Prolog Program Development via Enhanced Schema-Based Transformations*

Wamberto W. Vasconcelos†
Department of Artificial Intelligence
University of Edinburgh
Scotland, U.K.
wamb@aisb.ed.ac.uk

Norbert E. Fuchs
Department of Computer Science
University of Zürich
Switzerland
fuchs@ifi.unizh.ch

Abstract

We describe state-of-the-practice results in integrating a program transformation approach to a program development environment. We propose a formalisation for a specific kind of programming knowledge, the *programming techniques*. Our proposal employs *program schemata*, generic descriptions of programs addressing portions of predicate definitions in an economical manner, allowing important features to be highlighted and irrelevant details to be disregarded. This formalisation of programming techniques is employed in the definition of a program development framework which supports the preparation of individual predicates, and additionally provides a supervising facility which checks inter-procedural relations for possible efficiency improvements.

1 Introduction

In this work we propose a formalisation for a specific kind of programming knowledge, the *programming techniques*. Programming techniques are common code patterns of a programming language, independent of algorithms and problem domains, used systematically in different contexts and for different purposes [BBD+91], and extensively studied within the logic programming community (*e.g.* [Lak89], [SK93], [BRV+94], [Vas95]). Our formalisation of programming techniques is employed in the definition of a program development framework which supports the preparation of individual procedures using programming techniques, and additionally provides a supervising facility which checks the inter-procedural relations for possible efficiency improvements.

In the rest of this section we explain the concept of programming techniques adopted in this work. In Section 2 we explain the notation used to formalise programming techniques, the *enhanced schema language*. In Section 3 we introduce our representation scheme for programming techniques and define a techniques-based program development framework. In Section 4 we conclude and give directions for future research.

1.1 Prolog Programming Techniques

The compactness of Prolog programs allows the easy detection of commonly occurring patterns in the code. These patterns are loosely named *techniques* [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. Prolog programming skills consist, to a large extent, in the capacity to exploit the relatively simple syntax of Prolog in elaborate manners. Prolog program writing can be profitably seen as the proper assemblage of standard code patterns, the programming techniques [BRV+94, BBD+91].

Prolog programming techniques will be considered as being associated with the argument positions of a procedure: each argument position is either a programming technique on its own or is part of a more complex technique together with other argument position(s). For instance, the following definition of the *count*/2 predicate

```
count([],0).
count([H|T],Count):-
      count(T,CountT),
      Count is 1 + CountT.
```

which holds if its second argument is the number of elements of the list comprising the first argument, is such that a technique to manipulate a list has been used in the first argument, that is

*Submitted to the 7th Workshop on Logic Programming Environments, held in conjunction with ILPS'95. This work was partially carried out within the European Human Capital and Mobility Scheme (contract No. CHRX-CT 93-00414/BBW 93.0268) while the first author was visiting the Institut für Informatik, Universität Zürich.

†Sponsored by the Brazilian National Research Council (CNPq, grant No. 201340/91-7), on leave from UECE/Ceará, Brazil.

```
count([],...).
count([H|T],...):-
      count(T,...).
```

and another technique to count the iterations of a loop has been employed in the second argument position, that is

```
count(...,0).
count(...,Count):-
      count(...,CountT),
      Count is 1 + CountT.
```

Our approach represents a technique by formalising the context of its application, *i.e.*, how the procedure looks like *before* the technique is applied to it, and its prescribed alterations, *i.e.* how the procedure looks like *after* the technique is applied to it. In order to represent this proposal, we define a language, the *enhanced schema language*, which permits to represent logic programming constructs in a generic and economic fashion, in which important parts can be brought into focus and irrelevant details can be disregarded.

2 Enhanced Schema Language

In order to describe logic programming constructs we make use of the enhanced schema language defined in our previous work [VF95b, VF95a], an extension of Gegg-Harrison's proposal [GH93]. Logic program schemata are generic descriptions of logic programs, or of classes of logic programs, in a suitable Horn-clause notation.

