Interactive Programming Environment for ML

Laurence Rideau, Laurent Théry

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Interactive Programming Environment for ML

Laurence Rideau and Laurent Théry

N˚ 3139

March 1997

THÈME 2

ISSN 0249-6399

Interactive Programming Environment for ML

Laurence Rideau and Laurent Théry

Thème 2 — Génie logiciel
et calcul symbolique
Projet CROAP

Rapport de recherche n ° 3139 — March 1997 — 27 pages

Abstract: This paper presents our experiment in building a programming environment for ML. The approach is based on reusability. From existing compilers we derive some tools that assist users in developing ML programs. ML being a strongly typed language, its typechecking algorithm plays a central role. So we present some tools that show how to make use of type information in a programming environment.

Key-words: ML, functional programming, programming environment, typechecking

(Résumé : tsvp)

Unité de recherche INRIA Sophia Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex (France)
Téléphone : 04 93 65 77 77 – Télécopie : 04 93 65 77 65

Environnement de Programmation Interactif pour ML

Résumé : Ce papier présente le résultat de nos expériences sur la construction d'un environnement de programmation pour le langage fonctionnel ML. Notre approche se fonde sur la réutilisabilité. Nous dérivons de compilateurs déjà existants des outils qui assistent l'utilisateur dans le développement de programmes ML. ML étant un langage fortement typé, son algorithme de typage y joue un rôle important. Ainsi nous proposons différents outils qui montrent comment les informations de typage peuvent être utilisées dans un environnement de programmation.

Mots-clé : ML, langage fonctionnel, environnement de programmation, inférence de type

1 Introduction

This paper presents our experiment in building a programming environment for ML. Our initial motivation was to understand to what extent a programming environment could benefit from some of the characteristics of ML, such as strong typing and modular organization. We have tested our ideas on two different implementations of ML: SML/NJ [1] and CamlLight [14]. This ensures that what is presented here is independent of a particular implementation of ML.

Even though we were experimenting, our concern was (and still is) to obtain a real programming environment for ML and not for a subset of it. This decision has had an important impact on the design. When dealing with a subset of the language it is possible to build tools like a parser or typechecker from scratch, while for the full language this is unrealistic. For this reason, we based our approach on software reuse, linking an existing programming environment generator Centaur [3] to an existing ML compiler. Note that it is possible *only if* the compiler is designed in such a way that it is easy to modify. Following the terminology of SML, we call such a compiler *open*. Obviously the two compilers we have been using are open.

The paper is organized as follows. We first present the different tools we have developed and we show how they integrate into the environment. Then we concentrate on how these tools work and what the implementation problems are. For homogeneity, the examples that illustrate these two sections use CamlLight syntax.

2 Presentation

Figure 1 presents a screen dump of the environment that we obtain. It is composed of four different windows. The upper left window is an editor that contains an incomplete program. The program is the result of the editor reading the file `test.ml`. As this file happens to be syntactically incorrect, the program is displayed with holes. The holes are represented by `SYNTAXERROR` keywords annotated with the piece of text that has been rejected. Furthermore while typechecking the program (even though it was incomplete), an error and a warning have occurred. The messages are displayed in the upper right window. Selecting the first message has highlighted the identifier `tll` that has caused the error. The lower left window is another editor, which contains a more substantial program. Associated to it, the lower right window gives the explanation of the type of the function `tags_ml`. The three dots in the explanation denote a reference to a part of the program. Selecting the first three dots of the explanation has highlighted the application of `open_in` to `filename`.

Now that we have given an overview of the kind of environment we are aiming at. We present the different tools we have developed, splitting them in two categories. Syntactic tools take into account only syntactic properties of ML programs, while semantic tools use some semantic aspects, mainly type information.

