

A complete and realistic implementation of quotations for ML

Michel Mauny
INRIA-Rocquencourt*

Daniel de Rauglaudre
INRIA-Rocquencourt*

Abstract

Quotations have been introduced in the very first versions of ML [6] in order to represent propositions in the LCF logic under a concrete form. ML was the *metalanguage* of this logic, and the logic was its *object language*.

In this paper, we describe a new implementation of quotations in a variant of Caml-Light [8]. We address several important problems, namely the impact of such a mechanism on the design of ML, the problem of separate compilation and the possibility of implementing support for quotations in compilers that do not feature dynamic linking.

The novelty of this work is that our implementation is realistic, efficient and complete. It includes arbitrary quotations and antiquotations, usable as expressions as well as patterns. It is compatible with module systems and preserves separate compilation. Furthermore, our language is its own object language, and this allows for manipulating arbitrary ML abstract syntax trees (ASTs, for short) without ever mentioning their data constructors.

Introduction

Quotations have been introduced in the very first versions of ML [6] in order to represent propositions in the LCF logic under a concrete form. ML was the *metalanguage* of this logic, and the logic was its *object language*. Such a feature followed the evolution of Edinburgh ML (Cambridge, INRIA) and was still present in the Caml [15] system, where several object languages could coexist.

Caml quotations are not completely satisfactory for several reasons. One is that all object languages are forced to use Caml's lexical analyser, restricting the kind of object languages that can be dealt with. Other reasons include some incompleteness in the definition of Caml as its own object language, and an unclear interaction with the module system.

In this paper, we use a version of ML, derived from Caml-Light, in the design of which we devoted a particular attention to concrete and abstract syntaxes. We present here only the implementation of quotations, and we think that the concrete syntax that we use is close enough to other ML versions for the reader not to be confused.

We start by giving the motivations for this work and the constraints to be satisfied by our design (section 1).

* Author's address: INRIA, B.P. 105, F-78153 Le Chesnay Cedex. Email: {Michel.Mauny,Daniel.de_Rauglaudre}@inria.fr

Then, we explain the basic mechanisms of our implementation (section 2). Programming quotation expanders is outlined in section 3. In section 4, we present ML as its own object language together with the impact on the design of the ML abstract syntax. Problems such as pretty-printing ML programs containing quotations, or retrieving the source location of a quotation are addressed in section 5. Separate compilation, which was our original goal, must remain supported, and we show in section 6 how ML programs containing quotations can be compiled by a batch compiler. Alternative implementations are presented in section 7, for compilers that do not feature dynamic linking. Finally, the interaction with module systems is made more precise in section 8. We give some examples of usage of quotations in appendix.

1 Motivations and constraints

Although quotation mechanisms have been proposed before [10, 1, 2, 12], we recall what motivations guided our work.

1.1 Terminology

In the following, we call *quotation* an expression or a pattern of the form $\langle\langle e \rangle\rangle$, where e denotes a character string. The general form of a quotation is $\langle:S\langle e \rangle\rangle$, where S is a *syntax name*: a legal ML identifier. Such a quotation is called a *named quotation*. The former kind of quotations refers to some default syntax name S .

When a quotation is used in expression (resp. pattern) position, we call it *quoted expression* (resp. *quoted pattern*). We call *expression* (resp. *pattern*) *expander* an ML function of type $\text{String} \rightarrow \text{MExpr}$ (resp. $\text{String} \rightarrow \text{MLpatt}$), where MExpr and MLpatt are the concrete types of ML ASTs.

We call *syntax extension* a pair of expanders (one for quoted expressions and one for quoted patterns), together with a syntax name S .

1.2 The need for quotations

Quotation mechanisms (including antiquotation facilities) developed for ML [10, 1, 2, 12] allow for the manipulation of complex data structures by using a concrete syntax. Typical examples are ASTs: it is by far more pleasant to manipulate them under a dedicated concrete form rather

than using directly their representation as ML data structures. A typical example is the direct implementation of denotational semantics.

The usage of quoted expressions (instead of their ML representations) also offers a kind of abstraction in the sense that they allow for a program text (containing quotations) to be “parameterised” by the corresponding expander. For example, Pascal constants such as:

```
const MAX=256;
```

can be easily obtained by a quotation mechanism, allowing for the textual replacement of `<:Macro<MAX>>` by 256. More importantly, huge sets of constants, such as the X event names, can be used in ML programs and be expanded into ML constants (integers in the case of X events).

Another example is the implementation of assertions *à la C*. In a C program, one would write:

```
...; assert(n > 0); ...
```

Using quotations, one would write:

```
...; <:Assert< n > 0 >>; ...
```

assuming the proper definition¹ for the `Assert` syntax extension.

More generally, the advantages of C macros can be obtained with quotations.