In our enhanced schema language we provide a notational device which allows reference to arbitrary argument positions. It consists of adding the construct "$\#n$", where n is the argument position, to the term being referred to. This circumvents the restriction of Gegg-Harrison's schema language of allowing only the representation of consecutive initial argument positions of procedures. Since we want program schemata to stand for generic descriptions of actual programs, we shall assume that our addressed programs are translated into an *explicit argument format*, in which subgoals of the form $p(t_1, \ldots, t_n)$ are replaced by $p(t_1\#1, \ldots, t_n\#n)$ and appropriately translated back into the conventional Prolog format after the analysis. A similar explicit notation is offered to represent the order of clauses within a program schema. Vectors of clauses, denoted by $\vec{C}$, can be used to represent possibly empty sequences of clauses.

The adopted schema language allows the description of large classes of constructs sharing particular features, ignoring unimportant variations. The construct $t\#n$ represents the constraint that term t occupies position n in the subgoal; the symbol $\vec{\mathtt{A}}_i$ stands for a possibly empty sequence of arguments; $\vec{\mathtt{G}}_i$ represents a possibly empty sequence of *non-recursive* subgoals; $\vec{\mathtt{G}}_i^*$ represents a possibly empty sequence of any sort of subgoals; $x\#1\ \mathtt{is}\ \mathtt{F}(\vec{\mathtt{A}}_n, y, \vec{\mathtt{A}}_m)\#2$ represents a Prolog `is` subgoal in its explicit argument position format, such that the expression on the right-hand side has at least one reference to the variable symbol y: all references to y are economically represented by a single occurrence of that symbol.

The enhanced schema language also allows the abstract description of functors and their arguments, constants and system predicates which serve as tests. Constants are generically denoted as $\mathtt{ct}(n)$ where n differentiates distinct constants in the same schema. Prolog tests, viz. >, <, >, *integer*/1, *ground*/1, *number*/1, and so on, are generically denoted by $\mathtt{T}_n$ where n differentiates distinct tests in a schema and may be omitted. Using these additional shorthands even larger classes of programs can be depicted by simple schemata. The schema below depicts those programs manipulating singly-recursive data structures, such that its termination occurs when a constant is found:

$$
\boxed{
\begin{array}{l}
\mathtt{S}(\vec{\mathtt{A}}_1,\mathtt{ct}(\_)\#\mathtt{N},\vec{\mathtt{A}}_2)\mathtt{:-}\\
\quad \vec{\mathtt{G}}_1.\\
\mathtt{S}(\vec{\mathtt{A}}_3,\mathtt{F}(\vec{\mathtt{A}}_4,\mathtt{Xs},\vec{\mathtt{A}}_5)\#\mathtt{N},\vec{\mathtt{A}}_6)\mathtt{:-}\\
\quad \vec{\mathtt{G}}_2,\\
\quad \mathtt{S}(\vec{\mathtt{A}}_7,\mathtt{Xs\#N},\vec{\mathtt{A}}_8),\\
\quad \vec{\mathtt{G}}_3.
\end{array}
}
$$

Ordinary Prolog constructs can be used within program schemata, in which case they are considered as constraints, since they are specific syntactic patterns. We use frames around schemata to improve their visualisation.

2.1 Programming Techniques as Enhanced Schema-Based Transformations

Programming techniques can be conveniently formalised by means of enhanced schema-based program transformations. The context of the application of a programming technique can be depicted by a program schema and the prescribed alterations as another schema. For instance, the technique to count the iterations of a singly-recursive loop can be formalised as an enhanced schema-based transformation which maps programs of the form shown in the left-hand schema onto programs of the form shown in the right schema:

```
P(A⃗1):-
   G⃗1.
P(A⃗2):-
   G⃗2,
   P(A⃗3),
   G⃗3.
```

