rest of the program, i.e. the text behind the `let`-expression. If such a new type escapes the local declaration by occurring in the result τ, then it could have the same "personal identification number" as some other type, introduced at a different place. This loses type-safety.

```
let datatype A = C of bool -> bool
in
        fn (C x) => x true
end
let datatype B = C of int
in
        C 2
end
```

The expression above should elaborate according to the static semantics, but the dynamic semantics of it tries to apply the number 2 to `true`. Both `let`-expressions are elaborated in the *same* static context, which means that the datatypes `A` and `B` could be given the same type name. The whole expression only elaborates if this is the case, which *forces* them to have the same type name. In the dynamic semantics constructors evaluate to themselves[7], which in the example means that matching succeeds. Finally, the expression `x true` is evaluated in an environment in which `x` is bound to 2.

The easiest fix of this problem would be to disallow type names to escape `let`-expressions, that is to change rule 6 as follows.

$$\frac{C \vdash dec \Rightarrow E \qquad C \oplus E \vdash exp \Rightarrow \tau \qquad \text{tynames}\,\tau \subseteq T \text{ of } C}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau} \tag{6}$$

This version of the rule is a bit more restrictive than necessary (and desirable, see the next section), because the only thing that has to be taken care of is that fresh type names are really fresh.

Instead, one could have a notion of state for the static semantics, where a state is just a set of type names. Introduction of a new type name changes the state. A state convention similar to the one of the dynamic semantics would then give the rules in their full form. The idea of keeping track of type names using (solely) a state does not work well together with the constructor value idea described earlier, because constructor values of different evaluations of a datatype declarations would then have to be distinguished.

9.2 Too many type names are different

Any newly introduced datatype is attached with a "fresh" type name. Unfortunately, "completely fresh" is not always the right kind of freshness. Types of atomic patterns are mainly *guessed*, in particular the type of a variable pattern, rule 35. This rule could guess that the type of a variable includes type names which have not been introduced yet by type definitions.

However, those guesses may remain unresolved after a declaration has finished. Usually, one can replace all remaining unresolved guesses by bound type variables, making the declaration polymorphic. But this is not possible for declarations that only have weakly principal environments.

[7] The introduction of constructor values as described in part I, section 4.2, would change that. The example would instead evaluate to the packet [Match].

```
let
    val x = ref[]
    datatype a = B
    val y = x:=[B]
in  B
end
```

This example does not elaborate, because the last value declaration only elaborates if `x` has type `a list ref`, where `a` actually stands for its semantic value, i.e. its type name. Thus, the first value binding has to make this correct guess. The type name for `a` occurs then in the environment produced by the first declaration, which makes it non-fresh as far as the datatype declaration is concerned. All three declarations form a vicious circle.

But there is of course nothing wrong with the above declaration sequence from an intuitive point of view, and it would complicate the standard algorithm for type inference considerably to mirror the behaviour the Definition requires.

To adjust the static semantics, we had to be more explicit about the way type names are kept track of, i.e. we would not use $\oplus$ any longer for this purpose. For example, each declaration could explicitly produce a set of type names. Datatype declarations and abstract type declarations would be the only elementary declarations that produce non-empty sets of type names. For example, the rule for declaration sequences could then look like:

$$\frac{C \vdash dec_1 \Rightarrow E_1, T_1 \qquad C + (T_1, E_1) \vdash dec_2 \Rightarrow E_2, T_2}{C \vdash dec_1 \ \langle ; \rangle \ dec_2 \Rightarrow E_1 + E_2, T_1 \cup T_2} \tag{25}$$

In this form, the associativity of the semantic function for $\langle ; \rangle$ is easy to show: it follows directly from the associativity of $+$ on finite maps and of $\cup$ on finite sets.

That problem with the freshness of guessed types also appears in the suggested fix for `let`-expressions (last section), because the premise tynames $\tau \subseteq T$ of C would disallow τ to contain guesses of datatypes which have yet to be defined. Having sentences of the form $C \vdash dec \Rightarrow T, E$ gives us direct access to type names introduced by type declarations in *dec* and allows us to reformulate the `let`-rule to make it slightly more permissive:

$$\frac{C \vdash dec \Rightarrow E, T \qquad C + (T, E) \vdash exp \Rightarrow \tau \qquad \mathrm{tynames}\,\tau \cap T = \emptyset}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau} \tag{6}$$

Thus, type names *introduced by datatype bindings* in *dec* are not allowed in τ, but τ is allowed to guess type names not occurring in the static context.

A consequence of this slightly more permissive way of dealing with guessed types is that two claims in the Definition are not longer true:

- On page 23, the last two paragraphs before the rules have to be reformulated. One possible reformulation is their removal.
- At the beginning of section 5.14 (page 37), the third paragraph after "... the following Theorem can be proved:" is not longer true and has to be removed. Guessed type names would not enter the T component of a context, but at Module level all guesses have to be resolved, such that the other claims of the Theorem are not affected.

