Miscellaneous Design Issues in the ML Kit

Nick Rothwell

September 10, 1992

1 Introduction

This document covers a number of assorted design topics involving the ML Kit. These issues are sufficiently small that they do not merit reports of their own, and yet it is important that they be covered in order to gain a more complete understanding of how the Kit operates. The sections are as follows:

Section 2 covers the evaluation of the Modules language and the implementation of the associated dynamic environments;

Section 3 describes some of the higher levels of the interpreter, and the way that the execution and evaluation heirarchy is structured;

Section 4 describes the mechanism used for top-level printing;

Section 5 details the implementation of the pervasive functions of the Kit, corresponding to APPLY in the Definition;

Section 6 describes the detection and reporting of elaboration errors;

Section 7 describes file nesting and inclusion, and the semantics and implementation of the `use` function;

Section 8 describes the implementation of records, specifically the notion of flexible record which is needed to support the "{...}" notation in record patterns;

Section 9 lists the incomplete and unimplemented parts of the system.

2 Evaluation of the Modules

`EvalTopdec` is the evaluator for the Modules language. Its signature (`EVALTOPDEC`) is shown in Figure 1. This signature is common to both the Interpreter and the Compiler, since they share the same interface; `EvalTopdec` is specific to the Interpreter. (The Compiler has its own implementation.) Top-level declarations are

```
signature EVALTOPDEC =
  sig
    type topdec
    type DynamicBasis
    type Pack

    val RE_RAISE: Pack -> unit

    exception UNCAUGHT of Pack
    val pr_Pack: Pack -> string

    val eval: DynamicBasis * topdec -> DynamicBasis
    val FAIL_USE: unit -> unit

    type StringTree
    val layoutDynamicBasis: DynamicBasis -> StringTree
  end
```

Figure 1: The signature `EVALTOPDEC`

evaluated in a basis, yielding a new basis just as in rules 191 to 193 of the Definition. Any exception raised in the evaluation which propagates to the top level causes the meta-exception `UNCAUGHT` to be raised with a packet. (The exception `UNCAUGHT`, as well as the type `Pack` and the other exception manipulation facilities, are actually propagated from `EvalDec`.) `EVALTOPDEC` also contains pretty-printing functions for printing dynamic bases.

The other functions in `EVALTOPDEC` (`RE_RAISE` and `FAIL_USE`) relate to the implementation of the built-in function `use`, and are documented in Section 7.

The `EvalTopdec` functor takes as arguments the abstract syntax (signature `TOPDEC_GRAMMAR`), the dynamic objects of the Modules (signature `MODULE_DYNOBJECT`), the Core level evaluation machinery (`EVALDEC` and `CORE_DYNOBJECT`), and other utility structures. The top-level grammar is resolved such that identifier classes are known; this must be the case because `EvalTopdec` requires that the declaration type (`dec`) of `TOPDEC_GRAMMAR` shares with that of `EvalDec`, which requires a resolved grammar.

The implementation of Modules execution (functor `EvalTopdec`) follows fairly directly from the rules in the Definition. The topmost function `eval` corresponds directly to the final three Modules evaluation rules of the Definition (191 to 193). The lower level functions are mutually recursive, and correspond to the rest of the semantic rules, as follows:

`evalStrexp` (structure expressions) for rules 160–163;

`evalStrdec` (structure declarations) for rules 164–168;

`evalStrbind` (structure bindings) for rule 169;

`evalSigexp` (signature expressions) for rules 170 and 171;

`evalSigdec` (signature declarations) for rules 172–174;

`evalSigbind` (signature bindings) for rule 175;

`evalSpec` (specifications) for rules 176–183;

`evalStrdesc` (structure descriptions) for rule 186;

`evalFunbind` (functor bindings) for rule 187;

`evalFundec` (functor declarations) for rules 188–190.

Rules 184 (value descriptions) and 185 (exception descriptions) are essentially dealt with "in line" in `evalSpec`, using the operations `Vars_in_Int` and `Excons_in_Int` (from `ModuleDynObject`) respectively. Rules 191–193 are rolled into the main function `eval`. The Inter and $\downarrow$ operations are implemented in `ModuleDynObject`.

