DESIGNATING SUBTREES IN CENTAUR

D. Clément, SEMA–METRA
L. Hascoët, INRIA Sophia-Antipolis
Aug 10, 1987

Abstract

We present here an alternative method for designating subtrees of a given tree. The existing method uses pointers towards nodes of the tree. Our method uses access paths, and is thus more independent from the implementation than the existing one. We also show examples of the use of access paths, especially when they are more efficient than pointers, or when they can solve new problems.

1. THE "OFFICIAL" METHOD

When manipulating trees in a syntax directed editor, for instance CENTAUR [5], we constantly find the need to designate places in the program, that is, subtrees of the global tree. Following are some examples of situations that illustrate this need.

a) If the user points on some region of the program by clicking the mouse, instead of using the basic navigation routines (Up, Down, Right, Left), the system might want to know how to point to this same region by itself, for instance to help the user to come back to this place. Another interesting situation is when the system performs some operation on the tree, using a system based on inference rules called TYPOL [3][4]. This operation might need a designated subtree as an argument, so we have to provide a way for the system to know about this subtree.

b) If we want to take a photograph of a given program, including the highlighting of some subtree(s), we need to remember in some way the relation between every highlighted node and the root of the tree.

c) For every incremental computation, we need to know all the parts of the tree that were modified, in order to infer which computations have to be done again. Moreover, we would like to know, for each node of the tree, whether any subnode was modified, *ie* if the corresponding subtree was modified.

Of course, CENTAUR already provides a way of designating subtrees, which is the following: Every node of a syntax tree can be located by a pointer. Each time one wants to point to or to remember a subtree, one has only to remember the pointer to this subtree.

This solution is simple and easy to use. Most of the CENTAUR primitives work with the pointers. Highlighting a subtree, for instance, works all right with the pointer to the designated subtree. Remembering such a pointer is easy since it takes very little amount of storage.

However, when we want to do more complex things, we are too strictly limited by this method. In the next section, we shall describe why this standard solution is not strong enough for our needs.

2. WHY POINTERS ARE NOT STRONG ENOUGH

2.1. The main problem

Let us admit that we designate subtrees by pointers. To talk about the whole program, we just use a particular pointer, which goes to the top of the tree. The first problem we meet is that it is not easy to guess the parental relation between two pointers, if any. In particular, if we are given the pointer to the top of the tree, and the pointer to one subtree, we are unable to go from the top to the subtree using only basic moves.

Conversely, when traversing a tree, in preorder for instance, while executing a TYPOL program, it is easy to compute the sequence of movements from the top of the tree to the current node. But if we want to highlight the currently visited node, we first need to translate the sequence of movements to obtain the actual pointer, and only then can we highlight the subtree corresponding to this pointer.

To say it shortly, the main drawback of the pointer method is: we can't simply guess the relation between two different subtrees by just knowing their pointers. The other problems we are going to describe are, in some way, derived from this weakness.

2.2. Consequence for plain designating of subtrees

Some times one wants to talk about, let us say, the second son of a node, whatever modifications were done to this node or its sons. But, when replacing a node by another one, all the pointers towards this node and its sons are changed, since it is a new abstract syntax tree. So the old pointer to the second node is no longer useful to designate it, and it is necessary to "go there" by some atomic moves.

2.3. Consequences for incremental computation

Let us consider again the questions of incremental recomputation. We may remember the upper nodes of all modified subtrees by keeping the list **M** of pointers towards them. But then, we want to know, for any given node **N**, if some node under it was modified or not. The only way for that is to compute the pointers towards all subtrees of **N**, then to check if any of them is member of the list **M**. It is clear that this method is extremely heavy, especially if we need to apply it to every node of the tree.

Another problem is that we can't simplify the list **M** of modified nodes. Suppose some subtree **S** has three subtrees, $\mathbf{S_1}$, $\mathbf{S_2}$ and $\mathbf{S_3}$. If we have already modified the first and third subtrees, then **M** contains both $\mathbf{S_1}$ and $\mathbf{S_3}$. Now if we modify **S** in turn, we just add **S** to **M**. But now, $\mathbf{S_1}$ and $\mathbf{S_3}$ are redundant in **M**. Moreover, they may point to subtrees that don't exist any longer, just because the whole subtree **S** was modified. Therefore, we want to remove them from **M**, to come back to some kind of "normal form". But this is just as tough as the previous problem, since we want to know if some pointer is "ancestor" of another one. So we can't easily find the "normal form" of **M**.

2.4. Consequences for memorizing states of the screen

We have often felt the need to store and retrieve a given state of the screen, that is, the currently shown tree, along with the current position of the highlighted zone. So let us suppose we did that with pointers.

If we just remember the values of the two pointers, we are soon faced with the following problem. If in the following operations on the tree, we modify in some way the structure of the tree —which is more than often the case—, then, the value of the stored tree is changed together, since it is in fact the same tree, the same pointer. That is not so good for a backup!

On the other hand, if we choose first to copy both the top tree and the subtree, we memorize something that is of course independent from any further modification, but we have lost the parental relation between the two trees. Therefore, it is impossible to restore the previous state of the screen.

What we need here is to store, instead of both trees, a copy of the whole tree, plus the relation between the two trees, this relation depending only on the structure of the global tree.

We are now going to describe our solution to these problems.

3. THE NEW METHOD FOR DESIGNATING SUBTREES

3.1. The notion of PATH

We are more interested in considering the relation between two trees than in working with pointers towards them. Relations between trees seem more natural, and more independent from the implementation technique used for trees. They depend only on the structure of the total tree.

These relations should allow us to designate any subtree, or any list of subtrees. We are also interested in having some kind of normal form, to avoid repetitions, and to detect if one subtree is equal to another.

We also want to know, for every node of the tree, if some of its subtrees can be found in the list of designated zones. Moreover, if it is the case, we want to know how to find, how to access to these subtrees.

In fact, all that doesn't leave us much choice.

We propose to designate any subtree, or any collection of subtrees of a given tree, by what we call a PATH. The PATH is the link from the top of the tree to the top of every designated subtree. It can be viewed as the skeleton of the global tree, when we remove from it all the subtrees which are not designated.

3.2. The abstract syntax of a PATH

If the path is **all**, it means that the whole considered tree is designated. If not, then the path is the list of the sons of the considered tree that have some designated sub-zone. In particular, when the path is an empty list, ( ), it means that no subtree of the current tree is designated. For each son in the path, we give recursively the path from this son to its own designated subtrees.

Thus, the abstract syntax we use to write paths is:

sorts

PATH STARTED_PATH ATOMIC_STEP INTEGER

functions

crossroad	:	STARTED_PATH*	$\rightarrow$	PATH
all	:		$\rightarrow$	PATH
start_path	:	ATOMIC_STEP$\times$PATH	$\rightarrow$	STARTED_PATH
son	:	INTEGER	$\rightarrow$	ATOMIC_STEP
left	:		$\rightarrow$	ATOMIC_STEP
right	:		$\rightarrow$	ATOMIC_STEP

Remark:

crossroad is a list operator, so there are two equivalent ways of writing the same *crossroad* tree:
crossroad[PATH$_1$, PATH$_2$] or *crossroad*[PATH$_1$. *crossroad*[PATH$_2$. *crossroad*[ ]]]

3.3. The problem of lists

Now, we have to explain the use of the operators *left* and *right*. CENTAUR provides two ways of manipulating trees, and in particular lists. These two ways are the internal representation, VTP [6], and TYPOL, (or PROLOG). In the case of lists, this leads to some ambiguity. While list operators, seen from VTP, behave as n-arity nodes, with special operators like insert, delete or go to the $\mathbf{n^{th}}$ element, the same lists, seen from TYPOL, are built with a binary operator, just like the LISP cons.

Now we have to choose one of those two notations, and to convert to the other one when necessary. We chose the TYPOL way, just because we intend to use our paths mainly from TYPOL. An other reason is that this is the more powerful way. In VTP, there is no way to talk about the "rest" of a list, while it is possible in TYPOL. On the other hand, it is always possible to go to any element of a list with a correct sequence of **car**'s and **cdr**'s. We shall see later that this was the only solution to compute paths inside TYPOL rules.

Here, the ATOMIC_STEP *left* means the equivalent of the **car** operator, and goes to the first son of the current list. On the other hand, *right*, goes to the **cdr** of the list, *ie* the list without its first element.

