Formalising the Knowledge of a Prolog Techniques Editor

Wamberto Weber Vasconcelos*
Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
e-mail: wamb@aisb.ed.ac.uk

May 5, 1994

Abstract

A Prolog Techniques Editor (PTE) [7, 9] is a knowledge-based software development tool which provides its users with simple Prolog programs depicting common flows of control and ways to enhance them into more elaborate and useful code. However, the expert Prolog programmers who provide these components are offered no methodology or notation to help them formalise, test, compare and organise their programming knowledge in order to be supplied to the PTE. In this paper we propose the use of a repertoire of commands [4] and their combination in ordered sequences as a way to formalise the PTE's programming practices.

1 Introduction

Tools have been proposed [1] to help programmers in the task of writing/debugging/analysing Prolog programs. Knowledge-based software development tools offer the advice of an expert programmer to their users, giving them the opportunity to devise their own programs in a quicker and more reliable way.

One of these knowledge-based software tools is the *Prolog Techniques Editor* (PTE) [7, 9] which incorporates an adaptation of the *stepwise refinement method* to the logic programming paradigm. The PTE works by guiding its users through a series of interactions (windows and menus): basic Prolog programs (named *skeletons*) depicting most commonly found flows of control are offered to the user who chooses one to start with; then a succession of syntactic transformations (named *techniques*) adding parameters, goals or clauses is carried out upon the chosen skeleton, eventually yielding a more complex and useful program. The outcome of the session is then tested on sample Prolog queries: if the devised program does not meet the needs of the user, he/she can backtrack to any of the decision points and change options previously made.

One important issue arises regarding the logic programming knowledge (skeletons and techniques) of a PTE: no methodology, formalism or representation schema has been proposed so

*On leave from State University of Ceará, Ceará, Brazil; Sponsored by Brazilian National Research Council (CNPq), under grant no. 201340/91-7

far. Such facilities would enable the expert Prolog programmer devising the PTE to formalise, test, compare and organise the practices they use when writing their own programs.

Presently skeletons and techniques are empirically determined by the expert's ingenuity and experience, which sometimes might be misleading and wrong. As a result, one may end up with repeated, wrong (*e.g.*: skeletons which loops or techniques which inadvertedly change the order of arguments in a term) or far too specific components of the knowledge. Also, due to the lack of a proper notation or representation, the expert is left with the option of encoding the skeletons and techniques directly into the PTE using some symbolic programming language. Although the resulting PTE might exhibit an interesting behaviour in helping less experienced programmers, there are no clues of how generic the solution is, whether it can be shifted to other domains, how it relates to the way humans carry out the same task or if it can be further extended.

A way to uniformly represent skeletons and techniques is proposed in this paper: they will be encoded as sequences of commands chosen from a small and powerful repertoire. It is hoped that this suggestion will help overcome the problems mentioned above — the logic programming knowledge of the PTE will not be simply encoded as part of the system without clear-cut boundaries between the skeletons and techniques themselves and idiosyncratic features used to implement them. Instead, the set of skeletons and techniques will be represented *explictly* in a (logic programming) *knowledge base*.

Next section describes in more detail a PTE, with straightforward ways to represent skeletons and techniques and discusses some problems these representations face. Section 3 describes an alternative approach to the representation of Prolog programming knowledge – the use of *Abstract Program Templates* to represent skeletons and techniques. Section 4 drives some conclusions and shows directions of research activities.

2 Prolog Techniques Editors

Current trends in the development of logic programs, particularly of Prolog programs, consist of supplying the programmer with a range of tools (*e.g.*, debuggers, tutoring systems, tracers,

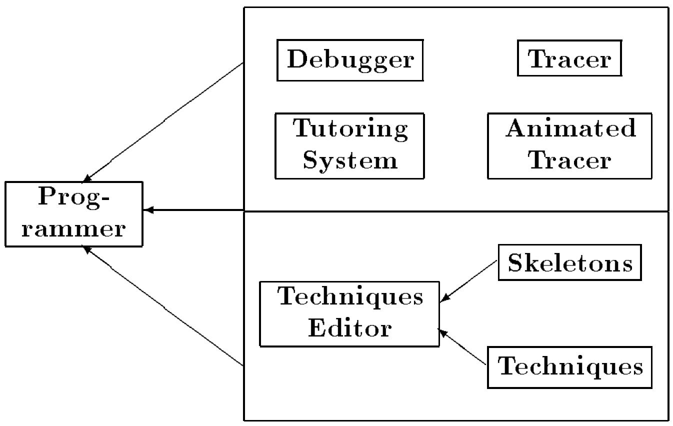

Figure 1: Prolog Program Development Environment

animated tracing tools, etc. [1]) with which the programmer can detect and fix bugs, get advice/help when in trouble, test and compare different versions of programs and follow the execution of the devised code. A pictorial representation of such an environment may be given by Figure 1.

