Mistakes and Ambiguities in the Definition of Standard ML – Addenda –

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

Version of December 18, 1996

Abstract

This paper lists errors in the definition of Standard ML [MTH90] that have been discovered since [Kah93] came out; it should be read together with that report. I also discuss some problems related to problem solutions suggested in [Kah93].

Page and rule numbers below refer to [MTH90] except where indicated.

Errors discovered since the 29th June 1994 (the first version of this document) are especially indicated.

Syntax for the Core

The syntax of a layered pattern is defined as *var* `as` *pat* (if we leave out the optional parts). However, this suggests that the following is defined despite the use of a constructor as *var*:

```
datatype t = A | B
val A as B = B
```

Implementations usually reject this, because layered patterns are typically implemented as *pat* `as` *pat* to resolve a shift-reduce conflict in the grammar (see section 8.4 in [Kah93]); an exception is MLKit which indeed allows the above declaration. A justification for the restriction is the claim on page 5 that "no value declaration can make a hole in the scope of a value or exception constructor". The above declaration does just that.

*The research reported here was partially supported by SERC grant GR/J 07303.

Syntax for Modules

Page 10, section 3.2, first line: "syntax classes" $\mapsto$ "identifier classes".

Static Semantics for the Core

The operator "Abs", defined in section 4.9, needs an additional argument (e.g. a static context C) to make sense of the notion of "new distinct type names". This requires a corresponding change in rule 20 when "Abs" is used.

In section 4.10, page 23, "where A is usually an environment or a context" should be replaced by "where A is usually a context", because it is actually *never* an environment (analogously 2 lines below).

On the same page, the "precise result" that "may easily be demonstrated" does not actually follow from the rules: elaboration of core phrases does not involve any principality requirements and so we *can* elaborate the declaration

```
fun I x = x
```

in a context C obtaining the environment $\{\texttt{I} \mapsto t \to t\}$ where t is a nullary type name not in C.

A serious soundness problem is in rule 16 on page 25: the use of + instead of $\oplus$ in the premise of that rule allows the type of formal parameters of a function abstraction not to be distinct from types local to its body. It is a bit difficult to come up with an example where this causes problems, provided `let`-expressions are disallowed to generate free names (as suggested in [Kah93]; without the proviso it's easy), but here is one:

```
fun d b f =
     if b then
            let datatype A = C of int
            in  f (C 2)
            end
     else   let datatype A = C of bool -> bool
            in  d true (fn C x => x true)
            end
val xxx = d false (fn _ => true);
```

The problem is: the type of `f(C 2)` is simply the result type of `f` (`bool`), i.e. the type name for A is not contained in it and the side-condition for `let`-expressions is satisfied. The same applies for the recursive call (`d true` ...).

The type names of both A can be shared because they are type-checked in the same static context. These type names escape the `let`-expression via the formal parameter `f`. The type of `d` is `bool` $\to$ (`A` $\to$ `bool`) $\to$ `bool` where `A` is the escaped (shared) internal type. The soundness of the type system is compromised, because the evaluation of the declaration of `xxx` leads to the application of 2 to `true`.

This problem can be solved by replacing + in the premise of rule 16 by $\oplus$: the example will not type-check, because the two datatypes can no longer share their type name. Notice that this is only a problem of the semantics, not of implementations. All

SML implementations use a global counter for the generation for new type names; as a consequence, the two branches of the `if`-expression are elaborated in *different* contexts.

There are a couple of problems with sequences: rules 28 and 29 on page 27 are problematic as they identify syntactic sequences *tyvarseq* with semantic sequences $\alpha^{(k)}$. The former (defined on page 7) involve *syntactic* parentheses and commas, the latter do not. Related to that is a problem with rule 49 on page 29: the premise $tyseq = ty_1 \cdots ty_k$ is ill-formed if $k > 1$. Strictly speaking, one needs three rules here to cover the three alternatives of sequences. The situation is slightly worsened by the ambiguity of the case $k = 1$.

