Chain Programs for Writing Deterministic Metainterpreters

DAVID A. ROSENBLUETH
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Universidad Nacional Autónoma de México
Apdo. 20-726, 01000 México D.F.

Abstract

Many metainterpreters found in the logic programming literature are *nondeterministic* in the sense that the selection of program clauses is not determined. Examples are the familiar "demo" and "vanilla" metainterpreters. For some applications this nondeterminism is convenient. In some cases, however, a *deterministic* metainterpreter, having an explicit selection of clauses, is needed. Such cases include (1) conversion of OR parallelism into AND parallelism for "committed-choice" processors, (2) logic-based, imperative-language implementation of search strategies, and (3) simulation of bounded-resource reasoning.

Deterministic metainterpreters are difficult to write because the variables in a logic program are local to each clause, and hence the programmer must be concerned about the set of unifiers of the children of a node in the derivation tree. We argue that it is both possible and advantageous to write these metainterpreters by reasoning in terms of object programs converted into a syntactically restricted form that we call "chain" form, where we can forget about unification, except for unit clauses. We give two transformations converting logic programs into chain form, one for "moded" programs (implicit in two existing exhaustive-traversal methods for committed-choice execution), and one for arbitrary definite programs. As illustrations of our approach we show examples of the three applications mentioned above.

1 Introduction

Perhaps the most common use of metalogic is the definition and implementation of metainterpreters (Safra and Shapiro 1986, Abramson and Rogers 1988, Kowalski 1990, Apt and Turini 1995). Many applications of metainterpreters are based on concise definitions, like that of the "vanilla" metainterpreter, which can be easily elaborated as required. Other applications, however, have been neglected, possibly because of employing convoluted definitions. Examples are deterministic metainterpreters exhaustively traversing search spaces. Our purpose will be to present a technique simplifying the design of deterministic metainterpreters. This technique converts the object program into a form severely restricted in its syntax, thereby facilitating reasoning about its search space.

Early works exploiting metainterpreters to great advantage are for example Bowen and Kowalski's amalgamation of language and metalanguage (Bowen and Kowalski 1982), Sergot's "query-the-user" facility (Sergot 1982), and Shapiro's "algorithmic debugger" (Shapiro 1982). These metainterpreters, just as the familiar vanilla and demo metainterpreters, have nondeterministic definitions. Consider for example the following *demo* predicate (Kowalski 1990, Kowalski 1995):

$$demo(T, P) \leftarrow axiom(T, P \leftarrow Q),\ demo(T, Q)$$
$$demo(T, P \wedge Q) \leftarrow demo(T, P),\ demo(T, Q)$$
$$demo(T, true) \leftarrow$$

This definition is nondeterministic because it is not determined, in the first clause of the definition, what axiom in the theory T, having conclusion P, might be needed to demonstrate P (Kowalski 1995, p. 229).

For some applications this nondeterminism is convenient. For others, however, a *deterministic* metainterpreter, having an explicit selection of axioms, is desired. A problem amenable to deterministic metainterpretation is that of exhaustively traversing, using a committed-choice processor, the search space generated by a logic program and a goal (Ueda 1987, Tamaki 1987). Another problem is that of describing search strategies with logic programs equivalent to flowcharts, like the programs of Clark and van Emden (Clark and van Emden 1981). Yet another application is Kowalski's approach for reconciling reactive and rational agents with bounded-resource metainterpreters (Kowalski 1995).

It is of course possible to write deterministic metainterpreters for logic programs. Clark and Gregory were perhaps among the first to publish (Clark and Gregory 1985) one such metainterpreter. We show a slightly modified version of their metainterpreter in Fig. 1. The intended meanings of some of the predicates in this metainterpreter are as follows. Assume that the set of answer substitutions to the goal having A as its only subgoal is $\{\theta_1, \ldots, \theta_n\}$. Then the predicate *single_call_set*$(A, A\theta s)$ is intended to hold when $A\theta s$ is a list of the form $[A\theta_1, \ldots, A\theta_n]$.