One of these proposed tools is the *Techniques Editor* ([7, 9], also shown in Figure 1) which help programmers devise their own code. Its idea is to offer the programmer a set of syntactic entities, called *skeletons*, which describe the flow of control of the most commonly found Prolog programs, and a set of syntactic transformations (selection mappings), called *techniques*, to be applied upon the skeletons. The techniques describe the programming practices (for example, adding parameters, binding them properly, adding goals or clauses) carried out by experienced programmers upon very simple programs processing data structures (for example, traversing or manipulating lists) in a straightforward way.

$$\underbrace{s \xrightarrow{t_1} e_1 \xrightarrow{t_2} e_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n}}_{\text{intermediate programs}} e_n \equiv p$$

Figure 2: Stepwise enhancement — starting with Skeleton s, and through the successive application of Techniques t_i yielding enhancements e_i, a final enhancement e_n (the program which was aimed at) is obtained

The *modus operandi* of this tool is a stepwise enhancement procedure in which programmers isolate, first of all, the flow of control needed to solve the problem. Then, they choose amongst the skeletons that one which best suits their needs.

After having chosen one skeleton, the programmer customises it to obtain the final program. This customisation process is achieved via the application of the techniques upon the skeleton yielding one *enhancement* (*i.e.*, the original skeleton enhanced with extra arguments, goals and/or clauses). The enhancements can be re-used as skeletons allowing one to repeat the process of application of techniques until the final program is obtained. This idea may be depicted graphically as in Figure 2.

2.1 Skeletons

Skeletons are basic syntactic constructs depicting the flow of control of a Prolog program.

Example A: Skeleton for searching a list:

```
search_n([X|Xs]):- a1(X), search_n(Xs).
search_n([X|Xs]):- a2(X), search_n(Xs).
        ⋮
search_n([X|Xs]):- an-1(X), search_n(Xs).
search_n([X|Xs]):- an(X).
```

Example B: Skeleton for traversing a list:

```
traverse_n([X|Xs]):- b1(X), traverse_n(Xs).
traverse_n([X|Xs]):- b2(X), traverse_n(Xs).
        ⋮
traverse_n([X|Xs]):- bn(X), traverse_n(Xs).
traverse_n([]).
```

Figure 3: Examples of Skeletons

They are programs themselves processing data structures in a very simple way. Figure 3 shows two examples of skeletons taken from [7]: $\mathtt{a}_i$ and $\mathtt{b}_i, 1 \leq i \leq n$ are case analysis. Although the sample skeletons below strongly resemble each other, one shall notice how their base cases differ. Example **A** depicts skeleton `search_n` in which the flow of control traverses the input list until a certain condition $\mathtt{a}_n$ describing what is actually being searched for is met. Example **B** shows a skeleton whose flow of control goes through the input list until it becomes empty: the empty list as the base case of the recursion ensures that the entire list will always be processed.

2.2 Techniques

Techniques may be seen as selection maps from a skeleton to one of its extensions [7]. They act upon skeletons, adding arguments to goals or heads, goals to clauses or clauses to the skeleton. Skeletons can be thought of as the mechanism which controls the program in terms of the order the data structure is processed (flow of control) and the techniques determine what is to be done with the data while the control flow is followed.

$$
\mathit{Collect}\left(\begin{array}{l}
\mathtt{traverse\_n([X|Xs]):-\ b_1(X),\ traverse\_n(Xs).}\\
\mathtt{traverse\_n([X|Xs]):-\ b_2(X),\ traverse\_n(Xs).}\\
\qquad\vdots\\
\mathtt{traverse\_n([X|Xs]):-\ b_n(X),\ traverse\_n(Xs).}\\
\mathtt{traverse\_n([]).}
\end{array}\right) =
$$

$$
\begin{array}{l}
\mathtt{traverse\_n\_coll([X|Xs],[X|Ys_1],Ys_2,\ldots,Ys_n):-\ b_1(X),traverse\_n\_coll(Xs,Ys_1,Ys_2,\ldots,Ys_n).}\\
\mathtt{traverse\_n\_coll([X|Xs],Ys_1,[X|Ys_2],\ldots,Ys_n):-\ b_2(X),traverse\_n\_coll(Xs,Ys_1,Ys_2,\ldots,Ys_n).}\\
\qquad\vdots\\
\mathtt{traverse\_n\_coll([X|Xs],Ys_1,Ys_2,\ldots,[X|Ys_n]):-\ b_n(X),traverse\_n\_coll(Xs,Ys_1,Ys_2,\ldots,Ys_n).}\\
\mathtt{traverse\_n\_coll(}\underbrace{[],[],\ldots,[]}_{n+1}\mathtt{).}
\end{array}
$$

Figure 4: Technique *Collect* applied to skeleton `traverse_n`

Examples of techniques [7] are:

1. *Collect*: this technique is used with skeletons whose clauses are mutually exclusive (at least one of the input arguments will be used for determining which clause to take) and there is a special data item to be collected. It adds an extra argument for each clause (case) in the skeleton. As the program traverses the data, the argument associated with the clause which is fired collects the current object in question into a list. One example of the application of this technique is that of Figure 4;

2. *Count*: it adds one to the value returned from the recursive call. In order to be applied, there ought to be a numeric argument being returned from a recursive call;

3. *Sum*: this technique is similar to the *count* technique, however instead of always adding one to the argument being returned, it adds the value of some argument.