1.3 Independence from data type definitions and parsing techniques

Quotations do not need to be attached to a fixed parsing technology as they are in [1]: when debugging a real-size compiler (for a language, say L) implemented in ML, we must be able to use a variant² p' of the parser p of L programs as a quotation expander. The parser p' can be written directly, or generated by a parser generator: we must be able to use p' as a quotation expander, whatever technology it uses.

The use of quotations for simulating Pascal constants implies that they cannot be systematically attached to a type definition as they are in [1]. For representing X events, ML integers might be preferable to a dedicated enumerated type, since events can be passed to or received from other programs, and coding/decoding integers to/from a dedicated type can be painful.

1.4 Locations in source programs

Although we want a general syntax extension mechanism, we still want to be able to pretty-print an ML program containing quotations, and to find the location of the original source code of quoted expressions or patterns (for printing error messages or debugging).

¹ Given in appendix.

² Such a variant is obtained by replacing the parts of p building the L ASTs—the semantic actions of grammar specification, for instance—by their ML quotation.

2 Basic mechanisms

The expansion of quotations *occurs at parse-time*, which guarantees the absence of run-time penalty for user programs. The basic mechanisms of syntax extensions in ML are extremely simple. We need:

- “hooks” in the ML concrete syntax for expressions and patterns, where quotations may occur;
- a mechanism to fill these hooks with user-defined functions with named objects (syntax extensions).

Let us examine in turn each of these ingredients.

2.1 Quotation “hooks”

Lexical analysis. Lexical tokens are annotated with source locations (type `Loc`). A new token “`Quotation String String Loc`”³ is introduced.

The input `<:Sn<str>>` at source location `loc` produces the token “`Quotation Sn str loc`”.

From the lexical point of view, quotations are similar to string constants, and inner occurrences of “`>>`” must be escaped.

Parsing. The expansion of quotations is the ML parser task. Given a token “`Quotation Sn str loc`”, the ML parser checks that the syntax `Sn` exists, and calls one of the `Sn` expanders (depending on whether we are parsing an expression or a pattern) on the string `str`. The result of this call is an ML AST, which is returned as result.

Syntaxes. A *syntax* is a triple composed of a name `S` (any legal ML identifier), an expression quotation expander and a pattern quotation expander. A set of syntax extensions can be implemented as an association list (private to the compiler) with syntax names as keys, and pairs of expanders as values.

Tools are provided in order to register a new syntax binding and change the default syntax name (for anonymous quotations). The function `new_syntax` is such that “`new_syntax sname (eexpand, pexpand)`” declares a new syntax extension named `sname`, and composed of the two expanders `eexpand` and `pexpand`. The directive “`!syntax S`”, executed at compile-time, sets the default syntax to `S`.

3 Quotation expanders

Quotation expanders are arbitrary functions taking strings as arguments, and returning ML ASTs. Expanders can be as simple as the functions used for coding Pascal or C constants mentioned in section 1.2, or as complex as parsing functions generated by sophisticated parser generators.

In case an expander raises an exception, that exception is trapped by the ML toplevel parser, and reported as a syntax error.

³ Quotation is the data constructor, and it is followed by the type of its arguments.

3.1 Programming quotations expanders

If we come back to the simple example of Pascal constants, defining quotations expanders can be as simple as functions matching the string "MAX", and returning the expression `<:MLepr<256>>`, *i.e.* the ML AST of the constant 256. Besides being types, `MLepr` and `MLpatt` are also predefined syntax extensions (*cf.* section 4) making our language its own metalanguage. We first define the two Macro expanders:

```
#value macro_expr = fun [
# "MAX" -> <:MLepr<256>>
# | s -> raise (Failure("Undefined \"~s~\""));
macro_expr : String -> MLepr = <fun>

#value macro_patt = fun [
# "MAX" -> <:MLpatt<256>>
# | s -> raise (Failure("Undefined \"~s~\""));
macro_patt : String -> MLpatt = <fun>
```

Then, we declare the new syntax, and make it the default one:

```
#new_syntax "Macro" (macro_expr, macro_patt);
- : Unit = ()
#!syntax Macro;
```

Finally, we can use our new syntax extension:

```
#<<MAX>> - 1;
- : Int = 255

#value overflow n = (n >= <<MAX>>);
overflow : Int -> Bool = <fun>

#value meets_max =
# fun [ <<MAX>> -> True | _ -> False];
meets_max : Int -> Bool = <fun>
```

One drawback of the string matching solution, is that quotations such as:

```
#<< MAX >>;
> Toplevel input:
><< MAX >>;
> ~~~~~
> Macro syntax failure: Undefined " MAX "
```

fail to expand correctly. This is because of the leading or trailing blanks occurring in the string that is passed to the expander. String matching functions are easy to define but might be too naive. Furthermore, such functions cannot process realistic concrete syntaxes. In these cases, a more sophisticated technology is desirable. Parsers can be written by hand, or using a parser generator in the style of Yacc.

