A Table-Driven Compiler for Pretty Printing Specifications

Laurent Théry

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

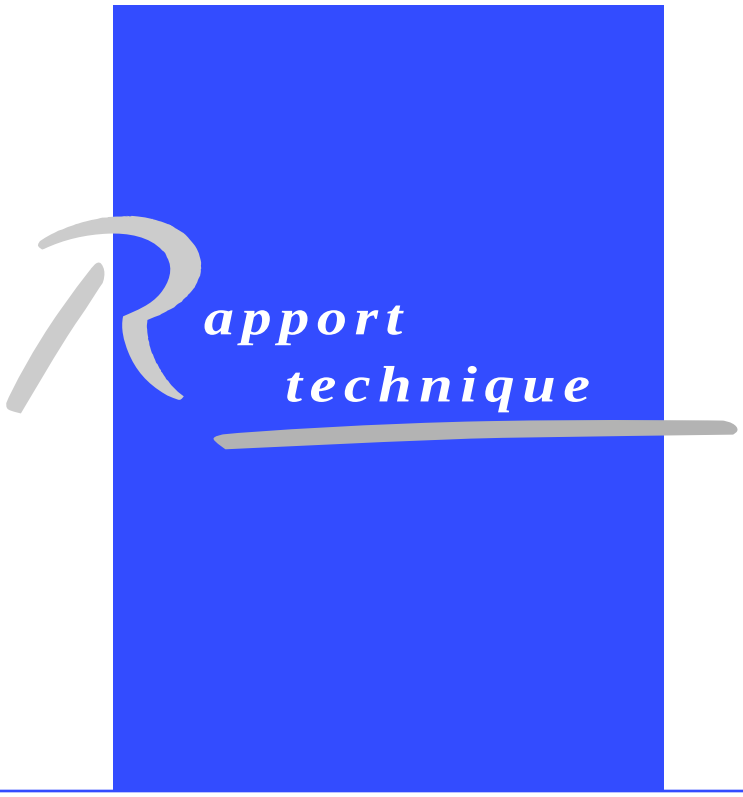
A Table-Driven Compiler for Pretty Printing Specifications

Laurent Théry

N° 0288

Octobre 2003

THÈME 2



ISSN 0249-0803

A Table-Driven Compiler for Pretty Printing Specifications

Laurent Théry

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lemme

Rapport technique n° 0288 — Octobre 2003 — 31 pages

Abstract: In this paper we present the design and the implementation of a compiler for Aïoli, a toolkit to build interactive and symbolic applications. This compiler produces executable code from pretty printing specification written in Ppml. The originality of the compilation is that it maintains dynamic links between the internal structure of the object that is displayed and what it is shown on the screen. It is then easy to build mechanisms such as structured navigation and incremental updating.

Key-words: Compiler, Pretty Printing, Incrementality, Ppml, Aïoli

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Un Compilateur Dirigé par les Tables pour des Spécifications d'Affichage

Résumé : Dans ce papier nous présentons la conception et l'implantation d'un compilateur pour Aïoli, une boîte à outils pour la construction d'applications symboliques et interactives. Ce compilateur permet de générer du code pour exécuter une spécification d'affichage écrite en Ppml. L'originalité de cette compilation est qu'elle permet non seulement la traduction de la structure de données vers la structure d'affichage mais aussi la navigation structurée et les modifications incrémentales.

Mots-clés : Compilation, Affichage, Incrementalité, Ppml, Aïoli

1 Introduction

When designing a system that iteratively manipulates and visualizes data, it is always good practice to have two separate structures: one for the manipulation and one for the visualization. This usually simplifies the implementation and also gives freedom in choosing the appropriate data representations to optimize manipulation and visualization separately. AÏOLI [1], a generic toolkit to build interactive and symbolic applications, uses this approach. It has a tree-like structure called VTP to represent data and a box structure called FIGUE to represent the layout. The connection between the two is expressed by a set of translation rules using the PPML [2] specification language.

In this paper, we are interested in building a compiler that turns PPML specifications into executable programs. AÏOLI provides support for building applications where the user follows or monitors transformations that are performed by a symbolic engine. It has been used for example to build the PCOQ proof environment [6] for the COQ proof engine [3]. In this context, the compilation has to do a bit more than a direct translation of the PPML rules. It is mandatory to be able to relate the data that is displayed with what is presented on the screen. This means that the compilation must generate extra information in order to preserve dynamic links between the object and its corresponding layout. In this respect, two basic operations are of special interest. The first one is structured navigation. This is the capability of navigating inside an object that is displayed on the screen in terms of the underlying structure of the object. The second one is incremental updating. This is the ability to update the layout of a modified object without having to recompute the whole pretty printing displayed structure. In the following, we show how these two operations have been made possible with our compilation schema.

The paper is structured as follows. In Section 2, we give a quick overview of the PPML language. In Section 3, we introduce the notion of path that is central in our compilation schema. In Section 4, we present the design of the compiler in a simplified framework where no list nodes are permitted. In Section 5, we illustrate on a small example how the compiler output can be used to get structured navigation and incremental updating. In Section 6, we give some details on how list nodes are handled. Finally, in Section 7, we give some factual information on the current implementation of the compiler.

2 Pretty Printing specifications

In our compilation, displaying an object is done by translating the object data structure into the layout data structure. We first introduce these two data structures before explaining what PPML specifications are.

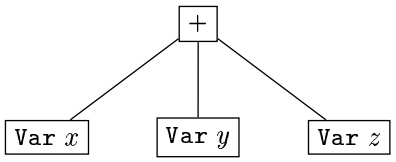
2.1 Trees

For the translation, we take as an assumption that the object to be displayed can be considered as a tree. This means that it has a *root* that corresponds to the top of the data structure

and each *node* of the tree has a name and a finite number of *subtrees* or *sons*. We consider that these subtrees are ordered so we can talk about the n^{th} son of a tree. We also consider some simple typing on nodes. There are three different kinds of nodes:

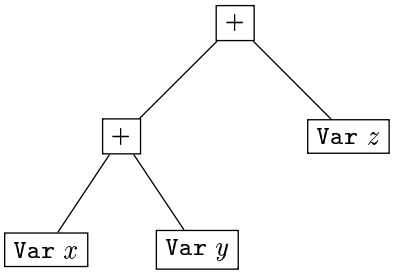
1. the regular nodes that have a fixed number of sons
2. the list nodes that have an arbitrary number of sons
3. the atomic nodes that have no sons but hold a value (integer, string).

Now let us consider some examples to get a feel for our definitions. First consider the expression $x + y + z$. If we choose to have the operator $+$ as a list node, its graphical representation would be:



The root of the tree is the expression $x + y + z$. It is composed of a list node $+$ that has 3 sons. Each son is an atomic node of type `Var` with an associated value.

An alternative representation of the expression $x + y + z$ is to consider $+$ as a binary node that is left associative. Thus, $x + y + z$ is the same as $(x + y) + z$. In this case we have the following tree representation:



The root is composed of a binary node $+$ with two sons $(x + y)$ and z. The first son is a node $+$ that has two atomic sons x and y.