Skeletons and techniques can be seen as part of the knowledge an experienced logic programmer employs in writing programs. The programmer, following her/his experience, devises simple programs conveying commonly occurring patterns of control flows, the skeletons. (S)he also describes, in a purely syntactic way, the practices — the techniques — (s)he does to obtain useful programs from the skeletons. A very important ability experienced programmers also have is knowing which skeleton to use with which technique(s) in order to get interesting programs — current proposals for techniques editors do not contemplate this level of expertise.

Skeletons and techniques comprise *sets* supplied by an expert programmer to be used by the techniques editor. These sets are best seen as separated entities from the editor itself, as pictorially suggested by Figure 1, in a well-delineated distinction between the components of the editor itself (interface, explanation facilities, testing and backtracking routines, etc.) and the knowledge it uses.

If such a distinction is made, one may see the techniques editor as a simple generic system — it guides the user through a series of windows and menus showing the various options of what to use next, what has been done so far and allowing the user to make experiments with the supplied programming knowledge, backtracking to previous versions of the program being devised if an unfortunate choice is made. In this modular approach, the same techniques editor could be offered to two-week Prolog students learning the basics of list processing or professionals developing large symbolic applications — it is enough to provide distinct skeletons and techniques sets to suit each user.

2.3 Representation of Skeletons and Techniques

Skeletons may be easily represented in many ways. Figure 5 shows a simple way of encoding skeleton `traverse_n` using Prolog code and a list data structure. The representation of Figure 5

```
skeleton(traverse_n,Pred,[],[Base]):- Base =..  [Pred,[]].

skeleton(traverse_n,Pred,[Case|Cases],[(Hd:- Tst,Rec)|Clauses]):-
                                        Hd =..  [Pred,[X|Xs]],
                                        Tst =..  [Case,X],
                                        Rec =..  [Pred,Xs],
                                        skeleton(traverse_n,Pred,Cases,Clauses).
```

Figure 5: A Prolog Representation of Skeleton `traverse_n`

is such that if one types

```
?- skeleton(traverse_n,flag_colours,[yellow,red],Program).
```

Prolog supplies the following instantiation to variable `Program`:

```
Program = [(flag_colours([X|Xs]):- yellow(X), flag_colours(Xs)),
           (flag_colours([X|Xs]):- red(X), flag_colours(Xs)),
            flag_colours([])]
```

One may easily devise interesting alternatives to encoding skeletons using ordinary data structures. In [9] skeletons are economically represented as DCG's. These approaches share the common drawback of imposing too heavy a burden on the techniques: these will become extremely complex because they will have to go through programming "tricks" (such as the predicate `=..` used above to circumvent Prolog's inabilility to handle second-order variables) in order to get to the skeleton itself. Also, in order to describe the repetition of a component of the skeleton an intensional device (such as a recursive call in the example above) must be used, adding extra difficulty to the techniques which can only refer generically to the repeated clauses.

Techniques, on the other hand, do not have the syntactic nature of skeletons. They are syntactic operations carried out over skeletons: one way of representing this kind of knowledge is by means of *rewriting rules* depicting patterns and how to rewrite them. If the symbol $\vec{G}$, with or without subscripts, is used as a short-hand for a (possibly empty) sequence of goals, the symbol $\vec{A}$, with or without subscripts, as a short-hand for a (possibly empty) sequence of arguments (variables, functors or constants), the small greek letter μ as an arbitrary predicate symbol (*i.e.*, a second-order variable), and the capital greek letters Ψ and Θ as arbitrary Prolog variables, one can, for instance, represent the technique *Count* which adds one to a value returned from the recursive call (as mentioned previously) as the pair of rules depicted in Figure 6: first rule shows how the base case clause pattern on the left-hand side of the $\Rightarrow$

$$\boxed{\mu(\vec{A}_1[]\vec{A}_2)} \quad \Rightarrow \quad \boxed{\mu(\vec{A}_1[]\vec{A}_2\texttt{,0})} \tag{1}$$

$$\boxed{\mu(\vec{A}_1[\Psi|\Theta]\vec{A}_2)\texttt{:-}\,\vec{G}\mu(\vec{A}_3\Theta\vec{A}_4)} \quad \Rightarrow \quad \boxed{\mu(\vec{A}_1[\Psi|\Theta]\vec{A}_2\texttt{,C)}\texttt{:-}\,\vec{G}\mu(\vec{A}_3\Theta\vec{A}_4\texttt{,C1),C is C1+1}} \tag{2}$$