9.3 Non-expansive Expressions

A value binding is only allowed to bind imperative type variables if its body is a *non-expansive* expression. Non-expansive expressions are defined on page 20, section 4.7. In particular:

> Any variable, constructor and `fn` expression, possibly constrained by one or more type expressions, is non-expansive; all other expressions are said to be expansive.

This should probably read: "Any (possibly long) variable, value constructor and...". Without the insertion "possibly long", qualified identifiers had to be considered expansive, according to a general comment about qualified identifiers on page 4 — this is surely not the intended meaning. The insertion "value" has the purpose to disambiguate the term "constructor", which does not have a meaning on its own in the Definition. Remark: it does not affect the rules whether exception constructors are considered expansive or not, provided there are no polymorphic exceptions in the static context[8].

9.4 Principal Environments

On page 30, the Definition locally restricts the meaning of $\succ$:

> For the present section, $E \succ E'$ may be taken to mean $SE = SE' = \{\}$, $TE = TE'$, $EE = EE'$, $\mathrm{Dom}\, VE = \mathrm{Dom}\, VE'$ and, for each $id \in \mathrm{Dom}\, VE$, $VE(id) \succ VE'(id)$.

The "$SE = SE' = \{\}$" has to be replaced by "$SE = SE'$", because the structure environment a Core declaration elaborates to can be non-empty. This happens when the Core declaration opens a structure that contains substructures.

10 Static Semantics for Modules

10.1 Free Imperative Type Variables

On the bottom of page 44 there is a comment about rules 100 to 102:

> (100)–(102) The side-conditions ensure that no free imperative type variables enter the basis.

This is only true if "the basis" means here the basis for the elaboration of top declarations. Other bases, intermediately created for the elaboration of (for example) local structure declarations, may well contain free imperative type variables.

```
local
    val x = ref []
in
    val y = map (fn _ => 1) (!x)
end
```

[8] Exception constructors are never polymorphic in static contexts occurring in proofs for sentences of the form $B_0 \vdash program \Rightarrow B$, where B_0 is the initial static basis. But the Definition does not enforce the use of B_0 — see below the section about programs.

The above example should[9] elaborate as a top declaration in the initial basis. Parsing both value declarations as structure declarations leaves the elaboration of the first value declaration with a free imperative type variable, because rule 57 enforces principality. This type variable enters the basis for the elaboration of the second declaration (rule 59), but does not appear in the result of the whole declaration, which is $\{\texttt{y} \mapsto \texttt{int list}\}$. Thus, the side-condition of rule 100 is satisfied.

Remark: the side-condition in rule 101 is redundant, because signatures cannot contain free imperative type variables anyway.

The side-conditions have an unexpected effect (many implementations get it wrong) for the elaboration of sequential declarations as programs: sequential declarations separated by a semicolon have to be parsed separately as *topdec* (because of a restriction for top-level declarations on page 14), enforcing the side-condition separately for both declarations, whilst sequential declarations separated by space have first to be parsed as a single *strdec* (or *fundec*, *sigdec*), enforcing the side-condition only once.

```
exception A of '_a
exception A
```

elaborates successfully and

```
exception A of '_a;
exception A
```

has to be rejected.

10.2 Identifier Status

The Definition itself does not conclusively define how the status of an identifier (constructor, exception constructor, value) in an expression is determined in the presence of structures and signatures. As mentioned earlier (part I), the Commentary fills this gap with its appendix B and this part of the Commentary should be understood as a part of the Definition.

However, type-safety is affected by those status maps, although only in rather pathological cases in a negative sense. One can argue which part of the semantics is most closely related to this loss of type-safety — I prefer to relate it to the static semantics for Modules, because I prefer to rule out the pathological cases rather than to repair their behaviour. According to the rules, the following signature declaration should elaborate:

```
signature SIG =
    sig datatype t = A of int
        type u; sharing type t=u
        type t
        val f: u -> bool -> int
        val B: (bool -> int) -> u
    end
```

[9] Poly ML rejects it.

Although the `datatype` description of `t` has been overwritten by a later `type` description, the value constructor `A` remains part of the signature interface. Usually the requirement of *type explication* (rule 65) excludes to overwrite types that occur in the signature interface. In the example type explication is not violated, because `u` provides the required type structure containing the type name occurring in the result type of `A`. An important detail is that the status map obtained for the signature expression assigns constructor status to `A`, although `A` is not contained in the constructor environment of `u`. This becomes a problem in the following instantiation:

```
structure STRUCT: SIG =
    struct
        local datatype t = A of bool -> int
        in    type u=t
              fun f(A x) = x
              val B=A
        end
        type t=unit
        fun A x  = B (fn _ => x)
    end
open STRUCT
```

In the instantiation, the identifier `A` is realised by a non-constructor, but imposing `SIG` on `STRUCT` turns `A` into a constructor, which basically means that the function definition of `A` will be ignored. As a result, the (well-typed) expression `f (A 2) true` is not evaluated to 2, evaluation tries instead to apply 2 to `true`, i.e. to use it as a function.