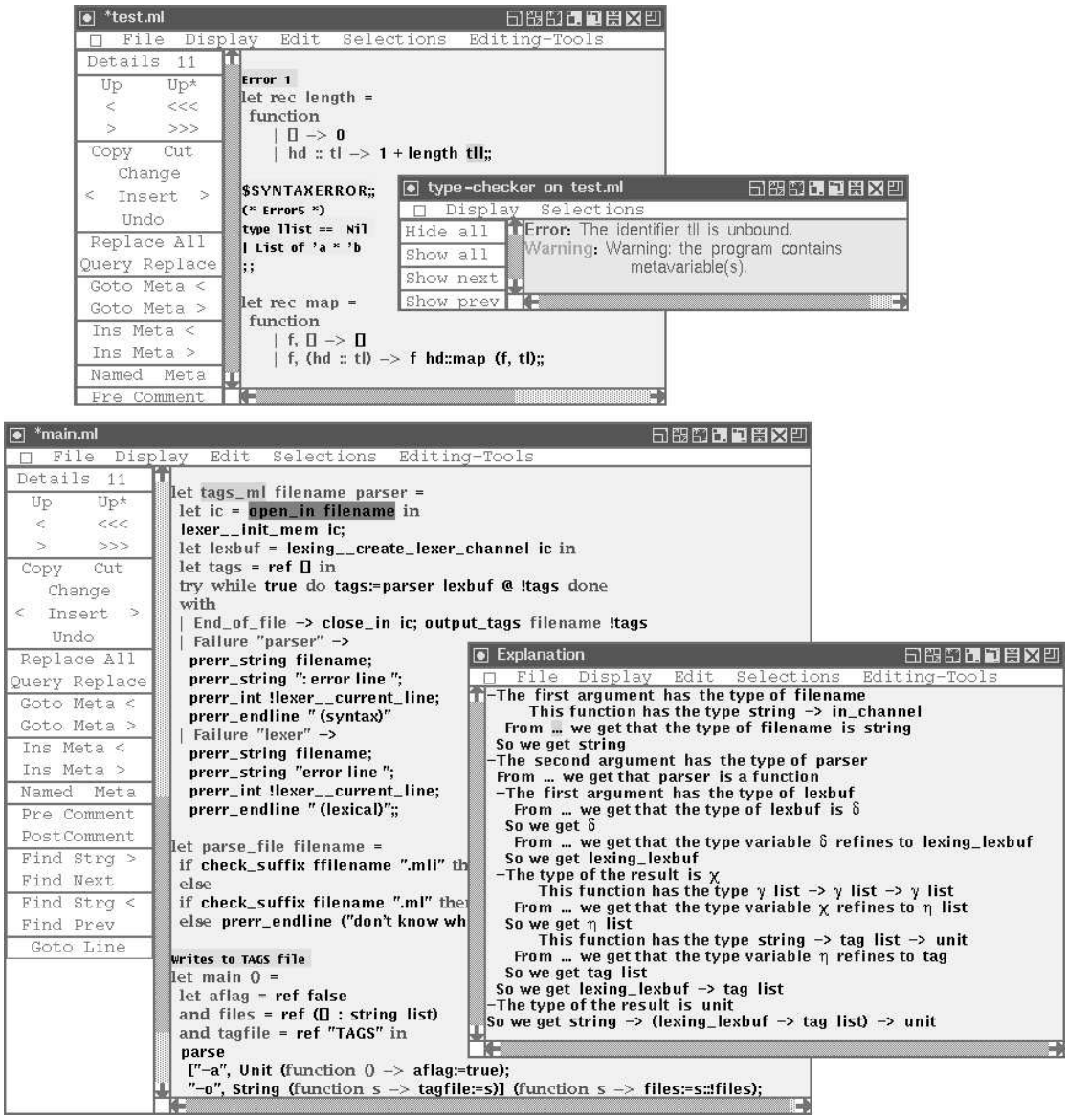


Figure 1: A screen dump

2.1 Syntactic tools

Syntactic tools provide the basic functionalities of the structured editor such as pretty printing, selection, and editing.