Figure 6: Representation of Technique *Count* using Production Rules

has to be rewritten — an extra parameter 0 is added as a last argument; second rule shows the recursive clause being rewritten with an added `C` parameter in the head, a `C1` parameter in the recursive call and an added goal `C is C1+1` at the end. If technique *Count* is applied to the `flag_colours` version of `traverse_n`, one would obtain the following code:

```
flag_colours([X|Xs],C):- yellow(X), flag_colours(Xs,C1), C is C1 + 1.
flag_colours([X|Xs],C):- red(X), flag_colours(Xs,C1), C is C1 + 1.
flag_colours([],0).
```

which traverses a list checking the "yellow-ness" and "red-ness" of each element *and* counting them simultaneously.

Techniques were presented through an informal description in English (with the outcome of an application of the technique to a particular skeleton) in a previous section and in [7, 10]. This is far from what is needed to embed them in a computer program. The rewriting rules representation above seems to be successful to formalise techniques of which syntactic manipulations are "localised", that is, the rewriting of a clause does not need information of other constituents of the skeleton. However, some techniques, like the *Collect* technique shown before (information regarding the total number of recursive cases is required to alter the skeleton), make use of information of the *whole* program or skeleton or of information of clauses other than the one presently being rewritten. In such cases, the rewriting rules approach falls short in providing a convenient notation. Enhancing the expressiveness of the rules to refer to other parts of the skeleton would make the rules very complex, if feasible at all.

Any attempt to represent techniques must take into account the representation of skeletons and vice-versa: skeletons are conceived to be manipulated by techniques and techniques are conceived to manipulate skeletons — one plays an essential part in the definition of the other. A useful notation would be a high-level one which abstracted away irrelevant details (names of predicates, functors or variables, etc.) and offered facilities of easily addressing important components of the skeleton (head of clause, all the base-case clauses, recursive goal, etc.). Such formalism should be economic and unambiguous and should be aimed at helping expert Prolog programmers devise their sets of skeletons and techniques.

3 Abstract Syntactic Manipulation of Programs

The proposed approach advocates the use of *Abstract Program Templates* as a notation with which experts may write skeletons and techniques in a more formal and reliable way. The structure of the abstract template allows a more generic, economic and convenient specification of parts of programs and provides a means to store skeletons in the knowledge base. This form of representation allows the organisation of the knowledge base (sets of Skeletons and Techniques) in a hierarchy of specialisation of elements (from more general to more specific versions of skeletons and techniques) with customisation operations as annotations on the links. Techniques, according to this proposal, consist of an abstract program template describing the generic structure of the skeleton they act upon (which works as a requirement — only those skeletons that fit into the description may be used as input for the technique) and a finite sequence of commands (syntactic manipulations altering the constituents of the template) chosen from a small repertoire.

Following this view, the set of Prolog skeletons (see Figure 1) consists of a hierarchy of abstract program templates. Let T be a template and $\mathcal{T}_i, \mathcal{T}_j, \mathcal{T}_k$ and $\mathcal{T}_l$ be sequences of customisation operations. A pictorial example suggesting how this hierachisation takes place is given by Figure 7. Each layer of the hierarchy may be seen as a class of more specialised templates derived from an element of upper layers through a constrained form of second-order substitutions.

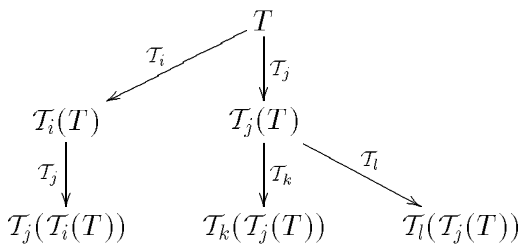

Figure 7: Organisation of a Prolog Knowledge Base (Skeletons) through Program Templates T and Customisations $\mathcal{T}_i, \mathcal{T}_j, \mathcal{T}_k, \mathcal{T}_l$

3.1 Abstract Program Templates

Abstract program templates are *Attribute-Value Matrices* (AVM) [6] which have Prolog programs parts as their constituents. Through this representation, programs are abstracted onto its main constituent parts (*e.g.*, head, body and recursive call) which are conveniently named and labeled. This offers a convenient, efficient and economical way of addressing each one of the important components of the skeleton when applying techniques to it.