Another consequence: the expression `(fn A x=>x)(B (fn _ => 2))*2` is wrongly treated. It successfully parses and elaborates, but then its evaluation tries to multiply a function with 2.

There are several ways to fix this problem. The most permissive is to turn constructor status into value status in a status map obtained from a signature expression, if the corresponding constructor (of a specified type) is not part of some constructor environment in the principal signature Σ of that signature expression. In the example, `SIG` would no longer assign constructor status to `A`, and the expression `f(A 2)true` would safely evaluate to 2. The other ill-treated sample expression would not even pass the syntax-check, as it uses `A` as a constructor in a pattern.

The methodological disadvantage of this solution is that it makes static analysis influence the syntax check. More in the spirit of the SML definition may be the following principle:

> A signature $(N)S$ is *constructor-explicit*, if for any substructure S' of S and any identifier *var* in $\mathrm{Dom}(VE \text{ of } S')$ that has constructor status, where $(VE \text{ of } S')(var) = \langle\tau \to\rangle\alpha^{(k)}t$ and $t \in N$, there is some substructure of S containing a type environment TE with $TE(tycon) = (t, CE)$ and $CE(var) = (VE \text{ of } S')(var)$ for some *tycon*.

In rule 65 of the Definition, we could then add another premise:

$$(N)S \text{ is constructor-explicit}$$

The effect of this new premise is to disallow "dangling" constructors — each constructor in a signature has to occur in a constructor environment of that signature, such that any structure matching the signature is forced to realise the specified constructors by "real" constructors.[10]

10.3 Exception Environments

The problem of the last section was caused by the lack of connection between value constructors in a variable environment and the constructor environment of their type. There is a similar problem with exceptions, caused by the lack of connection between exception constructors in a variable environment and the exception environment.

The purpose of exception environments in structure/signature matching is to require exception constructors to be matched by other exception constructors. Without this requirement there would be a misbehaviour of pattern matching, because rules 146, 147, 156, and 157 assume that looking up an exception constructor in the environment results in an exception name.

Unfortunately, this misbehaviour can still occur in some pathological cases:

```
signature EXC =
    sig
        exception A of int
    end;

structure S =
    struct
        exception B of int
        val A = fn x => B(x+1)
    end;

structure T: EXC =
    struct
        exception A of int
        open S
    end
```

The second structure binding is likely to be rejected in an implementation, because the exception constructor `A` appears to have been overwritten by a value variable, and signature `EXC` requires an exception constructor `A`. But this overwriting only took place in the *variable* environment, not in the *exception* environment. Thus, the exception

[10] A less sophisticated way to solve the problem is to disallow *any* overwriting of specifications, i.e. to require that the environments E_1 and E_2 obtained for $spec_1$ and $spec_2$ in a sequential specification $spec_1 \langle ; \rangle\ spec_2$ have disjoint domains, rule 81. However, occasionally the overwriting of specifications may be useful, e.g.

```
sig
    include SIG1; type u; sharing type u=t
    include SIG2
end
```

where the signatures `SIG1` and `SIG2` both specify a type `t`.

constructor `A` still exists in the *EE* component of the structure `T`, and the second structure binding should elaborate. The exception constructor `T.A` is now bound to a closure, not to an exception name. Therefore, the expression

```
(fn T.A x => x)(T.A 0)
```

is well-formed (`T.A` has status `e`), but it does not evaluate (not even to a packet), because rules 156 and 157 fail to find an exception name for `T.A` in the environment.

The implementations treat the example as follows: Poly ML and New Jersey ML do not elaborate the second structure declaration, complaining that the value `A` is not an exception constructor — they give exception environments less significance than the Definition does. Poplog ML elaborates successfully and it even evaluates the expression to 0 — it seems to work with a modified dynamic semantics for structure/signature matching.

A possible fix (along the lines of Poly ML) would be to eliminate exception environments altogether and instead to supply each entry in a variable environment with the information whether this is an exception constructor binding or not. In structure/signature matching we have to require that this attribute is preserved, similarly as a realisation has to preserve the equality attribute of types.

10.4 Others

Page 41, line -2:

`sharing s=t` $\longmapsto$ `sharing type s=t`

In the example, `s` and `t` are types, not structures.