3.4. An example

Let us illustrate that on an example. We want to consider the abstract tree whose graphical representation is:

To designate the three subtrees $\mathbf{T_1}$, $\mathbf{T_2}$ and $\mathbf{T_3}$, we use the following PATH, which we represent graphically as:

and whose abstract syntax is:

```
 crossroad
  [start_path(son(2),
              crossroad[start_path(son(2),all())
              ])
   start_path(son(3),
              crossroad[start_path(son(1),
                                   crossroad[start_path(right(),
                                                        crossroad[start_path(left(),all())
                                                                  start_path(right(),
                                                                             crossroad[start_path(left(),all())
                                                                             ])
              ])])])
   ]
```

3.5. Concrete syntaxes of a PATH

Now, those paths are going to be manipulated from different languages. Namely, when we work in CENTAUR, we are allowed to manipulate trees directly in VTP (which is written in LISP), and we can also translate them into TYPOL (PROLOG), to manipulate them with all the power of PROLOG for pattern-matching, unification, and backtracking. In this last case, we may want to manipulate paths written in PROLOG too. For now, these are all the languages we need.

It follows that we need to define a concrete syntax form, or a pretty-printing routine, for each of the manipulation languages. For LISP, we chose to pretty-print as much as possible with the LISP list operator. The pretty-printing rules are:

- The operator *crossroad* is just the lisp list.
- The operator *start_path* is the lisp cons.
- The operator *all*, which is pretty-printed as the atom **'all**.
- The operators *left* and *right* become respectively the atoms **'l** and **'r**.
- The operator *son* is pretty-printed as the integer atom which is the rank of the son. This is more natural, and leads to no ambiguity, since the abstract syntax is very simple. The previous example path becomes now:

'((**2** . ((**2** . **all**)))(**3** . ((**1** . ((**r** . ((**l** . **all**)(**r** . ((**l** . **all**))))))))))

For PROLOG, we chose that the *crossroad* operator should be represented as the PROLOG list. The *son* operator becomes, as usual, the integer atom representing the rank of the son. All the other operators are written in the usual PROLOG prefix notation, with some abbreviations. The previous path can be written in C-PROLOG [1]in the following way:

[sp(2, [sp(2, all)]), sp(3, [sp(1, [sp(r, [sp(l, all), sp(r, [sp(l, all)])])])])]

In the next section, we shall try to relate our notion of paths with other, classical ideas.

3.6. PATHs as finite automata

Let us consider the set S_1 of all sequences of movements (ATOMIC_STEP), that are allowed in the global tree. The set S_2 is the subset of S_1, whose elements lead to designated zones in the tree. Here, "designated" may mean highlighted, or modified. In other words, the designated zones are the subtrees we want to remember with a path P_1.

What we want to emphasize is that P_1 represents a finite automaton that recognizes exactly the subset S_2 of S_1.

In the starting state, the automaton reads the first movement of the given sequence s. Transitions are provided only for the ATOMIC_STEPs which can be found in the first level of the *crossroad*. For the example described above, there are transitions only for $son(2)$ and $son(3)$. Any sequence s starting with something else is simply not recognized.

After a transition, say $son(3)$, the new state is given by the PATH that follows $son(3)$ in P_1. It gives us a new path P_2. In this new state, the provided transitions are to be read in P_2, and in our example, we can see there is only $son(1)$, and so on. When the state reached is described by the PATH $all()$, The state is a termination state. But this state has all possible transitions provided, leading back to this state itself. It means that inside a designated tree, every sub-node of it is designated too. The automaton corresponding to our example path above is:

We find here for the first time the notion of sub-path. The path P_2 is a sub-path of P_1. It is also a sub-part of the original automaton, which is classically called the derivation of the automaton P_1 with regard to the initial sequence *son*(3)

There is a close relation between the division of automata and the notion of sub-path. The result of the derivation of an automaton A_1 by the sequence s is a new automaton A_2. In the language recognized by A_1, some chains may start with the initial sequence s, and finish with a sequence f. The automaton A_2 recognizes the set F of these sequences f. The sub-path P_2 can be interpreted in a parallel way. If the user stops to consider the global tree, by moving to some subtree we wants to focus his attention on, we "derive" the path P_1 by the sequence of moves the user did. Later we will say "divide" instead of "derive", to avoid confusion at least. Now the result of that is P_2. P_2 recognizes the finishing sequences, *ie* is the path that relates the new position "of the user", to the designated zones he still can reach from there.

4. OPERATIONS ON PATHS

We have already found some nice, real-size, applications of our PATHs. These examples have guided us in the choice of the basic operations we should provide the users of PATHs.We shall try to highlight, for each case, the advantages (and disadvantages) of paths *vs* pointers.

4.1. Putting a PATH into normal form

The classical use of a normal form for paths is to check if two paths are equal, *ie* if they designate exactly the same subtree(s) of a given tree. At first sight, it seems easier with pointers, since one just has to check equality of two sets of pointers. But we have already shown the problems found when two pointers designate a node and an ancestor of that node.

We are going to show here the rewriting rules which transform a path to its normal form.

First, here are some rules to define a standard order between the STARTED_PATHS of a *crossroad* node. We chose the natural order: 1st, 2nd and 3rd son, in the case of fixed arity nodes, and *left* first, then *right*, in the case of lists. Of course any mixture of lists and fixed arity notation is illegal, and we will not bother about these forbidden cases where in a *crossroad*, one talks about the $\mathbf{n^{th}}$ son and the left or right son.

$$(\mathbf{R_1})\quad \textit{crossroad}[\textit{start\_path}(\textit{son}(2), P_2), \textit{start\_path}(\textit{son}(1), P_1)\ .\ C] \\ \rightarrow \quad \textit{crossroad}[\textit{start\_path}(\textit{son}(1), P_1), \textit{start\_path}(\textit{son}(2), P_2)\ .\ C]$$

$$(\mathbf{R_2})\quad \textit{crossroad}[\textit{start\_path}(\textit{son}(3), P_3), \textit{start\_path}(\textit{son}(2), P_2)\ .\ C] \\ \rightarrow \quad \textit{crossroad}[\textit{start\_path}(\textit{son}(2), P_2), \textit{start\_path}(\textit{son}(3), P_3)\ .\ C]$$

$$(\mathbf{R_3})\quad \textit{crossroad}[\textit{start\_path}(\textit{son}(3), P_3), \textit{start\_path}(\textit{son}(1), P_1)\ .\ C] \\ \rightarrow \quad \textit{crossroad}[\textit{start\_path}(\textit{son}(1), P_1), \textit{start\_path}(\textit{son}(3), P_3)\ .\ C]$$

$$(\mathbf{R_4})\quad \textit{crossroad}[\textit{start\_path}(\textit{right}(), PR), \textit{start\_path}(\textit{left}(), PL)\ .\ C] \\ \rightarrow \quad \textit{crossroad}[\textit{start\_path}(\textit{left}(), PL), \textit{start\_path}(\textit{right}(), PR)\ .\ C]$$

Now we want to express that a *crossroad* node cannot have two sons starting by the same ATOMIC_STEP. In that case, the two *start_path* must be merged into a single one, starting by the same ATOMIC_STEP.

$$(\mathbf{R_5})\quad \textit{crossroad}[\textit{start\_path}(A\_S, \textit{all}()), \textit{start\_path}(A\_S, P_2)\ .\ C] \\ \rightarrow \quad \textit{crossroad}[\textit{start\_path}(A\_S, \textit{all}())\ .\ C]$$

$$(\mathbf{R_6})\quad \textit{crossroad}[\textit{start\_path}(A\_S, P_1), \textit{start\_path}(A\_S, \textit{all}())\ .\ C] \\ \rightarrow \quad \textit{crossroad}[\textit{start\_path}(A\_S, \textit{all}())\ .\ C]$$

$$(\mathbf{R_7})\quad \textit{crossroad}[\textit{start\_path}(A\_S, P_1), \textit{start\_path}(A\_S, P_2)\ .\ C] \\ \rightarrow \quad \textit{crossroad}[\textit{start\_path}(A\_S, P_3)\ .\ C]$$

Where P_3 is the classical concatenation of the two PATHs, P_1 and P_2, which are both starting with a *crossroad* operator.
The next step is to say that a *start_path* that leads to nowhere is equivalent to no *start_path* at all. That is:

$$(\mathbf{R_8}) \quad \mathit{crossroad}[\mathit{start\_path}(\mathrm{A\_S}, \mathit{crossroad}[\ ]),\ .\ \mathrm{C}] \quad \rightarrow \quad \mathrm{C}$$

We could also write rules like

$$\mathit{crossroad}[\mathit{start\_path}(\mathit{left}(), \mathit{all}()), \mathit{start\_path}(\mathit{right}(), \mathit{all}())\ .\ \mathrm{C}] \quad \rightarrow \quad \mathit{all}()$$

but we think it is a too fast simplification, and besides, it is not exactly the same thing to designate all sons of a node and to designate this node itself.