A more realistic example is the following syntax extension for pure λ -terms producing values and patterns for the following data type:

```
#type Lambda = [
# Var String
# | Abs String Lambda
# | App Lambda Lambda
# ];
Type Lambda defined.
```

Concrete λ -terms follow the usual syntax, excepted for abstractions, where abstracted variables are enclosed in square brackets. For example we write "[x] x" instead of the more classical "\x.x". We omit the definition of the expanders `lambdae` and `lambdap`:

```
#new_syntax "Lambda" (lambdae, lambdap);
- : Unit = ()
#<:Lambda< [x] [y] y >>;
- : Lambda = Abs "x" (Abs "y" (Var "y"))
```

Usually, expanders make heavy use of `MLepr` or `MLpatt` (anti)quotations.

3.2 Programming antiquotations

Given an object language, one can limit oneself to the production of closed ML ASTs such as:

```
#value lambda_id = <:Lambda<[x] x>>;
lambda_id : Lambda = Abs "x" (Var "x")
```

One can also design "hooks" in the concrete syntax and produce *contexts*. We choose pairs of "~" as Lambda antiquotation marks.

The following function returns the list of sub-terms of its argument:

```
#value rec subterms = fun [
# <:Lambda<[_~] ^t1~>> as t -> [t :: subterms t1]
# | <:Lambda<^t1~ ^t2~>> as t
# -> [t :: subterms t1 @ subterms t2]
# | v -> [v] (* Variable case *)
# ];
subterms : Lambda -> List Lambda = <fun>

#subterms <:Lambda< [x] [y] x y >>;
- : List Lambda = [Abs "x" (Abs "y" (App (Var "x") (Var "y"))); Abs "y" (App (Var "x") (Var "y"))]; App (Var "x") (Var "y"); Var "x"; Var "y"]
```

The `lambda_x` function takes a term and returns the abstraction of the variable `x` on it:

```
#value lambda_x = fun term -> <:Lambda<[x] ^term~>>;
lambda_x : Lambda -> Lambda = <fun>

#value delta = lambda_x <:Lambda<x x>>;
delta : Lambda = Abs "x" (App (Var "x") (Var "x"))
```

A simple technique for programming such antiquotations is to parse what appears between antiquotation marks as a string, and to pass this string to the ML parser for expressions (or for patterns), using the ML AST produced for building the final result.

4 Predefined syntaxes: MLepr and MLpatt

Two predefined syntaxes provide a simple way of building ML ASTs. `MLepr` produces ASTs for expressions, and `MLpatt` for patterns.

The simplest form of ML quotations is used to produce constant ASTs. In section 4.2, we introduce antiquotations in ML quotations: antiquotations allow for building ML contexts (*i.e.* expressions with holes).

4.1 Closed ASTs

Arbitrary ASTs may be built using the `MLepr` syntax. As a simple example, the following examples evaluate to ML ASTs (toplevel responses omitted):

```
#<:MLepr< 1+2 >>;
```

Quotations may be used recursively, as in:

```
#<:MLepr< do <:Assert< n > 0 \>>; return n >>;
```

4.2 Contexts

Expressions with holes are more interesting in that they allow the extraction of subexpressions (when used as patterns) and holes can be filled in by actual parameters of functions. Typically, most values (ML ASTs) returned by user-programmed expanders are built using ML quotations and antiquotations.

Simple antiquotations. The “#” character being free in our language, we use it as left and right antiquotation mark. As a first example, consider a function extracting arguments from a syntactic addition into two parts. (The default syntax is `MExpr`.)

```
#value split_add = fun [
# <<#e1# + #e2#>> -> (e1,e2)
# | _ -> raise (Failure "Not an addition");
split_add : MExpr -> MExpr * MExpr = <fun>
```

On the other hand, it is possible to use antiquotations in order to build ASTs:

```
#let f5 = <<fact 5>> and e = <<10>>
#in << #f5# + #e# >>;
- : MExpr = MLE_app 0 (MLE_app 0 (MLE_lid 0 "+") (MLE_app 0 (MLE_lid 0 "fact") (MLE_int 0 5))) (MLE_int 0 10)
```

A typical usage of `MExpr` and `M Patt` antiquotations is the specification of values returned by a user-defined quotation expander. For example, the result of parsing an application in the expression expander for the syntax `Lambda` is written `<:MExpr<App #e1# #e2#>>`, where `e1` and `e2` are the results of parsing the operator and the operand of the `Lambda` application.

Named antiquotation marks. For contexts to be really usable, we must be able to build and de-structure arbitrary ML ASTs using only their concrete form (quotations).

Many constructs pose no problem. For example, the conditional is correctly matched:

```
#fun [
# <<if #e1# then #e2# else #e3#>> -> (e1,e2,e3)
# | _ -> raise(Failure("Not a conditional"));
- : MExpr -> MExpr * MExpr * MExpr = <fun>
```

The `if`, `then` and `else` keywords provide a context which is sufficient to delimit the holes without ambiguity.