$\Longrightarrow$

```
P(A⃗1,0#_):-
   G⃗1.
P(A⃗2,Count#_):-
   G⃗2,
   P(A⃗3,RecCount#_),
   Count#_ is (RecCount + 1)#_,
   G⃗3.
```

The left-hand schema describes the class of programs the technique is aimed at, *viz.* singly recursive procedures with two clauses, and the second schema describes the alterations the technique performs to those procedures matching the initial schema, appropriately adding an argument position in each clause and a subgoal in the recursive clause. The actual explicit argument positions in the resulting program are not relevant, since the program will be translated back to its conventional form free of explicit argument positions.

Our proposal employs an initial partial schema substitution to constrain the matching process and to have generic constructs in the program schema appropriately defined. A schema substitution maps the schema constructs to actual program components – in [VF95b] it is described how to obtain schema substitutions for enhanced schemata. A technique to sum the elements of a data structure, for example, can be formalised as the following pair of enhanced program schemata

$\langle$

```
P(A⃗1,ct(_)#N,A⃗2):-
   G⃗1.
P(A⃗3,F(X,Xs)#N,A⃗4):-
   G⃗2,
   P(A⃗5,Xs#N,A⃗6),
   G⃗3.
```

,

```
P(A⃗1,ct(_)#N,A⃗2,0#_):-
   G⃗1.
P(A⃗3,F(X,Xs)#N,A⃗4,Sum#_):-
   G⃗2,
   P(A⃗5,Xs#N,A⃗6,SumXs#_),
   Sum#_ is (SumXs + X)#_,
   G⃗3.
```

$\rangle$

If, however, there are more than one argument manipulating data structures in the program the technique is applied to, then there are more than one possible way to match the initial program schema against the program. One way to solve this is by allowing generic constructs to be specialised by an initial schema substitution Θ_0. In our example above, $\Theta_0 = \{\mathtt{N}/n\}$, where n is a positive integer.

A programming technique is formalised as a special form of enhanced schema-based transformation [FF92, VF95b, VF95a]. Programming techniques are triples of the form $\mathcal{T} = \langle \Theta_0, \mathbf{S}_{in}, \mathbf{S}_{out} \rangle$, where Θ_0 is a possibly empty initial schema substitution, $\mathbf{S}_{in}$ is the input program schema depicting the program before the technique is applied (the *context* of the technique) and $\mathbf{S}_{out}$ is the output program schema depicting the initial program with the changes prescribed by the programming technique.

During the application of a programming technique to a program some degree of interaction with the user of the system must take place. A number of design decisions require human intervention and cannot be fully automated. In our previous example, the integer n supplied as the value of the variable `N` should be provided by the person involved in the program development task. Many design choices require human participation, the problem to which a program is being developed motivating particular decisions. As we shall see below, our program development environment supports an interactive component during the application of a programming technique, specialising some of the abstract constructs of the input and output program schemata.

3 Program Development via Enhanced Schema-Based Transformations

Program writing can profitably be seen as the task of retrieving, adapting and combining programming techniques of existing programs. Programming techniques provide ready-made solutions for subtasks of programming; they provide more abstract pieces of code that can be used to implement part of a specification. The largely informal and error-prone task of writing a program can be converted into that of finding a technique or a combination of techniques which implements the solution for a given problem [BBD+91].

In this section we use the formalisation of programming techniques given above to define a program development framework. Our techniques-based program development environment defined below provides a simple formalisation for existing related work, such as Kirschenbaum and colleagues' [KLS89, SK93], Robertson's [Rob91] and Bowles' [BRV+94].

3.1 Initial Programming Techniques

Special programming techniques provide a simple program which serves as the starting point of the program development process. These simple programs establish the flow of execution, by means of straightforward loop constructs. Examples of such simple programs are list-processing programs, a program to visit the nodes of a binary tree in pre-order, and so on. Initial programming techniques can be seen as an alternative formalisation for the *skeletons* of programs of Kirschenbaum and colleagues [KLS89, SK93].