11 Dynamic Semantics for the Core

11.1 Basic Values

The set of all basic values is defined in Section 6.4 on page 48 of the Definition. Basic values are functions not expressed by SML declarations, for example `=` or IO operations. In practice it is undesirable to have *all*[11] basic values specified by the Definition, because this prohibits implementations from providing further facilities of an operating system which are not expressible in terms of the other operations in BasVal. Implementations seem to ignore this restriction anyway.

11.2 Variable Environments

The correction in the Commentary about reducing the syntax does not go far enough (see section 4.1 on page 5 in this paper). A similar correction is necessary for abstract types:

[11] The Definition does not use the word "all", but Appendix D says: "We now describe the effect of APPLY upon each value $b \in$ BasVal.", indicating that there are no other basic values.

Page 46: remove the third bullet.

After 129.1 we add another rule for abstract types:

$$\frac{\vdash \mathit{datbind} \Rightarrow \mathit{VE} \qquad E + \mathit{VE} \vdash \mathit{dec} \Rightarrow E'}{E \vdash \texttt{abstype}\ \mathit{datbind}\ \texttt{with}\ \mathit{dec}\ \texttt{end} \Rightarrow E'} \qquad (129.2)$$

11.3 Application of Basic Values

There are two little problems with rule 116, the application of basic values, i.e. the rôle of built-in functions:

$$\frac{E \vdash \mathit{exp} \Rightarrow b \qquad E \vdash \mathit{atexp} \Rightarrow v \qquad \mathrm{APPLY}(b, v) = v'}{E \vdash \mathit{exp}\ \mathit{atexp} \Rightarrow v'} \qquad (116)$$

The rule implicitly assumes that the result of an application of a basic value is always a value. Appendix D makes clear that this is not always the case, the result may well be a packet ($\equiv$ raised exception), for example [`Div`] for division by zero. The exception convention does not apply here, because $\mathrm{APPLY}(b, v) = v'$ is not a sentence but a side-condition. An easy correction is to replace v' by v'/p.

There is another problem: the state convention does not apply too, for the same reason. This means that APPLY can neither depend on the state nor change it. For almost all functions, this is a safe assumption, but not for input and output — they clearly depend on the state, example:

```
val p = (open_in "file",1);
val x = input p
and y = input p
```

According to the semantic *rules*, `x` and `y` have to be bound to the same values, because they are evaluated in the same (SML) state and in the same environment. We can deduce this as follows: both `input p` are evaluated in the same environment, see rule 135. Expanding the state convention, it is also clear that both are evaluated in the same SML state. `input` evaluates in both cases to the basic value `input`, `p` is looked up twice in the same environment. The side-condition of rule 116 requires $\mathrm{APPLY}(b, v) = v'$, but for both applications of `input` b and v are the same, as we have already seen, and thus the two v' have to be the same too, by symmetry and transitivity of $=$.

The intention is surely different, as can be seen in appendix D — `x` and `y` are supposed to be bound to the first and second character of file `file`.

To mirror this intended behaviour, we have to extend the semantic class State by another component, the "outside world" W and allow APPLY to depend on it and to change it. The modified rule then looks as follows:

$$\frac{\begin{array}{c} s_1, E \vdash \mathit{exp} \Rightarrow b, s_2 \qquad s_2, E \vdash \mathit{atexp} \Rightarrow v, s_3 \\ \mathrm{APPLY}(b, v, W \text{ of } s_3) = v'/p, W' \qquad s_4 = s_3 + W' \end{array}}{s_1, E \vdash \mathit{exp}\ \mathit{atexp} \Rightarrow v'/p, s_4} \qquad (116)$$

In the appendix D, one could then be more specific what "outside world" actually means and how it is affected by the application of basic values.

There is one discrepancy between SML Definition and any SML implementation. All implementations provide a function `use` of type `string` $\rightarrow$ `unit` which reads declarations from a file. Such a function cannot *exist* in SML, because function application has no effect on the environment. Making such functions possible would require a non-trivial redesign of the dynamic semantics.

11.4 Others

A typo in rule 132 on page 53: the $longstrid_n$ must be $longstrid_k$.

There is a small oversight in the header of Exception Bindings (for rules 138, 139): a packet can never occur here, so the header should be:

Exception Bindings $\boxed{E \vdash exbind \Rightarrow EE}$

If the exception convention is expanded, this little change removes three redundant rules.

12 Dynamic Semantics for Modules

The Definition reduces the syntax for describing the Dynamic Semantics for Modules by the following convention:

- Qualifications "`of` *ty*" are omitted from exception descriptions.
- Any specification of the form "`type` *typdesc*", "`eqtype` *typdesc*", "`datatype` *datdesc*" or "`sharing` *shareq*" is replaced by the empty specification.
- The Modules phrase classes TypDesc, DatDesc, ConDesc and SharEq are omitted.