Problems arise for constructs that provide no syntactic context. For instance, integer constants and identifiers provide no context at all. A function such as:

```
#fun [ << #n# >> -> n ];
- : a -> a = <fun>
```

intended to match ML ASTs reduced to integers constants, matches everything, as shown by its type. Another occurrence of this problem can be found with arrays. The pattern `<<[#e#]>>` could match either the syntax of arrays with only one element, binding `e` to that element, or match any array and bind `e` to the list of its subexpressions.

An elegant solution to this problem has been proposed by Aasa [2], but it involves an interaction between parsing and typechecking, which we think is undesirable because this solution reduces the possible implementations of the

quotation mechanism (*cf.* section 7). Furthermore, there are cases where type information does not help at all (when no type constraint is imposed by the surrounding context on variables occurring in such antiquotations).

Our solution consists in proposing special antiquotation marks (which are equivalent to type constraints in Aasa’s work). The cases that we have to deal with are:

- leaves of ML ASTs, holding base type values, such as strings (for string constants as well as for identifiers), integers, etc.;
- optional keywords corresponding to a boolean value, like `rec`;
- constructs C derivable from one of the grammar schemes:

$$C ::= \text{lp } S \text{ rp} \quad \text{or} \quad C ::= \text{lp } MExpr \text{ } S \text{ rp} \\ S ::= \epsilon \mid MExpr \text{ } s \text{ } S \quad S ::= \epsilon \mid s \text{ } MExpr \text{ } S$$

where $MExpr$ denotes any ML expression, ϵ is the empty alternative, and `lp`, `rp` and `s` are terminals (standing respectively for left, right parenthesis, and separator). The last case occurs for lists, tuples, arrays, match cases, records and `let` bindings.

For the leaves of ML ASTs holding base values, we adopt the following antiquotation marks: `#:string` for string constants, `#:int` for integers, and similarly for other base values; `#:uid` for capitalised identifiers and `#:lid` for other identifiers.

For recursive constructs such as tuples, etc., we adopt a uniform convention for antiquotations, built from parentheses (`lp` and `rp` above) and the separator `s`. As an example, the opening antiquotation mark for arrays is “`#:;`”. Intuitively, it introduces a list of expressions separated by a semicolon. Therefore, the pattern `[#:; e#]` matches any array expression, binding `e` to its list of subexpressions.

For optional keywords, we use the antiquotation mark `#:kwd`, where `kwd` is the keyword.

We admit that these new antiquotation marks could have received less cryptic names, but our goal was to show that the problem of ambiguous quoted patterns could be solved in a systematic way.

We conclude this section by mentioning that the problems addressed above may occur also in user-defined expanders and types (typically ASTs), and the same kind of solutions may be applied.

4.3 Impact on the design of ML

We consider in this section the possible impacts of “ML as its own metalanguage” (*i.e.* the `MExpr` and `M Patt` syntax extensions) on the design of ML at the level of its abstract syntax.

Abstract syntax trees are not only an unambiguous way of representing programs: they often carry other kind of information. In the following, we call *purely syntactic information* the information concerning syntax (*i.e.* the information which is mandatory in order to print a program equivalent to the original one). Other kind of information will be called *annotations*, and do not interfere with the syntactic information.

ML is a quite complex language, with many syntactic constructs. In this section, we argue for the ASTs to be *as close as possible* to their concrete representation.

Derived forms. It must be clear, from the definition of the language, whether a syntactic construct is a derived or a primitive form. This is the case for SML [11], for instance. In such a case, it is desirable to have a one-to-one correspondence between ASTs and quotation patterns. If this was not the case (for example, if the conditional was a primitive form, but expanded to a `match` construct), one could be tempted to write functions such as

```
fun [ << match #e# with #cases# >> -> ...
    | << if #test# then #yes# else #no# >> -> ...
    ...
```

where the second case is useless, because its pattern is an instance of the first one. Note also that some derived forms, while being justified from a semantic point of view, can be counter-intuitive to a programmer. For example, the SML Definition specifies that the sequence is a derived form for applications of functions discarding the value of their argument [11, p. 67]. It may be hard to impose to non-specialists a matching on applications in order to catch sequences.

Therefore, the notions of derived and primitive forms are not only semantic notions, but have a counterpart in the representation of ASTs, and a good trade-off has to be found for the correspondence between derived forms and primitive ones to be reasonably intuitive.

Non-syntactic information should appear as annotations. Compiler writers tend sometimes to design ASTs in such a way that they carry informations useful at code-generation time. For example, a function applied to n arguments may lead to application nodes with the operator on one side, and a list with n elements on the other side. The problem is that in the concrete syntax, there is no notion of arity for functions: applications are juxtapositions which associate to the left. It can be surprising for the user to obtain a list of expressions, instead of an expression as the result of extracting the operand of an application node.