Note that the assumption we made to consider objects as trees does not preclude the internal structure of an object sharing sub-parts or infinite cycles. It just implies that for pretty printing it will be processed as a regular and so possibly infinite tree.

2.2 Boxes

For the layout, we use nested boxes. A box is composed of a combinator and list of sons. The combinator gives the directives on how the components in the box should be combined. Two typical combinators are the horizontal combinator and the vertical combinator. Terminal elements in boxes are string values. If we take our previous example $x + y + z$, its layout

is composed of five strings `x`, `+`, `y`, `+`, `z`. Putting these strings in a box with a horizontal combinator[1] we get:

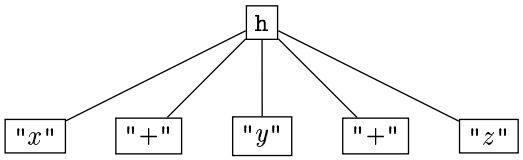
$$x + y + z$$

Alternatively having a vertical combinator would give:

$$\begin{array}{c} x \\ + \\ y \\ + \\ z \end{array}$$

The expressiveness of the layout structure relies obviously on the library of combinators. Having just the horizontal and vertical combinators is merely sufficient to display programs adequately. Getting a correct layout for more elaborated objects, such as mathematical objects, requires a far richer collection of combinators. This aspect is explained in [4].

As box structures can also be seen as trees, we use the same graphical representation as before. For example, if we consider the first layout, a horizontal box with five sons, its graphical representation is:



It is composed of a node with the horizontal combinator and five terminal strings.

2.3 Ppml

PPML [2] is a dedicated language for describing translations between tree structures and box structures. A PPML specification is composed of a sequence of rules. Each rule has a *pattern* and a *skeleton* separated by an arrow:

Pattern `->` *Skeleton*`;`

Patterns belong to the world of trees while skeletons belong to the world of boxes. Given a tree, a rule is applicable if its pattern filters the tree. The result of the application is described by the skeleton. When in a PPML specification more than a rule is applicable, it is the first one in the sequence order that is triggered.

The pattern syntax is the usual applicative one: nodes are applied to sons. For regular nodes with fixed arity, the parentheses are used. For list nodes, the square brackets are used. For atomic trees, the value is put directly after the node type between double quotes. To

[1] In order to simplify the presentation, we do not take into account the spacing information. It is given as a parameter of the combinator.

give some examples, first consider the tree $x+y+z$ where $+$ is a list node. Its representation as a pattern is

```
plus[var "x", var "y", var "z"]
```

With the binary representation instead, it would be:

```
plus(plus(var "x", var "y"), var "z")
```

Thus far we can only write patterns that represent a given tree. In practice, patterns should capture not a single tree but a whole family of trees. For this we use variables. In the same way that there are three types of nodes, there are three kinds of variables. A regular variable stands for any tree. It is represented by a name prefixed by a star. A sublist variable can only appear under a list node and stands for an arbitrary number of consecutive sons. It is represented by a name prefixed by a double star. An atomic variable stands for any atomic value. It is represented by a name prefixed by a star. Using variables, we can now give examples of more elaborated patterns. The following pattern:

```
plus[var *x, **y, *z]
```

accepts a tree whose root node is `plus` and that has at least two sons, the first one being an atomic node `var`. If we try to filter the tree `plus[var "x", var "y", var "z"]` with this pattern, it succeeds: the atomic variable `*x` is associated with the atomic value `"x"`, the sublist variable `**y` is associated with the sublist composed of the single tree `var "y"` and the tree variable `*z` is associated with the last son `var "z"`.

Now let's turn our attention to other part of our PPML rules, the skeleton. For the syntax of skeletons, the square brackets, the angle brackets and the double quotes are used as delimitators for boxes, combinators and strings respectively. To write the skeleton corresponding to a horizontal box with 5 strings for the expression $x + y + z$ we get:

```
[<h> "x" "+" "y" "+" "z"]
```

When the pattern associated with a skeleton introduces variables, we can use these same variables in the skeleton. A tree variable in a skeleton represents a recursive call to the set of rules with the corresponding subtree. An atomic variable in a skeleton represents the terminal string corresponding to the atomic value. For sublist variables, the situation is slightly more complicated since sublist variables may represent more than one tree. To handle them, there is a dedicated iterative constructor. The iteration is syntactically delimited by parentheses. They indicate the part of the skeleton that has to be duplicated as many as times as there are subtrees in the sublist. To give a concrete example, let us consider the following two rules:

```
plus[*x,**y] -> [<h> *x ( "+" **y )];
var *x -> *x;
```

If we use these rules to translate the tree `plus[var "x", var "y", var "z"]` we get the corresponding layout structure `[<h> "x" "+" "y" "+" "z"]`. Note that had we chosen + as a binary node, it would not have been possible to get such a flat structure. For example, the rules

```
plus(*x,*y) -> [<h> *x "+" *y ];
var *x -> *x;
```

produces a nested box structure `[<h> "x" "+" [<h> "y" "+" "z"]]`.

In order to accommodate alternative ways of pretty printing objects, a PPML specification defines only a specific way of pretty printing for a specific language. This information is given at the beginning of the specification. For example, if our nodes `plus` and `var` belong to a language `exp` of arithmetic expressions, the previous two rules can be declared as being the standard way (`std`) of pretty printing arithmetic expressions by:

```
pretty printer std of exp is

plus(*x,*y) -> [<h> *x "+" *y ];
var *x -> *x;

end pretty printer
```

It is also possible to extend existing pretty printers with new rules. For example, we can defined the specific (`spc`) pretty printer extending the standard one:

```
pretty printer spc of exp extends std

plus(plus(*x,*y),*z) -> [<h> "(" *x "+" "*y" ")" "+" *z ];

end pretty printer
```

The resulting pretty printer behaves as if we had written:

```
pretty printer spc of exp is

plus(plus(*x,*y),*z) -> [<h> "(" *x "+" "*y" ")" "+" *z ];
plus(*x,*y) -> [<h> *x "+" *y ];
var *x -> *x;

end pretty printer
```

Other aspects of PPML that are worth mentioning are the holophrasting mechanism, the context management and the annotation mechanism. The holophrasting gives the possibility to elide some part of the data structure during pretty printing. Holophrasting is represented in the PPML specification by a number. It is specified using the keyword `!`. For example the following rule

```
*x!0 -> "..."
```

indicates that if any tree reaches the holophrasting level 0, it is printed as "...". The computation of the holophrasting level works as follows. At the beginning of the pretty printing phase, the holophraste level is set to an initial value fixed by the user. Then at each rule application, the level is decremented by one. This is the default behavior. However, at any time in a skeleton it is possible to modify the level explicitly by using the keyword `!`. For example, the rule:

```
plus(*x,*y) -> [<h> *x "+" *y ]
```

is equivalent to the rule with explicit holophrasting:

```
plus(*x,*y) -> [<h> *x!-1 "+" *y!-1 ]
```

By slightly modifying this rule as follows:

```
plus(*x,*y) -> [<h> *x "+" *y!+1 ]
```

we ask the detail level to be incremented by one instead in the second recursive call. The result is that the second son of a `plus` will always be printed with more detailed than the first one.

