Object Language Embedding in Standard ML of New Jersey

Konrad Slind
University of Calgary

December 3, 1991

Compiler support for user-defined parsers and prettyprinters has been added to SML/NJ. The notions of *quotation* and *antiquotation* and their implementation in terms of *frag lists* are explained. For prettyprinting, the 1980 TOPLAS algorithm of Oppen is now used by the compiler, and has been made available for users.

1 Object Languages

Standard ML is often used to implement another language L, *e.g.*, the syntax of the HOL logic in hol90 or the syntax of CCS for the Concurrency Workbench. Typically, one defines the abstract syntax of L by a `datatype` declaration. Then useful functions over the datatype can be defined (such as finding the free variables of a formula when L is a logic). Soon afterwards, one concludes that concrete syntax is easier for humans to read than abstract syntax, and so writes a parser and prettyprinter for L.

In the situation just outlined, Standard ML is called the *metalanguage*, and L is called the *object language*, or OL. Edinburgh/INRIA/Cambridge ML, the precursor to Standard ML, was originally a programming metalanguage for a particular object language, the LCF logic. ML had automatic facilities for parsing and prettyprinting LCF formulae, terms and types. The work in this paper generalizes those facilities and will soon be available in the SML/NJ compiler.

2 Quotation and Antiquotation

A *quotation* is a special form of literal expression that represents the concrete syntax of an OL phrase. The backquote character (`) is used to delimit quotations. Note that adopting this notation probably requires that the backquote be removed from the set of characters from which SML symbolic identifiers are formed, thus potentially "breaking" some programs.

For a running example, suppose our OL is a simple propositional logic with propositions represented as values of abstract type `prop`. A quotation such as:

```
`A /\ B \/ C`;
```

would be an expression of type `prop`, and its evaluation would invoke an OL parser (% say) to parse, enforce precedence, etc., eventually returning the SML representation of the proposition.

```
- val p = %`A /\ B \/ C`;
val p = -: prop
```

The most common approximation to quotation is strings. This is not pleasant at times, especially when dealing with backslashes and newlines. Still, strings are bearable. Strings are not adequate, however, for the following idea.

The ML-OL relationship invites a notion of *antiquotation*: the temporary abandonment of parsing so that an ML value can be spliced into the middle of a quotation. Operations like this have cropped up under various names in various places: antiquote is due to Milner; Quine had a version called *quasi-quotation* in his 1940 book; and I have been told that Carnap used a notation much like it. It also closely resembles the Lisp *backquote* facility.

We shall assume that an antiquote is written as a caret (^) followed by either an SML identifier or a parenthesized SML expression. Antiquotation can be used to conveniently express *contexts*, which are often used as a descriptive tool for syntax. A context could be defined as a function taking a `prop` and directly placing it at a location in a quotation.

```
- fun foo a = %`^a ==> A`;
val foo : prop -> prop
```

In this case, `foo p` would denote the same proposition as

```
%`(A /\ B \/ C) ==> A`
```

Antiquotations can have nested quotations (which may contain antiquotes of their own, etc.):

```
- let val K x y = x
      val I x = x
  in
  %`A /\ ^(K (%`B`) (I (%`C`)))
      \/ C`
  end;
```

gives the same **prop** as that denoted by **p**. We note in passing that the power of the OL parser is completely up to its author: for example, in the framework offered here, one could write an OL "parser" for Scheme that parses program plus arguments, evaluates the program on the arguments, and finally prints the returned value.

2.1 Implementations of quote/antiquote

We view the SML compiler as mapping from concrete syntax to (SML) abstract syntax, and thence to values. We describe three ways of implementing quotation and antiquotation. These are loosely categorized by when evaluation occurs: the "top" two paths in Figure 1 involve pre-processing quotations, while the adopted method post-processes them. Detailed discussion of the pre-processing alternatives is left to the full report.

The upper path corresponds to string macro expansion: parse_ol has type `:string -> string`; thereafter, the SML compiler is unchanged. This is the approach taken in PolyML. The middle path corresponds to macro expansion in the SML abstract syntax tree; parse_ol has type `:AST -> AST`, where `AST` is the type of SML abstract syntax trees. This is the approach taken in the original ML and in CAML. The last approach swaps the last two maps in the middle path: the antiquotations (if any) that the the OL parser is invoked on are SML *values*. This is the approach taken in the current SML/NJ implementation. It depends on the *frag* datatype:

```
datatype 'a frag = QUOTE of string
                 | ANTIQUOTE of 'a
```