Initial programming techniques do not need input program schemata. They do not work by altering an existing program, but by defining an initial very simple program, without referring to existing constructs. In initial programming techniques we shall denote the input program schema as the empty set $\emptyset$. The initial schema substitution must provide important details such as the name of the newly defined program, specific value(s) of constant(s) symbols, and so on. For instance, the initial programming technique below

$$\langle \{\mathtt{P}/p\}, \emptyset, \boxed{\begin{array}{l}\texttt{P([]\#\_).}\\ \texttt{P([X|Xs]\#\_):-}\\ \quad\texttt{P(Xs\#\_).}\end{array}} \rangle$$

depicts a very simple list-processing program traversing a list until it is empty. The initial schema substitution stores the instantiation of predicate variable `P` to a concrete predicate symbol p supplied by the person developing the program. The application of initial programming techniques should supply us with very simple but fully executable Prolog programs.

Some degree of user interaction is necessary. The repertoire of initial programming techniques is offered to the users of our environment and they must choose one among them. Support should be given to allow the user make experiments and test different possibilities. The content of the initial schema substitution Θ_0 must also be supplied by the user. An adequate interface mechanism to support the specification of Θ_0 should hence be provided.

3.2 Program-Augmenting Programming Techniques

Program-augmenting programming techniques add further functionalities to existing procedures. These techniques can be applied to the simple programs obtained by initial programming techniques, to more complex procedures previously augmented with other programming techniques, or to ready-made procedures devised by other means. The distinguishing feature of a program-augmenting technique is its non-empty input program schema depicting the form a procedure should have to in order to have the technique applied to it. The output program schema depicts the syntactic additions that should be incorporated when the technique is applied to a procedure matching the input schema.

Some program-augmenting programming techniques are such that no further details need to be provided via the initial schema substitution Θ_0. The programming technique to count the iterations of a singly-recursive loop using an accumulator pair is one of such techniques, being represented by the enhanced schema-based transformation

$$\langle \emptyset, \boxed{\begin{array}{l}\texttt{P(}\vec{\texttt{A}}_1\texttt{):-}\\ \quad\vec{\texttt{G}}_1\texttt{.}\\ \texttt{P(}\vec{\texttt{A}}_2\texttt{):-}\\ \quad\vec{\texttt{G}}_2\texttt{,}\\ \quad\texttt{P(}\vec{\texttt{A}}_3\texttt{),}\\ \quad\vec{\texttt{G}}_3\texttt{.}\end{array}}, \boxed{\begin{array}{l}\texttt{P(}\vec{\texttt{A}}_1\texttt{,Count\#\_,Count\#\_):-}\\ \quad\vec{\texttt{G}}_1\texttt{.}\\ \texttt{P(}\vec{\texttt{A}}_2\texttt{,Acc\#\_,Count\#\_):-}\\ \quad\vec{\texttt{G}}_2\texttt{,}\\ \quad\texttt{NewAcc\#\_ is (Acc + 1)\#\_,}\\ \quad\texttt{P(}\vec{\texttt{A}}_3\texttt{,NewAcc\#\_,Count\#\_),}\\ \quad\vec{\texttt{G}}_3\texttt{.}\end{array}} \rangle$$

All the details needed for the successful application of the technique above are depicted by the patterns of its enhanced schemata. The initial schema substitution is not necessary and an empty set is thus assigned to it.

3.3 A Techniques-Based Program Development Framework

A stepwise program development framework can be defined incorporating the formalisation of programming techniques given previously. This approach has been advocated in [KLS89], based on the work developed by Wirth [Wir71] for the procedural paradigm. In the stepwise program development a simple program depicting the control flow of the execution is initially chosen and further augmented until a final program which solves a given problem is eventually obtained.

