An Environment for Building Prolog Programs Based on Knowledge about their Construction

María Vargas-Vera and Dave Robertson
Department of Artificial Intelligence
University of Edinburgh,
80 South Bridge,
Edinburgh EH1 1HN, U.K.
email: {mariav,dr}@aisb.ed.ac.uk

Program combination can be used to promote the reuse of software by allowing complex programs to be built by repeated combination of other programs.

Previous attempts at automatic systems which assist programmers in the task of combining programs have generally required lots of interaction from a user, who also typically needs a good understanding of the particular program transformation process being applied. A system for transforming programs expressed as recursion equations is given in [BD77], but its use requires intervention of a human with a good understanding of program transformation methods. In procedural languages, there are ways [HPR88] to merge programs derived from an initial generic template, however, this approach is restricted to a limited class of programs. In [TS83, TS84], an unfold/fold based transformation system was given, but requires user intervention and is restricted to programs with the same flow of control. In [LS87, SL88] methods were given for combining Prolog programs with the same basic flow of control (meta-interpreters) but these also require user intervention. The method of [FF91] employs basic schemata (supplied by an expert) to combine list-processing programs. The method is very efficient, however it can present the user with a difficult choice between possible output schemata. The method in [PP92] combines logic programs with the same flow of control using basic fold/unfold transformations, but relies on user-guidance and its efficiency depends on the decisions made by the user.

Given these problems of over-reliance on the user, and lack of flexibility, we started out with three basic ideas as to how the situation could be improved.

- Possibly, having a high level description of the workings of the programs would be useful.
- Such a high-level description was potentially available from Prolog techniques editors, by modifying them so as to record the history of the program development (which we call the *program history*). The advantage of using the *program history* is that it would give information about the program that would otherwise be difficult to re-extract from the finished program.
- We should develop many special purpose combination methods rather than attempt to find a small number of general purpose methods. This was motivated by initial experiments in combining various standard programs with the known methods.

To investigate these ideas we took Sterling's notion of initial control flows[1], extended it with additional "mutations", and developed it into a program classification scheme.

We then designed combination methods that were typically only to be used for particular classes. We found that this had the advantage that the method could then make most of the choices necessary in the combination process (choices that the user would otherwise have to make), and also work efficiently for its intended class of programs. The methods were able to do this using the knowledge contained in the program history. Furthermore, the program history itself could be used to classify the program and so select the correct method.

The initial system was then further improved by refinements to the classification system and the development of new methods that were tailored to make an effective use of the program history, (whilst also ensuring that we could still use the program history to select the correct method).

In some cases, when the user had underspecified the requirements of the combined program, the classification of programs (using the high-level description in the program history) implicitly allowed the method to make an informed estimate of what were the users likely intentions, see the example presented later. The methods were also able to combine programs with different flows of control, and for a common class of programs with arithmetic operations the system can deduce which arithmetic laws can be applied in order to get a more optimal combined program. More detailed description of these methods can be found in [VV94].

Thus, our ideal program construction system consists of a Prolog *techniques editor* and a *composition system*. We have not built a techniques editor because such systems already exist [Rob91, Bow93], and could easily be modified to record the choices that the user makes when building a program. In these editors, such choices include selecting the initial control flow (skeleton) and the techniques (standard Prolog practices). These choices are stored in the program history, which therefore contains a high-level description of the program that is close to the way that an expert might think about the program (simply because the editors are designed to use methods that experts might use themselves, but allow novices to use them easily).

The *composition system* allows users to construct more complex programs by combining *simpler programs* which have been built by means of a techniques editor as described above. A component of this system is the *selection procedure* which automatically selects a combination method according to the program histories of the simpler programs. The selected method then controls the application of transformation rules (taken from a library), making its decisions according to the information it finds in the program history. We have designed the methods so that when they meet an underspecified combination, then they will tend to do transformations that preserve the spirit of the program histories (and hence their functionalities) rather than the simplest logical alternative, again we refer the reader to the example later. We believe this is more likely to match user intentions (in the future

[1]This basis for the classification scheme was not chosen randomly, but was based on our studies in which we found that knowledge of the flow of control was important in guiding the combination process.

we would also allow user confirmation at this stage).