This rewriting system will help us to check that all our operations give results in normal form, but we will not implement it directly. We just want to keep it as a reference. Therefore, we will not go into any demonstration about it. Anyway, we think this system clearly has the Church-Rosser property.

4.2. Adding a PATH to another

If we define an incremental routine, we have to chose a way to remember exactly which are the nodes of the tree that were modified. We already saw the main drawback of the pointers method, which is the difficulty to compute the normal form.

Let's suppose now that the list of already modified subtrees is stored in the path P_1. Now we modify something more. If the system is consistent, we must suppose that we get another path P_2, which corresponds to the newly modified subtrees. Now we have to compute the new path to store P_3. We shall say that P_3 is the sum of P_1 and P_2.

The method to compute P_3 is very simple. P_1 and P_2 are both paths starting from the same place, *ie* the top of the tree. The first thing to do is only to concatenate the two lists of STARTED_PATHs, with some care for the *all* operator. Thus, the definition of the concatenation is in the following inference rules:

$$\mathit{all}()\ ,\ \mathrm{PATH} \ \rightarrow\ \mathit{all}()$$

$$\mathrm{PATH}\ ,\ \mathit{all}() \ \rightarrow\ \mathit{all}()$$

$$\frac{\mathrm{PATH} \neq \mathit{all}()}{\mathit{crossroad}[\ ]\ ,\ \mathrm{PATH} \ \rightarrow\ \mathrm{PATH}}$$

$$\frac{\mathrm{PATH} \neq \mathit{all}() \qquad \mathrm{C}\ ,\ \mathrm{PATH} \ \rightarrow\ \mathrm{C}_2}{\mathit{crossroad}[\mathrm{SP}\ .\ \mathrm{C}]\ ,\ \mathrm{PATH} \ \rightarrow\ \mathit{crossroad}[\mathrm{SP}\ .\ \mathrm{C}_2]}$$

Now, all that remains is to put P_3 in normal form again. In the way we would implement it, those two steps would be mixed, for more efficiency.

4.3. Checking if some subtree was partially modified

Again, the classical situation in which this question arises is incremental recomputation. Now that we have a stored path value, that indicates the modified subtrees, if any, we want to use this value.

Using the pointers method would oblige us to traverse the whole considered subtree, and for each node encountered, check whether its pointer is or not in the stored list. The complexity of that operation is $m \times n$, where n is the size of the subtree (number of nodes), and m is the number of pointers in the stored list.

With the PATH method, we just have to check if the stored path, P, is or not equal to *crossroad*[ ].

4.4. Checking if some subtree is totally new

It is nearly the same case as above, except that we just need to check if P is or not equal to *all*(). This problem was also an easier one with pointers, since one just needed to check if the pointer towards the considered tree was in the stored list

4.5. Computing a sub-PATH

In the two previous paragraphs, all the checkings can be performed on only one particular node, *ie* the one for which we know the PATH. Now, this is a little bit silly, since the only node for which we know the

PATH is the top of the whole tree. So we need to give means to compute a sub-PATH, *ie* the PATH that relates a given subtree of the global tree to its own modified subtrees.

Comparison with the pointers method is difficult here. With pointers, that problem could be seen as partitioning the set of modified pointers of the whole tree into subsets, these subsets containing, for instance

- In the first subset, all the modified pointers that are under the first son of the tree.
- In the second subset, all the modified pointers that are under the second son of the tree.
- And so on...

To compute a sub-PATH, we need a path P and a subtree. This subtree has to be designated, for instance by another path P_2. Of course P "starts" from the same place as P_2 does. In fact, this latest path is a little bit special, for it points to one and only one subtree. So it may be considered as only a list of ATOMIC_STEPs. The operation we will apply, which we like to call *divide* (since it divides a path into sub-paths), is very simple. Informally, three cases may happen.

- Either P equals $all()$. Then the result of the division is always $all()$.
- Either P, which is now a list of STARTED_PATHs, contains an element that starts by the same ATOMIC_STEP as P_2. Then the computation restarts from the corresponding path and the rest of P_2.
- Else P doesn't contain anything in the required "direction", and the result is the empty path $crossroad[]$.

When at last, the path P_2 is $all()$, this means that the target subtree has been reached, and the division is finished. The corresponding inference rules are given below:

$$all()\,/\,\mathrm{P} \;\rightarrow\; all()$$

$$\frac{\mathrm{P}\,/\,\mathrm{P}_2 \;\rightarrow\; \mathrm{P}_3}{crossroad[start\_path(\mathrm{AS},\mathrm{P})\,.\,\mathrm{C}]\,/\,crossroad[start\_path(\mathrm{AS},\mathrm{P}_2)] \;\rightarrow\; \mathrm{P}_3}$$

$$\frac{\mathrm{AS}' \neq \mathrm{AS} \qquad \mathrm{C}\,/\,crossroad[start\_path(\mathrm{AS},\mathrm{P}_2)] \;\rightarrow\; \mathrm{P}_3}{crossroad[start\_path(\mathrm{AS}',\mathrm{P})\,.\,\mathrm{C}]\,/\,crossroad[start\_path(\mathrm{AS},\mathrm{P}_2)] \;\rightarrow\; \mathrm{P}_3}$$

$$crossroad[\;]\,/\,\mathrm{P} \;\rightarrow\; crossroad[\;]$$

$$\mathrm{P}\,/\,all() \;\rightarrow\; \mathrm{P}$$

The result needs not to be put into normal form again. It is easy to prove that, if the two arguments are paths in normal form, then the result is too.

Now we can improve the comparison between pointers and paths. When given a global tree, of size s, a list of modified pointers, of length m, and also a subtree, located l levels deep in the tree, the size of the subtree is approximately $s \times 2^{-l}$. To determine if the subtree is partially modified has the complexity $m \times s \times 2^{-l}$. With paths, the same operation has the complexity of the division of the two paths, which is proportional to the length of P_2, which is proportional to l. In most cases, the path method wins, except perhaps for very deep subtrees.

4.6. Maintaining the PATH to the currently visited node

When traversing a tree with TYPOL, we may want every step of the inference to know the subtree it is working on. In particular, we often want to be able to highlight the currently visited node.

Let us admit first that, here, pointers are just as good as paths are. We only want to present our solution to this well-known problem.

Of course, the computation starts on the top node. The corresponding path is of course $all()$. Now if we know the path P to the current node, all we need to know is the path to some direct son of the current node. Let's suppose that the ATOMIC_STEP leading to this son is AS. We have to build the new path using the following inference rules. It is obvious they are not efficient, because the paths start from the top of the tree, while the new steps we want to add to paths are closer and closer to the bottom of the tree. It is a classical tradeoff; If we had chosen the reverse order for paths, the reverse problem would have arisen somewhere else.

$$\mathit{all}()\ ,\ \mathrm{AS}\ \rightarrow\ \mathit{crossroad}[\mathit{start\_path}(\mathrm{AS}, \mathit{all}())]$$

$$\frac{\mathrm{P}\ ,\ \mathrm{AS}\ \rightarrow\ \mathrm{P}_2}{\mathit{crossroad}[\mathit{start\_path}(\mathrm{AS}_2, \mathrm{P})]\ ,\ \mathrm{AS}\ \rightarrow\ \mathit{crossroad}[\mathit{start\_path}(\mathrm{AS}_2, \mathrm{P}_2)]}$$

There is a classical method that partially solves the problem of efficiency. The heart of the method is a little hack, which is to replace the *all*() at the end of the path by a free variable. In that case, one just needs to remember the variable and to instantiate it by

$$\mathit{crossroad}[\mathit{start\_path}(\mathrm{AS}, \mathrm{FREE\_PATH})]$$

where FREE_PATH is just another new free variable, for next step. But this method requires to be very careful about unifications, and doesn't apply to all cases.

The conclusion of this section is that for this particular case, we find a weakness of the PATH method. If one is only interested in keeping relations between the top of the tree and only one subtree at a time, for highlighting purposes, one should as well keep the path in the reverse order, from the bottom up. In this very particular case, the reverse of paths seems much more efficient than paths.