2.1.1 Pretty Printing

The recipes to increase legibility of ML programs are the usual ones. First, a special care must be taken for the layout, so that indentation and line breaking are as meaningful as possible. In that respect we try to follow the style advocated in books such as [18] and [21]. Second, tokens in programs belong to different syntactic categories, so it is important to visually distinguish them using fonts or colors. Here is an example of the black and white postscript generated by our pretty printer:

```
(*Definition of the map function *)
let rec map =
 function
    | f, [] → []
    | f, (hd :: tl) → f hd::map (f, tl);;
```

Note that comments are in typewriter style while patterns are in italic.

There are two main differences between the Centaur approach and the usual pretty printing provided by editors such as Emacs [4]. In Centaur, the layout is based on the abstract syntax tree representation of programs. Given a source file, it is first parsed to produce a syntax tree. Then using a set of rules, the layout of the tree is computed. So the pretty printing is *automatic* and is performed *independently* of the initial source layout. The second difference is that fonts and colors are selected using an exact syntactic criterion and not an approximation such as regular expressions, which are used in the Emacs highlighting mode.

2.1.2 Structured Selection

One of the benefits of having a close connection between the program and its tree representation is that we get for free the structured selection, i.e., a selection that follows the structure. It means that simply by selecting the token `if` of a conditional expression it is possible to highlight the whole expression. Our experiments have shown that having such a selection is essential: it provides a *natural* and *precise* way to show and designate locations in programs.

2.1.3 Detail Level

Even if it is good practice in ML to build its program as a set of small and modular pieces, activities such as browsing existing libraries make it always necessary to comprehend big chunks of code. Once again having the tree linked to the layout provides a natural elision mechanism. Using the depth in the tree, it is possible to replace expressions that are too deep in the program by an elision symbol such as "...". Here is the same program as section

2.1.1 but with a lower detail level:

```
(*Definition of the map function *)
let rec map =
 function
    | f, [] → ...
    | f, (hd :: tl) → ...;;
```

In the special case of ML, it has been necessary to refine the simple criterion of depth. For example, cascading `if` constructs gives nested conditionals in the tree structure. Elision computation has then to be rebalanced so that each branch of the conditional has the same detail level. It is also often more pertinent to show the pattern part of a matching with more detail than its body. The user can then only expand the cases he or she is interested in.

2.1.4 Guided Editing

Editing in a structured editor takes advantage of the connection with the abstract syntax tree to propose a menu with relevant tree constructions, i.e., constructions that are syntactically compatible with the selected place. The notion of incomplete programs is captured by a special syntax node to represent place holders in the program. So constructing a program is just a progressive refinement of the program, replacing place holders by expressions which may also contain other place holders. For example, let us suppose that we want to edit an expression place holder to get a condition whose test is a conjunction, we start with the place holder[1]:

< Exp >

We then select the item `if` of the menu showing us all the constructions in the syntactic category `Exp`, so we get:

`if` **< Bool >** `then < Exp > else < Exp >`

The menu now displays the constructions of the category `Bool`. To get a conjunction, we select the item `and`:

`if < Bool > & < Bool > then < Exp > else < Exp >`

If the previous example is a top-down construction, it is also possible to construct programs bottom-up. For example, given an expression, the menu offers two possibilities allowing to encapsulate this expression in a conditional: selecting the first item on the expression

`a+b`

gives

`if < Bool > then a+b else < Exp >`

[1] Place holders are surrounded with brackets and the boldface indicates the expression that is currently edited.

while selecting the other gives

```
if < Bool > then < Exp > else a+b .
```

More generally the menu should also provide for any kind of structured transformations. An example of such a transformation is the rewriting of a conditional into a matching. Given the conditional expression

```
if a<b then b-a else a-b
```

triggering the transformation produces:

```
match a<b with true -> b-a | false -> a-b.
```

The guided editing provided by Centaur accommodates these three kinds of editing. Even though menus are customizable, special care has to be taken to provide pertinent default menus. More details will be given in the implementation section.