The context management allows one to define rules that can be applied only locally. A context is referred by a name. A rule belongs to a context if its pattern is prefixed by the name of the context followed by a colon. Rules with no prefix belongs to the empty context. A recursive call is done in a specific context if the variable in the skeleton is prefixed by the name of the context followed by a colon. By example, the rule

```
ctx1:plus(*x,*y) -> [<h> *x "+" ctx2:*y ]
```

is only applicable in the context `ctx1`. The first recursive call is made in the empty context, the second one in the context `ctx2`.

Finally annotations are bits of information that can be attached to a node but do not belong to the data structure. A typical application of annotations is the handling of comments. Comments are not relevant when manipulating the data structure but are essential when doing pretty printing. The keyword `^` makes it possible to take annotations into account in the PPML. For example, the rule

```
*x^comment -> [<v> ^comment *x]
```

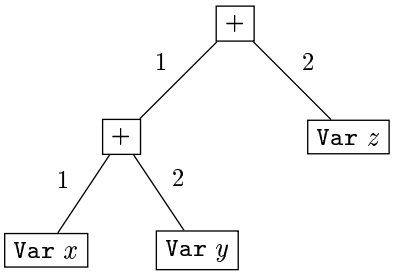
states that if a tree has an annotation of type `comment`, the value of the annotation should be printed in vertical mode before the tree.

3 Paths

The PPML specification defines a translation. Both the data structure and layout structure can be represented as trees. An important notion for our compiler design is the one of paths. Three different notions of path are needed. A single path denotes a single location in a tree. A multiple path denotes simultaneously different locations in a tree. A modification denotes a modification on a tree.

3.1 Single Path

To denote a location in a tree we make use of the fact that for each tree the list of its sons is ordered. Each subtree has a rank. The path associated with a selected tree is composed of the ranks of successive subtrees that are visited when going from the root to the selected tree. Syntactically the ranks are separated by dots and every single path ends with the symbol `s`. For example, let us annotate each subtree of the binary version of the expression $x + y + z$ with its rank, we get



It follows from the definition that the path of $x + y + z$ is `s`, the path of $x + y$ is `1.s`, the path of x is `1.1.s`, the path of y is `1.2.s` and the path of z is `2.s`

3.2 Multiple Path

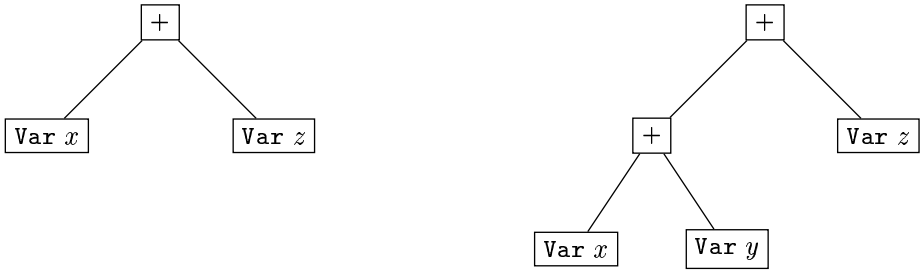
Multiple paths are a direct generalization of single paths where several locations are selected simultaneously. Rather than having just a list of single paths, common subpaths are shared. For example the multiple path that selects all the subtrees of the previous tree is

```
s
1.s
  1.s
  2.s
2.s
```

In this textual representation, indentation is used to indicate common subpaths.

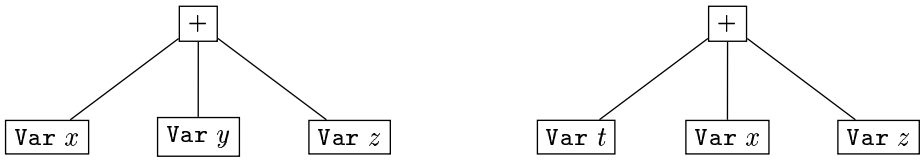
3.3 Modification Path

While single and multiple paths are used to denote locations on trees, modification paths are used to express transformation. We consider three types of modification: change, insertion, deletion. The change of a tree by another one is denoted by `c[`$\mathtt{v}_1$`,`$\mathtt{v}_2$`]` where $\mathtt{v}_1$ is the old tree and $\mathtt{v}_2$ the new one. If we consider the transformation from the tree on the left to the tree on the right



the corresponding modification path is `1.c[`x`,`$x + y$`]`. In the following we usually omit the arguments and simply write `1.c`.

The two other modifications are insertion and deletion. These concern only list nodes. We denote `i[v]` the insertion of the tree `v` and `d[v]` the deletion of the tree `v`. As deletion and insertion changes the node arity, it is necessary to fix some conventions for interpreting modification sequences. To understand why the interpretation could be problematic, let us consider the following transformation



One way to explain this transformation is to say that t has been added in first position and then y has been deleted. But when generating the path we have two choices for expressing the deletion, either consider the rank of y in the initial tree (2) or its rank after the insertion (3). The first choices gives the path:

```
1.i
2.d
```

while the second one gives:

```
1.i
3.d
```

In our implementation we favor the second interpretation. The main reason is that multiple modifications are usually created by accumulating progressively single modifications. Writing an accumulation algorithm in the second case is then simpler. The drawback of the second solution is that if we can still share common subpaths the order in the paths is now relevant. The paths

```
1.i
3.d
```

and

```
3.d
1.i
```

denote two different transformations.

4 Design of the Compiler

Now that the different actors of the compilation have been introduced, we can explain how the compilation schema works. The result of the compilation can be separated in three distinct components. The first component takes care of filtering. Given a tree it computes which rule has to be applied. The second component is the one that is in charge of building the layout structure. Given a rule and the values of the recursive calls it produces the complete structure. The third component is a table of correspondence between paths in the tree structure and paths in the display structure.

To simplify the presentation we first assume that we are not using list nodes. The discussion of how list nodes are dealt with is postponed until Section 6. We also use the same example all through this section. We use arithmetic expressions built using a binary node `plus` and an atomic node `var`. + is considered left associative. This means that $x + y + z$ is represented as `plus(plus(var "x",var "y"), var "z")`. In the following, we usually refer to the structure of the object as the data structure or the tree structure, we refer to the structure of the layout as the layout structure or the box structure.

4.1 Pattern Matcher

To compile patterns without list nodes we just need two operations on trees. The operation $\downarrow$ gives access to subtrees. It takes a tree and a single path and returns the tree denoted by the path. The operation n gives access to the name of a node. If we consider the expression of $x + y + z$, then $(x + y + z) \downarrow_{1.2.s} = y$ and $n(x + y + z) =$ `plus`.