According to the AVM in Figure 8, a program consists of n base-case clauses and m recursive clauses (left-hand upper corner part of figure). The order of the clauses may change and hence an annotation is provided with this purpose. Each one of the base-case clauses $b_i, 1 \leq i \leq n$ is made up of a *head* and a *body* consituents, the first of which describes the name of the predicate (*pred*) and its arguments (*args*). Constituent *args* is composed by a vector of arguments $\vec{A}_1$, the base-case value $\flat$ for this clause and another vector of arguments $\vec{A}_2$. Constituent *body* is the vector of goals $\vec{G}$.

$$T = \begin{bmatrix} base\_cases = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \\ rec\_cases = \begin{bmatrix} r_1 \\ \vdots \\ r_m \end{bmatrix} \end{bmatrix} \qquad b_i = \begin{bmatrix} head = \begin{bmatrix} pred = \mu \\ args = \begin{bmatrix} \vec{A}_1 = \_ \\ case = \flat \\ \vec{A}_2 = \_ \end{bmatrix} \end{bmatrix} \\ body = \begin{bmatrix} \vec{G} = \_ \end{bmatrix} \end{bmatrix}$$

$$r_j = \begin{bmatrix} head = \begin{bmatrix} pred = \mu \\ args = \begin{bmatrix} \vec{A}_1 = \_ \\ str = \begin{bmatrix} functor = \sharp \\ non\_rec\_args = \Psi \\ rec\_args = \Theta \end{bmatrix} \\ \vec{A}_2 = \_ \end{bmatrix} \end{bmatrix} \\ body = \begin{bmatrix} \vec{G}_1 = \_ \\ rec\_call = \begin{bmatrix} pred = \mu \\ args = \begin{bmatrix} \vec{A}_3 = \_ \\ rec\_args = \Theta \\ \vec{A}_4 = \_ \end{bmatrix} \end{bmatrix} \\ \vec{G}_2 = \_ \end{bmatrix} \end{bmatrix}$$

Figure 8: Attribute-Value Matrix for Abstract Program Template T, with generic base case $b_i, 1 \leq i \leq n$, and generic recursive case $r_j, 1 \leq j \leq m$.

Figure 8 gives a generic definition of a program which deconstructs an abstract recursive data structure $\sharp(\Psi, \Theta)$. The μ symbol is a second-order variable which stands for a generic name of a predicate — throughout the template, this symbol should have a uniform substitution. The $\sharp$ symbol stands for an arbitrary data structure functor, Ψ stands for an arbitrary Prolog variable which is the non-recursive part of the data structure and Θ stands for another arbitrary Prolog variable which is the recursive part of the data structure (capital greek letters will be used as generic Prolog variables, *i.e.*: they will be interpreted over the set of Prolog variables and non-capital greek letters will be used as generic names for predicates (second-order variables), *i.e.*: they will be interpreted over the set of Prolog predicate names). The symbol $\flat$ stands for any base-case value of a data structure (*e.g.* the empty list `[]` of list structures). The $\_$ symbol has the same meaning (anonymous variable) as in Prolog.

3.2 The Repertoire of Commands

Each technique $\mathcal{T}$ makes use of an abstract program template T describing the generic form of those (and only those) programs (skeletons and enhancements) which can be manipulated by the technique. A sequence C of commands (chosen by the expert programmer from the list below) depicts in a generic yet precise way the syntactic transformations the technique carries out.

This repertoire of syntactic operations is based on the commands of Bundy's *Recursive Techniques Editor for Prolog* [4, 5] — they enforce the preservation of syntactic and semantic correctness by making sensible adjustments in other parts of the template beyond that which is explicitly being altered by the user. These commands depicts basic operations carried out on templates, conveying little or no programming expertise, but ensuring the correctness of the newly obtained extension — it is their combination into useful and coherent sequences capturing common Prolog programming practices that gives rise to the techniques. The list of commands is:

- `rem_dependency(Clauses,Component)` — this command removes component `Component` from all the `Clauses` (`Clauses` may assume the values `base` for the *base_cases* clauses or `recs` for the *rec_cases* clauses). The entry `Component` may only be components of vectors of goals (*e.g.*: $\vec{G}, \vec{G}_1$);

- `rem_parameter(Parameter)` — this command removes all occurrences of `Parameter` in template T.

- `add_parameter(Clauses,Component,ListPar)` — it adds to `Component` of every element of `Clauses` (`Clauses` may assume the same values as in command `rem_dependency`) the list of parameters `ListPar`;

- `move_component(Clauses,Component,NewPos)` — moves `component` of every element of `Clause` (same values as in commands above) to new position `NewPos`;

- `rename(Scope,SubstList)` — performs the list of substitutions `SubstList` (of form $[old\_value_1/new\_value_1, \ldots, old\_value_n/new\_value_n]$) throughout the clauses of *scope*. The command ensures that the renaming is legitimate: Prolog meta-variables (*e.g.*: Ψ and Θ) must be renamed to Prolog variables (capital letters or names beginning with capital letters) and second-order variables (*e.g.*: μ and ξ) must be renamed to a Prolog predicate (words starting with non-capital letters). The *scope* of the substitution may be a single clause (b_i or r_j), groups of clauses (*base_cases*, *rec_cases*) or the template as a whole. Special symbols like μ and $\sharp$ are substituted consistently throughout the template;