2.1.5 Direct Editing

Structured editing should not rule out direct character based editing. They correspond to two complementary ways of constructing programs. In the Centaur system, structured editing is the default but direct editing is available as a special mode. When entering the direct editing mode, the user can freely modify the character string of the selected expression. Exiting the mode, the modified string is parsed and its abstract syntax representation replaces the subtree that was selected.

2.2 Semantic tools

The syntactic tools we have presented are very basic and they are a simple incarnation of generic tools provided by the Centaur system. Semantic tools are more specific to ML.

2.2.1 Type Information

ML is a strongly typed language. It means, in particular, that it is possible to associate a type with every expression of a well typed program. Knowing the type of an expression is a valuable information. It can be used as extra information to understand a piece of code but also as debugging information to understand why an expression is badly typed. ML uses type inference along with polymorphic types so discovering the type of an expression is not a straightforward operation. A tool to inspect types is then a natural need in a programming environment. How should this tool be integrated into the environment? A first attempt is to see the tool as a function that takes a subexpression of the program and returns its type. So for example, every time an expression is selected in the editor, its type appears in a dedicated type window. However, displaying only the type of the current expression is rarely sufficient as one often needs to see *simultaneously* the types of different expressions.

This is particularly true when polymorphic types are used intensively, it is then crucial to see which type variables are shared between expressions.

A second attempt is to use the program and the type constraint constructor to show types. Because ML uses polymorphic types, it contains a special constructor to constrain the type of an expression to a 'smaller' type than the one that would have been inferred. It is possible to use this constructor to hold the type of the expression, then inspecting types simply modifies the program. For example, given the following program:

```
let rec even x f g = (match (f x) with
                        0 -> true
                       |_ -> odd (g x) f g)
    and odd x f g = (match (f x) with
                        0 -> false
                       | 1 -> true
                       | _ -> even (g x) f g);;
```

and selecting the third occurrences of `f` and `g` we get:

```
let rec even x f g = (match (f x) with
                        0 -> true
                       |_ -> odd (g x) (f:'a->int) (g:'a->'a))
    and odd x f g = (match (f x) with
                        0 -> false
                      | 1 -> true
                      | _ -> even (g x) f g);;
```

The main drawback of this solution is code pollution, so after inspecting types a clean-up phase is necessary. A consequence of this pollution is that while inspecting types, the layout of the program is changing because of the insertion of types. In practice, the perturbation of the layout becomes quickly annoying.

Finally the solution we propose is to have a type window as presented in the first solution but to use it to display not a single type but a stack of types. Every time an expression is selected, its type is put on top of the stack. Moreover, types in the stack are linked back to the expression in the program, so that selecting a type in the stack highlights the expression that has the selected type. With this solution, the code is left unchanged and the types of different locations can be inspected simultaneously.

2.2.2 Type errors

Typecheckers provided by compilers only accept complete programs. So the typechecking phase is usually done once the program has been completely written. In our environment we manipulate incomplete programs, i.e., programs with place holders. It is natural to extend the typechecker so that it accepts incomplete programs. It is then possible to interact with the typechecker at any stage of the program construction leading to earlier detection of bad design decisions.

A second important characteristic of typecheckers is that all type errors of the whole program are to be reported. Not all compilers have this characteristic. For example the native CamlLight typechecker stops at the first error encountered. The main reason seems to be that a first type error could provoke other errors resulting in a large number of error messages. Our opinion is that it is arguable. First, examples such as the SML compiler show that there are techniques to restrain the influence of a type error. Second, handling a large number of error messages is only a problem in a poor programming environment. In a graphical environment, message browsers are adequate tools to handle a large quantity of error messages. Of course, this characteristic is a must when dealing with incomplete programs.