In this way the composition system produces as output the combined program and a new history for the combined program. These can be stored and used as input for further combinations. The main characteristics of our composition system are:

- The system decides the combining method by analysing the pair of programs to be combined (using the program history).
- The user does not need to take major decisions in the composition system such as which clauses need to be unfolded or folded (i.e. the user does not need to know about program transformation).

Let us now proceed to the promised example. Suppose we have two predicates:

`pos(List,Element,Pos)`:
finds `Element` in the `List` with position `Pos` (counting from the head of the list).

```
pos([X|_],X,1).
pos([_|T],X,N) :-
        pos(T,X,NP),
        N is NP+1.
```

`path(List,Element,Path)`:
finds `Element` in the `List` with path `Path` (meaning sequence of elements up to and including the `Element`).

```
path([X|_],X,[X]).
path([H|T],X,[H|R]) :-
        path(T,X,R).
```

Now suppose that the user requests that we combine these two programs. Most transformation systems will take the join specification to be the Prolog program

```
p_path1(L,X,P,LP) :-
            pos(L,X,P),
            path(L,X,LP).
```

and then convert this to the logically equivalent program

```
p_path1([X|_],X,1,[X]).
p_path1([X|T],X,1,[X|R]):-
            path(T,X,R).
p_path1([X|T],X,N,[X]) :-
            pos(T,X,NP),
            N is NP+1.
p_path1([H|T],X,N,[H|R]) :-
            p_path1(T,X,NP,R),
            N is NP +1.
```

In this case, if we give the query `p_path1([1,2,1,2],2,P,Path)` then Prolog will give four answers

P = 2	P = 4	P = 2	P = 4
Path = [1,2]	Path = [1,2,1,2]	Path = [1,2,1,2]	Path = [1,2]

In our opinion, this would usually not correspond to the users intentions. Instead, it seems more likely that the user would want to combine the functionalities of the programs to get

```
p_path2(List,Element,Pos,Path):
finds Element in the List with position Pos, and path
Path.
```

in which case the last two solutions would be unwanted, because they correspond to finding different copies of the same element in the list and returning the path for one but the position for the other. Systems that rely only on the join specification written in the Prolog form have no way to know that the user wanted exactly the same `Element`. That is, we want to synchronise the list search performed in `pos/3` and `path/3`. There is no way to express this requirement directly in Prolog using only the given predicates.

However, our system does not rely on the Prolog form of the join specification, but effectively has an extended join specification which we can write as

```
p_path2(L,X,P,LP) ⇐
        pos(L,X,P) ⋈
        path(L,X,LP).
```

where the underlined arguments provide flows of control which we want to synchronise. Since we assume knowledge of the history of development of the program we can check which arguments were intended to provide the flow of control and assess whether they will be compatible in combination. In this case it will observe (from the program history) that both programs have the same flow of control, called "search"[2], and only differ in that one has a count technique added, and the other an accumulator technique. We assume that the user would also like the flow of control of the output to be "search" and would simply like to have both the count and search techniques added. This gives the output program

```
p_path2([X|_],X,1,[X]).
p_path2([H|T],X,N,[H|R]) :-
        p_path2(T,X,NP,R),
        N is NP +1.
```

which will indeed only return the first two (desired) solutions and not the last two undesired solutions.

Let us look at this in a bit more detail. The search flow of control is

```
search([H|_],H) :- t1(H).
search([H|T],X):-
        t2(H),search(T,X).
```

where `t1/1` and `t2/1` are tests. The simplified versions of the program histories for the programs `pos/3` and `path/3` are

```
history(pos,  3, search, [[1,(true,true),no_test],[2,(search(T,X),pos(T,X,NP)),no_test]],
        count)
history(path, 3, search, [[1,(true,true),no_test],[2,(search(T,X),path(T,X,R)),no_test]],
        accumulator)
```

[2]So-named because it functions to search for a particular element

For each history, the first argument is the name of the program; the second is the arity; the third is the name of the initial control flow used in the construction of the program; the fourth argument is a list recording, for each clause, how the subgoals in the initial control flow were transformed; and the fifth is the technique used.

