Designing Prolog Programming Techniques

Wamberto Weber Vasconcelos*
wamb@aisb.ed.ac.uk
Department of Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN
Scotland — Great Britain

Abstract

We propose a medium in which expert programmers can design, test and organise Prolog programming techniques. The proposed approach employs simple single-argument program fragments and their combinations in order to represent techniques. The devised techniques can be made available to other programmers, by means of techniques-based editors.

1 Introduction

Due to the simplicity and compactness of the syntax of Prolog, commonly occurring patterns in programs can be detected automatically [Loo88, Bow92] or manually and used in a variety of ways, *e.g.* the detection of bugs [Loo88], the construction of (similar) programs [GH91, KLS89], the automatic explanation of programs [Gab92] and teaching purposes (automatic or human tutoring) [BBD+91, Rob91].

These patterns are loosely named *techniques* [Bow92, Brn91, BBD+91] and, together with knowledge of *when* and *where* to use them, provide a useful account of the body of knowledge necessary for the systematic development of correct Prolog programs. Such work is motivated by the prospect of transferring the largely informal and error-prone task of writing a program into that of finding a technique (or a set of techniques) which solves a given problem.

Current attempts to describe Prolog techniques remain largely informal or have limited coverage. Techniques have been depicted by means of examples (with an informal explanation in some natural language) showing instances of their successful applications [O'K90, BBD+91, KLS89, SS86, SK91]. If these techniques are to be put to use in tutoring systems or techniques editors, it is necessary to bridge the gap between their informal definition and a representation that can be used by a computer.

In this paper we propose an (albeit constrained) medium comprised by a notation, a suggested way to use this notation and a tool embodying it, in which expert programmers can express their knowledge and make it accessible

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

to others. We incorporate the stepwise enhancement method of Kirschenbaum *et al.* [KLS89] in our attempt to formalise programming techniques. Our ultimate goal is to provide an automated tool with which expert programmers could formally express, organise and test their programming knowledge and make it available to other users (novices or other expert programmers) by means of techniques-based editors.

In the following section a brief account of the stepwise refinement method is given and key concepts such as skeletons, techniques and additions are explained. In the third section we introduce the proposed representation for techniques and how a tool, the Prolog Techniques Meta-Editor, embodies it and makes it available to expert programmers. Finally, some conclusions are drawn and directions for future research are discussed.

2 Stepwise Enhancement using Skeletons and Techniques

In [KLS89], Wirth's *stepwise refinement method* is elegantly adapted to the context of Prolog programming. This adaptation uses a set of syntactic entities, called *skeletons*, describing the flow of control of a Prolog program, and a set of syntactic transformations, called *techniques*, to be applied to these skeletons. These techniques convey the programming practices (*e.g.* adding parameters and binding them properly) carried out by experienced programmers upon very simple programs processing input data (*e.g.* traversing a list) in a straightforward way.

This approach advocates the development of programs by means of stepwise enhancements in which programmers first isolate the flow of control needed to solve the problem. Then they choose from amongst the skeletons the one which best suits their needs. After the skeleton selection, the programmer customises it to obtain the final program. This customisation process is achieved by applying techniques to skeletons yielding *enhancements* (*i.e.*, skeletons enhanced with extra arguments, goals and/or clauses). The enhancements can be re-used, allowing one to repeat the process of application of techniques until the final program is obtained. If appropriate sets of skeletons and techniques are provided, the task of writing a program (largely informal and error-prone) could be transferred to that of finding a skeleton and a sequence of techniques.

2.1 Skeletons

Skeletons are basic syntactic constructs depicting the flow of control of a Prolog program [KLS89, Vas92]. They are presented in [KLS89] as basic Prolog programs themselves processing input data in a simple way, and are used to build more complex (and more useful) programs by establishing a flow of control which will be followed while computations are carried out. For instance, the list decomposition skeleton below

```
s([]).
s([X|Xs]):- s(Xs).
```

is found in programs traversing a list, as in the first argument of `reverse`:

```
reverse([],R,R).
reverse([X|Xs],R0,R):-
      reverse(Xs,[X|R0],R).
```

Skeletons do not perform any other computations apart from obtaining new values from an input and recursively iterating this process; terminating conditions (if any) may also be depicted. Techniques, explained below, are required for extra computations carried out as the flow of control of the skeleton is followed.

2.2 Techniques

Techniques are presented in [KLS89] as a mapping from a skeleton to a complete program using the same flow of control. The techniques act upon skeletons, adding arguments to goals, adding goals to clauses or clauses to a skeleton. Skeletons and techniques can be used to construct programs, with the skeleton controlling the manner in which the data is processed, and the techniques determining what is to be done with it, as the execution of the program progresses.

In this work we suggest a formalisation for a large and useful subclass of programming techniques, *viz.*, those techniques described by the arguments and subgoals (and their inter-relationships) added to existing skeletons. The approach advocated consists of isolating the arguments and subgoals comprising a technique and representing them by means of a sequence of single-argument programs along with an appropriate way to merge them and bind their variables. These single-argument programs are named *additions*.