NEW ERROR: 13 SEPTEMBER 1996

The Abs operator causes a problem w.r.t. principal environments. Consider the following example:

```
abstype rational = A of int * int
    with fun form(m,n)=A let val p=gcd(m,n) in (m div p,n div p) end
         fun eq(x,y) = x=y
    end
```

What is the principal type of `eq` after the `abstype` declaration? (All) implementations of SML will tell you: `''a*''a->bool`. This means that we cannot use `eq` for type `rational`, since abstract types do not admit equality. However, *within* the `abstype` definition the type `rational` is an equality type and the type system could have chosen `rational` instead of an equality type variable when instantiating the type of the polymorphic equality. This choice would have resulted in an environment in which we *could* use `eq` on type `rational`. Notice that we can force this effect by a type assertion in the definition of `eq`, e.g. by writing `y:rational` instead of `y` in the pattern.

What this means is that the type system really has a choice between giving `eq` the type `rational*rational->bool` and the polymorphic equality type, but it has no principal choice as `rational` is no longer an instance of `''a` after the `abstype` declaration. Therefore such declarations have to be rejected when principality is imposed (because they have no principal environment), and otherwise such functions have to be treated as overloaded. From an implementors point of view this request is not really acceptable.

NEW SOLUTION: 18 DECEMBER 1996

There is a solution to this problem which is surprisingly simple and fixes another problem at the same time. Do not allow abstract types to be equality types within their own definition! Technically this means to replace the premise "TE maximises equality" in rule 20 by the opposite condition which ensures that none of the fresh type names in TE is an equality type. As a consequence, the semantic Abs operation can be simplified, or even abandoned as its only remaining purpose is to remove the constructor environments from TE; we can eliminate the need for Abs, since this simplification can be done *before* evaluating *dec* — this is possible, because as a core declaration it cannot make any use of the constructor environments on its own.

The other problem which is fixed by this modification is that it is no longer possible to create a type structure that does not respect equality. Under the current semantics this arises in situations like the following:

```
abstype t = C with datatype u = D of t end
```

Under the modification, neither `u` does not admit equality, in the current semantics it does. It should be mentioned that this is not a soundness problem, only a methodological one. In a certain sense the current meaning of `abstype` is not abstract enough, some information about the implementing type can sneak out.

Static Semantics for Modules

Section 5.3, page 32, fails to define well-formedness of type environments, although it uses it. Such a notion does not make much sense, as type environments do not contain name bindings. However, it would make sense to include type structures in the well-formedness requirement of signatures, i.e. to require that if (t, CE) is a type structure occurring in $(N)S$ and $t \notin N$ then $\mathrm{names}(CE) \cap N = \emptyset$. Implementations usually do not impose this restriction.

New Error: 13 February 1995
The side-condition in rule 55 (functor application) is too weak: it should be $(N \text{ of } B \cup \text{names } S) \cap N' = \emptyset$, i.e. the fresh names in N' have also to be fresh w.r.t. the argument structure. Here is an example that shows how the weaker condition allows unsound elaboration:

```
functor F(type t; val g:t->bool) =
    struct
        datatype t = D of int
        val g = g
    end;
structure A =
    F(datatype t = D of bool->bool; fun g (D x) = x true)
```

If the above elaborates *and* the type names of both datatypes are shared (possible under the weaker condition) then the expression `A.g(A.D 2)` elaborates and leads to the application of `2` to `true`.

The comment for rules 66 and 96, about how the closure restrictions of section 3.6 could be enforced by modifying these rules is not quite right: for example in 66, it suggests replacing B in the premise by $B_0 + G \text{ of } B$. But this is not compatible with the full semantics, because B_0 may not be contained in B as identifiers of the initial basis (apart from `=`) can be re-bound. Thus, to enforce the restriction B should be replaced by $(B_0 \cap B) + G \text{ of } B$.