3 Linking

This section describes the very top level of the system, and the way that the execution and evaluation levels are linked together. These levels are above the Modules evaluator described in Section 2, and are in fact common to both the "pure" interpreted system and the lambda compiler. A companion report [3] gives more insight into the way that the execution levels for the compiler and interpreter are abstracted so as to appear identical to the rest of the system; we will only discuss this abstraction where it relates to the rest of the system.

The top-level of the system is a functor called `KitCompiler`. It takes as argument a string which is the name of a file to read as the *prelude* (Section 5); it has no other arguments. It returns a structure with the signature given in Figure 2. `KitCompiler` is also responsible for "tying the knot" in order to implement the `use` function, in the manner described in Section 7.

The main constituents of `KitCompiler` are `Linking`, which brings together the lower levels of the Kit together with the parsing, elaboration and evaluation levels, and `Interpreter`, which implements the top-level loop and some rudimentary file redirection. In its turn, `Linking` calls on the highest level functors for parsing (`TopdecParsing`), the static semantic objects of the language (`StaticObjects`), the elaborator (`Elaboration`) and the evaluation stage (`Execution`).

At this stage we should explain that `Elaboration` and `Evaluation` are regarded as *linking functors*, and form part of the "spine" which ties together the entire Kit. The spine functors live in the file `KitCompiler.sml`, and build various levels of the Kit (`Tools`, `Basics`, `TopdecParsing`, `StaticObjects`, `Elaboration`, `Execution`, `Linking`, `KitCompiler`). The style of these functors differs from those which comprise the body of the Kit proper: the linking functors refer to the functors of the Kit freely, and each linking functor refers to the linking functor in the level below it. Each linking functor contains substructures obtained by applying the linking functors below it; at each level, access might be required to several stages of the linkage, and the functors cannot be applied each time without the resulting types and structures violating the sharing constraints required by the rest of the Kit.

There are also other linking functors which do not form a part of the main spine. The lambda compiler contains a small number of linking functors (one for the top level of the compiler and one for the match compiler), and `Evaluation` is also a linkage stage. This is because it is at the level of `Evaluation` that the Compiler and Interpreter versions of the Kit diverge. There are two `Evaluation` functors, with the same result signature and the same set of arguments (although neither functor is required to use all of them). Each `Evaluation` functor is a linkage functor for its respective subsystem. The Compiler's `Evaluation` functor is described in the compiler paper [3]; we consider only the Interpreter version here.

In fact, the internal structure of `Evaluation` is simple. It draws together the elements needed for Modules-level evaluation (basic and special values, dynamic objects for the Core and Modules, evaluators for Core and Modules, and top-level printing). It exports a structure for the dynamic basis (signature `DYNAMIC_BASIS`), a top-level declaration evaluator (`EVALTOPDEC`) and top-level value printing (`VAL_PRINT`)—these are the three elements whose implementation differs between the Compiler and the Interpreter.[1] We describe `ValPrint` in Section 4.

The signature `INTERPRETER` is shown in Figure 3. The functionality provided is roughly equivalent to the interface provided by `KitCompiler`. `Interpreter` performs its own stream handling (indirectly, via the operations provided by `Parse`). It provides the persistent top-level basis (all the lower levels of the system are "pure" in that they take environmental objects as arguments and return new, incremental ones as results). It initiates parsing, elaboration and execution, and detects and reports errors in any of the three phases.

[1] In fact, `ValPrint` should be common between them, and use a lower-level runtime value traversal module with a common interface between the Compiler and the Interpreter. At present, `ValPrint` just inspects the runtime representation directly, with no reference to the static types of values.

```
functor KitCompiler(val prelude: string)
   : sig
      val parse: unit -> unit
      val elab: unit -> unit
      val eval: unit -> unit

      val parseFile: string -> unit
      val elabFile: string -> unit
      val evalFile: string -> unit
   end =
struct
   ...
end
```

Figure 2: The functor `KitCompiler`

```
signature INTERPRETER =
   sig
      datatype Mode = PARSE_ONLY | ELABORATE_ONLY | EVALUATE

      val interpretStdIn: Mode -> unit -> unit
      val interpretFile: Mode -> string -> unit
      val interpretString: Mode -> string -> unit
   end
```

Figure 3: The signature `INTERPRETER`

4 Top-level Printing

Any SML system needs a way of reporting top-level bindings to the user. This involves two processes: the printing of dynamic values, and the printing of static environments and types. Some of the entities to be presented to the user are, for our purposes, purely static in nature. For example, signatures and functors are reported merely as static objects; their dynamic (runtime) representation is not used. Other entities contain both a static and a dynamic component: most core-level value bindings, for example, are reported as a combination of value and type. (Those which are not include functions, and values with abstract types.)