Collecting values and building data structures with them, or counting the number of iterations of a loop are two examples of interesting computations that may be performed, once the flow of control is established by a skeleton. For instance, technique *count-down*, counting the number of times a loop is performed can be represented as:

```
t(...Count,Total):-
      ⋮
      Total = Count.
t(...Count,Total):-
      ⋮
      Count1 is Count + 1,
      t(...Count1,Total).
```

If it is applied to the skeleton above, it yields:

```
s([],Count,Total):-
      Total = Count.
s([X|Xs],Count,Total):-
      Count1 is Count + 1,
      s(Xs,Count1,Total).
```

Technique *count-down* can be seen as the appropriate merging of additions `q` and `r`:

```
q(Count):-
      Var = Count.
q(Count):-
      Count1 is Count + 1,
      q(Count1).
```

```
r(Total):-
      Total = Var.
r(Total):-
      r(Total).
```

Addition `q` uses its argument in input mode, adding one unit at each recursive iteration and binding its final value to `Var` in the base-case clause. Addition `r` passes its argument down the recursive call, and in the base-case has a `Var` bound to it. These two additions are merged together and the `Var` variables are unified, giving rise to technique *count-down*.

3 Designing Skeletons, Additions and Techniques

The stepwise enhancement approach has been put to use in [Rob91], by means of a *Prolog Techniques Editor* (PTE). This knowledge-based software development tool provides its users with a set of skeletons and applicable techniques and guides the user through a series of interactions at the end of which a program is obtained and tested. If the resulting behaviour differs from the intended one, the user may backtrack to previous decision points and examine different alternatives.

During the preparation of the skeletons and techniques which will be offered to the users of the PTE, much ingenuity and labour is required, for no formalisation or methodology has been proposed as an aid. After studying patterns frequently found in programs and acclaimed techniques informally described in Prolog textbooks and papers, the PTE developer chooses a number of skeletons and techniques and manually encodes them in a way that the PTE can use. Later expansions of the PTE by adding, removing or altering skeletons and techniques may involve much work.

An interactive system, the *Prolog Techniques Meta-Editor* (or Meta-Editor, for short), aimed at *expert* programmers, with which one can create and manage a library of skeletons and programming techniques is proposed here. Such a tool is aimed at helping expert programmers to devise and test new skeletons and additions, and define relationships (the programming techniques) between them. Figure 1 below shows the relationship between a PTE and the herewith proposed Prolog Techniques Meta-Editor. The expert programmer prepares and tests skeletons, additions and techniques by means of the Meta-Editor. These constructs are stored in a *library* of programming practices and supplied to the users of the Prolog Techniques Editor. The PTE can be seen as a generic system which could take an arbitrary body of programming knowledge (for example, list-processing programming techniques or programming techniques for novices) and offer it to its users. The Meta-Editor uses ideas of the Recursive Techniques Editor [BGB91] and adapts them to the systematic program development advocated by Kirschenbaum *et al* [KLS89]. The Meta-Editor provides a repertoire of commands similar to that of the Recursive Techniques Editor but aimed at enabling expert programmers to define and manage a library of programming techniques.

Skeletons, additions and techniques are an attempt at representing the com-

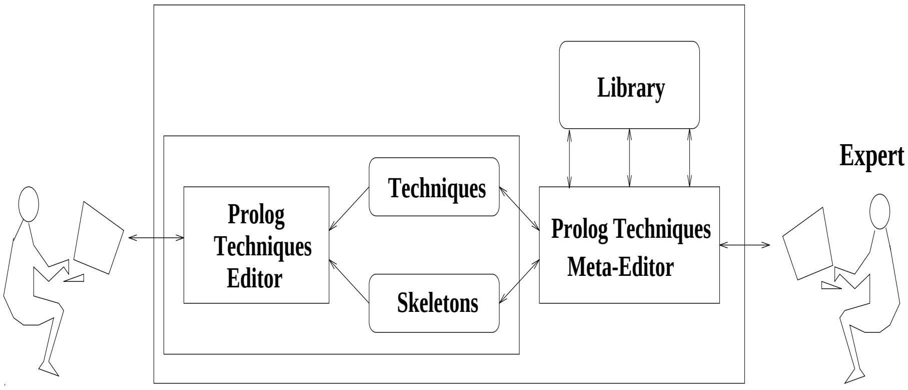

Figure 1: Prolog Techniques Editor and Prolog Techniques Meta-Editor

mon programming practices programmers make use of when devising Prolog programs. In this work, the issue of whether every conceivable programming practice can be devised (completeness) is not a main concern. The usability of the approach, measured by the usefulness of the programming practices that can be devised is, however, an important point. The proposed formalism allows re-using components and devising more complex entities in an incremental manner, also defining natural hierarchies of components. These are advantages which might help the users of the Meta-Editor.