4.7. System-related input and output of PATHs

What we call *input* and *output* of paths is:

- For *input*, all the possible ways to compute the path relating a tree and a given list of subtrees. These trees have to be designated before in some way. That way might be anything, for instance the position of the video pointer or the current highlighting pointer. These notions are related to the CENTAUR system, but others systems have other notions close to the CENTAUR ones. The result of our *input* routines is always a path. This path may be represented in LISP or PROLOG concrete syntax, according to the way we want to use it later.
  The input routine itself may be written in LISP or in PROLOG, according to efficiency reasons, for example.
- For *output*, all the possible ways of displaying the value of a path on the screen. If the tree corresponding to the top of the path is displayed on some CENTAUR view, the way to output the path may be to highlight the designated subtrees. Of course, this operation is linked to the actual implementation of the CENTAUR system. Here again, all languages can be chosen to write the path to output, and to write the *output* routine itself.

Another close problem is to translate a LISP path to the equivalent PROLOG path, back and forth. This a technically difficult problem, because we need a communication mechanism between a LISP process and a PROLOG process that are interleaved.

It would be tedious to give the actual implementation of all these routines. On the other hand, the underlying principles are already given, and are rather trivial. Besides, we never wrote all these routines. We just write one of them when we need it. Right now, what exists already is:

- An input routine, which gives the path, from the top of the tree displayed in a given view, to its highlighted zone. It is written in LISP, and gives a path in its LISP form. Its current name is **get_access**.
- A translation routine, from a LISP path to the PROLOG equivalent one, written in LISP, called **prolog_access**.
- A PROLOG predicate, called **prolog_access**, which is an input routine, and which unifies its only argument to the PROLOG form of the path read from the current view.
- An output routine that, given a path and a tree in its VTP form, gives back the list of the designated subtrees, in its VTP form too. If one wants to see the tree and its highlighted zone on a given view, one just has to "send" the global tree and the result of the routine to that view, using ordinary VTP primitives. Everything is written in LISP, and this function is named **send_access**.

We understand one being worried about the apparent complexity of all these combinations of input, output and implementation languages, plus the associated translations: This problem disturbs us too!

4.8. And so on...

We don't intend to be exhaustive about all the uses of paths. Therefore, we will stop here our list of examples. We just think that we have presented and justified the main operations on paths, and shown that they are more powerful than pointers.

The rewriting rules and inference rules we have shown here are not the actual implementations, which may be done in PROLOG, or in LISP, with perhaps some little hacks around. Rules are all right for explaining.

For further examples, we are going to show in the next chapter, some more complex uses of paths, involved in more "real programming" situations

5. MORE COMPLEX USES OF PATHS

The following examples are real uses of paths, and we are not going to use them to introduce or illustrate operations on paths. Instead, we want to show them as examples of realizations, along with our choices of actual implementation. The three following uses are thus considered as real-size applications.

5.1. Storage of the screen state

We want to write a LISP toplevel function that saves the state of some "view" from the CENTAUR screen. What we want is to provide a checkpoint mechanism. When one saves the contents of a view, nothing happens in the view. One may then do every modification one wants on the displayed program. If one wants to undo the modifications, and come back to the exact stored state, we shall provide another LISP function, which restores on the screen the previously stored state of the view. The first function will be called **photograph**, and should put the saved state into the global variable called **picture**. The LISP code uses the input routine **get_access**.

```
(defvar picture ())

(de photograph (view_name)
    (let ((view ({name}:view view_name)))
        (setq picture (cons ({tree}:copy (view-tree-root view))
                                                ;This is the copy of the entire tree
                            (get_access view))
                                                ;This is the path to the designated subtree(s)
        )
    )
)
```

The reverse function, which will be called **restore_picture**, will use the output routine **send_access**.

```
(de restore_picture (view_name)
    (let ((view ({name}:view view_name)))
        (view-tree-root view (car picture)
                                    ;This is to write the entire tree in the view
        (view-tree-k     view (car (send_access (cdr picture)
                                                (car picture)))
        )                           ;This is to set the highlighted zone
                                    ;to the first designated subtree
        ({interface}:affiche view)
                                    ;This is to refresh the view
    )
)
```

These functions may be called from the user level, or embedded in a bigger program. This example is still short, and could not be solved with the "pointer" method.

We chose to apply functions written in LISP for many reasons:

- In CENTAUR, the user level is a LISP level.
- LISP is lower level, therefore faster than PROLOG, and we don't need to traverse the paths by ourselves, which could be more cleanly done in PROLOG.
- The routines **get_access** and **send_access**, since they use the VTP routines, and since VTP is written in LISP, are easier to write in LISP.

5.2. Incremental recomputation of synthesized attributes

Let's suppose we have a TYPOL predicate that computes a synthesized value on the given tree. This means that every computed value depends only on the subtree it was computed on, and that the result does not change when the corresponding subtree is not modified. This is an ideal case to recompute this value incrementally.

For instance, this predicate could compute the value of an expression that contains no variables. In fact, it may also evaluate expressions with variables, provided the environment containing the values of these variables is given, and does not change between a computation an the incremental reevaluation. This predicate could also be a type-checking, provided the type-checking algorithm was modified to remove all inherited attributes, which is possible in some cases.

Let's also suppose that, for every node in the tree, the more up-to-date computed value of the corresponding subtree is stored is some way, and can be read or re-written. We suppose this, just not to mix many different problems, but the fact is that it may be easier, and faster, not to store these values everywhere, but only on some nodes. That does not change the solution of the problem we focus on here. Let's write our predicate, for instance:

$$\mathrm{FIXED\_ENVIR} \vdash \mathrm{TREE} : \mathrm{VALUES}$$

The typical TYPOL rules that define our predicate are:

(1)
$$\frac{\rho \vdash \mathrm{T}_1 : v_1 \qquad \rho \vdash \mathrm{T}_2 : v_2 \qquad \rho \vdash \mathrm{T}_3 : v_3 \qquad v \;=\; v_1 \star v_2 \star v_3}{\rho \vdash \mathrm{node}(\mathrm{T}_1, \mathrm{T}_2, \mathrm{T}_3) : v}$$

(2)
$$\frac{\rho \vdash \mathrm{T_l} : v_l \qquad \rho \vdash \mathrm{T_r} : v_r \qquad v \;=\; v_l \bullet v_r}{\rho \vdash \mathrm{list\_node}[\mathrm{T_l} \,.\, \mathrm{T_r}] : v}$$

(3)
$$\rho \vdash \mathrm{atomic\_value}\ x : x$$

Now we want to modify these rules to obtain an incremental system. The principle we will use is based on the question: "Was the currently visited node modified partially since the latest evaluation, or is it totally intact since then?"

So the first thing to write is a system that collects all the successive modifications, and gathers them in a path. All the modifications are seen more easily from the LISP side of CENTAUR, so this part will be written in LISP. First, we define a global variable, **modified**, that will contain the path to the modified subtrees. Since we want to keep track of modifications since the last evaluation, we must check that **modified** is reset to () (LISP concrete syntax for *crossroad*[ ]), after every evaluation, incremental or not.

Now, every operation that modifies a subtree must be trapped in some way. This depends on the implementation of CENTAUR, and will not be discussed here. We just suppose that some call is made after every modification, which is:

(**update_modified path**)

where **path** designates the newly modified subtree, to be added to **modified**. The principle of adding two paths has been explained earlier, and programmed in LISP in a more efficient way. The definition of **update_modified** is then:

```
(defvar modified ())

(de update_modified (added_path)
    (setq modified
          (add_paths modified added_path)
    )
)
```

Now, at some moment, the user asks for a recomputation, by calling the execution of the TYPOL predicate. So let's modify this TYPOL program. Since we are going to work in TYPOL now, we need to use the PROLOG version of the path. The new first rule of the TYPOL program will be:

$$\frac{\text{get\_modified}(\pi) \qquad \pi \ \rho \vdash \text{TREE} : v \qquad \text{reset\_modified}}{\rho \vdash \text{TREE} : v}$$

where "get_modified" first takes the value of the LISP variable **modified**, then translates it to a PROLOG term, and unifies it with the free variable π. Now all the rules of the original TYPOL program must be modified, since we want every predicate to be called with the value of the path, from the current node, towards its modified subtrees. We will thus need the predicate "divide", implemented in TYPOL exactly like it was defined earlier, but for PROLOG syntax. For instance the predicate

$$\text{P} \ / \ \mathit{crossroad}[\mathit{start\_path}(\mathit{son}(2), \mathit{all}())] \ \rightarrow \ \text{P}_2$$

will now be written:

$$\text{P} \ / \ [\mathbf{sp}(\mathbf{2}, \mathbf{all})] \ \rightarrow \ \text{P}_2$$

and will be abbreviated into:

$$\text{P} \xrightarrow{2} \text{P}_2$$

The new TYPOL rules are then:

(1′)

$$\frac{\pi \xrightarrow{1} \pi_1 \quad \pi \xrightarrow{2} \pi_2 \quad \pi \xrightarrow{3} \pi_3 \quad \pi_1 \ \rho \vdash \text{T}_1 : v_1 \quad \pi_2 \ \rho \vdash \text{T}_2 : v_2 \quad \pi_3 \ \rho \vdash \text{T}_3 : v_3 \quad v \ = \ v_1 \star v_2 \star v_3 \quad \text{store}(v)}{\pi \ \rho \vdash \text{node}(\text{T}_1, \text{T}_2, \text{T}_3) : v}$$

(2′)

$$\frac{\pi \xrightarrow{\text{l}} \pi_l \qquad \pi \xrightarrow{\text{r}} \pi_r \qquad \pi_l \ \rho \vdash \text{T}_\text{l} : v_l \qquad \pi_r \ \rho \vdash \text{T}_\text{r} : v_r \qquad v \ = \ v_l \bullet v_r \quad \text{store}(v)}{\pi \ \rho \vdash \text{list\_node}[\text{T}_\text{l} \ . \ \text{T}_\text{r}] : v}$$

(3′)

$$\frac{\text{store}(x)}{\pi \ \rho \vdash \text{atomic\_value} \ \ x : x}$$

The rule (2') is the typical example where we need to consider lists in our way, and not as n-arity nodes. If we consider the chain of inference rules that leads to, say, the third son of a list, TYPOL will force us to apply three inference rules. Now if lists were n-arity nodes, the ATOMIC_STEP to apply would be *son*(3). But the computing of this *son*(3) is not local to one of the inference rules, and cannot be done before execution time. In other words, we need to use some kind of a global variable, or an index, to decide, in the rule (2), what is the actual rank of the **"car"** son T_l, and the actual rank at which the **"cdr"**, T_r, will start. In our solution, the sequence of $left()$ may be considered as this index. So it is compulsory for us to consider *son*(3) as two $left()$, and one $right()$.

Now our modified program is still equivalent to the first one, and is not incremental yet. To make it incremental at last, we just need to add one rule that traps the case where the current subtree was not modified at all:

$$(0) \qquad \frac{\text{read\_store}(v)}{[\,]\ \rho \vdash \text{TREE} : v}$$

To finish this section, let's just remark that, in the case where values are not stored everywhere, this rule becomes

$$(0') \qquad \frac{\text{has\_store(TREE)} \qquad \text{read\_store}(v)}{[\,]\ \rho \vdash \text{TREE} : v}$$

5.3. Forward-backward source-code mapping

This is our biggest example so far. It is also the one which requires the most sophisticated tools on paths. On the other hand, it can be viewed just as a special case of the previous section. What we have is a translator, written in TYPOL. Our example is the translator from ASPLE towards SML (Stack Machine Language), which is just similar to any translator or small compiler, provided it doesn't use a global, inherited, environment (*cf* previous section), and provided there is no optimization performed. Therefore, the translator can be considered as the evaluator of a synthesized attribute, which is the translated version.

Now what we want is a little different from an incremental translator that we could obtain like in the previous section. What we want is to define the mapping from some subtree of the source file, towards the corresponding part of the code file, forward and backwards. This is why we can't handle simplifications, because it destroys the links between parts of the source, and parts of the code.

This mapping can be seen as a major improvement to an incremental translator: while the incremental evaluator constantly stores and reads the values of every subtree, which can waste memory space and cpu time, we may here take profit of the fact that the result of the computation is already stored in the translated or compiled file. So this source-code mapping gives us the equivalent to the "read_store" predicate of the previous section.

Now, how shall we manage to obtain this mapping? The input must be a subtree, in the source or in the code file. So must be the output. In our context, subtrees are nothing but paths. So we need to transform the translator to obtain a program that translates paths in the source into paths in the code, and conversely.

5.3.1. *The given translator*

We are going to show here some TYPOL rules, which define the translator we are starting from. This translator goes from a type-checked ASPLE program to SML code, which looks a little like assembly language.

We must emphasize that, when the result of a translation is the concatenation of two list of machine instructions, the translator doesn't really concatenate these two lists, but instead puts them side to side in a higher level list. This can be considered a lazy method, but it has no bad consequences. Only, when traversing a list, one must be careful to traverse also the elements of this list, when they are lists themselves. Moreover, when we pretty-print the code, these extra levels of list do not appear, and everything looks as if the lists were concatenated.

On the other hand, this "hidden" structure of the code will be of great help for us, because the concatenation would have lost some structure, and the mapping would have been harder.

We show here only some representative rules. The ASPLE operators are written in normal style, while the SML ones are in *italic* style:

$$(1) \qquad \vdash \text{stms}[\ ] \rightarrow \mathit{coms}[\ ]$$

(2) $$\frac{\vdash \mathrm{STM} \to c_1 \qquad \vdash \mathrm{STMS} \to c_2}{\vdash \mathrm{STM; STMS} \to coms[c_1, c_2]}$$

(3) $$\frac{\vdash \mathrm{EXP} \to c \qquad \vdash \mathrm{STMS}_1 \to c_1 \qquad \vdash \mathrm{STMS}_2 \to c_2}{\vdash \text{if EXP then } \mathrm{STMS}_1 \text{ else } \mathrm{STMS}_2 \to coms[block(coms[c, fjp1, c_1, ujp2, lbl1, c_2, lbl2])]}$$

(4) $$\vdash \mathrm{deref(id\ X)} \to coms[ldo(sml\_id\ \mathrm{X})]$$

(5) $$\frac{\vdash \mathrm{EXP} \to c}{\vdash \mathrm{deref(EXP)} \to coms[c, ind()]}$$

(6) $$\vdash \mathrm{id\ X} \to coms[lao(sml\_id\ \mathrm{X})]$$

The rules (1) and (2) are about the translation of the "list of statements" construct. They are rather obvious, and we may underline the fact that the result is not a flattened list of instructions, because we do not concatenate the two lists of instructions c_1 and c_2. As a result, it is easier to distinguish in the translated code, what comes from the first statement (c_1), and what comes from the rest (c_2). Another remark is that the tree structure of the translation is different from the tree structure of the source. That is what interests us in mapping paths from the source to the code.

The rule (3) is just another example of classical compilation of the "if then else" construct. Again, the resulting list of statements is not flat. We want to emphasize here that, since this translator is more of a compiler, information that was stored in the structure of the tree (the "if then else"), is transformed into instructions (the *jumps* and *labels*) that are at the same level than any other instruction. That can be a difference between plain translators and compilers.

Conversely, some parts of the compiled code, which can now be designated in the code because they are subtrees by themselves, do not correspond to something precise in the source. On the source side, the "if" itself could not be designated, because it was not a subtree by itself, but the top node of a bigger subtree. In that case, the problem is: What can we do if the user asks for the code→source mapping of the $fjp1$ instruction. The most precise thing to show in that case is the whole "if then else" subtree.

The rules (4), (5) and (6) are here to show an code optimization that doesn't upset us. The special case of deref(id X) gives a more efficient code than the rules (5) and (6). This optimization is not a problem because it is done locally, during the normal traversal of the tree, and doesn't need an environment or methods of "peep hole".

Except for these simple cases, optimizations give us many problems. In the case where the translation needs an environment, this environment given by another part of the source (the declarations for instance), we need a little extra care to define the source-code mapping. For instance, if the source file doesn't change, we need to memorize this environment, and give it as a parameter to the mapping predicate.

One interesting problem doesn't show up in the ASPLE→SML program. Suppose a translator goes from a language where declarations may be grouped if of the same type, towards another language, where declarations are just one list of binary mappings identifier→type. In that case, we might find the following rules:

(7) $$\frac{\vdash \mathrm{TYPE} \to t \qquad \vdash \mathrm{IDENTS},\ t \to c}{\vdash \text{var IDENTS : TYPE} \to c}$$

(8) $$\frac{\vdash \text{IDENT} \rightarrow c_1 \qquad \vdash \text{IDENTS},\ t \rightarrow c_2}{\vdash \text{IDENT}, \text{IDENTS},\ t \rightarrow lmaps[mapsto(c_1, t).c_2]}$$

The TYPE subtree maps to many subtrees in the translated part. That is why paths are particularly relevant, since they can designate many subtrees at a time. On the other hand, if the user points on only one t subtree in the translated part, one has to tell him that this t is closely linked to the others, and that it cannot be modified alone. So one has to be careful to write the source↔code mapping in a clever and useful way.