The printing of values is not simply a case of presenting the static and dynamic components in a neat and tidy way: for example, the dynamic component might be irrelevant, and the static component might need to be reorganised in order to print it in the way that the users wants to see it, which usually corresponds to the source language syntax (an obvious example is functor bindings).

Value printing is driven at the top-most level by the traversal of a basis, which contains a static basis and a dynamic basis (as described in the Definition), as well as the infix basis. `TopLevelReport.report` takes a basis as argument, along with a flag indicating whether dynamic values should be printed as well: if the Kit is being run purely as an elaborator, then there will be no values to print. The result of printing a top-level basis is a *report* (as described in Section 6), which is a representation of the lines to be printed. The report is generated by a traversal of the infix basis (in order to report infix bindings) onto which is appended a traversal of the static and dynamic bases.

In theory (and assuming that both bases are present), the printout is driven by a traversal of the static basis and the dynamic basis in parallel. The traversal is a recursive one over both data structures, although the dynamic basis is ignored for the upper levels of the language (since the Modules objects are not printed as dynamic values at all). The dynamic basis is used when the data values of the core are reached: most of the core values need to be printed as a (dynamic) value with a (static) type.

In practice, this parallel traversal is decoupled and the printout is driven by a traversal of the static basis, with the dynamic basis being traversed at the leaves of the static basis. This is a simpler solution because dynamic basis traversal is sufficiently simple that it can be detached from static basis traversal (which is quite complex), and also makes it easy to report a top-level basis when no evaluation has been done, or where there is no dynamic entity corresponding to the static one (for example, in the case of signatures and functor bodies).

The reporting functions (in `TopLevelReport`) are mutually recursive, and take as argument a function called `render` for performing a dynamic basis traversal at the appropriate leaves of the static basis. For example, the type of `reportStr` is (if we were to label the arguments):

```
signature VAL_PRINT =
   sig
   type id and strid
   type DynamicBasis
   type TypeScheme
   type Val

   val locate:  DynamicBasis * strid list * id -> Val
   val print:  Val * TypeScheme -> string
end
```

Figure 4: The signature VAL_PRINT

```
    {render: StrId list * id * TypeScheme -> string,
     pathR: StrId list,
     str: Str,
     bindings: bool
    } -> Report
```

`render` takes a path, an identifier and a type scheme, and prints the dynamic value of that identifier, accessed via the path from the topmost dynamic basis, according to its type as indicated in the type scheme. `pathR` is a (reversed) record of the nested structures traversed to reach the leaf node. The other arguments vary between the different reporting functions, but the argument `bindings`, if present, indicates whether the dynamic value should be retrieved via the `render` function and printed.

`TopLevelReport` makes calls to lower-level functions in other modules. `iterateVE` (in `Environments`) is a generic traverser of value environments; `iterateSE` is the equivalent for structure environments. `ModuleEnvironments` provides traversers for signature and functor environments (which take callback functions for dealing with the individual signatures and functor signatures). Finally, `ValPrint` (whose signature is given in Figure 4) provides the ability to traverse a chain of structure identifiers in a dynamic basis yielding a dynamic value, and to print a value according to a type scheme. These are the two operations which will vary in implementation between the interpreter and the compiler.

5 Implementation of the Pervasives

Let us refer to the identifiers that are defined in the initial basis of Standard ML as *pervasives*. The pervasives are listed in Appendices C and D of the Definition;

they include type constructors (e.g. `int`), value constructors (e.g. `true`) and value variables (e.g. `map`). They also include the basic values, such as `+` and `open_in`.

Some of the pervasives are expressible in terms of ML declarations that use a smaller set of pervasives; this applies to `map`, for example. Such pervasives are defined in a *prelude* which the Kit reads when building itself.

Other pervasives cannot be dealt with in this way; for instance, `+` is a pervasive value which cannot be expressed in terms of more primitive operations. We say that such pervasives are *primitive*. Pervasives are used by the Kit (for example, `true` is needed in the expansion of derived forms), so pervasives must be accessible to the Kit itself.