This is not correct, for similar reasons as datatype bindings cannot entirely be thrown out of the Core semantics. A datatype description gives rise to a variable environment in the static semantics and so it should evaluate here to a set of variable names — the domain of the static variable environment. One can easily show that the above principle loses type-safety:

```
val A = 2;
signature B = sig datatype t=A of bool end;
structure S:B = struct datatype t=A of bool end;
open S;
val C = A true
```

According to the Definition, the above declaration should elaborate (which is fine). However, the dynamic semantics defines the value of `A` in the last declaration still to be 2. The signature evaluates to the empty interface, because the datatype description is replaced by the empty specification. As a result, the structure evaluates to the empty environment, similarly `open S`. Thus, the old value of `A` is still stored in the environment when it comes to the evaluation of the last declaration and 2 is used as a function.

The correction is rather obvious. First, the bullets in the section 7.1 "Reduced Syntax" become:

- Qualifications "`of` *ty*" are omitted from constructor descriptions and exception descriptions.
- Any specification of the form "`type` *typdesc*", "`eqtype` *typdesc*" or "`sharing` *shareq*" is replaced by the empty specification.
- The Modules phrase classes TypDesc and SharEq are omitted.

We have to insert a rule for datatype descriptions (after rule 176).

$$\frac{\vdash \mathit{datdesc} \Rightarrow \mathit{vars}}{\mathit{IB} \vdash \texttt{datatype}\ \mathit{datdesc} \Rightarrow \mathit{vars}\ \mathrm{in\ Int}} \qquad (176.1)$$

Also rule 177 has to be changed, for similar reasons — obvious, if one compares it with rule 74. The new rule is:

$$\frac{\vdash \mathit{exdesc} \Rightarrow \mathit{excons} \qquad \mathit{vars} = \mathit{excons}}{\mathit{IB} \vdash \texttt{exception}\ \mathit{exdesc} \Rightarrow (\{\}, \mathit{vars}, \mathit{excons})} \qquad (177)$$

After the section with value descriptions (rule 184), we have to add two new sections and rules.

Datatype Descriptions $\boxed{\vdash \mathit{datdesc} \Rightarrow \mathit{vars}}$

$$\frac{\vdash \mathit{condesc} \Rightarrow \mathit{vars} \qquad \langle \vdash \mathit{datdesc} \Rightarrow \mathit{vars}' \rangle}{\vdash \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{condesc}\ \langle \texttt{and}\ \mathit{datdesc} \rangle \Rightarrow \mathit{vars}\ \langle \cup\ \mathit{vars}' \rangle} \qquad (184.1)$$

Constructor Descriptions $\boxed{\vdash \mathit{condesc} \Rightarrow \mathit{vars}}$

$$\frac{\langle \vdash \mathit{condesc} \Rightarrow \mathit{vars} \rangle}{\vdash \mathit{con}\ \langle\ |\ \mathit{condesc} \rangle \Rightarrow \{\mathit{con}\}\ \langle \cup\ \mathit{vars} \rangle} \qquad (184.2)$$

13 Programs

The Definition defines an initial static and dynamic basis, but it does not use these bases for anything. The only exception is section 3.6, which explains several restrictions an implementor may impose on modules. Hence, if an implementor does *not* want to impose any restrictions but implement the full language, then the initial basis seems to be not of his or her concern.

In particular, the section about programs does not require to start the execution of a program in the initial basis. Let us call for the rest of this section the basis in which the execution starts B_1, and the (combined static and dynamic) initial basis B_0.

It may probably be intended to allow $B_0 \neq B_1$, e.g. to encourage implementors to provide richer libraries. For several reasons this freedom of having B_0 and B_1 completely unrelated goes a bit too far:

- The semantic object class Basis includes many bases that are inconsistent in an intuitive sense. For example: the static and dynamic bases may be unrelated; closures in the dynamic basis may contain ill-typed expressions; the static environment may bind exception constructors to non-imperative type schemes; some type structures may not respect equality; some signatures may not be type-explicit, contain free names, etc.

- It is not clear that B_0 is included in B_1, i.e. it is not clear that the identifiers bound in B_0 are defined in B_1 as well, and even if they are, it is not clear that they have the same meaning.

- It is similarly unclear which status an identifier initially has — does it have infix status or not, is it a value variable, a value constructor, or an exception constructor?

The first of these points can be considered to be in the responsibility of the implementor. If he or she chooses a basis different from B_0 to start execution in, then it is up to him or her to make sure that this basis works smoothly with the rest of the Definition. However, it would be nice if the Definition provided explicit criteria B_1 has to match that guarantee sound behaviour.

The second point is annoying, because without this inclusion we cannot rely on the presence or even the given meaning of the predefined operations, and this is surely not the intent. We should require $B_0 \subseteq B_1$, where "$\subseteq$" is pointwise subset of the components (for tuples), and subset of the corresponding graphs for finite maps.