5.3.2. *The principles of our mapping*

We want to define a predicate that means that two paths, one in the source, one in the code, correspond to each other. The PROLOG philosophy drives us to consider the same predicate, with differently instantiated variables, for different purposes. For instance, if we want to check that two given paths correspond, the two PROLOG variables standing for these paths will be instantiated from the start. If we want to compute the code mapping corresponding to a given source path, we instantiate only the variable standing for the source path, and the predicate solving should instantiate the other variable with the code path. Also, if we instantiate the code path, we should obtain as a result the source path.

When one looks at TYPOL rules in a very theoretic way, this gives no problem, since the order in which predicates are solved is not specified. Unsolved predicates are just constraints that give the result. But this doesn't run correctly, because TYPOL is compiled into C-PROLOG [1], and the order in which predicates are solved is fixed arbitrarily at compile time. So the translator, which could be so nice and simple "theoretically", has to be written in a more heavy way, to control the order of resolution by hand.

Some hope may be in using other, more sophisticated controls, like those of PROLOG-II [2], or MU-PROLOG [7]. In these languages, one resolution may be frozen, until some of its parameters are instantiated. But we haven't tested this deeply enough yet, and we fear some new problems may arise.

So one parameter of our mapping will be the direction of the mapping. Another important parameter is the source program itself. The reason is that it doesn't make any sense, to ask if two paths correspond, if the program itself is not given. So the source is needed. On the other hand, it is redundant to give the code; since we are working with a translator, and we have the source, we are at least able to guess the code!

So the typical predicate we shall write is:

$$d \vdash \text{TREE} : \pi_s \Leftrightarrow \pi_c$$

which reads: "In the case where the direction of the mapping is d, the path π_s in the source tree TREE corresponds to the path π_c in the code (or conversely, according to d)"

At once, we may define the most straightforward cases, where one of the paths is known and either empty or **all**. In that case, the value of the other path is obviously the same. Then we may write:

(S_1) $$\frac{\text{is\_empty\_path}(\pi_s) \qquad \text{set\_empty\_path}(\pi_c)}{\rightarrow \vdash \text{TREE} : \pi_s \Leftrightarrow \pi_c}$$

(S_2) $$\frac{\text{is\_empty\_path}(\pi_c) \qquad \text{set\_empty\_path}(\pi_s)}{\leftarrow \vdash \text{TREE} : \pi_s \Leftrightarrow \pi_c}$$

(S_3) $$\frac{\text{is\_all\_path}(\pi_s) \qquad \text{set\_all\_path}(\pi_c)}{\rightarrow \vdash \text{TREE} : \pi_s \Leftrightarrow \pi_c}$$

(S_4) $$\frac{\text{is\_all\_path}(\pi_c) \qquad \text{set\_all\_path}(\pi_s)}{\leftarrow \vdash \text{TREE} : \pi_s \Leftrightarrow \pi_c}$$

or even, if we don't fear the order of resolution (which is the case here):

$$(\mathrm{S}_5) \qquad \frac{\text{empty\_path}(\pi_s) \qquad \text{empty\_path}(\pi_c)}{d \vdash \text{TREE} : \pi_s \Leftrightarrow \pi_c}$$

$$(\mathrm{S}_6) \qquad \frac{\text{all\_path}(\pi_s) \qquad \text{all\_path}(\pi_c)}{d \vdash \text{TREE} : \pi_s \Leftrightarrow \pi_c}$$

where (S_5) is short for (S_1) and (S_2), and (S_6) for (S_3) and (S_4). Now, for each rule of the translator, we are going to see what we can deduce for the source↔code mapping. Let us consider the rule (3):

$$(3) \qquad \frac{\vdash \text{EXP} \to c \qquad \vdash \text{STMS}_1 \to c_1 \qquad \vdash \text{STMS}_2 \to c_2}{\vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 \to \mathit{coms}[\mathit{block}(\mathit{coms}[c, \mathit{fjp}1, c_1, \mathit{ujp}2, \mathit{lbl}1, c_2, \mathit{lbl}2])]}$$

If we know the path in the source, we can divide it by $\mathit{son}(1)$, $\mathit{son}(2)$ and $\mathit{son}(3)$, to obtain the sub-paths in EXP, STMS_1 and STMS_2. Now, let's suppose that we know how to map these three sub-paths in their three sub-trees, and it gives us three paths in the code, π_0, π_1 and π_2. Now that we know the paths in the subtrees c, c_1 and c_2, it is easy to compute the resulting path in:

$$\mathit{coms}[\mathit{block}(\mathit{coms}[c, \mathit{fjp}1, c_1, \mathit{ujp}2, \mathit{lbl}1, c_2, \mathit{lbl}2])]$$

Of course, from code path to source path, things are to be done in the reverse order. Since we must control the evaluation by hand, we cannot just write down the constraints on paths, but we have to order them differently according to the direction. This is tedious, and will be isolated in the two predicates PATH_BEFORE and PATH_AFTER. The rule (3) gives us the new rule:

$$(3') \qquad \frac{\begin{array}{c}\text{PATH\_BEFORE}{<}d, \pi^s \Leftrightarrow \pi^c, [\xrightarrow{1}; \xrightarrow{2}; \xrightarrow{3}] \Leftrightarrow [\xrightarrow{\text{l1l}}; \xrightarrow{\text{l1rrl}}; \xrightarrow{\text{l1rrrrrl}}], [\pi^s_0, \pi^s_1, \pi^s_2] \Leftrightarrow [\pi^c_0, \pi^c_1, \pi^c_2], \mathit{status}{>} \\ d \vdash \text{EXP} : \pi^s_0 \Leftrightarrow \pi^c_0 \qquad d \vdash \text{STMS}_1 : \pi^s_1 \Leftrightarrow \pi^c_1 \qquad d \vdash \text{STMS}_2 : \pi^s_2 \Leftrightarrow \pi^c_2 \\ \text{PATH\_AFTER}{<}\mathit{status}, d, \pi^s \Leftrightarrow \pi^c, [\xrightarrow{1}; \xrightarrow{2}; \xrightarrow{3}] \Leftrightarrow [\xrightarrow{\text{l1l}}; \xrightarrow{\text{l1rrl}}; \xrightarrow{\text{l1rrrrrl}}], [\pi^s_0, \pi^s_1, \pi^s_2] \Leftrightarrow [\pi^c_0, \pi^c_1, \pi^c_2]{>}\end{array}}{d \vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 : \pi^s \Leftrightarrow \pi^c}$$

Of course, every other rule from the translator is going to be transformed exactly in the same manner. We give to the calls to PATH_BEFORE and PATH_AFTER all the information they might need, and in fact more that that in many cases.

For instance, if we translate paths in the source→code direction, then PATH_BEFORE is going to perform the first part of what we wrote previously, *ie* divide the given path π^s by the three sequences of ATOMIC_STEPs : 1, 2 and 3. That will be all. We can see that all the other arguments were not used, but we couldn't guess it before execution time. Conversely, PATH_AFTER only has to deduce from the three paths π^c_0, π^c_1 and π^c_2 and the three sequences of steps l1l, l1rrl and l1rrrrrl, the value of π^c.

But if we translate paths in the reverse order, PATH_BEFORE is going to do some other work, which uses the other arguments, and PATH_AFTER will also change its behavior.

The use of the variable *status* is different. It is here to solve the problem we described previously: What happens when the user wants the code→source mapping of $\mathit{fjp}1$? This problem must be detected during the PATH_BEFORE operation. When the process of dividing the input path by all the sequences of steps *loses* something, *ie* when some designated subtree doesn't appear any longer in the computed sub-paths, then it means that the result must be the corresponding global subtree. In that case, PATH_BEFORE gives back sub-paths that are all equal to **all**, and PATH_AFTER gives always **all** as its result. the *status* is here for PATH_AFTER to know what PATH_BEFORE decided.

The sequences of steps that appear in the rule (3') are computed easily from rule (3). For instance the sequence l1rrl is the sequence of moves that goes from the top of the generated code:

$$\mathit{coms}[\mathit{block}(\mathit{coms}[c, \mathit{fjp}1, c_1, \mathit{ujp}2, \mathit{lbl}1, c_2, \mathit{lbl}2])]$$

to the subtree c_1 corresponding to STMS_1, and it is also the sequence by which one must divide the path π^c into the generated code, to obtain the sub-path π^c_1 into the generated sub-code c_1.