Using these two operations it is easy to check if a rule is applicable. For example checking if a tree t is filtered by `plus(plus(*x,*y),var *z)`is equivalent to the formula:

$$n(t) = \texttt{plus} \quad \&\& \quad n(t \downarrow_{1.s}) = \texttt{plus} \quad \&\& \quad n(t \downarrow_{2.s}) = \texttt{var}$$

A naïve method for finding which rule applies is to check the tree against the formula of the first rule of the specification. If it fails, we repeat the same operation with the formula of the second rule, and so on, until finding one rule whose formula is valid for the given tree. This method is correct but clearly inefficient. As described in [7], we can instead build a decision tree in order to check all the formulae in one shot. Let us explain how it works on the following three rules:

```
plus(*x,plus(*y,*z)) -> "1"
plus(plus(*x,*y),*z) -> "2"
plus(var *x, var *y) -> "3"
```

The corresponding formulae are:

$n(t) = \texttt{plus} \quad \&\& \quad n(t \downarrow_{2.\texttt{s}}) = \texttt{plus}$
$n(t) = \texttt{plus} \quad \&\& \quad n(t \downarrow_{1.\texttt{s}}) = \texttt{plus}$
$n(t) = \texttt{plus} \quad \&\& \quad n(t \downarrow_{1.\texttt{s}}) = \texttt{var} \quad \&\& \quad n(t \downarrow_{2.\texttt{s}}) = \texttt{var}$

The decision tree is built by adding the formulae one by one. Starting from the first formula, the initial decision tree looks like the following using a JAVA-like syntax:

```
if (n(t) == plus) {
  if (n(t ↓2.s) == plus) {
    return 1;
  }else{
    return 0;
  }
}else{
  return 0;
}
```

The program returns 1 if the first rule is applicable, 0 otherwise. For adding the second formula, we only need to consider the two statements that returns zero. For the first one, we are already in a context where the top node is a `plus`, so only the second half of the second formula needs to be checked. For the second statement, it concerns tree whose top operator is not a `plus`, so it does not concern the second rule. The new decision tree is then:

```
if (n(t) == plus) {
  if (n(t ↓2.s) == plus) {
    return 1;
  }else if (n(t ↓1.s) == plus) {
    return 2;
  }else{
    return 0;
  }
}else{
  return 0;
}
```

In the same way, the third formula refines even further the decision tree:

```
if (n(t) == plus) {
  if (n(t ↓2.s) == plus) {
    return 1;
  }else if (n(t ↓1.s) == plus) {
    return 2;
  }else if (n(t ↓1.s) == var) {
    if (n(t ↓2.s) == var) {
      return 3;
    }else {
      return 0;
    }
  }else{
    return 0;
  }
}else{
  return 0;
}
```

Once built checking which rule, if any, applies to a tree corresponds to executing the decision tree.

4.2 Box Builder

The second component is a far simpler component. It is used for building boxes. Given a rule referred by its index, it returns a box structure where variables are replaced by dummy boxes `?`. The purpose of dummy boxes is just to preserve correct ranking in boxes. For example given the skeleton

```
[<h> *x "+" *y]
```

the box builder inserts the box:

```
[<h> ?  "+"  ?]
```

at the rule index of the global table. The first dummy box represents the first element of the box and corresponds to the recursive call associated with the variable `*x`. The string `"+"` is the second element of the box. The second dummy box represents the third element of the box and corresponds to the recursive call associated with the variable `*y`.

4.3 Path Table

The third component is a table of correspondence. For each rule and for each variable in the rule, it associates the path of the variable in the pattern with the path of the variable in the skeleton. Let us illustrate it with the following example:

```
plus(*x,*y) -> [<h> *x "+" *y ];
```

This rule has two variables. The first one `*x` represents the *first* son of `plus` node in the pattern and also the *first* son in the horizontal box. The second variable `*y` represents the *second* son in the pattern and the *third* son in the box. The table has then two entries for the given rule:

```
1.s  ⟶  1.s
2.s  ⟶  3.s
```

The same principle applies to atomic variables. The only difference is that their entries are singled out in the table by the symbol $\longmapsto$ to indicate that they do not correspond to a recursive call but actually to a terminal string. For example the rule:

```
plus(var *x,var *y) -> [<h> *x "+" *y ];
```

generates the following two entries:

```
1.s  ⟼  1.s
2.s  ⟼  3.s
```

5 A Toy Example

We are now able to give the three resulting components for our very simple PPML specification:

```
plus(*x,*y) -> [<h> *x "+" *y];
var *x -> *x
```

The pattern matcher has the following shape:

```
if (n(t) == plus) {
    return 1;
} else if (n(t) == var){
    return 2;
} else {
  return 0;
}
```

the box builder:

```
1:
   [<h> ? "+" ?]
2:
   ?
```

and the path table:

```
1:
   1.s  ⟶  1.s
   2.s  ⟶  3.s
```

2:

```
s  ⟼  s
```

5.1 Translation

To show how the compilation results can be used to do pretty printing, let us consider the expression `plus(var x, plus(var y, var z))`. Using the pattern matcher, we get that the first rule applies. Using the box builder with one as an index we get:

```
[<h>  ?  "+"  ?  ]
```

There are two entries in the path table for the first rule. Both represent recursive calls. The first entry is

```
1.s  ⟶  1.s
```

This tells us that the first son of the tree, in our case `var x`, corresponds to the first element of the box (i.e, the first dummy box). The second entry is

```
2.s  ⟶  3.s
```

This tells us that the second son of the tree, in our case `plus(var y, var z)`, corresponds to the third element of the box (i.e, the second dummy box).

We are now free on how we proceed. Suppose we want first to fill the second dummy, we then call the pattern matcher on `plus(var y, var z)`. It tells us again that the first rule applies. Calling the box builder refines our result:

```
[<h>  ?  "+"  [<h> ?  "+" ?  ] ]
```

We have now three dummies corresponding to the pretty printing of `var x`, `var y`, and `var z` respectively. Let us proceed with `var x`, the pattern matcher tells us that the second rule applies. Calling the box builder gives us the dummy box `?`. The path table tells us that the one entry corresponds to an atomic variable, so we can directly replace the dummy box with the atomic value of the tree in our case `"x"`. In consequence, we now have:

```
[<h> "x"  "+"  [<h> ?  "+" ?  ] ]
```

Doing the same for the other two variables gives us the expected final result:

```
[<h> "x"  "+"  [<h> "y"  "+"  "z"  ] ]
```

As we have seen, the result of the compilation contains all the information needed to perform the translation. Having separated the filtering from the actual building of the layout structure gives us complete freedom to choose the appropriate strategy. It means for example that without changing our compilation schema, we could develop a lazy way of building the layout that would not require having the entire object when starting the translation. The layout could be constructed progressively as more is known about the object to display.

5.2 Structured Navigation

Structured navigation is made possible if we are able to relate subparts of the tree structure with corresponding subparts of the box structure. Rephrasing it in term of paths, it is made possible if we are able to translate paths of the tree structure into corresponding paths of the box structure. The key information that is needed in order to do that is a trace of the different rules that have been applied during the pretty printing translation. We will refer to this trace as the box-rule trace. The trace has a tree-like structure that mimics the recursive calls of the pretty printing phase. Coming back to our previous example, the trace corresponding to the translation of `plus(var "x", plus(var "y", var "z")` is the nested list `(1 (2) (1 (2) (2)))`. The top rule was the first one. It has two recursive calls. For the first one, the second rule applied with no recursive call. For the second recursive call, the first rule applied and generated two recursive calls with the second rule.