There is a subtle problem with bound names in rules 95 and 99. Page 31 says: "The prefix (N), in signatures and functor signatures, binds both type names and structure names." This remark suggests that objects like $(N)S$ are considered modulo α-congruence and some definitions (like application of realisations) indeed underline this perception. Quoting section 5.13, "a signature $(N)S$ is *principal for sigexp in* B if, choosing N so that $(N \text{ of } B) \cap N = \emptyset$,..." Now the problem is that a signature $(N')S'$ not satisfying the proviso is still principal if it is α-congruent to another principal signature. But this is not good enough for rules 95 and 99: both rules use S from $(N)S$ without the binder. Thus, both rules need the additional side-condition $N \cap (N \text{ of } B) = \emptyset$.

Dynamic Semantics for the Core

Page 47, definition of VarEnv: Var $\mapsto$ (Var $\cup$ Con $\cup$ ExCon). This change makes the presentation consistent with the static semantics. Neither static nor dynamic semantics distinguish identifier classes in the domain of a variable environment.

Rule 126, page 53, should have E, v instead of E in its conclusion.

Dynamic Semantics for Modules

Rule 185, page 61: in the conclusion, the keyword `and` is missing.

Derived Forms

The definition of derived forms for lists in figure 16 on page 67 does not say what happens to an expression `[1,2]` if the meaning of the identifiers `nil` and `::` changes. One can observe implementations doing different things: in SML/NJ (and MLKit), the expression `[1,2]` leads to a type error after the declaration

```
datatype t = nil of int | :: of int
```

while PolyML sticks to the type `int list`. However, SML/NJ and MLKit do not go all the way and claim a type error for `[1,2]` if the infix status of `::` is changed to left-associative, etc. Related to that is a weird bug in SML/NJ that occurs when one tries to open a structure in which `::` does not have constructor status.

Full Grammar

As opposed to the grammar of the bare language, the full language grammar employs auxiliary non-terminals *infexp* and *appexp* to deal with infix expressions, disallowing certain expressions in the full language. In addition to expressions, a similar construction is necessary for patterns (*infpat, apppat*, etc.), otherwise one could write patterns like `x:t ++ y:u` if `++` is an appropriate infix constructor. This is surely not intended, and it is not supported by any implementation I know.

Remarks on the Corrections in [Kah93]

Correcting bugs in software is not a one-time activity, in particular bug-fixes have the tendency to introduce further bugs. Unsurprisingly, this also happened to my corrections in [Kah93]; fortunately, all necessary amendments are minor.

On page 15 in [Kah93], I clarified how the word "precedence" has to be understood, i.e. that a sequence of lexical items reduces to a non-terminal (phrase-class) by the alternative form with *lowest* precedence. This works fine with most of the grammar; however, all forms of declarations include the empty word in two ways, as the empty declaration and as the concatenation (with omitted ";") of two empty declarations. So we have to give the empty declaration lower precedence than declaration sequencing to

achieve the desired effect. In the grammar, this means listing sequential declarations *before* empty declarations.

In section 10.2, page 21f, I explained a problem regarding identifier status and suggested two different ways of fixing it. There is a third solution, which is probably better and closer to practice. This solution adds status information to static environments and modifies the notion of enrichment accordingly, such that constructors are guaranteed to be implemented by constructors. But this may not be the best form of presentation, since the static semantics would be messed up with status information; for the semantics of Extended ML [KST94], we chose therefore to incorporate the treatment of identifier status in the semantics for derived forms.

Acknowledgement

People who have knowingly or unknowingly contributed to this update are: James McKinna, Claudio Russo, Don Sannella, Andrzej Tarlecki and Mads Tofte.

References

[Kah93] Stefan Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, 1993.

[KST94] Stefan Kahrs, Don Sannella, and Andrzej Tarlecki. The definition of Extended ML. Technical report, University of Edinburgh, 1994. Draft.

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