The last characteristic of typecheckers is the pertinence of its error messages. Elaborate solutions have been proposed to improve pertinence such as [13]. Our approach has been more pragmatic. In practice, compilers propose much simpler solutions. So we started from these simple solutions and tried to understand how they could benefit from the possibility of showing locations in the program. First of all, every error message needs to be associated with the *exact* location in the program where the error occurs. It appears also that some classes of errors can benefit from the possibility of showing several locations simultaneously. It is the case for applications where the type of the left part does not match the type of the argument. There is a conflict between the type that is expected and the type that is computed. To be more concrete, let us take a simple example with the following program

```
let rec f x = ..............
           ....(f (1,2))..
            ............
            ............
            (f (1,2,3))
```

and the proviso that `f` does not occur in the dotted part of the definition of `f`. While typechecking this program, an error occurs at the second application of f telling that the expected type is $int * int * int \rightarrow \alpha$ while the type that is computed is $int * int \rightarrow \alpha$. Our refinement improves this message by showing three locations:

```
let rec f x = ..............
            ....  (f (1,2)2 )..
            ............
            ............
            (f (1,2,3)3 )1
```

The first location is the place where the error occurs: the second application. The second location is the expression that has constrained the expected type to differ from the computed type. Here it is the pair `(1,2)` that forces `f` to accept a pair as first argument. The third location is the equivalent of the second location but for the computed type. So here it is the

triple `(1,2,3)`. This extra information is clearly valuable as both occurrences of application of `f` could be arbitrary distant.

2.2.3 Type Explanation

Even with the type inspection and the message browser there are still cases where finding the source of a type error is difficult. The typical situation is where the type that is inferred differs for what we expect and we cannot figure out why. Because of polymorphism, typechecking is very similar to prolog execution: during typechecking, the type of an expression is progressively unified against other types. To understand the type of an expression it is important to figure out which expressions in the program have contributed directly, or indirectly, to the result.

This idea of type slicing has originally been proposed as a way to explain type inference by Mitchell Wand in [20]. More recently, Dominic Duggan and Frederick Bent in [11] have extended the technique to handle several kinds of typechecking including ML typechecking. Another influential work in a related area has been the one described in [5]. In this paper, Coscoy et al. investigate the possibility to generate a natural language explanation from a mechanized proof done in Coq [8], a prover based on type theory. In that context, explaining a proof becomes explaining why a program has a given type. We have tried to equip our environment with a tool that follows these lines. Once a program has been typechecked, it is possible to have an explanation for the type of any expression in the program. Let us consider the following program

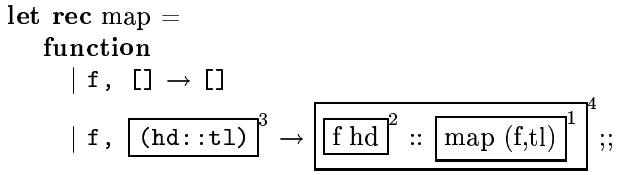
```
(*Definition of the map function *)
let rec map =
 function
     | f, [] → []
     | f, (hd :: tl) → f hd::map (f, tl);;
```

Selecting the first occurrence of `map` and pressing the button `Explanation` updates the window that is dedicated to explanation as follows:

From **1** we get that map is a function
– The type of the first argument is (*typeof* f) ∗ (*typeof* tl)
  From **2** we get that the type of f is (*typeof* hd) → β
  So we get ((*typeof* hd) → β) ∗ (*typeof* tl)
  From **3** we get that the type of tl is (*typeof* hd) list
  So we get ((*typeof* hd) → β) ∗ (*typeof* hd) list
– The type of the result is α
  From **4** we get that the type variable α refines to β list
  So we get β list
Altogether we have ((*typeof* hd) → β) ∗ (*typeof* hd) list → β list

Numbers represent hyperlinks that are connected to the program. Selecting one of these numbers highlights the corresponding part of the program. To emulate this behavior in this

presentation, we simply represent the correspond parts by boxes with the link number in superscript.



The explanation is presented as structured text. Each section introduces the type of a program expression which is then progressively refined. After each refinement, the type of the expression is repeated. Note also that the keyword *typeof* is used to link type variable to variable identifier in the text.