We think that multiple applications should be detected later on (at typechecking time, for example), and represented as such at the level of the intermediate language, or at parsing time, but stored under the form of annotations. In the latter case, these annotations must be optional, *i.e.* the correction of the typing and compilation processes must not rely on their presence.

Independence of syntax trees from the typing context. Since data type declarations are always processed (being in a toplevel session or during a batch compilation), it is tempting to encode as soon as possible some type information in ASTs. At parse time, data type declarations can be recorded, allowing the parser to make an early distinction between data constructors and ordinary identifiers. This way, syntax trees can possess distinguished data constructors with their type information, simplifying a bit the subsequent typechecking phase.

If such information is part of the purely syntactic information, it does not fit well with quotations, since a

typing context is necessary to reconstruct it. The reason is simple: quotations (more precisely quotation expanders) are compiled once for all in a context different from their context of use. Because of this separate compilation of expanders, syntax trees built by expanders cannot take advantage of their context of use, unless the ML parser “patches” them after the quotation expander returns (trees must be copied in this case, because of a possible sharing of the same tree between different expander calls).

Here again, there are several design possibilities:

- the first one has been (sometimes vigorously) discussed for a long time, adopted by the Haskell designers, but let aside by ML implementors: *data constructors must be capitalised*. Then, the distinction between value identifiers and data constructors is done at the lexical level, and type information can be added as optional annotations (but *not* as part of the syntactic information).
This is the approach that we adopted in our design, and some other improvements of the ML syntax can be obtained by pushing further this idea.
- the other one consists in having a single syntax node for identifiers (of any kind), and to add category and type information under the form of optional annotations. This latter task can be left to the type synthesizer, in which case no annotation at all is produced.

Again, all type information contained in ASTs must not interfere with purely syntactic information, and must be optional (*i.e.* the typechecker should be able to live without them).

5 Annotations

`Mlexpr` data constructors carry an integer value which, when strictly positive, is called the *address* of the node. Our ML parser generates new unique numbers for each node of regular programs. Using these addresses as keys, and hash tables, for example, useful informations called *annotations* can be stored and retrieved, provided that these integers are unique.

All nodes allocated by the toplevel ML parser have a non-null address. All nodes allocated by the `Mlexpr` expanders have no address (*i.e.* carry the integer 0).

Pretty-printing. In our implementation, results of expander calls are *copied* before being plugged in the current AST. This allows for a *relocation* of these nodes to take place: all nodes of the ML AST corresponding to the quotation are assigned a unique address. For pretty-printing quotations correctly, the top node of an expanded quotation can be annotated with its original source text, and such annotations can be checked and retrieved by a carefully written ML pretty-printer.

Source location. In the same way as we store pretty-printing information as annotations while relocating a syntax node, we store its source text location (or an approximation thereof). This implies that when the source of a typing error is inside a quotation, either the whole

quotation or a part of it is underlined by the type checker as being the source of the error.

6 Separate compilation

So far, all our examples assumed that we were in an ML interactive session. Actually, our goal is to propose a quotation mechanism compatible with separate compilation, and the introduction of quotations at toplevel is just a consequence of that. We address the problem of separate compilation in this section.

Even under the toplevel loop, we can think of a syntax extension as an extension of the compiler. More precisely, we can think of the ML compiler *compile* as being parameterised by a list of syntax extensions. At each toplevel phrase, the compiler that is actually invoked is (semantically) the partial application (*compile exts*), where *exts* is the current list of syntax extensions. The declaration of the new syntax extension (*S*,(*expand*,*peexpand*)) implies that the next instance of the compiler to be called will be (*compile* ((*S*,(*expand*,*peexpand*)):*exts*)).

This view of the toplevel compiler leads to the same considerations for a batch compiler. This led us to:

- force syntax extensions to be modules (or sets of modules) depending only on the pervasive environment. This allows for linking them with the initial compiler.
- add `-L` options to the compiler, followed by an object code file to be loaded before actually compiling the source files. This option can be seen as implementing the partial application mentioned above.

Of course, this scheme works only because of the possibility of dynamic loading of code in the compiler. This is not always possible, and we outline below three other possible implementations (two of which we actually developed).

7 Other possible implementations

When dynamic loading of code by the batch compiler is not possible, we can think of relinking the whole compiler object files together with compiled syntax extensions. This generally implies that syntax extensions are defined in the same language as the compiler itself (fortunately, this language is usually ML itself). There is not much to be said about this possibility: it is always possible, but a complete relinking of the compiler can be slow in some ML implementations. We actually tested this possibility, which is quite close to our final implementation (which is bootstrapped).

We explore below two other possibilities: the integration in the compiler of an ML interpreter (powerful enough to interpret syntax extensions), and a more classical solution which consists in preprocessing source files as the C preprocessor does.