Now suppose we want to know which part of the layout corresponds to the subtree `var "y"`. We proceed as follows. We start by computing the path of the subtree `var "y"` in the tree structure. It is `2.1.s`. The box-rule trace tells us that the top rule was the first one. We look up for the path table entries of this rule. It gives us:

```
1:
   1.s  ⟶  1.s
   2.s  ⟶  3.s
```

As our path starts with a 2, only the second entry is relevant. It gives us the information that the corresponding path in the box structure starts with a 3. We can then proceed with the remaining subtree path `1.s` for `var "y"` and the box-rule trace associated with the second recursive call `(1 (2) (2))`. Again the first rule was applied, we have to consult our path table with the same two entries. This time the first entry is relevant and, given the path tree correspondance, it adds a `1` to the path in the box structure. As all our initial path has been completely consumed, we are done. The resulting path is `3.1.s`.

In exactly the same way we can perform the translation the other way around, taking a path in the box structure and translating it progressively using the trace. Having this capability of translating forth and back between paths gives us a notion of structured selection for free. To give a concrete example consider the case where PPML is used for displaying programs of some language with a conditional statement. A typical rule for pretty printing such a statement is the following:

```
if(*test,*then,*else) -> [<v> [<h> "if" *test "{"]
                                *then
                              "}else{"
                                *else
                              "}"
                          ]
```

When selecting with the mouse on the screen the `"if"` token, using path translation the corresponding conditional expression is found. Translating back the path of the conditional expression gives the box to highlight in order to show the whole conditional statement.

Two aspects are worth noticing with respect to this translation mechanism. The first one concerns the linearity of PPML rules. So far in our examples variables have appeared only once in the pattern and once in the skeleton. Nothing forbids having multiple occurrences of the same variables. An example where having multiple occurrences in the skeleton is interesting is when pretty printing a section that has a name. A natural PPML rule is the following:

```
section(*name,*block) -> [<v> [<h> "begin" *name]
                              *block
                              [<h> "end"   *name]
                          ]
```

We use the name twice to start and close the section. Multiple occurrences could also be handy in patterns to denote identical subterms. For example we could have the following ad-hoc rule:

```
plus(*x,*x) -> [<h> "2" *x]
```

Our translation mechanism still works in the presence of multiple occurrences of variables. What is lost is just the one to one correspondence between paths in the tree and paths in the box. A single path in the tree may correspond to a multiple path in the box and conversely.

The other aspect worth noticing is that links between the data structure and the layout structure are computed dynamically using paths. This means that our schema is an ideal candidate for a distributive environment where the process that holds the data and the one that visualizes it are distant. The two processes only need to communicate via paths. We could even easily accommodate the situation where several processes visualize the same shared data. Furthermore, for a process to perform path translation locally it just needs to have the box-rule trace that is a much simpler object to transfer than the whole data structure.

5.3 Incremental Updating

To get incremental updating we simply need to push a little bit further the idea of translating paths given in the previous section. Modification paths are not so different from selection paths. In the previous section's example, the translation of the tree path `2.1.s` resulted in the box path `3.1.s`. It is easy to see that the translation of the tree modification path `2.1.c` should give the box modification path `3.1.c`. So the same translation mechanism seems to work. This is almost always true except when, given that patterns can be arbitrarily elaborate, a modification of a subtree makes rules that were applied earlier in the layout box construction no longer applicable. An extreme case is the one of the nonlinear rule:

```
plus(*x,*x) -> [<h> "2" *x]
```

Suppose we have translated a `plus` tree with two identical sons. The rule was then applicable and is recorded in the box-rule trace. Now any modification, at any depth, in only *one* of

the two sons invalidates the rule application. This means that when walking down the modification path using the box-rule trace, it is also necessary to check whether the rules that were applied are still valid. So while the path translation could be done without the actual data structure, for incremental updating the data is needed. Note that updating is still incremental in the sense that we do not recompute the whole layout, but only reevaluate the layout on the path leading to the modification. Rather than being proportional to the length of the data structure, applying a modification is proportional to the size of the modification path.

Incremental updating relies heavily on the fact that a rule's applicability for a given term only depends on its subterms. This means that to perform incremental updating we just need to be able to construct the appropriate modification paths. Without sharing the situation is rather simple. A single modification on the tree structure corresponds to a single modification path. The situation becomes more delicate with shared data structures. A single modification on the tree structure could correspond to a multiple modification path. Generating appropriate modification paths in the presence of shared data would then require a very high control on sharing which is unlikely to happen in practice.

6 Adding Lists

In Sections 4 and 5, we have purposely omitted list nodes. List nodes make the compilation more difficult. Before explaining why it is so difficult, let us say a few words on why having list nodes is good thing in the first place. The alternative to list nodes is nested lists à la LISP which are somewhat less intuitive. Not having list nodes would mean not being able to reflect data structures having arrays. From the point of view of efficiency, having nested lists implies having data structures that are more deep than broad. This has some impact on our architecture since we heavily rely on paths. Paths for nested lists are significantly longer since they indicate the depth of the subtree. Most of our operations being proportional to the size of the path, using nested nodes would result in being less efficient.

In the following we present how list nodes have been integrated into the different compiler components always using our arithmetic expression example but this time with a `plus` node implemented as a list node.

6.1 Pattern Matcher

To handle lists, our pattern matcher needs to be able to determine a tree node's arity, in addition of being able to access the subtrees and the node's name. Thus we add a third operation, a, that gives access to the node's arity. If we consider the expression $x+y+z$, we have $a(x+y+z) = 3$. Checking if a tree t is filtered, for example, by `plus[plus[*x,*y],**z]` is equivalent to the formula:

$$n(t) = \texttt{plus} \quad \&\& \quad 1 \leq a(t) \quad \&\& \quad n(t \downarrow_{1.\texttt{s}}) = \texttt{plus} \quad \&\& \quad a(t \downarrow_{1.\texttt{s}}) = 2$$

When building the decision tree it is necessary to dispatch the arity information properly. For example let us consider the following three rules:

```
plus [**x, var *y] -> "1";
plus [var *x, var *y, **z] -> "2";
plus [*x, var *y, *z, var *t, **u] -> "3";
```

When the tree is of arity one, only the first rule is applicable. When its arity is between 2 and 3, the first and the second are applicable. When the arity is more than 4, all rules are applicable.

Because of sublist variables, it is convenient to allow negative numbers in path when using the access operation $\downarrow$. From a tree t, a move $-n$ denotes the subtree at rank $a(t) + (1 - n)$. With this extension we can express the pattern `plus[**x,var *y]` with the following formula:

$$n(t) = \texttt{plus} \quad \&\& \quad 1 \leq a(t) \quad \&\& \quad n(t \downarrow_{-1.\texttt{s}}) = \texttt{var}$$

Note that not all patterns can be translated into a formula. For example the pattern `plus[**x, var *y, **z]` has no equivalent. This is not a problem because PPML is deterministic by nature and the previous pattern is not. The tree $x + y + z$ could be filtered in three distinct ways.