For practical reasons, it should also be required that the status maps of B_0 and B_1 agree on non-qualified identifiers, and that an identifier has exactly the same infix/nonfix status in B_0 and B_1. The motivation for this rather strong requirement can be seen in the portable definition of the identity function:

```
nonfix id x;
signature garbage =
    sig type garbage
        val id: garbage
        val x: garbage
    end;
structure garbage: garbage =
    struct datatype garbage = id | x end;
open garbage;
fun id x = x
```

"Portable" means here: will successfully elaborate and evaluate with the same meaning in any basis B_1. The mentioned restriction for non-qualified identifiers supports portable programs without this sort of `garbage`. Qualified identifiers are not involved in this problem, because they are never infix and because pattern variables are always non-qualified.

To connect execution of a program with B_1, we can do the following:

1. Define some properties of B_1 and the starting state s_1 in relation to the initial basis and state, e.g. along the lines sketched above.
2. Introduce a new syntactic class Root and a syntax rule:
   root ::= program
3. Add a rule section after rule 196:

   Root $\boxed{\vdash root \Rightarrow B, s}$

   together with the following new rule:

$$\frac{s_1, B_1 \vdash program \Rightarrow B, s}{\vdash program \Rightarrow B, s} \tag{197}$$

The purpose of all these changes and restrictions is to achieve the following portability goal: whenever a program successfully (i.e. without using rules 194 and 195) executes in the initial basis and state, then it does so in s_1, B_1 and the resulting bases agree on their common domain.

Notice that this new rule 197 also requires the definition of an initial *state*. We can define a state s_0 as $(\{\}, \mathrm{BasExName})$ and allow s_1 to be an extension of it, similarly as for the bases. The exception names and addresses occurring in B_1 should all occur (be defined) in s_1.

14 A Appendix: Derived Forms

The section does not say anything about the meaning of optional phrases in rewrite rules. This is a problem, as they can neither be treated as optional phrases in grammar rules nor as options in sentences. Therefore a clarification is desirable, for instance the following (to be inserted before "In the derived form for tuples", page 66, line 9):

> Each row that contains k optional phrases in the left column is an abbreviation for 2^k rules, one for each combination of presence or absence of the optional phrases. An optional phrase on the right-hand side of such a rule is present, iff the *corresponding* phrase on the left-hand side is. A phrase corresponds to itself and $\langle\texttt{:}\ sigexp\rangle$ corresponds to $\langle\texttt{:}\ sigexp'\rangle$.

We have to slightly change the table for function-value bindings (page 68, figure 17):

Function-value Bindings *fvalbind*

<table>
<tr>
<td>$\langle\texttt{op}\rangle var\ \ atpat_{11}\cdots atpat_{1n}\langle\texttt{:}ty_1\rangle\ \texttt{=}\ exp_1$
$\texttt{|}\langle\texttt{op}\rangle var\ \ atpat_{21}\cdots atpat_{2n}\langle\texttt{:}ty_2\rangle\ \texttt{=}\ exp_2$
$\texttt{|}\quad \cdots \quad \cdots$
$\texttt{|}\langle\texttt{op}\rangle var\ \ atpat_{m1}\cdots atpat_{mn}\langle\texttt{:}ty_n\rangle\ \texttt{=}\ exp_m$
$\langle\texttt{and}\ fvalbind\rangle$</td>
<td>$\texttt{op}\ var\ \texttt{=}\ \texttt{fn}\ var_1\texttt{=>}\ \cdots\ \texttt{fn}\ var_n\texttt{=>}$
$\texttt{case}\ (var_1\texttt{,}\cdots\texttt{,}var_n)\ \texttt{of}$
$(atpat_{11}\texttt{,}\cdots\texttt{,}atpat_{1n})\ \texttt{=>}\ exp_1\langle\texttt{:}ty_1\rangle$
$\texttt{|}(atpat_{21}\texttt{,}\cdots\texttt{,}atpat_{2n})\ \texttt{=>}\ exp_2\langle\texttt{:}ty_2\rangle$
$\texttt{|}\quad \cdots \quad \cdots$
$\texttt{|}(atpat_{m1}\texttt{,}\cdots\texttt{,}atpat_{mn})\ \texttt{=>}\ exp_m\langle\texttt{:}ty_n\rangle$
$\langle\texttt{and}\ fvalbind\rangle$</td>
</tr>
</table>

$(m, n \geq 1;\ var_1, \cdots, var_n$ distinct and new$)$

There are two changes: first, the `op` on the right-hand side is not longer optional but compulsory — otherwise we could not say whether it should be present or not if some of the `op` on the left are present and some are absent; the other change is that the type expressions have been supplied with indices. This makes a corresponding change necessary in the full grammar on page 72.

