MOST SPECIFIC LOGIC PROGRAMS

K. MARRIOTT, L. NAISH * and J.-L. LASSEZ

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, USA

Abstract

More specific versions of definite logic programs are introduced. These are versions of a program in which each clause is further instantiated or removed and which have an equivalent set of successful derivations to those of the original program, but a possibly increased set of finitely failed goals. They are better than the original program because failure in a non-successful derivation may be detected more quickly. Furthermore, information about allowed variable bindings which is hidden in the original program may be made explicit in a more specific version of it. This allows better static analysis of the program’s properties and may reveal errors in the original program. A program may have several more specific versions but there is always a most specific version which is unique up to variable renaming. Methods to calculate more specific versions are given and it is characterized when they give the most specific version.

Keywords: Logic programs, specific logic programs.

1. Introduction

People usually write programs which are as general and simple as possible. However for efficient execution it may be better to use a less general program. Here we investigate one way in which a logic program can be transformed into a more efficient, less general program called a more specific version of the program.

A more specific version of a program is obtained by replacing each clause in the program by an instance of the clause, or by removing the clause if it is not used in any successful derivations. The definition of more specific version has been chosen so that any successful derivation from the original program has a corresponding derivation from the more specific version of the same length and with the same answer. Consider the program $member$, $member(A, A \cdot B)$. $member(C, D \cdot E) ← member(C, E)$. From inspection, in any successful derivation for $member$ the second argument must be bound to a term of the form $X \cdot Y$. Thus in the second clause, $E$ must

* Department of Computer Science, University of Melbourne, Parkville 3052, Australia.

© J.C. Baltzer A.G. Scientific Publishing Company
eventually be bound to \( X \cdot Y \) in any successful derivation. It follows that a more specific version of \( \textit{member} \) is

\[
\textit{member}(A, A \cdot B).
\]

\[
\textit{member}(C, D \cdot X \cdot Y) \leftarrow \textit{member}(C, X \cdot Y).
\]

It follows from the definition that a more specific version of a program has the same set of positive consequences as the original program. Failure in a derivation may occur more quickly with the more specific version as each clause in the more specific version is an instance of a clause in the original program. For example, the goal \( \textit{member}(a, b \cdot []) \) will fail more quickly with the more specific version of \( \textit{member} \) than with the original. Thus more specific versions have the same size or smaller derivation trees than the original program and so, in this sense, they are more efficient.

Another advantage of a more specific version is that it makes explicit the information hidden in the original program about the successful variable bindings. This is important for two reasons. Firstly, it allows better static analysis of the program properties by other programs. Thus, for example, we would expect that derived wait [12], mode [10] or type [13,24] statements and determinacy [14] information will be more precise for the more specific version than for the original. Secondly, the more specific version may make visible errors hidden in the original program. An example of such an error might be a clause body which, because of a type mismatch, can never succeed or only succeeds with the empty list rather than with all lists.

The idea of specializing clauses in a program to give an equivalent program may be considered as a type of partial evaluation [3]. However, the ideas presented here are quite different from the standard partial evaluation technique suggested for Prolog in which bodies of clauses are repeatedly unfolded [5,20,23].

More specific clauses are related to the item set of Sato and Tamaki [17] in which the clause instances used in any successful derivation from a given goal are described with a finite set of terms of bounded depth. Sato and Tamaki use the item set to specialize the program in much the same way as we use the more specific version of the clause bodies. However, they specialize the program for a particular goal, do not have a “best” specialization and do not consider negation-as-failure.

In the next section of this paper we introduce some notation. In the third section more specific versions are formalized and their properties are discussed. It is shown that every program has a most specific version which is unique up to variable renaming. A more specific version of a program may have an increased set of finitely failed goals and so, when negation-as-failure is used, may behave differently from the original program. More specific versions which preserve infinite derivations are introduced and proved to have the same behaviour as the original program when negation-as-failure is used. Any program in which each clause is used in at least one successful derivation is shown to have a most