In principle, the primitives should include all the basic values (BasVal is defined in [1, Section 6.4]). However, the large number of basic values is a problem in a real implementation. It would be very inconvenient if each basic value were to have its identifier built into the system (separately as an identifier and as a variable), its type built into the elaborator, its value built in as a basic value and its operation built into the APPLY function.

Therefore, the Kit has a special built-in value, called `prim`, which is used to implement most of the basic values in a uniform way. More precisely, most of the basic values (i.e. the members of BasVal) are declared in the prelude, using calls to `prim`. `prim` has type

```
(int * 'a) -> 'b
```

where the `int` indicates which primitive function is required. It is essentially overloaded, implementing a large number of primitive functions according to the value of its first argument (rather than according to its type, which is the usual discriminant for overloaded operations).

Figure 5 gives a small portion of the prelude, showing some uses of `prim`. The second argument to `prim` is either a single value or a tuple expression, depending on the number of arguments the primitive operation takes. Each use of `prim` has to be accompanied by type constraints, since it is a type-unsafe construct, and since the pervasives must be given the types dictated by the Definition. Note also that exception-raising pervasives take exceptions (i.e. values of type `exn`) as extra arguments; the implementation of APPLY can raise these exceptions as required. The equality predicate is implemented using `prim`. (It is prim zero, a fact assumed by the lambda code generator.) The prelude makes all the standard fixity declarations, declares all the pervasive exceptions[2], and declares the types `unit`, `bool` and `list`.

There are a number of things that the prelude cannot implement. These are the following:

- The following types: `int`, `real`, `string`, `exn`, `ref`. The types `int`, `real` and `string` need to be built-in because ML has constants of those types.

[2]including `Match`, `Bind` and `Interrupt`, although these are not known by the interpreter at present

```
fun op = (x: ''a, y: ''a): bool = prim(0, (x, y))
    and abs(x: int): int = prim(1, (x, Abs))
    and floor(x: real): int = prim(2, (x, Floor))
    and real(x: int): real = prim(3, x)
       ...
```

Figure 5: Use of `prim` in the prelude

In addition, the type of `prim` contains `int`. `exn` is built-in because of the primitive nature of the exception mechanism. `ref` is built-in because of the special nature of the `ref` constructor, and also because the `ref` type constructor is treated specially when testing types for equality admission.

- The following values: `ref`, `prim`. The need to predefine `prim` should be obvious. `ref` has special static and dynamic properties. Statically, it has type `'a -> 'a ref` in patterns and `'_a -> '_a ref` in expressions. Dynamically, it creates a new address on application, and has a specific semantic rule (rule 114) indicating this.

In addition, some of the non-primitive identifiers are known internally by the system. These are `nil`, `::`, `true`, `false` and `it`. These are needed for derived-form expansion (specifically, `if/then/else`, list expressions, and top-level expressions). All except `it` are needed for the dynamic evaluation phase as well, since some of the primitive operations return boolean constructors or lists. Note that the types (and identifiers) `bool` and `list` are not known internally. Note also that "`:=`" and "`!`" are non-primitive, and are defined in the prelude with the types required by the Definition.

Perhaps surprisingly, the types `instream` and `outstream` are not primitive. Figure 6 shows their definition in the prelude. Their types are introduced using an `abstype` declaration, which makes them suitably opaque. The pervasive operations on streams are provided as primitives (using `prim`) with integers denoting the streams themselves. The interpreter (specifically, the functor `Apply`, using a stream package `IOStreams`) provides the required functionality over streams accessed as integers. This approach frees the rest of the Kit from having to know anything about the types of input and output streams.

The evaluation of the primitives is implemented by the functor `Apply`, whose signature is given in Figure 7. The set of basic values BasVal contains a single value, `prim`. `Apply` provides the application function APPLY which takes a BasVal (which must be `prim`) and an argument value (which must be a pair whose first argument is an integer denoting the primitive function required and whose second argument is of the correct form), and returns a result value. `Apply` also provides

```
abstype instream = INSTREAM of int
    and outstream = OUTSTREAM of int
with
   val std_in = INSTREAM 0
   val std_out = OUTSTREAM 0

   fun open_in(f: string) =
      INSTREAM(prim(33, (f, Io("Cannot open " ^ f))))

   fun open_out(f: string) =
      OUTSTREAM(prim(34, (f, Io("Cannot open " ^ f))))

   fun input(INSTREAM i, n: int): string = prim(35, (i, n))

   fun lookahead(INSTREAM i): string = prim(36, i)

   fun close_in(INSTREAM i): unit = prim(37, i)

   fun end_of_stream(INSTREAM i): bool = prim(38, i)

   fun output(OUTSTREAM i, str: string): unit =
      prim(39, (i, str, Io "Output stream is closed"))

   fun close_out(OUTSTREAM i): unit = prim(40, i)
end
```