We are ready now to show the definitions of our two fundamental predicates PATH_BEFORE and PATH_AFTER:

set PATH_BEFORE is

$$(\mathrm{PB}_1) \qquad \frac{\mathrm{divide}{<}\pi^s, \mathrm{STEPS}^s, \mathrm{PATHS}^s, \mathit{status}{>}}{{<}\rightarrow, \pi^s \Leftrightarrow \pi^c, \mathrm{STEPS}^s \Leftrightarrow \mathrm{STEPS}^c, \mathrm{PATHS}^s \Leftrightarrow \mathrm{PATHS}^c, \mathit{status}{>}}$$

$$(\mathrm{PB}_2) \qquad \frac{\mathrm{divide}{<}\pi^c, \mathrm{STEPS}^c, \mathrm{PATHS}^c, \mathit{status}{>}}{{<}\leftarrow, \pi^s \Leftrightarrow \pi^c, \mathrm{STEPS}^s \Leftrightarrow \mathrm{STEPS}^c, \mathrm{PATHS}^s \Leftrightarrow \mathrm{PATHS}^c, \mathit{status}{>}}$$

end PATH_BEFORE;

set PATH_AFTER is

$$(\mathrm{PA}_1) \qquad \frac{\mathrm{multiply}{<}\mathrm{PATHS}^c, \mathrm{STEPS}^c, \pi^c, \mathit{status}{>}}{{<}\mathit{status}, \rightarrow, \pi^s \Leftrightarrow \pi^c, \mathrm{STEPS}^s \Leftrightarrow \mathrm{STEPS}^c, \mathrm{PATHS}^s \Leftrightarrow \mathrm{PATHS}^c{>}}$$

$$(\mathrm{PA}_2) \qquad \frac{\mathrm{multiply}{<}\mathrm{PATHS}^s, \mathrm{STEPS}^s, \pi^s, \mathit{status}{>}}{{<}\mathit{status}, \leftarrow, \pi^s \Leftrightarrow \pi^c, \mathrm{STEPS}^s \Leftrightarrow \mathrm{STEPS}^c, \mathrm{PATHS}^s \Leftrightarrow \mathrm{PATHS}^c{>}}$$

end PATH_AFTER;

where the predicates "divide" and "multiply" have rather intuitive meanings, and will be precisely defined in the next section.

This is exactly what we required for the source→code and the code→source directions. Please notice that we did not include yet the definitions of theses two predicates for the two other important cases:

- When the two paths are given, and that we just want to check if they do correspond. In this case, we will write $d = \otimes$, and we should add the corresponding rules to PATH_BEFORE and PATH_AFTER. PATH_BEFORE should do all the work, and PATH_AFTER has nothing to do. So we add:
  to PATH_BEFORE:

$$(\mathrm{PB}_3) \qquad \frac{\mathrm{divide}{<}\pi^s, \mathrm{STEPS}^s, \mathrm{PATHS}^s, \mathrm{ok}{>} \qquad \mathrm{divide}{<}\pi^c, \mathrm{STEPS}^c, \mathrm{PATHS}^c, \mathrm{ok}{>}}{{<}\otimes, \pi^s \Leftrightarrow \pi^c, \mathrm{STEPS}^s \Leftrightarrow \mathrm{STEPS}^c, \mathrm{PATHS}^s \Leftrightarrow \mathrm{PATHS}^c, \mathrm{ok}{>}}$$

  and to PATH_AFTER:

$$(\mathrm{PA}_3) \qquad {<}\mathit{status}, \otimes, \pi^s \Leftrightarrow \pi^c, \mathrm{STEPS}^s \Leftrightarrow \mathrm{STEPS}^c, \mathrm{PATHS}^s \Leftrightarrow \mathrm{PATHS}^c{>}$$

- When no path is given, it means that the user wants to generate (all the) couples of corresponding paths. In this case, we will write $d = \leftrightarrow$, and we will add the rules:
  to PATH_BEFORE:

$$(\mathrm{PB}_4) \qquad {<}\leftrightarrow, \pi^s \Leftrightarrow \pi^c, \mathrm{STEPS}^s \Leftrightarrow \mathrm{STEPS}^c, \mathrm{PATHS}^s \Leftrightarrow \mathrm{PATHS}^c, \mathrm{ok}{>}$$

  to PATH_AFTER:

$$(\mathrm{PA}_4) \qquad \frac{\mathrm{multiply}{<}\mathrm{PATHS}^s, \mathrm{STEPS}^s, \pi^s, \mathrm{ok}{>} \qquad \mathrm{multiply}{<}\mathrm{PATHS}^c, \mathrm{STEPS}^c, \pi^c, \mathrm{ok}{>}}{{<}\mathit{status}, \leftrightarrow, \pi^s \Leftrightarrow \pi^c, \mathrm{STEPS}^s \Leftrightarrow \mathrm{STEPS}^c, \mathrm{PATHS}^s \Leftrightarrow \mathrm{PATHS}^c{>}}$$

We would like to insist once more: all that is heavy, and we should try to find new ways of controlling PROLOG execution, in order to simplify these two predicates. When we find a way of writing just a set of constraints which get solved as soon as possible, then the program will be much clearer.

In some cases, where the rule to transform is more complex, like the rules (7) and (8) of the previous section, the predicates PATH_BEFORE and PATH_AFTER are still controlling the execution too weakly. We have then to put the "divide" and "multiply", according to the chosen value of the direction d. This means that there will be two rules (7') and (8'), in the case where $d = \rightarrow$, and two other, (7") and (8") in the case where $d = \leftarrow$. It is not hard to write these rules. The method is to wait for dividing or multiplying paths until all the input arguments are known. That is easy when the direction d is known.

Now we can build the final transformed TYPOL program, which will be made with the rules (S_5) and (S_6), then all the original rules modified like (3'), then the definitions of PATH_BEFORE and PATH_AFTER. To finish, we have to put two top rules, to interface with the user. These rules will look like:

$$(T_1) \qquad \frac{\text{gettree}<\textit{asple}, \textit{root}, \text{TREE}> \quad \text{get\_access}<\pi^s> \quad \rightarrow\vdash \text{TREE} : \pi^s \Leftrightarrow \pi^c \quad \text{show\_code}<\pi^c>}{<\rightarrow>}$$

$$(T_2) \qquad \frac{\text{gettree}<\textit{asple}, \textit{root}, \text{TREE}> \quad \text{get\_access}<\pi^c> \quad \leftarrow\vdash \text{TREE} : \pi^s \Leftrightarrow \pi^c \quad \text{show\_source}<\pi^s>}{<\leftarrow>}$$

The actual rules may be slightly different because of the centaur implementation, but we will not care here. We are now going to discuss the routines used to manipulate paths, which are:

- Input-Output routines, namely “get_access”, “show_source” and “show_code”.
- Test routines, all_path and empty_path.
- Destructor and Constructor routines, which are “divide” and “multiply”. This “divide” is more complex than the one we described in previous chapters.

5.3.3. *The higher level manipulations on* PATHS *we need*

The first thing to do is to choose in what language we are going to define these “predicates”. Two main choices appear.

- Since we have started from a TYPOL program, and we produce another TYPOL program, we may choose to do everything in PROLOG. In that case, only the input-output routines have to be defined in LISP, since they are system-dependent, and the CENTAUR system kernel is written in LISP. The transformed program runs in PROLOG of course, and we program all test and destruction-construction routines in PROLOG too. In this case, the best choice is to manipulate paths in their PROLOG syntax.
- The other choice is to leave the paths in their LISP form. In that case, the input-output routines are nearly the same, except that they use paths in their LISP syntax. That might be a little easier since these routines are in LISP. Now, the test and destruction-construction predicates, which are still called from the PROLOG side, could translate their inputs, then compute, and at last translate back to LISP. This is awfully slow. The best method is to have these predicates call directly LISP routines. From the PROLOG side, all that can be found in the paths is a pointer to a LISP term, and we know how to call a LISP routine from PROLOG, which is especially easy when the arguments are already known from the LISP side. What happens in that case is that the actual paths never cross the LISP-PROLOG border, and they are never translated. This should save time. All the complex computations are done in LISP, which is ordinary considered faster than PROLOG.

So we tried both methods, and we were a little surprised that the all-PROLOG method is not much slower than the all-LISP one. This is because PROLOG is sometimes very good, for traversal of structures for instance, and because the PROLOG calls to LISP are sometimes slow. Nevertheless, the all-LISP solution pleases us more, because we think the interfacing can be improved, and that we still believe LISP is more efficient for very complex manipulations. Another reason is that the real structure of paths is hidden from the user.

For the remaining of this paper, we will fix our choice on the all-LISP solution. Input-output routines are not very interesting to describe here, although it was not easy to write them, because they are related to the CENTAUR implementation, and one has to know a lot about VTP to write them correctly. We will focus on test, destruction and construction routines. What interests us here is the central algorithm, and the communication between PROLOG and LISP, again, will not be described.