These other possible implementations can be explained in light of refined views of the compiling process. Call *compile* a batch compiler including a set of syntax extensions. We said above that the compiler could be seen as

parameterised by a list of syntax extensions. In fact, if we go further in this direction, we can parameterise the ML parser only by a parsing process in charge of expanding quotations.

Traditionally, an ML batch compiler can be decomposed as follows:

$$\underbrace{gencode \circ typecheck \circ parse}_{compile}$$

where *parse* denotes the ML usual parser enriched with a set of syntax extensions.

This parser can be seen as the application of a raw ML parser *mlparse* parameterised by a mechanism of quotation expansion:

$$\underbrace{mlparse \text{ quotexpand}}_{parse}$$

Moreover, *quotexpand* itself can be decomposed in the application of an interpreter *exti* of extension definitions to a set of files containing syntax extension definitions:

$$\underbrace{exti \text{ extfiles}}_{quotexpand}$$

Therefore, a compiler without syntax extensions is:

$$\underbrace{gencode \circ typecheck \circ mlparse (exti [])}_{compile}$$

Finally, one can suppose that *mlparse* realises itself the application of the interpreter *exti* to extension files. Abstracting over extension files, we obtain:

$$\underbrace{gencode \circ typecheck \circ (mlparse \text{ exti})}_{compile}$$

Our implementation follows this decomposition: extension files are ML compiled files, and *exti* dynamically loads these files. Extension files typically contain calls to `new_syntax` that extend the current ML parser with new syntax bindings.

This decomposition of the compiling process appears to be useful for justifying the alternative implementations of quotation expanders given below.

7.1 Quotation interpreters

This implementation follows closely the decomposition above. It relies in the design of an extension language together with its interpreter *exti* (the language could be ML, but it could also be different). The interpreter is linked once for all with the compiler. Therefore, there is no need for dynamic loading of code. The resulting compiler, given a list of extension files, interprets these files, and is then able to compile source files containing quotations.

We did experiment with such a solution using a subset of ML as extension language, and the results were quite satisfactory, mainly because compilation times are considered as being less critical than execution times.

Different variants of this solution can be imagined, the most interesting of which would probably be an interpreter for simple grammars with ML quotations as semantic actions, and antiquotations restricted to single variables.

7.2 Preprocessing source files

Another possibility is to consider two applications: a text expander, acting as a preprocessor, composed of an ML extensible parser and a regular ML printer *regmlprint*, communicating with a regular ML compiler (*i.e.* with a regular ML parser *regmlparse*). The whole process is then the composition:

$$\underbrace{(gencode \circ typecheck \circ regmlparse)}_{\text{regular ML compiler}} \circ \underbrace{(regmlprint \circ (mlparse \textit{exti}))}_{\text{preprocessor}}$$

Because *regmlparse* \circ *regmlprint* is the identity function on ML ASTs, the whole process is functionally equivalent to:

$$gencode \circ typecheck \circ (mlparse \textit{exti})$$

The preprocessor can either be an ML parser statically linked to an interpreter of syntax definitions, or be obtained by a complete relinking of an ML parser with compiled syntax extensions (which is a lighter task than relinking a whole compiler).

This solution is certainly the least elegant of all because the source file actually compiled is the result of a preprocessing phase, in which source locations can be different from locations in the original source file. This could be avoided by omitting the creation of temporary files and exchange directly ASTs through a communication channel. This way, there is no need for a pretty-printer, nor for parsing expanded source files; two trivial programs for printing and parsing ASTs would suffice.

8 Interaction with module systems

The interaction with the Caml-Light module system is clear, since the quotation system is orthogonal to modules: quotations belong to the compiler, and the scope of syntax extensions is global to a compilation.

However, since quotation expanders are compile-time entities, they could be associated with (or be part of) other compile-time entities, as signatures in the SML module system. This would restrict the scope of syntax extensions to the scope of the signature they belong to.

It is useful to keep in mind that quotations allow for parameterising only *program texts*. One cannot think, for instance, of an SML structure or functor being parameterised by a syntax extension (*i.e.* a set of expanders): either the expanders are provided when compiling the implementation, and the compilation is possible, or the expanders are not known, and the implementation cannot be parsed, and therefore be compiled.

9 Related work

Caml quotations [15] were a generalisation to several object languages of the quotations of the original ML systems. Limited to the Caml lexical analyser, Caml quotations were strongly linked to the Yacc interface and Caml macros to the presence of dynamic types [15, 9]. Here, there are no such constraints: we have complete freedom

in the choice of lexing and parsing techniques, and no need for dynamic typing.

Slind presented in [14] the inclusion of quotations and antiquotations in SML-NJ. If used without antiquotations, this system implements quotation expansion just like a regular call of a user-defined parser to a string (*i.e.* expansion occurs at run-time).