Figure 6: Definition of streams from the prelude

```
signature APPLY =
  sig
    type BasVal and Val

    val APPLY: BasVal * Val -> Val
    val equal:  Val * Val -> bool

    val FAIL_USE: unit -> unit
  end
```

Figure 7: The signature APPLY

the equality function as a `prim` (which is used to implement "`=`", and is also needed for pattern matching), and some glue to deal with the `use` function (Section 7). The body of `Apply` is fairly straightforward. There are functions for unpacking and packing the primitive types and the important non-primitive types like `bool` and `list`. The function implementing APPLY takes a BasVal (which is ignores, since it can only be `prim`) and a value which is immediately decomposed to yield an integer denoting the primitive operation, and a value denoting the actual arguments to the primitive operation. `Apply` makes use of `IOStreams` to implement streams, and `CoreDynObject.Store` to manipulate references. (Only the `add` and `retrieve` operations are used by APPLY to implement assignment and dereferencing; new references (using `Store.unique`) are created as `ref` is applied, in `EvalDec`.)

`Apply` is also capable of raising exceptions from the primitive functions—it is passed a value `RAISE` of type `CoreDynObject.Val -> 'a` which can be called with exception values.

6 Error Detection and Reporting

As a general rule, whenever a Kit component wants to print some message to the user, it produces a value, called a *report*, which can later be combined with other reports from other parts of the Kit to form a complete message which can then be printed. This approach is very different to eagerly writing something on the screen each time there is some information that the use might want. By delaying printing, we have the possibility of editing reports (or omitting parts of reports altogether) before they are presented to the user.

A report is an abstract representation of a line of text. Reports can be created from strings, appended to each other, indented, decorated with prefix strings, and finally printed. This kind of functionality is needed to print some of the more complex objects of ML such as functor bindings. Rather than print lines of text,

```
signature ERROR_TRAVERSE =
  sig
    type topdec
    type Report

    datatype result = SUCCESS
                    | FAILURE of Report

    val traverse: topdec -> result
  end
```

Figure 8: The signature `ERROR_TRAVERSE`

a "printing" function will return a report; these reports are accumulated to the top level of the system (functor `Interpreter`) where they are finally printed.

In addition, some printing (or rather, report generation) is deferred. In particular, elaboration errors are not printed (i.e. turned into reports) as soon as they are detected; that would complicate the elaborator, as all the elaboration functions would have to return results containing reports. Instead, elaboration errors cause error info nodes to be planted on the abstract syntax. After elaboration, a separate post-pass runs over the abstract syntax to pick out and to generate a report from any errors which were detected. The interpretation process only continues if there were no errors.

Error traversal is done by `ErrorTraverse`, whose signature is given in Figure 8. `ErrorTraverse` examines all info nodes to see if they contain any error info. If so, this is extracted together with the source information (if it is present, which it should be in most cases) and built into a report. The result of each traversal is examined in the top-level loop, in `Interpreter`, where any errors are reported.

Reports are also used for top-level printout of bindings. The functor `TopLevelReport` takes a basis (comprising a static, a dynamic, and an infix basis) and returns a report. `TopLevelReport` is responsible for any indentation and formatting of Modules-level constructs (such as functors, which require some work in order to present them in an acceptable manner), but knows nothing about creating textual representations of Core values. This is done by `ValPrint`; the only assumption made by `TopLevelReport` (perhaps erroneously) is that a value will print into a one-line report.

7 File Inclusion and Implementation of `use`

Despite being present in most SML implementations, `use` is not part of the SML Definition. In addition, different implementations implement different functionality for it, and rarely specify its behaviour precisely. Since it is not part of the Definition, it should, strictly speaking, not be implemented in the Kit; however, the Kit's usefulness would be limited without it. Isolated source files could be read using `evalFile`, but it would be difficult to test the Kit on real-world programs since they either contain occurrences of `use`, or they are built on top of a library or environment (such as the Edinburgh SML Library, or its Make system) which requires `use`.