Supplying these type expressions with indices is a bit more liberal as it allows them to be *syntactically* different. The static semantics expects them to be *semantically* equal eventually, but this should not be handled on the level of syntax. For example, the old rule prohibits the following "declaration" to pass the syntax check:

```
fun f 0 : int = 1
| f x : (int) = x*f(x-1)
```

The problem is that `int` and `(int)` are syntactically different, i.e. different beings of the syntactic class Ty. Requiring syntactic equality is problematic anyway, because there are several levels of syntax: characters, and lexical items before and after expanding derived forms.

15 B Appendix: Full Grammar

On page 73, Figure 22, there is an (R) missing after "function type expression" — it was not forgotten in Figure 4.

The syntax rule for *fvalbind* does not describe all phrases that are supposed to reduce to *fvalbind*, because there is a note in Figure 20, page 72:

> Note: In the *fvalbind* form, if *var* has infix status then either `op` must be present, or *var* must be infixed. Thus, at the start of any clause, "`op` *var* `(`*atpat*`,`*atpat'*`)` $\cdots$" may be written "`(`*atpat var atpat'*`)` $\cdots$"; the parentheses may also be dropped if "`:`*ty*" or "`=`" follows immediately.

The syntax rule itself does not allow any form of infix notation for *fvalbind*, in other words: it is incomplete. Notice that the full grammar for expressions and patterns explicitly permits infix notation, so it should be made explicit here as well, for example as follows (introducing *fpat* as a variable for a new syntactic class Fpat):

$$
\begin{array}{lll}
\mathit{fvalbind} & ::= & \;\;\mathit{fpat}_1 \ \langle :\ \mathit{ty}_1 \rangle \ \texttt{=}\ \mathit{exp}_1 \\
 & & \texttt{|}\ \mathit{fpat}_2 \ \langle :\ \mathit{ty}_2 \rangle \ \texttt{=}\ \mathit{exp}_2 \\
 & & \texttt{|}\qquad \cdots \qquad \cdots \\
 & & \texttt{|}\ \mathit{fpat}_m \ \langle :\ \mathit{ty}_m \rangle \ \texttt{=}\ \mathit{exp}_m \\
 & & \qquad\qquad \langle \texttt{and}\ \mathit{fvalbind} \rangle \\
\mathit{fpat} & ::= & \langle \texttt{op} \rangle \mathit{var}\ \ \mathit{atpat}_1 \cdots \mathit{atpat}_n \\
 & & \texttt{(}\mathit{atpat}_1\ \ \mathit{var}\ \ \mathit{atpat}'_1\texttt{)}\ \ \mathit{atpat}_2 \cdots \mathit{atpat}_n \\
 & & \mathit{atpat}_1\ \ \mathit{var}\ \ \mathit{atpat}'_1
\end{array}
$$

The last alternative form for *fpat* corresponds to the last remark in the "Note". about dropping parentheses.

In this presentation, we have to additionally require that each *fpat* in an *fvalbind* has the same number of arguments, that is the index n of the *atpat* in the syntax rule for *fpat* is fixed for one *fvalbind*. This is also the reason for indexing "both arguments" of an infixed operator with 1, as they only constitute a single argument: a pair with two components.

In the syntax rule for an infixed *fpat* I have required the other components to be *atomic* patterns. This is more restrictive than the corresponding rules for *exp* and *pat*, but allowing arbitrary *pat* to be components of an infixed *fpat* would lead to some disambiguation problems — the precedence of an infixed value variable in an *fpat* had to be lower than any syntactic construct of *pat*, regardless of its fixity directive. One might consider the introduction of additional syntactic variables *apppat* and *infpat*, in analogy to *appexp* and *infexp*, to use *infpat* instead of *atpat* at the appropriate places in the rule for *fpat*. This would still leave the infixed value variable in *fpat* with a lower precedence than any infixed constructor, because "precedence does not decrease the class of admissible phrases".

The implementations do not agree on this matter — Poly ML accepts infix patterns as components of an infix *fpat*, New Jersey ML and Poplog ML do not. However, Poly ML requires the precedence (given by a fixity directive) of the infixed value variable to be lower than the precedences of the infixed constructors in the arguments.

16 C Appendix: The Initial Static Basis

The second bullet in Figure 23 on page 75 describes the resolution of overloading and what the occurrences of `num` in Figure 23 stand for.

Strictly speaking, a static environment having the properties required in the second bullet *cannot exist*, i.e. it cannot be built out of the semantic objects defined in section 4 of the Definition. There is simply no type scheme σ such that $\sigma \succ \mathtt{int} \to \mathtt{int}$ and $\sigma \succ \mathtt{real} \to \mathtt{real}$, but $\sigma \not\succ \tau$ for any other τ.