3.1 Designing Skeletons

The proposed Meta-Editor offers its users a repertoire of basic commands through which an initial skeleton consisting of a single general clause of the form below can be gradually customised:

$$\begin{array}{l} s(X) :- \\ \qquad test(X), \\ \qquad obtain(X, \vec{Y}, \vec{Z}), \quad \langle \mathit{offer}(\vec{Y}) \rangle \\ \qquad \overrightarrow{s(Z)}\ . \end{array}$$

$\vec{Y}$ and $\vec{Z}$ are (possibly empty) vectors of variables and $\overrightarrow{s(Z)}$ is a (possibly empty) vector of recursive calls, one for each variable $Z_i \in \vec{Z}$. In the clause above, a *test* is carried out using X, then a computation *obtain* is performed also using X obtaining the vectors $\vec{Y}$ of *Y-values* and $\vec{Z}$ of *Z-values*. Finally, a vector of recursive calls follows, one for each of the Z-values.

The $\vec{Y}$ stands for the values obtained from X to be used within the clause. The annotation $\langle \mathit{offer}(\vec{Y}) \rangle$, as shall be seen, is useful when relating additions to skeletons, listing the values offered within each clause. The $\vec{Z}$ stands for the values obtained from X to be used in the recursive call(s). In order to enforce the establishment of a flow of control, it is required that at least one recursive clause, *i.e.*, a non-empty vector of Z-values, must exist in a skeleton. The subgoal *test(X)* stands for a call to a clause of form

$$test(X) :\!- test_1(X), \ldots, test_t(X)$$

where each $test_i(X)$ can be a constant-dependent test (a test which depends on a constant C, *e.g.*, $X = C, X > C$, etc.), a system single-argument test (a test provided by the system which depends solely on X, *e.g.* `atom`(X), `call`(X), etc.) or a user-defined test (predicates which are not covered by the previous cases).

The subgoal $obtain(X, \vec{Y}, \vec{Z})$ can be specialised into a sequence of computations in terms of X, assigning values to $\vec{Y}$ and $\vec{Z}$. Each of these computations can be in the form of data structures decomposition (*e.g.*, list or binary tree decomposition), arithmetic calculations in terms of X (*e.g.* adding or subtracting a constant value from X) or auxiliary procedures (calls of system or user-defined predicates or retrieval of facts in the data-base).

Skeletons are devised by having the cardinality of $\vec{Y}$ and $\vec{Z}$ specified and the subgoals *test* and *obtain* of each clause defined and unfolded into the skeleton. It is, however, useful to consider more abstract forms of skeletons and hence we allow vectors and predicates to be left unspecified. Unspecified vectors and subgoals stand for *all* their possible instances.

A standard session would initially have the expert programmer defining the number of clauses of the skeleton. After that, the user would go through each clause and would define the number of components of the $\vec{Y}$ and $\vec{Z}$ vectors; these may as well be left unspecified. The predicates *test* and *obtain* are then specified and unfolded into the clause(s); here it may also be the case that some (or all) of these predicates are left unspecified. The Meta-Editor also offers the possibility of re-using previously edited components and further customising them, providing guidance in the definition of the predicates *test* and *obtain*: a

<table>
<tr><th>list</th><th>visit-graph</th><th>meta-interpreter</th></tr>
<tr><td>s(X):-
X = [],
Y1 = X. ⟨offer(Y1)⟩
s(X):-
X = [Y1|Z1], ⟨offer(Y1)⟩
s(Z1).</td><td>s(X):-
X = end,
Y1 = X.⟨offer(Y1)⟩
s(X):-
edge(X,Z1),
Y1 = X,⟨offer(Y1)⟩
s(Z1).</td><td>s(X):-
X = true,
Y1 = X.⟨offer(Y1)⟩
s(X):-
sys(X), call(X),
Y1 = X.⟨offer(Y1)⟩
s(X):-
clause(X,Z1),
s(Z1).
s(X):-
X = (Z1,Z2),
s(Z1),
s(Z2).</td></tr>
</table>

Figure 2: Some skeletons devised using the Meta-Editor.

number of appropriate stereotyped clauses are offered for each of them. Useful flows of control, such as those shown in Figure 2, can be devised using the initial template and the following commands:

1. *Y- and Z-Values*: specifies the actual number of elements of $\vec{Y}$ and $\vec{Z}$. In

the case of Z-Values, the number and arguments of the recursive calls are appropriately adjusted.

2. *Copy*: copies a clause n times.
3. *Fundamental*: inserts a copy of the initial (fundamental) template anywhere in the skeleton.
4. *Remove*: removes a clause (in order to avoid meaningless skeletons, at least one recursive clause has to be left).
5. *Define test and obtain*: *test* and *obtain* are defined and unfolded into the clause. Cliché definitions are offered to the users who customises them to suit their needs (*e.g.* defining the functor of a data structure decomposition).