The Kit does not make `use` pervasive; instead it is defined in a structure called `Kit`, as a call to `prim`. The (informal) semantics are as follows:

- Evaluation of the `use` function applied to a string causes the top-level loop to be immediately invoked on the file, such that the file's bindings (and side-effects) are established in the top-level environment before the call to `use` terminates. Because complete phrases are typechecked before execution, an expression such as

  ```
  (use "file.sml"; x + 1)
  ```

  has the same meaning and value regardless of any binding made to `x` in `file.sml`. The used file is executed in the enclosing basis upto but excluding the phrase containing the call to `use`.

- A compiled-time error (parsing or typechecking) in the used file causes the exception `Use` (in structure `Kit`) to be raised in the calling context. This can be handled in the usual manner.

- An exception which propagates uncaught to the top-level in the used file is passed uncaught into the context calling the `use`, or else reported at top-level.

- Nested occurrences of `use` work as expected.

Therefore, the exception `Use` can be handled in order to deal with compile-time failure for a used file (or any nested file), and runtime exceptions which propagate through a `use` can be handled.

`Interpreter` maintains the top-level basis in a reference variable (all the lower levels are pure in the way they manipulate bases). As each top-level declaration is executed and terminates successfully, its bindings are added to the top-level basis. This means that a file containing an error (compile time or runtime) will cause bindings to be established up to but excluding the offending declaration.

`use` is implemented using source readers [4]. This means that a call to `use` on a file causes the entire file to be read into memory before its contents are fed to the lexer. In practice, this should be no different behaviourally to the "obvious" scheme where the file is read from sequentially as the lexer is invoked.

The main implementation issue with `use` is that it requires self-reference in the interpreter. `use` is a primitive function which needs to be able to call the entire interpreter on a body of source text, but the interpreter needs to support the entire language, including all the primitive functions such as `use`. The most straightforward implementation technique is to use a reference to hold the `use` function. The reference (of type `(string -> unit) ref`) is provided in functor `KitCompiler`, and is created containing a function of the correct type but no effect. Thus, a function `USE` (of type `string -> unit`) can be passed to the lower levels of the system (the functors `Linking`, `Execution`, `Evaluation`, `EvalDec`, `Apply`). Once the functor heirarchy has been built, the reference variable can be assigned with a function which calls `KitCompiler.interpretFile`.

Exception propagation within `use` is handled as follows. `Apply` has a local exception `Use`, and there is a handler for this at the point in `Apply` where the `use` operation is executed. Should `Use` be raised, the handler translates this failure into a raising of the exception value `Use` which was passed as an argument to `prim` in the prelude. The facility to raise `Use` is exported from `Apply` as the function `FAIL_USE`. This function is passed through `EvalDec` and `EvalTopdec` to `Interpreter`, where the ability to raise `Use` (to signal compilation errors) is needed. For runtime exceptions, `EvalDec` provides a function `RE_RAISE` which the interpreter can use to raise uncaught packets into the surrounding `use` context. (There is no problem with unclosed source files since each used file is read completely into memory and closed before it is presented to the top-level loop.)

8 Record Typing and Flexibility

SML Implementations are required to deal with two kinds of overloading: arithmetic/logical operators, and record patterns. Operator overloading is a scheme which attaches one of several monomorphic functions to each occurrence of the operator, depending on the type at which the operator is being applied. In a similar manner, we consider record patterns such as {`A, B, ...`} to be overloaded; compilers (including the Kit's lambda compiler) tend to represent records as vectors of values, indexed according to the order of the labels, and so record wildcards have to be resolved in all contexts so that code generation can take place.

To support flexible records we have implemented an algorithm based on an earlier implementation by Mads Tofte, after the scheme described in a paper by Didier Rémy [2]. In principle, a fairly simple implementation would suffice, were we implementing type unification using update-in-place, but since we adopt a "pure" substitution-based algorithm, a more complicated scheme is needed so that substitutions can be built to map short records into longer ones.