To express overloading within the semantics we could do the following changes:

- On page 16, section 4.1, line -5, add at the end:

  There is a distinguished type variable `num`.

- On page 19, section 4.5, line 3, add after "imperative if α_i is imperative" the following:

  ; if $\alpha_i = \mathtt{num}$, then $\tau_i = \mathtt{int}$ or $\tau_i = \mathtt{real}$

- Finally, on page 75, we remove the second bullet and add to any type scheme containing a `num` the prefix "$\forall$`num`.".

Notice that these changes affect the existence of principal environments. We simply have to drop the claim of their existence (in section 4.12, page 30). As a consequence of that, rule 57 would be the natural place where overloading has to be resolved. Declarations that do not have a principal environment are declarations for which overloading cannot be resolved. This requires no change — such a declaration simply fails to satisfy the side-condition of rule 57.

Examples like the following are disallowed:

```
local
    structure s = struct fun g x = x+x end
in
    val h = s.g 2
end
```

There does not seem to exist any SML compiler that allows the above example anyway.

Another consequence of the change is that overloading in Core declarations *is* allowed, i.e. it is not longer a question of the mercy of the compiler writer. The following example is currently refused by Poly ML and Poplog ML, but accepted by New Jersey ML and the ML Kit compiler:

```
let fun g x = x+x
in  g 2
end
```

The described change means that the example *has to be* allowed.

Notice that the above approach does not distinguish between different *dynamic* values for `+`, etc. This is fortunately not necessary, because the Definition can consider `int` and `real` as disjoint sets, allowing `+` to be expressed as one function.

Another effect of dropping the (general) existence claim of principal environments is that disambiguation of wildcard pattern rows (syntax: `...`) could be located at rule 57. Currently, section 4.11 requires the type of a wildcard pattern row to be determined by "the program context". Taken literally, this mild requirement is quite a task for implementors, as a "program" properly includes top declarations.

Even disambiguation at rule 57 is rather subtle, because type inference (that is: the implementation of type checking, replacing guesses by logical variables) on the Core level has then to be able to deal with incomplete record types and type schemes. For instance, if an incomplete record type scheme σ is instantiated to an incomplete record type τ and then τ is unified with another (perhaps incomplete) record type, then this unification may extend the record domain of τ *and* σ.

Remark: enforcing principality (rule 57) or not (core semantics) are not the only two possible options to handle overloading. For example, one could require (or: allow implementations to require) the *existence* of a principal environment at rule 17 (value binding). This would mean that both kinds of overloading had to be resolved at this place. But there is a subtlety — it might interfere with guessing imperative types (weakly principal environments), a correct guess could resolve overloading:

```
val a = ref []
val b = fn y => case !a of x::xs => x+y | [] => y
val c = a:=[2]
```

Although *type inference* cannot resolve the overloaded `+` before the declaration of `c`, the *type system* does it earlier by having correctly guessed the type of `a`. In other words: overload resolution in type inference does not quite coincide with requiring the existence of principal environments.

17 D Appendix: The Initial Dynamic Basis

A problem concerning the initial dynamic basis is the way functions on integer or real numbers are defined, see [Har92]. The Definition says about special values in section 6.2: "Each integer or real constant denotes a value according to normal mathematical conventions." In appendix D, `sqrt`(r) is defined as: "returns the square root of r, or the packet [Sqrt] if r is negative."

Normal mathematical convention is that a real number is ... a real number, as opposed to a floating point number that approximates it. Of course, it was not the intention of the authors of the Definition to force some unusual powerful representation of some subset of real numbers that is closed under ordinary arithmetic, square root, natural logarithm and powers of e — equality would hardly be decidable, for instance.

To make the intentions clear, the Definition should rather refer to IEEE standards or the ISO standard for language independent arithmetic. This does not only affect the arithmetic operations, but also the meaning of special constants. Notice that it has to be explained what happens to overly large or overly precise special constants, possible choices are: compile-time error, run-time exception, rounding, truncation, etc.

For integers there is the similar problem that "normal mathematical conventions" for them do not know the concept of a "range". The Definition uses the term "out of range" without ever introducing it.

Acknowledgement

People who have knowingly or unknowingly contributed to this paper are: Bill Aitken, Andrew Appel, Dave Berry, Rob Harley, Robin Milner, Nick Rothwell, Don Sannella, Jon Thackray, and Mads Tofte.

References

[App92] Andrew W. Appel. A Critique of Standard ML. Technical Report CS-TR-364-92, Princeton University, 1992.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Har92] Rob Harley. Formalizing SML's arithmetic by tracking cumulated error bounds. CS4 Report, University of Edinburgh, 1992.

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

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

[Rot92] Nick Rothwell. Parsing in the SML Kit. Technical Report ECS-LFCS-92-236, University of Edinburgh, LFCS, 1992.