A concrete syntax quotation will be mapped by the SML compiler into a `frag list`. Intuitively, a `frag` is a contiguous part of a quotation: `` `A /\ B` `` maps to `[ QUOTE "A /\\ B" ]` while `` `^x /\ ^y` `` maps to

```
[ QUOTE "",ANTIQUOTE x, QUOTE "/\\", ANTIQUOTE y, QUOTE "" ].
```

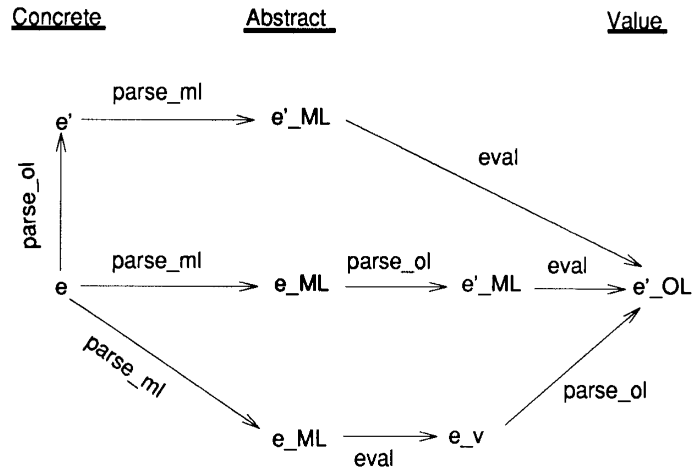

Figure 1: Evaluation alternatives for quotations

In this approach, the value of a quotation has type `:ol frag list` where `ol` is the type of object language expressions; the type of the OL parser is `:ol frag list -> ol`.

Example: Giving SML the input

```
- %`A /\ ^x \/ C`;
```

when `x` is in the SML environment with type `:prop` results in the following operations:

1. The SML lexer, which has been modified to recognize quoted objects, adds the lexemes

   ```
   ... (quote "A /\\ ");  (antiq "x"); (quote " \\/ C") ...
   ```

   to the stream emitted by the lexer.

2. The SML parser has been modified to recognize quotations as atomic expressions. It gathers up the above lexemes into the list

   ```
   [ QUOTEexp "A /\\", ANTIQUOTEexp x, QUOTEexp "\\/ C" ]
   :AST list
   ```

in which QUOTEexp is a function from strings to abstract syntax and ANTIQUOTEexp is a function from abstract syntax to abstract syntax. We further convert this (real) list into a LISTexp (list expression in the abstract syntax) and continue parsing.

3. The AST for the input expression is typechecked, then evaluated. The list expression is typechecked in the presence of the OL parser and hence is well-typed, with type `:prop frag list`. This gives for the example

```
[ QUOTE "A /\\",  ANTIQUOTE x, QUOTE "\\/ C" ]
```

   In step 2, the `x` in **`ANTIQUOTEexp x`** has type `:AST`; here it has type `:prop` because it has now been evaluated.

Advantages and disadvantages.

+ the quoted object is guaranteed to be a member of the OL datatype

+ small changes to the SML compiler

+ any number of object language quotations can be in existence at a time

+/- quotations have been decoupled from their parsing

- new primitive type constructor `frag` introduced

- some OL expansion may be left undone at compile time, *e.g.*, when a closure is built with a quotation inside it.

2.2 Further topics

Often one wants to parse stratified languages, such as first order logic, or typed lambda calculus, which requires a trick. Also, there is a bit of trickery when one wants to deal with ML-Yacc and ML-Lex, especially when functorizing the parser. These topics are covered in the full report.

3 Pretty Printing

There is a facility for installing user-defined prettyprinters over (monomorphic) user-defined types into the top-level loop of the SML compiler. The underlying algorithm is that of Oppen (1980). There are more expressive prettyprinting languages around, notably that of PPML, but the Oppen interface has the benefit of being efficiently implementable—it runs in constant

space. Thus it is a good option for modest prettyprinting tasks that need to be quick, such as the printing of SML values.

The high-level view is that the user will define a prettyprinter for a datatype and install it in a prettyprinter table. When it comes time for the compiler to print a value, it looks first in the prettyprinter table, to see if a prettyprinter is installed for that type. If so, it calls the prettyprinter on the value, otherwise, it calls the default printing routine.

The Oppen algorithm provides a *block* abstraction: a block establishes a level of indentation. Since blocks can be nested and offset from one another, levels of indentation can be achieved. A block can be broken up by one or more *breaks*, which mark possible places to add carriage returns. There are two styles of block: *consistent* and *inconsistent*. If a consistent block does not fit completely onto the current line, a carriage return will be added after each component of the block. If an inconsistent block does not fit completely onto the current line, a carriage return is added after the last item that does fit on the line; this style of block conserves on vertical space.