The interested reader is referred to the paper by Rémy for details of the algorithm. The implementation of the algorithm is purely local to `StatObject`, the module responsible for implementing the static semantic objects in the Kit. `StatObject` implements three kinds of substitution (type variable, record row variable, record field variable) internally, and exports a single substitution type. The rest of the Kit is oblivious to this complexity due to the abstract nature of substitutions: other modules (such as `ElabDec`) simply compose and apply substitutions as if they merely ranged over type variables.

9 The State of the System

This section lists the unimplemented and nonfunctional features of the Kit, as well as some matters of style and so on which could be improved.

Error conditions in the parsing tools. The shunting algorithm employed by the infix analyser panics on an illegal sequence of operators. It should provide error annotations. An example of offending input is

```
infix @@ val x = @@ 2
```

Lexical errors not handled. The lexer raises an error if it encounters an illegal token (such as a control character, or illegal string escape like `"\ "`). This should be caught and presented as a parsing error.

Overflow in integer constants. There should be a trap for the `Overflow` exception in the lexer, so that large integer constants can be faulted as bad tokens. Currently, the `Overflow` exception propagates.

Missing position information. Source position information is not planted into the abstract syntax created for derived forms. This results in error printing with the source position printed as "`(position unknown)`".

Duplicate identifiers. The so-called syntactic checks for duplicate identifiers (variables in pattern, constructors in datatype declaration, modules separated by `and`, record labels, and so on) are currently not made.

Redefinition of "`=`". Because the equality predicate is defined in terms of `prim` in the prelude, the Kit must allow "`=`" to appear in a declaration. Since the Definition outlaws this, the Kit should disallow subsequent declarations of "`=`". In the same vein, a qualified identifier containing "`=`" should be disallowed (as it can never be valid).

Numeric record labels. The Definition allows numeric record labels to be of arbitrary size (possibly overflowing an integer), and also syntactically restricts them to not start with a zero. The Kit currently regards record labels as integer constants.

Type path names. Currently, type constructors are printed merely as identifiers. They should be printed as complete paths depending on their accessibility. Inaccessible type constructors should be marked (as "`?.TyCon`" for example). The same should probably be done for exception and data constructors printed at top-level. Similarly, we could insist that values whose data constructors are completely inaccessible be printed opaquely; this would implement `abstype` properly.

Redefinition of `ref`. The Kit currently allows redefinition of `ref` as a data constructor (which is a problem since application of `ref` must be treated specially) and also as a type constructor (which is a problem since equality admission regards `ref` specially). Both kinds of redefinition should be outlawed.

Treatment of `Match` and `Bind`. Although `Match` and `Bind` are defined in the prelude, they are not raised by inexhaustive pattern matches and bindings.

Modules compiler. There is currently no lambda compiler for the Modules.

Core match compiler. The match compiler for the Core does not deal with exception matches properly. This is a difficult problem, and the current decision tree scheme needs to be rethought to deal with the sequential nature of exception matching.

Parsing cosmetics. There should be a distinct continuation prompt.

Use of library signatures. The "generic" library signatures such as `PRINT`, `EQUALITY` and so on should be used more. The main problem with this is weaknesses in SML/NJ's typechecker, which disallows repeated (shared) types in signatures.

Sets instead of lists. Many of the elaboration routines for the Core and Modules use lists in lieu of sets. These should be converted to use the Edinburgh Library's set operations. The main occurrences are those where `ListHacks` and `LIST_HACKS` are currently being used.

Phase problems when changing execution mode. At present, it is possible to run the Kit in elaborate-only mode, and then switch to execute mode with the same accumulated basis. This is insecure, since identifiers will be known statically with the incorrect, or no, runtime binding.

Occurrences of `op`. In `Infixing`, there should be a cosmetic check for occurrences of `op` in datatype and exception bindings. According to the Definition, `op` is required for identifiers with infix status, although the parser does not need this annotation.

Bad functor argument reporting (new, 22/Jul/92). The declaration "`functor F() = struct end`" gets turned into a report with what looks like the first line missing.

References

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

[2] Didier Rémy. Type inference for records in a natural extension of ML. Technical Report Research Report number 1431, INRIA, May 1991.

[3] Nick Rothwell. Functional compilation from the Standard ML core language to lambda calculus. Technical Report ECS–LFCS–92–235, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[4] Nick Rothwell. Parsing in the SML Kit. Technical Report ECS–LFCS–92–236, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.