3.2 Designing Additions

Techniques are comprised of the combination of additions, *i.e.* single-argument program fragments combined together performing computations as the skeleton defines the flow of control. The Meta-Editor encourages the methodical development of simple additions — the building blocks of a technique — which can be subsequently combined into more complex program fragments.

In order to help expert programmers devise additions, the Meta-Editor offers an initial generic template comprised of a single clause and commands for its customisation, in a similar way to the devising of skeletons. However, additions have different initial clauses depending on their intended use mode (input or output). An *input mode addition* (or input addition) has the following initial template:

$$
\begin{array}{ll}
a(X) :- & \langle required(\vec{R}) \rangle \\
\quad prepare(X, \vec{R}, \vec{Y}, \vec{Z}), & \langle offer(\vec{Y}) \rangle \\
\quad \overrightarrow{a(Z)}\ . &
\end{array}
$$

In this clause, the (possibly empty) $\vec{R}$ vector of *R-values*, together with the input value X, is used to *prepare* the vectors $\vec{Y}$ and $\vec{Z}$. The R-values are variable symbols which should be linked to offered values of other additions or skeletons, giving rise to more complex computations. An *output mode addition* (or output addition) has the following initial template:

$$
\begin{array}{ll}
a(X) :- & \\
\quad \overrightarrow{a(Z)}, & \langle required(\vec{R}) \rangle \\
\quad prepare(\vec{R}, \vec{Z}, \vec{Y}, X). & \langle offer(\vec{Y}) \rangle
\end{array}
$$

The ordering of subgoals in the template of the output addition reflects the fact that the argument in the head goal and in the recursive call(s) are initially free variables: it is necessary that the recursive call(s) come first in the clause, binding the vector $\vec{Z}$. Only after the recursive subgoals are executed is that the computation *prepare*, assigning values to $\vec{Y}$ and X, is carried out.

Different templates are necessary because of the strict left-to-right evaluation of Prolog: different orderings of subgoals may produce different results. The two templates are an attempt to enable different orderings of the subgoals to be devised. The template for input additions may also be specialised in an *interleaved* version in which each recursive call has its own *prepare* subgoal. For instance, if we take $\vec{Z} = (Z_1, Z_2)$ we have the following interleaved version of an input addition:

$$\begin{array}{lll} a(X) :- & & \langle required(\vec{R})\rangle \\ & prepare_1(X, \vec{R}, \vec{Y}, Z_1), & \langle offer(\vec{Y})\rangle \\ & a(Z_1), & \langle required(\vec{S})\rangle \\ & prepare_2(X, \vec{S}, \vec{W}, Z_2), & \langle offer(\vec{W})\rangle \\ & a(Z_2). & \end{array}$$

The subgoal $prepare(\vec{A}, \vec{B}, \vec{C}, \vec{D})$, as the *obtain* subgoal of skeletons, can be specialised into a sequence of computations in terms of $\vec{A}$ and $\vec{B}$, assigning values to $\vec{C}$ and $\vec{D}$, in the form of data structures compositions, arithmetic calculations and calls to auxiliary procedures.

The editing of additions is carried out in a similar way to that of skeletons. Depending on whether the user wishes to edit an addition to be used in input or output mode, one of the templates above is initially shown. A typical session would have the expert programmer initially specifying the number of clauses of the addition and then (s)he going through each clause defining $\vec{R}, \vec{Y}$ and $\vec{Z}$ (or leaving them unspecified and hence treated generically). The *prepare* predicates would then be specified and unfolded into the appropriate clauses. A repertoire of commands similar to that of the skeleton editing is available for devising additions. Some examples of useful additions are shown in Figure 3.

carry-value-up (-)	*build-list-up (-)*	*build-list-down (+)*

`a(X):-` $\langle required($`R1`$)\rangle$
  `X = R1.`
`a(X):-`
  `Z1 = X,`
  `a(Z1).`
	`a(X):-` $\langle required(\vec{R})\rangle$
	  `Y1 = [`$\vec{R}$`],` $\langle offer($`Y1`$)\rangle$
	  `X = Y1.`
	`a(X):-`
	  `a(Z1),` $\langle required(\vec{R})\rangle$
	  `X = [`$\vec{R}$`|Z1].`
		`a(X):-` $\langle required(\vec{R})\rangle$
		  `Y1 = [`$\vec{R}$`|X].` $\langle offer($`Y1`$)\rangle$
		`a(X):-` $\langle required(\vec{R})\rangle$
		  `Z1 = [`$\vec{R}$`|X],`
		  `a(Z1).`

Figure 3: Additions devised through the Meta-Editor: the *(+)* and *(-)* indicate that the addition is to be used in input and output mode, respectively.

3.3 Design of Techniques

After additions have been devised, the Meta-Editor offers the possibility of combining them and linking (Definition 3.5 below) their offered and required values. This provides the integration between different additions comprising a technique.