3 Implementation Issues

After having briefly described the different tools we have been developing, in this section we describe more deeply some tools stressing the problems that we encountered while implementing them.

3.1 Implementation of the user interface

As stated in the introduction, we develop an interactive programming environment on top of an existing ML compiler. We use Centaur, the programming environment generator, to derive most of the components of the environment such as editors, menus, pulldowns. If most of the graphical components we use could easily be implemented with other technologies such as Tcl/Tk [17] or Java/Awt [2], we rely heavily on structured editing. It means that the editor which is displaying the program under construction is aware of the underlying structure of ML programs. The editor does not consider a program just like a set of lines but it keeps a close link between what is displayed and the representation of the program as an abstract syntax tree.

3.2 Abstract Syntax and Parsing

Due to its distributed architecture, the Centaur system is open and can easily be connected to external components [10]. To communicate with external tools, Centaur provides a simple communication protocol. This protocol allows structured data transfer and remote operations.

Using these communication facilities, in our ML environment, parsing is performed by an ML parser that has been extracted from the ML compiler. This parsing produces ML abstract syntax trees that are then transferred to Centaur. On the ML side, we perform a

postorder tree traversal, sending to Centaur tree building commands according to a dedicated tree protocol defined in [9].

3.2.1 Overview of the Syntax Tree Protocol

In this section, we detail the protocol used to transfer an abstract syntax tree from ML to the programming environment. It assumes that the two components know the formalism[2] of the tree. The protocol design takes into account the potentially large size of the trees. To avoid problems during the transfer, the data is split into atomic data. With the abstract syntax tree protocol one has to send the tree node-by-node. The receiver re-builds the corresponding tree progressively. In case of a connection failure, the subtrees already sent need not to be re-sent. The data transfer can continue at any given subtree. The sender traverses the tree so that the receiver can incrementally construct the tree. The receiver manages a stack to store nodes needed for later constructions.

There are two types of nodes and, thus, two creation primitives: *make-leaf* and *make-node*.

- *make-leaf* <*formalism-name*> <*operator-name*> <*value*>
- *make-node* <*formalism-name*> <*operator-name*> <*number-of-sons*>

Note that the number of sons is used by the receiver to pop from the local stack. The *formalism-name* argument allows the transfer of multi-formalism trees. This formalism must then be known by both the sender and the receiver.

Furthermore, the transferred trees may be incomplete (i.e., containing placeholders to mark the location of the missing subtrees) or annotated (i.e., information is attached to the nodes).

3.2.2 Entry Points

When doing structured editing, entry points in the parser are crucial: they are used to parse sub-expressions. We don't want that a local modification forces the reparsing of the whole program. Unfortunately some compilers can compile only programs, the SML/NJ (resp. the CamlLight) parser has one single entry, the one for the declarations (resp. two entries, one for the declarations and one for the expressions).

To overcome this problem, we create a table that associates each entry point with two strings and a path. The two strings are used to encapsulate the string to be parsed so that the result is parsable at top level. For example, if we consider the entry point that corresponds to the types of the ML language, its two associated strings are `"let x = (y:"` and `")"`. Given a string that represents a type such as `"(int*int)list"`, the encapsulation creates `"let x = (y:(int*int) list)"` that represents a top level declaration. The path in the table is then used to extract the abstract syntax representation of the initial string from the resulting parsed tree.

[2] The formalism defines the abstract syntax of the trees.

3.2.3 Comments

The problem of comments is more delicate. Comments are useless for compilation, so they are usually thrown away during the parsing phase. This might not seem crucial as it is obvious that the comment mechanism can be replaced advantageously in an interactive programming environment by some fancier literate programming techniques *à la hypertext*. Nevertheless it seems rather unpleasant to get rid of this basic way to annotate programs.

Our solution takes advantage of the fact that most of the ML compilers need to keep extra location information inside parsing trees in order to be able to give more precise error messages. So during the lexical analysis we just record the list of comments with their locations. Then using the location information in the parsed tree, it has been possible to develop a good heuristic to hook comments back in the structured editor.