6.2 Box Builder

When building boxes, we need to handle iteration. This works as follows. In the box builder, the box generated for rules with iteration corresponds to the case where there is no iteration (the sublist is empty). Then an additional component is generated to give access to the elements of the iteration. For example given the skeleton:

```
[<h> *x ( "+" **y )]
```

the box builder for this rule returns:

```
[<h>  ?  ]
```

where the dummy box `?` stands for the variable `*x`. The additional component when asked for an iteration for the given rule returns a list composed of two elements:

```
"+" ?
```

6.3 Path Table

The path table is where the situation gets more complicated. To explain what the problems are, let us consider the following contrived example.

```
plus[*x,**y,*z] -> [<h> *x ("+" **y) "+" *z ];
```

The correspondence for the variable `*x` is no problem. It is as before:

`1.s` $\longrightarrow$ `1.s`

For the variable `**y` in the pattern, we have to encode the fact that if the tree is of arity n_t then this variable covers the sons from rank 2 to rank $n_t - 1$. For this, we introduce the notation $*_i^j$, which represents the sons from rank i to rank $n_t - j$ where n_t is the arity of the list in the tree structure. For the variable `**y` in the box structure, it covers the sons of rank 3, $3+2$, $3+4$, until $n_b - 2$ where n_b is the arity of the horizontal box structure. For this, we introduce another notation $*_i^{j,k,l}$, which represents the sons of rank i, $i+k$, $i+2k$, until $n_b - j$ for a box structure of arity n_b. The extra information l indicates that the sublist variable is the l^{th} positioned element in those generated by *one* iteration. It is used to compute the placement of the sublist variable in the box structure generated by an iteration. (We give another example at the end of this section.) With the two new notations, it is possible to give the association for the variable `**y`:

$*_2^1$`.s` $\longrightarrow$ $*_3^{2,2,2}$`.s`

Note that now in order to perform the translation we need to have the information about the respective arities of the tree structure or of the box structure. An interesting alternative is to know instead the number of iterations. For $*_i^j$, it covers the sons of rank from i until $i+(m-1)$ where m is the number of iterations. For $*_i^{j,k,l}$, it covers the sons of rank i, $i+k$, $\ldots$ $i+(m-1)k$.

For the third variable `*z`, we could use negative path:

`-1.s` $\longrightarrow$ `-1.s`

The only drawback of this solution is that it only works when the arity is known. Knowing the number of iterations is not sufficient to be able to convert the negative rank information into a positive one. As we want the result of our compilation to be usable both with rank and with iteration information, we introduce a redundant notation $-_i^j$ that accomodates both. This represents the son of rank n_t+1-i if n_t is the arity or the son of rank $m+j$ if m is the number of iterations. For paths in the skeleton, we also need the size of the iteration that is the number of sons added at each iteration, so the notation is $-_i^{j,k}$. It represents the son of rank n_b+1-i if n_b is the arity or $j+mk$ if m is the number of iterations. We can now write the correspondence list for the variable `*z`:

$-_1^2$`.s` $\longrightarrow$ $-_1^{3,2}$`.s`

Altogether for the rule

```
plus[*x,**y,*z] -> [<h> *x ("+" **y) "+" *z ];
```

we have the following entries in the path table:

Tree		*Box*	
Path	*Interpretation* (arity n_t, m iterations)	*Path*	*Interpretation* (arity n_b, m iterations)
i	son of rank i	i	son of rank i
$-_i^j$	son of rank $n_t + 1 - i$ or $m + j$	$-_i^{j,k}$	son of rank $n_b + 1 - i$ or $j + mk$
$*_i^j$	sons of rank i, $i+1$, until $n_t - j$ or $i + (m-1)$	$*_i^{j,k,l}$	sons of rank i, $i+k$, until $n_b - j$ or $i + (m-1)k$

Fig. 1. Path interpretations

$$\begin{array}{rcl} \texttt{1.s} & \longrightarrow & \texttt{1.s} \\ *_2^1\texttt{.s} & \longrightarrow & *_3^{2,2,2}\texttt{.s} \\ -_1^2\texttt{.s} & \longrightarrow & -_1^{3,2}\texttt{.s} \end{array}$$

As promised, we now look at another example involving the information provided by l is used. Let us consider a variation of the previous rule

```
plus[*x,**y,*z] -> [<h> *x "+" ( **y "+") *z ];
```

It generates the following entries in the path table:

$$\begin{array}{rcl} \texttt{1.s} & \longrightarrow & \texttt{1.s} \\ *_2^1\texttt{.s} & \longrightarrow & *_3^{2,2,1}\texttt{.s} \\ -_1^2\texttt{.s} & \longrightarrow & -_1^{3,2}\texttt{.s} \end{array}$$

The only difference between this new path table and the previous one is the parameter l of the $*_i^{j,k,l}$ notation for the second entry. Here the sublist variable is the first element of the iteration and not the second one as before. This similarity is not much a surprise since these two rules when applied to the same tree produce *exactly* the same result. So with $i = 3$ and $j = 2$ and $k = 2$ we know that the recursive calls for the sublist variable are at ranks i, $i+k$, $i + 2 * k$, until $n_b - j$, i.e 3, 5, 7, until $n_b - 2$. The information l is only used to know which elements of the box correspond to the same iteration. For example, for the n^{th} iteration, these elements have ranks $i + (n-1)k - l + 1$, $i + (n-1)k - l + 2$, until $i + nk - l$.

Figure 1 summarizes the different notations used in the path table.

6.4 Translation

In the translation we have to handle iterations. Given the path table and the arity of the tree it is easy to determine the number of iterations. Let us take a simple example with the tree `plus[var "x", var "y", var "z"]` and the rule

```
plus[*x,**y] -> [<h> *x ( "+" **y)];
```

The path table associated with this rule is

$$\texttt{1.s} \longrightarrow \texttt{1.s}$$
$$*_2^0\texttt{.s} \longrightarrow *_3^{0,2,2}\texttt{.s}$$

The second entry corresponds to the variable `**y`. The tree has arity 3 and the path of the sublist variable in the tree structure is $*_2^0$.s. It means that the variable stands for the trees from rank 2 until $3 - 0$. It implies that we have two iterations. To get the box with two iterations, we first call the box builder. It returns a box with no iteration

```
[<h> ? ]
```

The path of the sublist variable in the box structure is $*_3^{0,2,2}$.s. It indicates that the first element of the first iteration has the path `2.s`: the rank of the recursive call is 3 and it is the second element of the iteration. The component that gives access to elements of the iteration is then used twice to insert the consecutive elements starting from position `2.s`. We then get the expected result:

```
[<h> ? "+" ? "+" ? ]
```

6.5 Structured Navigation

As noted in Figure 1, paths can only be interpreted if one has information about arities or iterations. Fortunately almost all the time, the iteration information can be obtained from the box-rule trace. Let us take another look at our previous example:

```
plus[*x,**y] -> [<h> *x ( "+" **y)];
var *x -> *x;
```

The box-rule trace corresponding to translating the tree `plus[var "x", var "y", var "z"]` is `(1 (2) (2) (2))`. The fact that there were two iterations is reflected by the number of recursive calls associated with the application of the first rule. Looking at the path table, one recursive call comes from the first entry of the path table, that is, a regular tree variable, the other two recursive calls are generated by the two iterations.