An addition depicts the adjustments to be carried out on each clause of skeletons (or programs) and also works as a pattern describing those skeletons (or programs) to which it is applicable. In order to preserve the declarative meaning of each skeleton and addition, the combination of additions with other additions and their application to skeletons is only possible if their clauses are *compatible* as formalised by Definition 3.2 below.

The combination process is carried out by having the user of the Meta-Editor firstly choosing compatible additions which are then appropriately combined together (Definition 3.3 below). After this, the Meta-Editor guides the user through the definition of links between variables within the combined addition. For instance, additions *build-list-down* and *carry-value-up* shown in Figure 3 can be combined as:

```
a(A1,B1):-
      A2 = [C⃗|A1],   ⟨offer(A2), required(B2)⟩
      B1 = B2.
a(A1,B1):-
      A2 = [C⃗|A1],
      B2 = B1,
      a(A2,B2).
```

The variable names are changed to avoid accidental bindings between them. Also, annotations which are useless for that particular combination (such as $\langle \mathit{required}(\vec{C}) \rangle$ in the example above where no values are offered by *carry-value-up*) are suppressed. After the combined addition is shown, the user may wish to link the offered and required variables within each clause — in the current example, the user may authorise the link `B2` with `A2` in the first clause, obtaining the following *linked combination* (the annotation $\langle \mathit{required}(\vec{C}) \rangle$ is also shown for future combinations):

```
a(A1,B1):-           ⟨required(C⃗)⟩
      A2 = [C⃗|A1],   ⟨offer(A2)⟩
      B1 = A2.
a(A1,B1):-           ⟨required(C⃗)⟩
      A2 = [C⃗|A1],
      B2 = B1,
      a(A2,B2).
```

Linked combinations may subsequently have their remaining required variables linked to offered values of skeletons, giving rise to a programming technique. The linked combination shown previously can be used to define, for instance, a technique building a list with the nodes of a graph obtained by skeleton *visit-graph* (Figure 2):

```
a(X,A1,B1):-
      X = end,
      Y1 = X,
      A2 = [Y1|A1],   ⟨offer(A2)⟩
      B1 = A2.
a(X,A1,B1):-
      edge(X,Z1),
      Y1 = X,
      A2 = [Y1|A1],
      B2 = B1,
      a(Z1,A2,B2).
```

The technique itself is underlined, but for clarity it is shown applied to the skeleton. The required vector $\vec{C}$ was linked to `Y1` in both base-case and recursive clauses of the skeleton. A technique is fully defined when there are no more required values to be linked.

A programming technique can be further combined with other (combined) additions and/or techniques; however, the combination of techniques is only allowed if they share the same skeleton. Once a technique is defined for a skeleton, it is automatically defined for any of its specialisations (if the skeleton has unspecified parts).

The compatibility between two clauses is related to the number of recursive calls: only those clauses with the same number of recursive calls are permitted to be combined. A clause with a vector of recursive calls may be combined with another clause by having its vector of recursive calls first specialised to the same number of recursive calls of the other clause:

Definition 3.1 Clauses c_1 and c_2 are *compatible*, $comp_c(c_1, c_2)$, *iff*:

1. c_1 and c_2 have the same number (possibly zero) of recursive calls, or
2. c_1 or c_2 has an unspecified number of recursive calls (a vector).

The compatibility of each clause in a fragment (skeleton or addition) F_l with another clause in F_r is a necessary and sufficient condition to have them combined together:

Definition 3.2 Program fragment F_l is *compatible* with F_r, $comp_f(\mathsf{F}_l, \mathsf{F}_r)$, *iff* for every clause l of F_l there is exactly one clause r in F_r such that $comp_c(l, r)$.

It is worth noticing that F_r may have other clauses which are not compatible with any clause of F_l, however F_l is still compatible with F_r, but F_r is not compatible with F_l, *i.e.*, the compatibility of program fragments is not symmetric.

The combination of two compatible clauses is performed by pairing off the recursive calls and producing for each pair a combined recursive call with the arguments of the original calls. When combining clauses a distinction should be made between the left- and right-hand side clauses, because the non-recursive subgoals are slotted in the combined version according to the order of the original clauses, the left-hand side subgoals (in their original relative ordering) coming before the right-hand side ones (also in their original ordering). In the definition below $\vec{A}$ stands for a (possibly empty) sequence of arguments and $\vec{S}$ stands for a (possibly empty) sequence of non-recursive subgoals:

Definition 3.3 The *combination between two compatible clauses* $c_i, i = l, r$, of form

$$p(\vec{A^i}) :- \vec{S^i}\, p(\vec{A^i_0})\, \vec{S^i_0}\, \cdots\, p(\vec{A^i_m})\, \vec{S^i_m}$$

is $\dot{c} = c_l \cdot c_r$, of form

$$p(\vec{A^l}, \vec{A^r}) :- \vec{S^l}\, \vec{S^r}\, p(\vec{A^l_0}, \vec{A^r_0})\, \vec{S^l_0}\, \vec{S^r_0}\, \cdots\, p(\vec{A^l_m}, \vec{A^r_m})\, \vec{S^l_m}\, \vec{S^r_m}$$

If both clauses have unspecified vectors of recursive calls, then their vectors of recursive calls should be considered as a single recursive call and combined as such. If only one of the clauses has a vector of recursive calls, then this clause has its vector first customised to the number of recursive calls of the other clause and then they are combined as in the definition above. Whenever clauses are combined, variables will not get accidentally bound by virtue of their sharing the same name.