3.1 The Signature

The pervasive structure `PP` will meet the following signature:

```
sig
  val install_pp : ('1a -> unit) -> unit
  val pp : ('1a -> unit) -> '1a -> string

  datatype break_style = consistent | inconsistent
  val begin_block : break_style -> int -> unit
  val end_block : unit -> unit
  val add_break : (int*int) -> unit
  val add_string : string -> unit
  val add_newline : unit -> unit
  val set_line_width : int -> unit
  val set_pp_output_function : (string -> unit) ->
                               (string -> unit)
end
```

Note: `install_pp` and `pp` are pervasive, and have not yet been put into the PP structure.

`val install_pp : ('1a -> unit) -> unit.` This takes a prettyprinting function as its argument, and installs it in the SML compiler's

prettyprinter table. Note: the above type is what the type of **install_pp** *should* be, but not what it is: currently it has type **:string -> ('1a -> unit) -> unit**, where the string is the name of the datatype over which the prettyprinter operates. This can be disastrous if two different types with the same name are declared in different structures and values of those types come to be printed: one will print properly, the other will cause a core dump. This will be fixed shortly.

val pp : ('1a -> unit) -> '1a -> string. This takes a value and a prettyprinter and applies the prettyprinter to the value, dumping the result to a string. This is useful when one wants to prettyprint values as a computation progresses, rather than just at the end. What one does is prettyprint the value to a string, then call **output** on that string.

datatype break_style = consistent | inconsistent. This defines the type of breaks.

val begin_block : break_style -> int -> unit. This begins a new level of indentation, at the current offset from the left margin. The first argument determines how the block is to be broken. The second argument determines the new offset if the block gets broken.

val end_block : unit -> unit. Prettyprinting reverts to the previous level of indentation.

val add_break : (int*int) -> unit. Notify the prettyprinting engine that a carriage return is possible. The argument to this function is a pair; the first member is the **size** of the break, the second member is an **offset**. This "break" offset is used for finer control over indentation than that offered by the block offset. The current block style and the size of the block determine what action is to be taken:

1. **block_style = consistent** and the entire block fits on the remainder of the current line. Output **size** spaces.
2. **block_style = consistent** and the block does not fit on the rest of the line. Add a carriage return after each component in the block and add the block offset to the current indentation level. Each component will be further indented by **offset** spaces.
3. **block_style = inconsistent** and the next component of the current block fits on the rest of the line. Output **size** spaces.
4. **block_style = inconsistent** and the next component doesn't fit on the rest of the line. Output a carriage return and indent to the current indentation level plus the block offset plus **offset** extra spaces.

size is taken into account when determining if there is enough room to print the next component in the block.

val add_string : string -> unit. Outputs the given string.

val add_newline : unit -> unit. Forces the output of a carriage return.

val set_line_width : int -> unit. Allows the user to change the prettyprinting engine's notion of how wide the line is. This call re-allocates the token buffer used by the prettyprinting engine, and so should only be called when prettyprinting is not happening. The initial line width is 70 characters.

val set_pp_output_function :(string -> unit) -> (string -> unit). Prettyprinting eventually boils down to the output of strings; blocks and breaks are just instructions for artfully arranging the strings. This function allows one to change the output function that gets applied to strings. The returned value is the current output function. The initial output function is

```
fn s => output(std_out,s).
```

3.2 Examples

```
open PP;
datatype Num = Zero | Suc of Num;

fun pp_Num Zero = add_string "Z"
  | pp_Num (Suc n) = ( pp_Num n; add_string "'" );

val _ = install_pp "Num" pp_Num;

datatype Nexp = Atom of Num
              | Add of (Nexp*Nexp);

local
fun pr (Atom a) = pp_Num a
  | pr (Add(a1,a2)) = ( pr a1;
                        add_string " +";
                        add_break(1,0);
                        pr a2
                      )
in
fun pp_Nexp ne = ( begin_block consistent 0;
```

```
                    pr ne;
                    end_block()
                  )
end;

val _ = install_pp "Nexp" pp_Nexp;

- val A = Add (Atom(Suc(Suc(Suc(Suc(Suc(Suc(Zero)))))))
               Atom(Suc(Suc(Suc(Suc(Suc(Suc(Zero))))))));

val A = Z''''''''' + Z''''''''' :Num_exp

- set_line_width 20;

- A;
val it =
  Z''''''''' +
  Z'''''''''
  :Num_exp

- Add(A,A);
val it =
  Z''''''''' +
  Z''''''''' +
  Z''''''''' +
  Z'''''''''
  :Num_exp
```


Acknowledgements

This work was done in collaboration with Dave MacQueen. Also, Dave Berry pointed me to Simon Finn, who informed me of a bug in the original Oppen algorithm.