Antiquotations are “holes” in quotations that are parsed like regular ML expressions. Fixed antiquotation marks (“~” or “~(...)”) are used to escape from an object language to ML identifiers or expressions. Given a call “p ‘...’”, the ML parser produces the ML application of the parser p to a list of elements belonging to a special polymorphic data type ‘a frag. This list is an alternation of QUOTE and UNQUOTE data constructors.

The expansion of quotations at run-time forbids the usage of quoted patterns, and implies that a quotation occurring in a function body will be parsed at each function call, in the general case. Furthermore, all antiquotations in a frag list must possess the same type.

SML-NJ quotations can be easily implemented in our system. We omit the code of *mk_frag*, which explodes a string into a list of Quote/Antiquote elements.

```
# type Frag a = [ Quote String | Antiquote a ] ;
Type Frag defined.

# value mk_frag = ...;

# new_syntax "Quote" (mk_frag, failwith);
- : Unit = ()

# <:Quote<^(True) /\ ~(False)>>;
- : List (Frag Bool) = [Quote ""; Antiquote True; Quote
e " /\ "; Antiquote False; Quote ""]
```

Aasa *et al.* [1, 2] present a general solution to the ambiguity problem in quotation patterns. They use the cooperation of a specific parsing mechanism (Earley’s algorithm [5]) with the type synthesizer. In this work, data type definitions can occur under a concrete form (in which case data constructors are hidden to the user), and the type definition includes precedence information for disambiguation. Type definitions act therefore as definitions of concrete syntaxes of quotations and antiquotations.

Aasa’s work has been improved by Petersson and Fritzon [12], who use a more efficient parsing algorithm [7].

Finally, we must mention that we have no support, in AST manipulation, for avoiding unwanted name clashes, which are frequent in presence of some forms of macro expansion. In [4], a technique for having extensible syntaxes preserving lexical scoping is presented. This technique relies on a grammar formalism and presents several advantages: it is language independent and allows incrementality in the language definition.

Our technique is much more basic: we do not rely on a particular parsing technique, nor on a particular language definition formalism. On the other hand, we cannot obtain the same nice properties (incrementality, parsing always terminates, no need for quotation marks).

Our work does not have exactly the same goal: we are interested in arbitrary language manipulation in ML, including ML programs themselves. In our opinion, extensibility is another topic: it concerns each of the object languages that we manipulate. Extensibility can be seen as orthogonal to our work, since it concerns parsing techniques and language definition formalisms.

10 Conclusion

We have presented the integration and the implementation of a general quotation mechanism in ML.

We have shown that separate compilation can be preserved and that the interaction with module systems remains clear, if there is a clear distinction between the compiler and the program actually compiled. This distinction is indeed the one between compile-time and run-time objects.

Syntax extensions are extensions of the compiler, and more precisely of its parser. Based on this remark, we proposed several reasonable alternative implementations, in order to accommodate compilers not featuring dynamic code linking.

Finally, quotations provide a reasonable way of introducing macro expansion in ML, without needing any dynamic typing [9].

References

- [1] A. Aasa, K. Petersson, and D. Synek. Concrete syntax for data objects in functional languages. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 96–105, 1988.
- [2] Annika Aasa. *User Defined Syntax*. PhD thesis, Chalmers University of Technology, 1992.
- [3] A. Appel and D. MacQueen. A Standard ML compiler. In G. Kahn, editor, *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1987.
- [4] Luca Cardelli, Florian Matthes, and Martin Abadi. Extensible syntax with lexical scoping. In *Proceedings of the Fifth Workshop on Database Programming Languages*, 1993.
- [5] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2), 1970.
- [6] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadworth. A metalanguage for interactive proofs in LCF. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 119–130, 1978.
- [7] Jan Heering, Paul Klint, and Jan Rekers. Incremental generation of parsers. *ACM Sigplan Notices*, 24(7), 1989.
- [8] Xavier Leroy. *The Caml Light system, release 0.6 – Documentation and user’s manual*. INRIA, 1993. Distributed with the Caml Light system.
- [9] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4), 1993.
- [10] M. Mauny. Parsers and printers as stream destructors and constructors embedded in functional languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*. Addison-Wesley, 1989.
- [11] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. The MIT Press, 1990.
- [12] Mikael Petersson and Peter Fritzon. A general and practical approach to concrete syntax objects within ML. In *Proceedings of the 1992 Workshop on ML and its Applications*, 1992.
- [13] Tim Sheard and Leonidas Fegaras. Fold for all seasons. In *Proceedings of the 1993 ACM Conference on Functional Programming and Computer Architecture*, pages 233–242, 1993.
- [14] Konrad Slind. Object language embedding in Standard ML of New-Jersey. In *Proceedings of the 1991 Workshop on ML*, 1991.
- [15] P. Weis, M.V. Aponte, A. Laville, M. Mauny, and A. Suárez. The CAML reference manual. Technical Report 121, INRIA, 1990.