In the combination of program fragments one must also make the distinction between left- and right-hand side components because their combination is made up of individual combinations of each clause of the left-hand side fragment with one clause of the right-hand side program fragment:

Definition 3.4 The *combination between compatible program fragments* $\mathsf{F}_l = l_1, \ldots, l_m$ and $\mathsf{F}_r = r_1, \ldots, r_n$, is $\dot{\mathsf{C}} = \mathsf{F}_l \cdot \mathsf{F}_r = \dot{c_1}, \ldots, \dot{c_m}, \dot{c_i} = l_i \cdot r_j, \, comp_c(l_i, r_j)$.

One can see that the combination $\dot{\mathsf{C}}$ of program fragments F_l and F_r is still compatible with both F_l and F_r, $comp_f(\dot{\mathsf{C}}, \mathsf{F}_l)$ and $comp_f(\dot{\mathsf{C}}, \mathsf{F}_r)$.

The required values $\vec{R}$ of each clause (if they exist) should be assigned a value of some offered value $Y_i \in \vec{Y}$, after a combination takes place. This provides the integration between the components of a programming technique:

Definition 3.5 The *links* L between a clause c with required values $\vec{R}$ and a compatible clause c', with offered values $\vec{Y}$, is a finite set of pairs A/B where A is either the vector $\vec{R}$ or one of its elements $R_i \in \vec{R}$ and B is either the vector $\vec{Y}$ or one of its elements $Y_j \in \vec{Y}$ or a constant. We shall use c/L to denote clause c after its required values $R, R/Y \in \mathsf{L}$, are replaced by Y, and $\mathcal{L}$ is used to refer to a set of links $\mathcal{L} = \{\mathsf{L}_1, \ldots, \mathsf{L}_n\}$ between two program fragments.

When a link is created between a vector and a particular set of offered values (or a vector of offered values), the vector of required values is properly substituted for the set of offered variables (or the vector of offered values) within the clause. When a vector of offered values is linked to a set of required values, then the clause of the offered values is customised by having the vector of values appropriately substituted for a specific set of offered values with its cardinality the same as that of the required values.

The links are supplied by expert programmers and convey the knowledge used to integrate the independent additions comprising a programming technique. Each required value of a vector in an addition (or a combination of additions) should appear exactly once in the set L of links. The set of links can be provided in a generic way if vectors of values are found. To prevent experts from linking required values twice, once a variable $R \in \vec{R}$ is used, it is removed from the annotation *required*.

Definition 3.6 The *linked combination between compatible program fragments* $\mathsf{F}_l = l_1, \ldots, l_m$, and $\mathsf{F}_r = r_1, \ldots, r_n$, with sets of links $\mathsf{L}_1, \ldots, \mathsf{L}_n$ (between r_i and l_j) is $\widehat{\mathsf{C}} = \hat{c}_1, \ldots, \hat{c}_m, \, \hat{c}_i = l_i \cdot r_j/\mathsf{L}_j, \, comp_c(l_i, r_j)$.

Although the definitions above allow all sorts of combinations between compatible program fragments, not all of them make sense according to our adopted approach. Skeletons, for instance, are not allowed to be combined with other skeletons since they dictate the flow of control of a program and this must be unique. Some meaningless or wrong combinations/links between additions can also be ruled out on syntactic grounds:

1. *Non-empty links between additions* – Additions are only allowed to be combined and stored if the set of links between them is not empty. If

they do not share values, *i.e.* their set of links is empty, then they are not integrated and it is meaningless to combine them[1].

2. *Missing required or offered values* – In order to have two additions linked, there must be at least one offered value in one addition and at least one required value in the other one (in compatible clauses).

3. *Incompatible types of required and offered values* – If the computation(s) of an addition make(s) use of the built-in predicate `is` it can be inferred that its required and/or offered values are *numeric*. Required numeric values can only be linked to offered numeric values – attempts to combine additions with incompatible values are prevented.

4. *Incompatible ordering of required and offered values* – The links between required numeric values and offered numeric values (or vice-versa) are only permitted if in the combined clause the *offer* annotation comes before the *required* annotation. The reason for this constraint is that the required numeric values need to be instantiated, *i.e.* linked to offered values, before the computation (using predicate `is`) is executed.