A previous version of the compiler was using this technique to determine iterations and perform path translation. Unfortunately, this technique does not work in all cases. It is not correct when a sublist variable is used only for pattern matching purposes. Let us consider the following contrived example:

```
plus[**x,*y] -> [<h> "..." "+" *y]
```

As there is no recursive call with the variable `**x`, there is no way to get the number of iterations. This makes it impossible to translate paths for the variable `*y` correctly. The solution to this problem has been to annotate explicitly all rules in the box-rule trace with the number of iterations they have produced. This number is set to zero for rules with no iteration. With this convention the new box-rule trace for our example becomes $(1^2\ (2^0)\ (2^0)\ (2^0))$.

6.6 Incremental Update

Producing a correct modification path is the more difficult, and error-prone, part of the whole compiler construction. The problem comes from the fact that insertions and deletions propagate to all elements on their left in the list. Let us consider our favorite example:

```
plus[*x,**y] -> [<h> *x ( "+" **y)];
var *x -> *x;
```

where we have being translating the tree `plus[var "x", var "y", var "z"]`. Inserting a tree in last position gives the following tree path:

```
4.i
```

The corresponding box path is more elaborated since we must introduce the two new elements of the new iteration at the end:

```
6.i
7.i
```

The situation gets even more complicated if we insert an element in first position

```
1.i
```

In that case we not only have to insert a new iteration but also indicate that the first son has been changed:

```
1.c
2.i
3.i
```

A brute-force solution would be to automatically recompute the whole list in cases of insertion or deletion. A more sophisticated one, implemented in the compiler, consists in trying to generate the minimal modification path as given in the previous example. This is done by carefully computing the drift implied by the insertions and deletions of the modification path to the remaining elements of the list.

7 Implementation of the Compiler

The compiler is written in JAVA. The package is named `aioli.ppml` and is 7000 lines long. It uses the ANTLR library [5] for parsing PPML files. It is available with the distribution of AÏOLI at the following address:

```
http://www-sop.inria.fr/lemme/aioli
```

7.1 Overview

The two data structures are manipulated through interfaces. The interface `aioli.vtp.Tree` corresponds to the tree data structure. The interface `figue.box.GlyphInterface` corresponds to the box data structure. For paths, there is a class for each kind of path. The class `aioli.vtp.IPath` corresponds to the single paths. The class `aioli.vtp.MPath` corresponds to the modification paths for the tree data structure. The class `aioli.ppml.MBPath` corresponds to the modification paths for the box data structure. Finally the box-rule trace is manipulated through the interface `aioli.ppml.Skeleton`

A pretty printer is an object of the class `aioli.ppml.PPEngine`. Once a pretty printer is created, it provides:

- the method `build` to create a `Skeleton` from a `Tree`;
- the method `transferSk` to create a `GlyphInterface` from a `Skeleton`;
- the method `vtp2box` to create a `IPath` on the box structure from a `IPath` on the tree structure;
- the method `box2vtp` to create a `IPath` on the tree structure from a `IPath` on the box structure;
- the method `modif` to create a `MBPath` on the box structure from a `MPath` on the tree structure.

In the current version of the compiler we provide one strategy to generate layout. In a first step we construct the complete box-rule trace. This is the `build` method. In a second step we build the box structure using the box-rule trace. This is the the `transferSk` method. Once the layout is built, it is possible to translate between selection paths using the `vtp2box` and `box2vtp` methods and to translate modification paths using the `modif` method.

7.2 Creating a Pretty Printer

To illustrate how the generation works, consider the following file `exp-std.ppml`

```
pretty printer std of exp is
   plus[*x,**y] -> [<h> *x ( "+" **y)];
   var *x -> *x;
end
```

It defines the standard pretty printing for our arithmetic expression language `exp`. To compile the prettyprinter, we just need to call the JAVA with the appropriate parameters

```
% java aioli.ppml.Compile exp-std.ppml
```

This creates two files `BBstd.java` and `PPstd.java`. The first one holds all the basic methods that are only concerned with the tree data structure (the path table, the matcher, and the generation of terminals). The second one holds the actual prettyprinter. These files are given in Appendices A and B. Inside JAVA, a pretty printer can be created by the expresssion:

```
new aioli.ppml.PPEngine(new PPstd());
```

It is also possible to retrieve a pretty printer using its name and the language to which it applies. In our case, calling the static method

```
aioli.ppml.PPManager.getPprinter("exp","std");
```

returns an instance of our pretty printer.

7.3 The BBstd file

Here we give some information on the BBstd file. It defines a class `BBstd`. The purpose of this class is to hold all information relevant to the tree data structure: pattern matching and terminal values. The path table is kept in the variable `Table` as a three dimensional array of integers.

```
static final int Table [][][] ={
...
}
```

The first dimension corresponds to a rule's index. In `BBstd`, the entries relative to the first rule are given by the expression `Table[1]` whose value is:

```
{{2,2},
 {PPconst.List,1,-1,-1,0},{1},{1},
 {PPconst.Recur,1,-1,-1,0},{PPconst.listn,2,0},{PPconst.listn,3,0,2,2}}
```

The first element `{2,2}` contains two numbers. The first one indicates the number of entries in the table. The second indicates which entry corresponds to the sublist variable. If the rule does not have a sublist variable, the second number is 0.

Each entry is encoded by a sequence of three arrays. The first array indicates the type of the variable, it can be either an atomic variable (`Atomic`), a regular variable (`Fixed`), a regular variable under a list node (`List`), or a sublist variable (`Recur`). For example the first array of the second entry is

```
{PPconst.Recur,1,-1,-1,0}
```

It indicates that this line holds the sublist. The other integers in the array are information relative to holophrasting and context management.

The second array for an entry gives the path in the pattern. The path for the second entry is

```
{PPconst.listn,2,0}
```

It corresponds to $*_2^0$.

The last array gives the path in the skeleton. The path for the second entry is

```
{PPconst.listn,3,0,2,2}
```

It corresponds to $*_3^{0,2,2}$. The `terminal` method is used to generate all the terminals in the rule as an extension of PPML, which allows one to call external function to get terminal strings in boxes.

```
public String terminal(Tree tree, int rule, int elt) {
...
}
```

The `match` method is in charge of the pattern matching. To avoid the size limitation on `switch` constructors in JAVA, a submethod is created for each operator and for each context. The task of the main `match` method is simply to find out to which submethod we delegate the pattern matching.

```
public int match(Tree tree, int holo, AStack annot, String ct) {
...
}
```

7.4 The PPstd file

The `PPstd` file first defines a class `TTstd`. The purpose of this class is to hold all the information to build the box structure.