3.2.4 Syntax Error Recovery

A characteristic of parsers is that they generally stop at the first error encountered. Thanks to the structure of ML programs (list of declarations), it is always possible, when parsing a list of declarations, to continue the syntax analysis on the next declaration following a syntax error. Using the tree protocol facilities, in case of syntactic error, we send to the programming environment an incomplete syntax tree (with a placeholder marking the location of the missing declaration), annotated with the erroneous text where the syntax error has been detected.

With this technique, error recovery has a rough granularity, but the whole program is analysed, and errors can be corrected by editing the erroneous text in the programming environment.

3.2.5 Contextual information

In Centaur editing and prettyprinting are done in a context free manner. Unfortunately ML has some constructs that make it necessary to handle contextual information. For example, an infix or postfix declaration indicates that the parser should behave differently before and after the declaration. Rather than extending the general mechanism of editing and prettyprinting, we implement an intermediate solution where contextual informations are kept at the level of the abstract syntax tree. Then any expression within a given syntax tree is manipulated in the same context. For the implementation, this implies that the ML side that takes care of the parsing and typechecking has to maintain an environment for every program developed in the environment.

3.3 Pretty Printing

To describe the layout of programs, Centaur provides a pretty printing metalanguage called *PPML* [12]. A *PPML* specification is composed of a set of rules. The left part of the rule describes a pattern. The right part of the rule is a box description. The first element of a

box, written between brackets, is the *combinator*. It explains how the different elements of the box are to be displayed. An example of a rule is the following:

$$\text{plus}(*\text{x},*\text{y}) \longrightarrow [<\text{h } 1> *\text{x } \text{"+"} *\text{y}]$$

The pattern part of this rule is '`plus(*x,*y)`', variable prefixed by a star represents arbitrary term. The right part is composed of the box '`[<h 1> *x "+" *y]`'. The combinator of the box is `<h 1>`. The box contains three elements: the recursive call on the first son, the character "+", and the recursive call on the second son. When the rule is applied, the three elements are then displayed in horizontal mode with an interword spacing of one unit.

It is possible to associate graphic attributes (i.e., color and font) to elements of box. Symbolic names are used for attributes so that customization can be performed independently of the *PPML* specification. It is also possible to modify the computation of the detail level by matching the current level of the tree and changing the recursive calls. So for example, a modified version of the previous rule could be

$$\text{plus}(*\text{x},*\text{y})!^n \longrightarrow [<\text{h } 1> *\text{x}!^{n+1} \text{ in class} = \text{symbol: } \text{"+"} *\text{y}!^{n+1}]$$

In this case the character "+" will be displayed using graphical attributes of the class `symbol` while the sons of `plus` are recursively called with a detail level boosted by one unit.

Finally the selection mechanism is automatically derived from the PPML specification. It works as follows: when selecting a token of the pretty printer output, the rule that has produced the token is recovered. Then the tree that is selected is the one that has matched the left hand part of the rule. Applying this principle to the previous rule, selecting the "+" token will select the whole `plus` tree.

In both ML implementations, there are more or less one hundred operators in the abstract syntax. Pretty printing specifications are about one thousand lines, which correspond to two hundred rules. To our knowledge, none of the compilers appear to have a pretty printer for their abstract syntax, so it is difficult to evaluate the quality of the pretty printers we have obtained. Still, we believe our specifications are rather compact if we keep in mind that a specification describes not only the pretty printing but also the selection and the computation of the detail level.

3.4 Guided Editing Menus

The simplest menu that is provided once the abstract syntax is entered in the Centaur system is a hierarchical menu where the submenu of each syntactic category contains an item for each constructor. Such a menu is far from being usable, it gives a too detailed vision of the program construction.

A first operation is to regroup important constructs in a couple of pertinent items. An example of such a phenomenon is the '`let ...in ...`' constructor. It is represented in the abstract syntax by an operator that takes three arguments:

– a boolean that tells if the constructor is recursive or not,

- a list of bindings that associates values to local variables,
- an expression that delimits the scope of the previous bindings.