The Assert example

The `Assert` syntax extension provides a mechanism for checking the truth of some boolean expressions, and printing an error message allowing to find the source location of the guilty program in case an assertion failed. The function `locmsg` returns a string indicating the source location of the assertion. The `assert` expander is defined by:

```
#value assert s =
# let e = parse_mlexpr s in <:Mlexpr<
# if #e# then () else
# raise (Assert(#:string (locmsg())~s#))
# >>;
assert : String -> Mlexpr = <fun>
```

`Assert` has no pattern expander.

```
#new_syntax "Assert" (assert,
# (fun _ -> raise(Failure "Assert in a pattern")));
- : Unit = ()
```

Compiled with the previous definition of the `Assert` expanders, the factorial function checks that its argument is not negative. With an expander returning the ML expression `<:Mlexpr< () >>`, the compiler would generate a code without assertion checks.

```
#exception Assert String;
Exception Assert defined.
#value rec fact = fun [
# 0 -> 1
# | n -> do <:Assert<n>= 0>>;
#       return n*fact(n-1)
#];
fact : Int -> Int = <fun>
#fact (5+6);
- : Int = 39916800
```

In case of failure, the `Assert` exception carries the file name (standard input, here) and the character ending the quotation of the assertion.

```
#fact (5-6);
Uncaught exception: Assert "std_in, Char 60, n>= 0"
#fact (5-7);
Uncaught exception: Assert "std_in, Char 60, n>= 0"
```

The Printf example

ML has no equivalent to the C function `printf`. One way to simulate it is to provide a special syntax to generate the correct sequence of printing commands. A syntax extension called `Printf` can therefore be introduced. Quotations consist in a format string, in the style of C formats, and a sequence of ML expressions in the same order as the `%` characters in the format. In the format, `%s` denotes a string argument, and `%d` an integer:

```
#value fname = "fact";
fname : String = "fact"
#let x = 5 in
#<:Printf< "%s(%d+1) is %d\n" fname x (fact(x+1)) >>;
fact(5+1) is 720
- : Unit = ()
```

The previous quotation expands to the following sequence of printing commands:

```
do print_string fname; print_string "(";
  print_int x; print_string "+1) is ";
  print_int (fact(x+1)); print_string "\n";
return ();
```

If the `Printf` quotation expander is carefully programmed, taking care of the source location of parts of the quotation, the typechecker is able to locate correctly the source of type errors:

```
#<:Printf< "This (%s) is not a string\n" 3 >>;
> Toplevel input:
><:Printf< "This (%s) is not a string\n" 3 >>;
>
> Expression of type Int
> cannot be used with type String
#<:Printf< "Ill typed: %d\n" (fact "zero") >>;;
> Toplevel input:
><:Printf< "Ill typed: %d\n" (fact "zero") >>;
>
> Expression of type String
> cannot be used with type Int
```

Generation of “fold” functions

So far, we mentioned only `Mlexpr` and `MLpatt` as pre-defined syntax extensions, for ML expressions and ML patterns. ML type expressions possess also their quotation expanders:

```
#<:MLtexpr< [ Nil | Cons a (List a) ]>>;
- : MLtype_expr = MLTE_sum 0 [("Nil", []); ("Cons", [M
LTE_lid 0 "a"; MLTE_app 0 (MLTE_uid 0 "List") (MLTE_li
d 0 "a")])]
```

Used as patterns with antiquotation marks, they allow to de-structure arbitrary type expressions.

Let us assume the existence of a primitive provided by the compiler returning the type expression to which a type constructor is bound. It is possible to implement an automatic generator of “fold” functions associated to types [13]. We implemented a simple version of this, embedded it into a syntax extension named `Fold`. Given a type definition for binary trees:

```
# type Btree a b = [
# Leaf a
# | Node b (Btree a b) (Btree a b)
#];
Type Btree defined.
```

we can produce the following function:

```
#value fold_Btree = <:Fold<Btree>>;
fold_Btree : (a -> b) -> (c -> b -> b -> b) -> Btree a
c -> b = <fun>
```

The expression produced by the `Fold` quotation is:

```
fun fLeaf -> fun fNode ->
  let rec fld = fun [
    Leaf x1 -> fLeaf x1
    | Node x1 x2 x3 -> fNode x1 (fld x2) (fld x3)]
  in fld
```

We can now use it to produce an arithmetic evaluator:

```
#value compute =
# fold_Btree (fun x -> x)
# (fun [ "+" -> prefix +
#       | "*" -> prefix *
#       | _ -> raise(Failure "Unknown op.")]);
compute : Btree Int String -> Int = <fun>
#compute
# (Node "+" (Leaf 1)
# (Node "*" (Leaf 2) (Leaf 3)));
- : Int = 7
```