The program histories thus say that both programs were built starting from "search", that the test literals were trivially true, but that the techniques used were different. In this case, the composition system then selects the join–1–1 method that makes the combination by combining only corresponding clauses, rather than combining all pairs of clauses, thus achieving the desired synchronisation in the unpacking of the lists.

The basic results of this investigation were:

1. The classification in terms of control flows and techniques was a useful classification in that it allowed us to develop methods for each class in turn that were far more powerful, and required less user direction, that was possible for programs in general. This is not so surprising, after all, such classifications arose out of experts trying to reason about Prolog programs, and they would try to find useful reasoning methods.

2. Storing the program history is useful. It acts as a set of "high-level machine-readable comments" to the code. In this case these comments contained the information necessary to classify the program by the above classification, and hence select the correct combination method. These comments could also be used by the combination methods to further reduce the need for user interaction. In particular, the comments could be used to infer the users intentions in requesting the program combination, and so deal with cases of underspecification. So the use of the program history was a powerful way to render program combination more effective.

In [VVVR93, VVRI93] we describe our resulting set of methods, the corresponding program classification system and how these elements can be combined into an almost fully automatic program combination system. Of course, our set of methods is not complete but it does cover a wide range of combinations. A possible extension of this work would be to use these comments (in the program history) in order to aid a high-level dialogue with the user. The system could talk in the language of control flows, and their associated functionalities, instead of in terms of low-level guidance of the combination method.

We believe that many of these lessons should also apply to languages other than Prolog.

We would like to give acknowledgements to Andrew Parkes and Rolando Carrera for their useful comments on this paper.

References

[BD77] Rod Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal ACM*, 24(1):44–67, 1977.

[Bow93] Andy Bowles. A Techniques Editor for Prolog novices. Internal note submitted for publication, DAI, 1993.

[FF91] Norbert E. Fuchs and Markus P. J. Fromherz. Schema-Based Transformations of Logic Programs. In *Logic Program Synthesis and Transformation, Workshops in Computing*. Springer Verlag, 1991.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating Non-Interfering Versions of Programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[LS87] Arun Lakhotia and Leon Sterling. Composing Logic Programs with Clausal Join. Tr 87-25, Computer Engineering and Science Department, Case Western Reserve University, 1987.

[PP92] Maurizio Proietti and Alberto Pettorossi. Best-first Strategies for Incremental Transformations of Logic Programs. In *Second International Workshop on Logic Program Synthesis and Transformation*, 1992.

[Rob91] Dave Robertson. A Simple Prolog Techniques Editor for Novice Users. In G. A. Wiggins, C. Mellish, and T. Duncan, editors, *3rd UK Annual Conference on Logic Programming*, pages 190–205. Springer Verlag, April 1991.

[SL88] Leon Sterling and Arun Lakhotia. Composing Prolog Meta-Interpreters. In Kowalski and Bowen, editors, *5th Symposium of Logic Programming*, pages 386–403, 1988.

[TS83] Hisao Tamaki and Taisuke Sato. A Transformation System for Logic Programs which Preserves Equivalence. Tr 83-18, ICOT Research Center, 1983.

[TS84] Hisao Tamaki and Taisuke Sato. Unfold/Fold Transformation of Logic Programs. In *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Sweden, 1984.

[VV94] Maria Vargas-Vera. *Guidance during Program Composition in a Prolog Techniques Editor*. PhD thesis, Department of Artificial Intelligence, Edinburgh University, 1994. In preparation.

[VVRI93] Maria Vargas-Vera, Dave Robertson, and Robert Inder. Combining Prolog Programs in a Techniques Editing System. In *Third International Workshop on Logic Programming Synthesis and Transformation*. Springer Verlag, July 1993. Also published as DAI. Research Paper 636.

[VVVR93] Maria Vargas-Vera, Wamberto Vasconcelos, and Dave Robertson. Building Large-Scale Prolog Programs Using a Techniques Editing System. In *International Logic Programming Symposium*. The MIT Press, October 1993. Presented as a poster and also published as DAI. Research Paper 635.