In Centaur, lists are explicit operators, so in the definition of the abstract syntax, the operator `ValBinding list` represents the list of bindings. An element of this list must belong to the syntactic category `ValBinding` whose only operator is `valbinding` which takes two arguments a pattern and an expression. From this it follows that an operator `let` has always a `Valbinding list` as its second argument, which has at least one element whose head operator is `valbinding`. Also, the `let` constructor is heavily used in ML so it makes sense to provide the user with two items: one for the recursive version and one for the non recursive one. It is possible to incorporate these two remarks in two rules of the menu:

```
let :
  *x {Expression} ->
     let(false(),ValBinding list(valbinding(*y,*z)),*t)
letrec :
  *x {Expression} ->
     let(true(),ValBinding list(valbinding(*y,*z)),*t)
```

As in *PPML*, identifiers prefixed by a star represent arbitrary terms. The first rule associated with the item `let` expresses that any term belonging to the syntactic category `Expression` may be transformed in a non-recursive '`let ...in ...`' construction.

A second operation is to introduce bottom-up rules. Top-down rules are interesting when starting to build a program from scratch. Bottom-up rules are more suited for modifying existing programs. If we come back to our previous example, creating a local variable on top of an existing expression is a frequent operation. It can be obtained by refining the two rules we have given previously.

```
let :
  *x {Expression} ->
     let(false(),ValBinding list(valbinding(*y,*z)),*x)
letrec :
  *x {Expression} ->
     let(true(),ValBinding list(valbinding(*y,*z)),*x)
```

We simply reuse the variable identifier of the lefthand side and make it appear in the body of the `let` as its third argument.

A last operation is to add some useful transformations. In our example, it may be handy to pass from a non-recursive local declaration to a recursive one. It can be done using two new rules.

```
+rec :
   let(false(),*y,*x) {Expression} -> let(true(),*y,*x)
```

```
-rec :
   let(true(),*y,*x) {Expression} -> let(false(),*y,*x)
```

The menu proposed in the Caml version contains 140 rules, among which 100 top-down rules, 20 bottom-up rules, and 20 transformation rules. Although we concentrated our effort in developing one general purpose menu for our environments, we are convinced that the environment should provide not one, but several different kind of menus. For example, it is obvious that our general purpose menu is far too complex for a beginner in ML.

3.5 Typechecking

Before entering into the details of the different tools that use type information, it is first necessary to give a quick introduction on how the ML typechecker works. Our goal here is not to present completely the algorithm (for this see [7], for example) but rather to introduce enough material to make the following discussion fruitful.

The typechecking of a program is performed by a single traversal of the data structure that represents the program. The central part of the algorithm is the unification algorithm that makes the expected type of an expression equal to the type that has been effectively computed. Take as an example the application that is composed of a function and an argument. The typechecking of an application is done in seven steps.

1. The expected type of the function is a function type $\alpha \to \beta$.
2. The type of the function is computed and gives the value γ.
3. The expected and computed types of the function are unified to $\alpha_1 \to \beta_1$.
4. The expected type of the argument is then α_1.
5. The type of the argument is computed and gives the value α_2.
6. The expected and computed types of the argument are unified leading to α_3. This unification gives the type $\alpha_3 \to \beta_2$ to the function since the unification of α_2 may have changed type variables contained in β_1.
7. The return type of the application is β_2.

Unification substitutes a type variable either to a type value or to another variable. This last operation is called *aliasing*. Taking a simple example, suppose that we want to unify $\alpha * \beta \to \gamma$ with $int * \beta \to \eta$. The result of unification is to substitute α with *int*, leaves β unchanged and aliases γ and η. Usually type variables are represented by pointers, i.e., reference cells in ML, then unification simply updates pointers. Figure 2 shows a graphical representation of unifying these two terms. Boxes represent type variables, instanciation is represented by a thin arrow.