These cases capture some potential problems that may occur while the expert programmer is devising techniques. The Meta-Editor can offer support to its users by spotting these cases and rejecting the faulty combination. The envisaged setting is one in which "clusters" of properly combined and linked additions are offered to programmers who may test them on skeletons. These clusters are the *programming techniques*:

Definition 3.7 A *programming technique* T is the triple $\mathsf{T} = \langle \mathsf{S}, \widehat{\mathsf{A}}, \mathcal{L} \rangle$ where S is a skeleton, $\widehat{\mathsf{A}}$ is a linked combination of additions compatible with S, $comp_f(\mathsf{S}, \widehat{\mathsf{A}})$, and $\mathcal{L}$ is a (possibly empty) set of links between the clauses of $\widehat{\mathsf{A}}$ and S.

Programming techniques are devised with a particular skeleton (or class of skeletons) S in mind and sometimes there must be links between the skeleton and the combined addition so that offered values of the former can be linked with required values of the latter. It is worth noticing that once a technique is defined for a generic skeleton, then the technique is automatically defined for future specialisations of the skeleton.

Example: Given output addition *build-list-down* (Figure 3) and the following generic skeleton *single-recursion*:

$$
\begin{array}{lll}
\mathtt{s(X)\text{:-}} & & \\
 & test(\mathtt{X}), & \\
 & obtain(\mathtt{X}, \vec{Y}). & \langle offer(\vec{Y}) \rangle \\
\mathtt{s(X)\text{:-}} & & \\
 & test(\mathtt{X}), & \\
 & obtain(\mathtt{X}, \vec{Y}, \mathtt{Z1}), & \langle offer(\vec{Y}) \rangle \\
 & \mathtt{s(Z1).} &
\end{array}
$$

[1] Techniques, however, may have empty links between the skeleton and the addition(s) because an addition might simply use the flow of control of the skeleton, *e.g.* a technique which counts the number of times a loop is performed.

then we can define the technique $\mathsf{T} = \langle \textit{single-recursion}, \textit{build-list-down}, \mathcal{L} \rangle$, where $\mathcal{L} = \{\mathsf{L}_1, \mathsf{L}_2\}$, $\mathsf{L}_i = \{\vec{R}/\vec{Y}\}, i = 1, 2$, which builds a list with values provided by the skeleton. This definition is such that any further specialisations of the skeleton *single-recursion* (through specifying subgoals *test*, *obtain* and the cardinality of $\vec{Y}$) can be enhanced with it.

The combination of additions is carried out by specifying the left- and right-hand side components: only compatible additions are permitted to be combined. The combination process relies entirely on the expert programmer's expertise and in the current stage, the Meta-Editor simply rules out, on syntactic grounds, those additions that are not compatible. An incremental procedure takes place during the preparation of combinations: initially two basic compatible additions are chosen by the expert and combined/linked. The resulting combined/linked addition may be subsequently combined/linked with other additions or other (previously prepared) combined/linked additions.

The Meta-Editor supplies the user with the set of existing input and output additions and combined additions and the user informs which of them will play the role of the left-hand side program fragment in the combination process. After the left-hand side component is chosen, the user is offered those compatible input and output additions and compatible combinations and chooses one to play the role of the right-hand side program fragment. The Meta-Editor then displays their combination (with their variable names changed to avoid clashes) and, if there are any *required* annotations, the user may link them to offered values within the same clause. Combined additions may also be edited and further customised.

After a combined addition is defined, the user may define sets of techniques which make use of it by providing links between the combination and different skeletons. If the final combined addition is such that there are no more *required* annotations, then a technique is automatically defined with the most generic compatible skeleton as part of it. If the combined addition has *required* values to be linked, then the user must inform the skeleton and the links between the skeleton and the combined addition in order to define a technique.

4 Conclusions and Directions of Research

Existing attempts to describe Prolog techniques are largely informal, these normally being depicted by means of examples and an explanation in a natural language. In this paper, we have adopted the stepwise enhancement approach advocated in [KLS89] and suggested a simple and methodical way of formalising the notion of Prolog programming techniques and skeletons. A tool, the Prolog Techniques Meta-Editor, embedding the proposed approach, which assists expert programmers in devising their techniques was also presented.

The proposed approach uses simple one-argument program fragments (the skeletons and additions) expressed in a notation much similar to Prolog itself, and a small repertoire of commands to customise them. The way program fragments are represented encourages a methodical design in which more specific instances are gradually prepared. The devised program fragments can be incrementally combined into more sophisticated constructs giving rise to the programming techniques. The skeletons and techniques are then offered to users of a Prolog Techniques Editor. The components of a technique can be re-used to define other techniques and techniques can also be used to define other more complex techniques.

The notation used to formalise skeletons enables the expert programmer to concentrate on a single value (whether a complex data structure or a simple atom) and the different ways this value is used to obtain new values. The manner of formalising techniques encourages expert programmers to focus on the relationships between different arguments. The program (or class of programs) the technique is aimed at is specified by having the expert programmer providing a skeleton: the technique can be applied only to programs built from that skeleton.