- `insert(Clause,GoalVector)` — this command creates automatically a new goal to be inserted in component `GoalVector` of `Clause` (may be a single clause or whole groups of clauses). The new goal is depicted in terms of a second-order variable with all relevant arguments of the clause copied as arguments of the new goal;

- `multiply(Clause,N)` — it replicates `Clause` `N` times;

3.3 Skeletons as Specialised Program Templates

The following sequence of customisation operations $\mathcal{T}_{list}$:

$$\mathcal{T}_{list} = \langle \texttt{rename(template,}[\sharp/., \Psi/\texttt{X}, \Theta/\texttt{Xs}, \flat/\texttt{[]}])\rangle$$

customises generic templates to list processing skeletons and techniques. One may easily devise customisation techniques for different data structures: supposing one has a data structure whose functor is `day`, with two components `Hour` (non-recursive) and `RestHours` (recursive) and base-case values $\ddot{\smile}$ and $\ddot{\frown}$, the following customisation technique specialises generic templates to data structure `day`:

$$\mathcal{T}_{\texttt{day}} = \langle \texttt{rename(template,}[\sharp/\texttt{day}, \Psi/\texttt{Hour}, \Theta/\texttt{Hours}, \flat/\{\ddot{\smile}, \ddot{\frown}\}])\rangle$$

Using template T of Figure 8 as a starting point and carrying out the following sequence of specialisation operations:

1. Specialise $\vec{G}_1$ into $\xi(\Psi)$;

2. Apply customisation technique $\mathcal{T}_{list}$;

one will obtain the following specialised template:

$$\mathcal{T}(T) = \left[\begin{array}{ll} \mathit{base\_cases} = & \left[\begin{array}{l} b_1 = \left[\begin{array}{ll} \mathit{head} = & \left[\begin{array}{l} \mathit{pred} = \mu \\ \mathit{args} = \left[\begin{array}{l} \vec{A}_1 = \_ \\ \mathit{case} = \texttt{[]} \\ \vec{A}_2 = \_ \end{array}\right] \end{array}\right] \\ \mathit{body} = [\vec{G}_1 = \_] & \end{array}\right] \end{array}\right] \\ \mathit{rec\_cases} = & \left[\begin{array}{l} r_1^* = \left[\begin{array}{ll} \mathit{head} = & \left[\begin{array}{l} \mathit{pred} = \mu \\ \mathit{args} = \left[\begin{array}{l} \vec{A}_1 = \_ \\ \mathit{str} = \left[\begin{array}{l} \mathit{functor} = . \\ \mathit{non\_rec\_args} = \texttt{X} \\ \mathit{rec\_args} = \texttt{Xs} \end{array}\right] \\ \vec{A}_2 = \mathit{anon} \end{array}\right] \end{array}\right] \\ \mathit{body} = & \left[\begin{array}{l} \vec{G}_1 = \left[\begin{array}{l} \mathit{pred} = \boxed{\xi} \\ \mathit{args} = [\mathit{non\_rec\_args} = \texttt{X}] \end{array}\right] \\ \mathit{rec\_call} = \left[\begin{array}{l} \mathit{pred} = \mu \\ \mathit{args} = \left[\begin{array}{l} \vec{A}_3 = \_ \\ \mathit{rec\_args} = \texttt{Xs} \\ \vec{A}_4 = \_ \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

The $*$ as a superscript of r_1 means that the clause is repeated an unknown number of times, depending on the substitutions of variable ξ (which is enclosed in a box). This specialised template corresponds to skeleton `traverse_n`.

3.4 Some Examples of Techniques

Techniques are conveniently expressed as the pair $\langle T, C\rangle$, where T is an abstract program template describing the generic structure of the skeleton the technique is applicable to and C is a sequence of commands adequately chosen from the repertoire given previously. This subsection shows some examples of techniques described using the proposed representation.