The stepwise approach can be defined in terms of the concepts introduced above. An initial programming technique depicting the flow of execution is firstly chosen and a succession of program-augmenting programming techniques is applied to it:

$$stepwise(P^F) \longleftrightarrow initial(P) \wedge augment(P, P^F)$$

The definition for *initial* must provide a user-friendly interface to support the decision-making process. A facility to browse through the repertoire of initial programming techniques should be offered to the users, as well as a means to supply the initial schema substitution.

The stepwise construction of procedure P is defined as an initial procedure P' obtained via one of the initial programming techniques and its augmentation into a final procedure P. The augmentation process is defined recursively:

$$\begin{array}{lcl} augment(P, P^F) & \longleftrightarrow & (solves(P) \wedge P^F = P) \vee \\ & & (choose(\mathcal{T}, P) \wedge apply(\mathcal{T}, P, P') \wedge augment(P', P^F)) \end{array}$$

The augmentation ends when a procedure P which solves a given problem is eventually obtained (base case). A procedure P is gradually augmented by having a program-augmenting technique $\mathcal{T}$ chosen and properly applied to it, yielding P', and recursively using P' as the basis for future augmentations (recursive clause). Predicate *solves* provides support to the task of testing whether a procedure solves a given problem. A given set of queries and their answers, for instance, could be used to test if the procedure devised so far successfully solves the problem at hand. The *choose* predicate must also provide an interaction mechanism to help users choosing the next programming technique to be applied – the choice is dependent on procedure P devised so far. Predicate *apply* appropriately augments P with $\mathcal{T}$ thus obtaining P'; an interface mechanism should support users at this stage when supplying schema substitutions.

The stepwise approach can be incorporated to a program development environment and complete programs can be manipulated rather than isolated procedures:

$$\begin{array}{lcl} augment^{\Pi}(\Pi, \Pi^F) & \longleftrightarrow & (solves^{\Pi}(\Pi) \wedge \Pi^F = \Pi) \vee \\ & & (procedure(\Pi, P) \wedge choose(\mathcal{T}, P) \wedge apply(\mathcal{T}, P, P') \wedge update(\Pi, P, P', \Pi') \wedge augment^{\Pi}(\Pi', \Pi^F)) \end{array}$$

Predicate $augment^{\Pi}$ addresses full programs, augmenting a program Π into Π^F. The $solves^{\Pi}$ predicate checks if program Π solves a given problem and establishes the condition to stop the augmentation process. The augmentation process itself still addresses one procedure at a time: predicate *procedure* extracts procedure P to be augmented from Π, user participation being essential during this task, and support must be given to ensure that all predicates in the program are eventually defined. The *choose* and *apply* predicates are as above.

The *update* task appropriately updates Π, replacing P for P' if there are no references to P elsewhere in Π, or by inserting P' into Π, extending the program:

$$\begin{array}{lcl} update(\Pi, P, P', \Pi') & \longleftrightarrow & (reference(\Pi, P) \wedge \Pi'' = \Pi \cup P' \wedge transform(\Pi'', \Pi')) \vee \\ & & (\neg reference(\Pi, P) \wedge \Pi'' = (\Pi - P) \cup P' \wedge transform(\Pi'', \Pi')) \end{array}$$

In either case, a supervision facility *transform* is invoked: the new program is checked for inter-procedural relationships and opportunities for improving its efficiency are pursued. The *transform* predicate is as defined in [VF95b, VF95a]: opportunities for improving a logic program are systematically attempted, either by examining its procedures in an isolated fashion, or by checking for conjunctions within clauses that can be used as joint specifications. Opportunities are represented as enhanced schema-based transformations, generic descriptions of inefficient programming constructs and of how these should be altered in order to confer a better computational behaviour on the program.

4 Conclusions and Directions of Research

In [VF95b, VF95a] we proposed an enhanced schema language to generically represent logic programming constructs. In that work, the schemata were used to describe program transformations. A fully automatic opportunistic approach for program analysis and optimisation was defined based on the enhanced schema-based transformations.