The formalisation employed naturally defines a hierarchy of program fragments in which the topmost nodes are the initial templates offered to expert programmers; their offspring are more specialised forms of them and the leaf nodes are fully specified program fragments. The library of the Meta-Editor could be managed according to this hierarchical view. For instance, the skeleton *single-recursion* shown previously can be seen as a more generic form of skeletons *list* and *visit-graph* presented in Figure 2 — any technique devised for the former is automatically defined for the latter ones. A facility ("browser") to guide the expert programmer through an existing hierarchy of program fragments should be made available as one the services to manage the library of programming practices of the Meta-Editor.

In theory, given an appropriate set of skeletons and techniques every Prolog program can be devised. However, the size of such set and the complexity of its components greatly impair its use. If the programs built using the Prolog Techniques Editor are allowed to be further combined then the set of skeletons and techniques can be kept small and a large number of programs (more complex ones) can still be produced [VVRV93].

The formalism presented in this paper does not support the construction of every conceivable programming technique, and hence not every Prolog program may be developed in this approach. "Bad" programming practices (*e.g.* having a test after a sequence of subgoals which are not related to it – tests should be performed as early as possible so as to save computational effort) should not be able to be devised, and those programs incorporating them should not have their construction supported. However, by dropping some of the restrictions in the definitions given above, removing the concepts of skeletons and techniques and regarding only the additions (with tests) and their combinations as the building blocks of programs, a richer class of programming practices, and hence programs, can be devised.

Additions and techniques that change the flow of control were avoided. These can change the behaviour of the skeleton and their combination may require more careful analysis. Moreover, some techniques, making use of higher-level information such as the number of recursive clauses of the skeleton (for instance *collect* [KLS89]) or adding clauses (initialisation calls) were not addressed. They are material for future research.

The automatic acquisition of skeletons and techniques from a given program is a problem currently under investigation. A program would be supplied with the use mode of the arguments of each subgoal and would be decomposed into its program fragments which would then be matched against the existing library of programming practices. The closest match plus the necessary adjustments to transform it into the extracted program fragment would conveniently incorporate new components to the existing library.

Acknowledgements: Thanks are due to D. Robertson, J. Hesketh, A. Bowles and M. Vargas-Vera for invaluable discussions, to S. Simpson and I. Frank for the proof-reading and to the anonymous LoPSTr '93 referees for the constructive criticisms.

References

[BBD+91] P. Brna, P. Bundy, T. Dodd, C. K. Eisenstadt, M. Looi, H. Pain, D. Robertson, B. Smith, and M. Van Someren. Prolog Programming Techniques. *Instructional Science*, 20(2):111–133, 1991.

[BGB91] A. Bundy, G. Grosse, and P. Brna. A Prolog Techniques Recursive Editor. *Instructional Science*, 20(2):135–172, 1991.

[Bow92] A. W Bowles. *Detecting Prolog Programming Techniques Using Abstract Interpretation.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1992.

[Brn91] P. Brna. Teaching Prolog Techniques. Research Paper 530, Department of Artificial Intelligence, University of Edinburgh, 1991.

[Gab92] D. S. Gabriel. TX: a Prolog Explanation System. Msc dissertation, Department of Artificial Intelligence, University of Edinburgh, 1992.

[GH91] T. S. Gegg-Harrison. Learning Prolog in a Schema-Based Environment. *Instructional Science*, 20:173–192, 1991.

[KLS89] M. Kirschenbaum, A. Lakhotia, and L. Sterling. Skeletons and Techniques for Prolog Programming. Technical Report 89-170, Case Western Reserve University, Ohio, U.S.A., 1989.

[Loo88] C.-K. Looi. *Automatic Program Analysis in a Prolog Intelligent Teaching System.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1988.

[O'K90] R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[Rob91] D. Robertson. A Simple Prolog Techniques Editor for Novice Users. In *3rd Annual Conference on Logic Programming*, Edinburgh, Scotland, April 1991. Springer-Verlag.

[SK91] L. Sterling and M. Kirschenbaum. Applying Techniques to Skeletons. Technical Report 91-179, Center for Automation and Intelligent Systems Research, Case Western Reserve University, Ohio, U.S.A., Dec 1991.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.

[Vas92] W. W. Vasconcelos. Formalising the Knowledge of a Prolog Techniques Editor. In *9th Brazilian Symposium on Artificial Intelligence*, Rio de Janeiro, Brazil, October 1992.

[VVRV93] M. Vargas-Vera, D. Robertson, and W. W. Vasconcelos. Building Large-Scale Prolog Programs using a Techniques Editing System. Research Paper 635, Department of Artificial Intelligence, University of Edinburgh, 1993. To be presented as a poster in the International Logic Programming Symposium, Vancouver, British Columbia, Canada, Oct., 1993.