Technique *Collect*: $\mathcal{T}_{Collect} = \langle T, C\rangle$, where

- $T = \left[\begin{array}{l} \mathit{base\_cases} = [b] \\ \mathit{rec\_cases} = \left[\begin{array}{c} r_1 \\ \vdots \\ r_m \end{array}\right] \end{array}\right]$ [1]

- $C = \langle$ `make_copy(`m`,`$\flat$`,Copy)`, `add_parameter(`b`,`$\vec{A}_1$`,Copy)`,
  `for_each(`r_j`,[new_var(`r_j`,`m`,ListVar)`,
  `get_elements([1,`$j-1$`],ListVar,SubListVar1)`,
  `add_parameter(`r_j`,`$\vec{A}_2$`,SubListVar1)`,
  `get_elements(`j`,ListVar,Var`$_j$`)`,
  `add_parameter(`r_j`,`$\vec{A}_2$`,` $\sharp(\Psi,$`Var`$_j))$,
  `get_elements([`$j+1$`,`m`],ListVar,SubListVar2)`,
  `add_parameter(`r_j`,`$\vec{A}_2$`,SubListVar2)`,
  `add_parameter(`r_j`,`$\vec{A}_4$`,ListVar) ] )` $\rangle$

This technique works in a two-fold fashion: first the *only* base-case clause is altered (first two commands) and then the recursive clauses are processed, individually. For the base-case clause,

[1]For the sake of economy of space, the details of the template will not be shown — the structure of the *only* base-case clause b and each recursive clause $r_i, 1 \leq i \leq m$, are the same as that described in the template of Figure 8.

m copies of the base-case value $\flat$ are made and stored in variable `Copy` (command `make_copy`); these copies are then added to vector $\vec{A}_2$ of base-case clause.

The recursive-clauses $r_j, 1 \leq j \leq m$, are processed individually, although in an homogeneous manner — the operator `for_each(Clause,ListOfCommands)` builds a "cluster" of operations to be performed once for each clause r_j. The sequence of operations of each recursive clause ensures that a list of *new* Prolog variables `ListVar`, obtained through command `new_var`, is adequately inserted. In vector $\vec{A}_2$ the technique inserts the first $j-1$ variables (obtained through command `get_elements`), followed by a structure consisting of the functor $\sharp$ of the template, the non-recursive variable Ψ and the j-th new variable $\mathtt{Var}_j$ of `ListVar` and then the $m-j$ new variables left. In $\vec{A}_4$ the whole list of m new variables is inserted.

Technique *Count*: $\mathcal{T}_{Count} = \langle T, C \rangle$, where

- $T = \begin{bmatrix} base\_cases = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \\ rec\_cases = \begin{bmatrix} r_1 \\ \vdots \\ r_m \end{bmatrix} \end{bmatrix}^2$

- $C = \langle$ `add_parameter(`$base\_cases$`,`$\vec{A}_2$`,0),`
  `for_each(`r_j`,[new_var(`r_j`,2,[C,C1]), add_parameter(`r_j`,`$\vec{A}_2$`,C),`
  `add_parameter(`r_j`,`$\vec{A}_4$`,C1), put(`r_j`,`$\vec{G}_2$`,[C is C1 + 1])]) ` $\rangle$

This technique was represented in a previous section using Productions Rules. Its functioning — altering skeleton `traverse_n` — may also be found above. In this representation scheme, the abstract program template is the same as that of Figure 8. The sequence of commands is also two-fold, as in the previous example: the first command simply adds the constant `0` at the end of vector $\vec{A}_2$ of every base-case clause. The second part of the sequence is a cluster (operator `for_each`) of commands to be applied in every recursive clause, one at a time: two new variables `C` and `C1` are produced (command `new_var`); the first variable `C` is attached to the end of vector $\vec{A}_2$ and the second one, `C1`, is attached to the end of vector $\vec{A}_4$. Finally, the goal `C is C1 + 1` is added to the vector of goals $\vec{G}_2$.

3.5 The Prolog Techniques Meta-Editor

The approach presented here naturally gives rise to an automated tool which would help the Prolog expert programmer to manage the knowledge base of a Prolog Program Development Environment. This tool, a *Prolog Techniques Meta-Editor*, would offer a range of services to its users, such as:

- editing of templates and techniques;
- creation of customisation operations;
- testing of techniques;
- comparison and combination of templates and techniques;

[2] The details of the template are as in Figure 8.

- organising sub-sets of skeletons and techniques adequate to different classes of users of the Prolog Techniques Editor associated (beginners, advanced, list-processing programs, etc.)

The Meta-Editor would interface the Knowledge Base in which templates and techniques would be conveniently stored (according to different hierarchies) and the sets (files) of Skeletons and Techniques of a Prolog Techniques Editor. This idea is pictorially suggested through Figure 9.