In this work we have shown that the same enhanced schema language can be used for *developing* Prolog programs. Programming techniques can be economically depicted using enhanced program schemata. A framework for the stepwise development of Prolog programs can be defined in a straightforward fashion using the proposed formalism for programming techniques. We have also shown that the opportunistic program transformation system can be embedded into the program development environment and be used as a supervision facility that checks for inter-procedural relationships and tries to improve their efficiency.

A number of important issues concerning the repertoire of programming techniques should be considered:

- *Selection of a representative corpus of programming techniques* – the repertoire of initial and program-augmenting programming techniques should cover a representative corpus of programming practices, describing common flows of execution (lists, graphs, formulae, binary trees, and so on) and classic augmentations (difference lists, accumulator pairs, and so on).
- *Presentation of programming techniques* – depending on the experience of the user, alternative formats for the presentation of the programming techniques should be considered. For instance, if the user is a novice programmer, a cleaner visual representation should preferably be employed.
- *Organisation of repertoire of programming techniques* – a number of relationships can be used to help organising the programming techniques available to our system, such as the number of clauses and recursive calls, employed data structures, termination, and so on. The same repertoire of programming techniques could be organised differently, depending on the level of expertise of its users.
- *Theoretical basis* – initial programming techniques can be seen as defining execution trees which are then altered by the program-augmenting techniques. Initial programming techniques may define infinite execution trees, or simple, finite ones. Some program-augmenting programming techniques may change the flow of execution established by the initial programming technique: the execution tree is, in this case, altered. Programming techniques and Prolog programming development could be studied under this perspective.

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.

[BRV+94] A. W. Bowles, D. Robertson, W. W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying Prolog Programming Techniques. *International Journal of Human-Computer Studies*, 41(3):329–350, September 1994. Also as Research Paper 641, Dept of Artificial Intelligence, University of Edinburgh. Eletronic version available at `http://www.dai.ed.ac.uk:80/students/wamb/Publications.html`.

[FF92] N. E. Fuchs and M. P. J Fromherz. Schema-Based Transformations of Logic Programs. In *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LoPSTr'92)*. Springer-Verlag, 1992.

[GH93] T. S. Gegg-Harrison. Exploiting Program Schemata in a Prolog Tutoring System. Technical Report CS-1993-11, Department of Computer Science, Duke University; Durham, North Carolina, U.S.A., April 1993. Reformatted version of PhD dissertation with the same title and equivalent content.

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

[Lak89] A. Lakhotia. Incorporating 'Programming Techniques' into Prolog Programs. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, Ohio, U.S.A., October 1989.

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

[SK93] L. Sterling and M. Kirschenbaum. Applying Techniques to Skeletons. In *Constructing Logic Programs*. John Wiley & Sons Ltd, London, England, 1993.

[Vas95] W. W. Vasconcelos. *Extracting, Organising, Designing and Reusing Prolog Programming Techniques*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, August 1995. Eletronic version available at `http://www.dai.ed.ac.uk:80/students/wamb/Publications.html`.

[VF95a] W. W. Vasconcelos and N. E. Fuchs. An Opportunistic Approach for Logic Program Analysis and Optimisation using Enhanced Schema-Based Transformations. In *Proceedings of the Fifth International Workshop on Logic Program Synthesis and Transformation (LoPSTr'95)*, Utrecht, the Netherlands, 1995.

[VF95b] W. W. Vasconcelos and N. E. Fuchs. Opportunistic Logic Program Analysis and Optimisation: Enhanced Schema-Based Transformations for Logic Programs and their Usage in an Opportunistic Framework for Program Analysis and Optimisation. Technical Report 95.24, Institut für Informatik, Universität Zürich, August 1995. Eletronic version available at `http://www.dai.ed.ac.uk:80/students/wamb/Publications.html`.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.