- Test routines are easy to write. For instance, “all_path(π)” means that π unifies with the pointer to the lisp path **'all**. Thus, the definitions are:

$$\frac{\text{getsymb}<\text{"}\textit{all}\text{"}, \pi>}{\text{all\_path}(\pi)}$$

$$\frac{\text{getsymb}<\text{"( )"}, \pi>}{\text{empty\_path}(\pi)}$$

In fact, we have hacked a little further, as we shall mention later.

- The destruction routine is "divide". It is customary to call it a destruction routine because it splits a path into sub-paths, and it is the only one which is allowed to look into the structure of a path. For the other routines of the TYPOL program, a path is just an object whose structure is hidden. What we want is a function that divides a path by every element of a list of sequences of steps. It must give back a list of paths. Therefore this divide is a "map" of the previous divide, on a list of sequences of steps. This divide must also detect the case where the division looses something, and it must generate the correct *status*. Here follows our implementation. The top function is called **split_status**.

```
(de split_status (access lstart)
    (if (let ((access_bis (copylist access)))
              (lost access_bis lstart)            ;What part of the path is lost in division
       )
       (cons (all_all lstart) 'lost)              ;Then, all result paths are 'all
       (cons (split access lstart) 'ok)           ;Else, divide normally
    )
)

(de split (access lstart)
    (let ((index 0))
         (while lstart
                (vset globalvector index (split1 access (nextl lstart)))
                                                  ;This to divide by the first steps sequence
                (setq index (add index 1))
         )
         index                                    ;The result is in the global vector globalvector
    )
)

(de split1 (access start)
    (cond ((null start)        access)
          ((eq access 'all)    'all)
          (t                   (let ((good_way (assq (car start) access)))
                                    (when good_way (split1 (cdr good_way) (cdr start)))))
    )       ;This is the classical definition of the old "divide"
)

(de lost (access lstart)
               ;The function lost removes in access all that may be designated by lstart
    (ifn lstart (clean access)
               (lost (lost1 access (car lstart)) (cdr lstart)))
)               ;While lost1 removes only what is designated by one steps sequence

(de lost1 (access start)
    (cond ((null access)      ())
          ((null start)       ())
          ((eq access 'all)   'all)
          (t                  (let ((good_way (assq (car start) access)))
                                   (when good_way
                                         (rplacd good_way
                                                 (lost1 (cdr good_way) (cdr start))))
                                   access))       ;Similar to split1
    )
)
```

```
(de all_all (lstart)
    (let ((index 0))
        (while lstart
               (vset globalvector index 'all)
               (setq index (add index 1))
               (nextl lstart)
        ) index
    )
)

(de clean (access)              ;This is to put in normal form
    (cond ((null access)         ())
          ((eq access 'all)      'all)
          (t                     (let ((first_access (clean (cdar access))))
                                     (ifn first_access (clean (cdr access))
                                                    (rplacd (car access) first_access)
                                                    (rplacd access (clean (cdr access))))))
    )
)
```

- The construction routine is "multiply". It is also clear why we call it a "construction" function. The algorithm is to concatenate each sequence of atomic steps to the corresponding path, which gives a list of paths, then to add all these paths together, while being careful to keep this final path in normal form. The top function is called **merge**.

```
(de merge (vector lstart)
    (let ((index 0)
          (res ())
         )
         (while lstart
                (setq res
                      (merge_two res
                      ;This is to add two paths
                                 (attach (nextl lstart) (vref vector index)))
                                 ;And attach is to concatenate a sequence of steps in front of a path
                )
                (setq index (add index 1))
         )
         res
    )
)

(de merge_two (access1 access2)
    (cond ((null access1)      access2)
          ((null access2)      access1)
          ((eq access1 'all)   'all)
          ((eq access2 'all)   'all)
          (t                   (mapc (lambda (way) (setq access1 (merge_list access1 way)))
                                     access2)
                               access1)
                               ;For each STARTED_PATH in access2, add it to access1
                               ;in a way that keeps access1 in normal form
    )
)        ;cf the algorithm to add two paths, given in section 4.2.

(de merge_list (access way)
    (let ((same_way (assq (car way) access)))
         (ifn same_way
              (cons way access)
                   ;If no STARTED_PATH in access starts like way, then just add way
              (rplacd same_way (merge_two (cdr same_way) (cdr way)))
                   ;Else, add the two corresponding sub-paths
              access
         )
    )
)        ;cf the algorithm to add two paths, given in section 4.2.

(de attach (start access)
    (cond ((null access) ())
          ((null start)  access)
          (t             (list (cons (car start)
                                     (attach (cdr start) access))))
    )
)
```

5.3.4. *Final hacks and performances*

To increase the speed of the resulting program, we finally "hack" it in two ways. These two "dirty tricks" are:

- The predicates all_path and empty_path just disappear. The reason is that, inside a given CENTAUR session, the LISP pointers towards the atoms **()** and **'all** keep the same value. So we added a prolog rule on top of our program, whose function is

  1) First to discover the two values of these pointers

  2) Then to assert the new rule that replaces (S_5), which is

  $$d \vdash \text{TREE} : \pi_0 \leftrightarrow \pi_0$$

  where π_0 is the pointer to ()

  3) Then to do the same for **'all**

  4) Finally to retract itself from the PROLOG list of rules.

  This manipulation speeds the program up, because it reduces the number of resolution calls in prolog. Before, every call of the standard predicate could unify to (S_5) and (S_6), and then failed very often, and then only found the correct rule. Now, these calls are unified only when it is going to work, and one step of resolution is spared.

- The sequences of atomic steps, like l1rrl, are of absolutely no use on the PROLOG side, since the only thing we do with them is to pass them as arguments to the lisp functions **split_status** and **merge**. This is a waste of time, because it means we have to translate the PROLOG term l1rrl to its LISP form each time we use it. Why not, instead, keep it all the time in LISP syntax, and keep it on the PROLOG side only by an identifier. For instance, each time the sequence of steps "2" appears, which is often, we replace it by an identifier, say "id2", which is the name of a LISP variable, whose value has been set to **(2)**, once and for all. This saves a considerable amount of time, which was wasted in the PROLOG→LISP translations.

We finally present the performances of our source↔code mapping. Our reference is the time to translate an ASPLE program (A small program of about 60 lines) to its SML code, using the TYPOL translator. We measured times on a Sun3/75, with a 68020 processor.

Time of translation $\simeq$ 0.14

In fact, we should compare with a source↔code mapping, written without paths. This program would be some kind of a translator that checks, for each traversed node, whether the current pointer is or not equal to the searched one. Thus, we think such a program would be slower than a plain translator. Anyway, let's suppose the reference time is 0.14. The time taken by our mapping is nearly the same in both directions. It is nearly proportional to the depth of the designated tree in the source tree. The maximum depth of our total tree is about 30.

Depth	**Average Time**
1..5	**0.02** ±0.01
10..15	**0.07** ±0.02
15..20	**0.10** ±0.03
25..30	**0.15** ±0.03

We also have found good results for the behavior in limit cases. For instance, if we point on some region like the $fjp1$, which corresponds only to the global if-then-else subtree, then all this subtree is shown. Also, when a subtree corresponds to many, all the corresponding subtrees are shown in a row.

6. CONCLUSION

What we have presented here is just another technique to designate and manipulate subtrees. We needed it in the CENTAUR context, because the previous method was a little too weak and system dependent, but we think this path technique can be adapted to any system that manipulates trees. All the routines to use paths haven't been written yet, but we think they should be written only as the need for them arises.

REFERENCES

[1] **Bowen, D., Byrd, L., Pereira, L., Warren, D.**, *"C-Prolog user's manual, version 1.5"*, Edited by F.Pereira, SRI International, Menlo Park, California, (February 1984).

[2] **Caneghem, M. Van,** *"Prolog II, Manuel d'utilisation"*, Technical report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, Marseille, (1982).

[3] **Despeyroux, Th.,** *"Executable specification of static semantics"*, Semantics of data types, Lecture Notes in Computer Science, Vol. 173, (June 1984).

[4] **Despeyroux, Th.,** *"Spécifications sémantiques dans le système MENTOR"*, Thèse, Université Paris XI, (1983).

[5] **Kahn, G. et al.,** *"CENTAUR. The system"*, INRIA Report N^o 777 (December 1987).

[6] **Lang, B.,** *"The Virtual Tree Processor: Reference Manual"*, ESPRIT project 348, 3rd review report, (September 1986).

[7] **Naish, L.,** *"MU-Prolog 3.2 reference manual"*, Technical report 85-11, University of Melbourne, (1985).