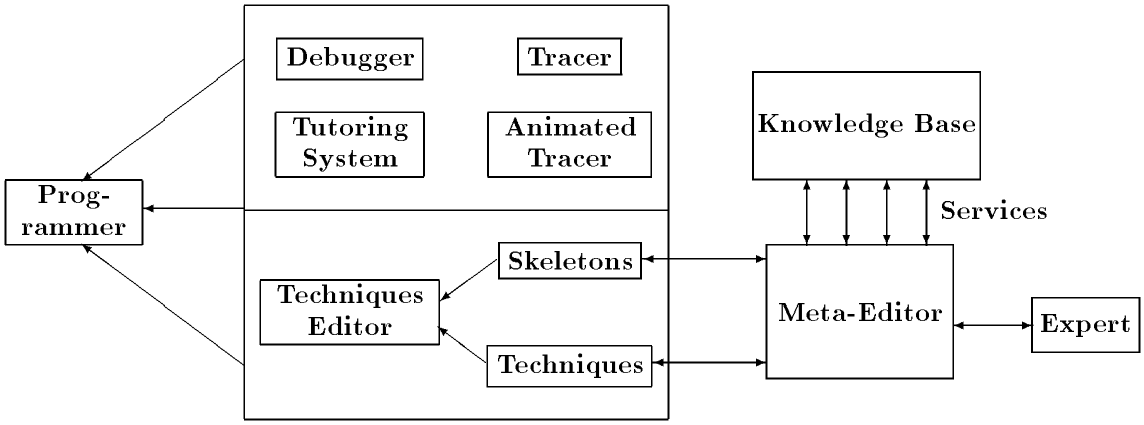

Figure 9: Prolog Program Development Environment and Prolog Meta-Editor

4 Conclusions and Discussion

The use of Skeletons and Techniques seems to be a promising approach to learning, teaching and standardising Prolog programs [10]. It also brings hope of automatising or semi-automatising the development of logic programs: Prolog Techniques Editors [9] are examples of tools for the semi-automated preparation of Prolog programs.

However, Expert Programmers responsible for providing skeletons and techniques for the PTEs are left alone in this pursuit for patterns of programs and programming practices. No methodology, formalism or representation schema has been proposed to help them with this task.

In this paper it is suggested a way to formalise and manipulate the knowledge encoded as skeletons and techniques. Using *Abstract Program Templates* (attribute-value structures which have parts of programs as their constituents) one is able to encode generic descriptions of programs and customise them to obtain skeletons. These templates are also used as part of the description of techniques: in this case a template stands for a pattern of those programs (skeletons or their enhancements) able to undergo the transformations of the technique. These transformations are expressed using commands which enforce the syntactic and semantic correctness of the altered program.

The proposed formalism suggest the conception of a tool, a *Meta-Editor* for Prolog skeletons and techniques, which would offer automated support to expert programmers while formalising the Prolog programming knowledge to be used in PTEs.

To make the templates more expressive and useful, work has been carried out towards cataloguing a larger set of skeletons and techniques, through the study of classic Prolog solutions found in the literature. The templates should de general enough to encompass them. The implementation of the commands and their interaction with the templates have been leading to an improvement of the repertoire of operations to be offered. The organisation of the Knowledge Base in various overlapping hierarchies (according to different customisation techniques) may suggest a plausible model for the way experienced Prolog programmers organise their programming knowledge (by making associations and learning new programs by combining and relating existing ones).

References

[1] Brna, P., Bundy, A., Pain, H. and Lynch, L.; *Programming Tools for Prolog Environment*; Research Paper no. 302; Department of Artificial Intelligence; University of Edinburgh; 1986.

[2] Brna, P.; *Teaching Prolog Techniques*; Research Paper no. 530; Department of Artificial Intelligence; University of Edinburgh; 1991.

[3] Brna, P., Bundy, P., Dodd, T., Eisenstadt M., Looi, C. K., Pain H., Robertsonv, D., Smith, B. and Van Someren, M.; "Prolog Programming Techniques"; *Instructional Science*, **20**(2):111–133; 1991.

[4] Bundy, A.; *Proposal for a Recursive Techniques Editor for Prolog*; Research Paper no. 394; Department of Artificial Intelligence; University of Edinburgh; 1988.

[5] Bundy, A., Grosse G. and Brna, P.; "A Prolog Techniques Recursive Editor"; *Instructional Science*, **20**(2):135–172; 1991.

[6] Johnson, M.; *Attribute-Value Logic and the Theory of Grammar*; Stanford, California; CLSI Lecture Notes no. 16; 1988.

[7] Kirschenbaum, M., Lakhotia, A. and Sterling, L.; *Skeletons and Techniques for Prolog Programming*; Technical Report 89-170; Case Western Reserve University; Ohio, U.S.A.; 1989.

[8] Robertson, D.; *A Preliminary Description of a Techniques Editor for Prolog*; IKBS Group Notes, no. 5; Department of Artificial Intelligence; University of Edinburgh; Edinburgh; 1991.

[9] Robertson, D.; *A Simple Prolog Techniques Editor for Novice Users*; Research Paper no. 523; Department of Artificial Intelligence; University of Edinburgh; Edinburgh; 1991.

[10] Sterling, L. and Kirschenbaum, M.; *Applying Techniques to Skeletons*; Technical Report 91-179; Center for Automationv and Intelligent System Research, Case Western Reserve University; Ohio, U.S.A.; Dec. 1991.