The `buildBox` method creates a box structure from a skeleton.

```
final public GlyphInterface  buildBox(Skeleton sk) {
 ...
}
```

The `iterBuildBox` method is a local method that returns the element located at the given `rank` when iterating the given `rule`.

```
final public GlyphInterface  iterBuildBox(int rule, Skeleton sk, int rank) {
  ...
}
```

The `PPstd` file also defines the class `PPstd`. This class is an extension of the class `TTstd` to handle PPML extensions. In this class, the `match` method first looks if the local matcher finds a rule otherwise the pretty printers in the extension list are tried in order. (Cf. the use of `pl` in Appendix B.)

```
public void match(Tree tree, int holo, AStack annot,
                  String context, RuleIndex rx) {
  ...
}
```

8 Conclusion

We have presented the design and the implementation of a compiler for pretty printing specification. Its main originality lies in seeing that pretty printing as a translation from the tree-like structure of the data to the tree-like structure of the layout. Every time such a translation is done, enough information is kept in order to keep a correspondence between data and what is displayed on the screen. The links between the two structures are computed symbolically and dynamically using paths. The implementation of the compiler is straightforward. The only difficult part as explained is the treatment of list nodes.

The first benefit of this approach is the simplicity of the design. Very few assumptions are made about the data structure and about the layout engine. They must bear a tree-like structure and accommodate the notion of paths. This makes our solution very generic and keeps the data and the box structures as separate as possible. The layout engine and the symbolic engine can then be developed independently from one another.

The second benefit is that the same architecture could run without any problem in a distributed environment. The engine that manipulates the data could be physically distant from the engine that visualizes the data.

References

1. Aïoli. A Toolkit to Build Interactive and Symbolic Applications, Available at `http://www-sop.inria.fr/lemme/aioli/`.
2. Patrick Borras et al. Centaur: the system. *Software Engineering Notes*, 13(5), 1988.
3. Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant: A Tutorial: Version 6.1. Technical Report 204, INRIA, 1997.
4. Hanane Naciri and Laurence Rideau. Figue: Mathematical formula layout with interaction and mathml support. In *the Fifth Asian Symposium on Computer Mathematics ASCM'2001*, September 2001.
5. Terence Parr and Russell Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software Practice and Experience*, 13(7):789–810, 1995.
6. Pcoq. A Graphical User-interface to Coq, Available at `http://www-sop.inria.fr/lemme/pcoq/`.
7. Simon Peyton-Jones. *Implementing Functional Languages: A Tutorial.* Prentice Hall, 1992.

A BBstd

```
class  BBstd {
   static final int Table [][][] ={
      /* Rule 0 */
      {{2,2},
       {PPconst.Atomic,1,0,-1,0},{},{1},
       {PPconst.Recur,1,0,-1,0},{PPconst.listn,1,0},{PPconst.listn,2,0,1,1}},
      /* Rule 1 */
      {{2,2},
```

```
     {PPconst.List,1,-1,-1,0},{1},{1},
     {PPconst.Recur,1,-1,-1,0},{PPconst.listn,2,0},{PPconst.listn,3,0,2,2}},
    /* Rule 2 */
    {{1,0},
     {PPconst.Atomic,1,-1,-2,0},{},{}}
};

public final int [][][] getTable() {
   return Table;
}
public String terminal(Tree tree, int rule, int elt) {
   switch (rule) {
   case 0:
      switch (elt) {
      case 1:
         return lang.ppaux.operatorName(tree);
      default: return tree.atomValue().toString();
      }
   case 2:
      switch (elt) {
      case 1:
         return lang.ppaux.identitypp(tree);
      default: return tree.atomValue().toString();
      }
   default: return tree.atomValue().toString();
   }
}
static String[] _ccontext= {"0"   };
static String[] _acontext={   };
static aioli.ppml.Context _context= new aioli.ppml.Context(_ccontext,_acontext);
public  aioli.ppml.Context getContext() {return _context;}
public int match(Tree tree, int holo, AStack annot, String ct) {
   int res = 0;
   int context = _context.getContext(ct);
   for(int i=0; i< _ccontext.length;i++)
      if (_ccontext[i].equals(ct)) {context=i;break;}
   switch (context) {
   case 0: {
      switch(tree.operator().getCode()) {
      case formalism.op_plus: {
         return ct0hDefopplus(tree,annot);
      }
```

```
         case formalism.op_var: {
            return ct0hDefopvar(tree,annot);
         }
         default:
            return 0;
         }
      }
      default: return 0;
      }
   }
   public int ct0hDefopplus(Tree tree,AStack annot) {
      int atree = tree.length();
      if (atree>=1) {
         return 1;
      }
      return 0;
   }
   public int ct0hDefopvar(Tree tree,AStack annot) {
      return 2;
   }
}
```

B PPstd

```
class  TTstd extends BBstd implements PPtable {
   final public GlyphInterface  buildBox(Skeleton sk) {
      int length = sk.getLength();
      int rule = sk.getRule();
      GlyphInterface fm0= null;
      GlyphInterface fm1= null;
      GlyphInterface fm2= null;
      try {
      switch (rule) {
      case 0 : {
      fm0 = new Vertical(10,10);
      for(int i = 0; i < length;i++)
         for(int j = 0; j <1 ;j++)
            fm0.addChild(sk.iterBuildBox(sk.down(i),j));
         break;
      }
      case 1 : {
      fm0 = new Horizontal(0);
```

```
      fm1 = sk.down(0).buildBox();
      fm0.addChild(fm1);
      for(int i = 1; i < length;i++)
         for(int j = 0; j <2 ;j++)
            fm0.addChild(sk.iterBuildBox(sk.down(i),j));
         break;
      }
      case 2 : {
      fm0 = sk.down(0).buildBox();
         break;
      }
      }
      }
      catch(Exception e) {}
      return fm0;
   }
   final public GlyphInterface  iterBuildBox(int rule, Skeleton sk, int rank) {
      GlyphInterface fm0= null;
      GlyphInterface fm1= null;
      GlyphInterface fm2= null;
      try {
      switch (rule) {
   case 0:
      if (rank == 0) {
      fm0 = sk.buildBox();
         return fm0;
      }
      return fm0;
   case 1:
      if (rank == 0) {
      fm0 = Atom.newAtom("+");
         return fm0;
      }
      if (rank == 1) {
      fm0 = sk.buildBox();
         return fm0;
      }
      return fm0;
      }
      }
      catch(Exception e) {}
      return fm0;
```

```
   }
}
public class  PPstd implements Pprinter {
   public String version() { return "0.98";}
   public String name() { return "std";}
   PPtable _table = new TTstd();
   Pprinter[] pl = {};
   boolean[] plb = {};
   public String[] extensionList() {
      String [] res = new String[pl.length];
      for(int i =0; i< pl.length; i++){
         res[i]=pl[i].name();
      }
      return res;
   }
   public boolean isActive(String ppml) {
      for(int i =0; i< pl.length; i++){
         if (pl[i].name() == ppml) return plb[i];
      }      return false;
   }
   public void activate(String ppml, boolean b) {
      for(int i =0; i< pl.length; i++){
         if (pl[i].name() == ppml) {plb[i]=b;return;}
      }
   }
   public void match(Tree tree, int holo, AStack annot, String context, RuleIndex rx) {
      rx.set(_table,_table.match(tree,holo,annot,context));
      if (rx.getIndex() != 0)
         return;
      for(int i =0; i< pl.length; i++){
         if (!plb[i]) continue;
         pl[i].match(tree,holo,annot,context,rx);
         if (rx.getIndex() !=0)
            return;
      }
      return;
   }
}
```

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803