The predicate *set*$(A, Bs, A\theta s)$ is a generalization of *single_call_set*. In this case the goal Bs, which may have more than one subgoal, is of the form $B_1 \wedge \ldots \wedge B_m \wedge$ *true*, and $A\theta s$ is of the form $[A\theta_1, \ldots, A\theta_n]$, where $\{\theta_1, \ldots, \theta_n\}$ is the set of answer substitutions for Bs. (Bs terminates by *true* to simplify the code.)

The predicate *all_set*$(ABs, A\theta s)$, in turn, can be viewed as a generalization of *set*, where ABs is a list of clauses $[A_1 \leftarrow Bs_1, \ldots, A_m \leftarrow Bs_m]$ and $A\theta s$ is of the form $[A_1\theta_{1,1}, \ldots, A_1\theta_{1,n_1}, \ldots, A_m\theta_{m,1}, \ldots, A_m\theta_{m,n_m}]$, where $\{\theta_{i,1}, \ldots, \theta_{i,n_i}\}$ is the set of answer substitutions for Bs_i. We refer the reader to (Clark and Gregory 1985) for a thorough discussion of this metainterpreter.

As can be seen in Fig. 1, the programmer must be concerned about *the set of unifiers of the children of a node in the derivation tree* (cf. the *term_instances* predicate). The reason for this concern is that variables in a logic program are local to each clause. This is an additional difficulty, absent in usual nondeterministic metainterpreters.

The study of deterministic metainterpreters can be viewed as an attempt to narrow the "gap" between the "don't-know" form of logic programming, needed for user-level applications, and the "don't-care" form, useful for controlling the execution of logic programs. Kowalski's observation (Kowalski 1993) that the FGCS

Example of object-program representation:

$$
\begin{array}{l}
\mathit{definition}(ap(\_,\_,\_), [ap([\,], L, L) \leftarrow \mathit{true}, \\
\qquad\qquad ap([A|L'], M, [A|N]) \leftarrow (ap(L', M, N) \wedge \mathit{true}) \\
\qquad\qquad] \\
\quad) \leftarrow
\end{array}
$$

Example of goal: $\leftarrow \mathit{single\_call\_set}(ap(X, Y, [a, b]), A\theta s)$

Metainterpreter:

$$
\begin{array}{l}
\mathit{single\_call\_set}(A, A\theta s) \leftarrow \mathit{definition}(A, ABs), \\
\qquad \mathit{all\_matches}(A, ABs, AB\theta s'), \\
\qquad \mathit{all\_set}(AB\theta s', A\theta s) \\
\mathit{all\_set}([\,], [\,]) \leftarrow \\
\mathit{all\_set}([A \leftarrow Bs|ABs], A\theta s) \leftarrow \mathit{set}(A, Bs, A\theta s_1), \\
\qquad \mathit{all\_set}(ABs, A\theta s_2), \\
\qquad \mathit{append}(A\theta s_1, A\theta s_2, A\theta s) \\
\mathit{set}(A, \mathit{true}, [A]) \leftarrow \\
\mathit{set}(A, B \wedge Bs, A\theta s) \leftarrow \mathit{single\_call\_set}(B, B\theta s), \\
\qquad \mathit{term\_instances}(B, B\theta s, A \leftarrow Bs, AB\theta s), \\
\qquad \mathit{all\_set}(AB\theta s, A\theta s) \\
\mathit{all\_matches}(A, [\,], [\,]) \leftarrow \\
\mathit{all\_matches}(A, [A\theta \leftarrow B\theta s|ABs], [A\theta \leftarrow B\theta s|AB\theta s']) \leftarrow \mathit{copy}(A, A\theta'), \\
\qquad \mathit{unify}(A\theta, A\theta'),\ !, \\
\qquad \mathit{all\_matches}(A, ABs, AB\theta s') \\
\mathit{all\_matches}(A, [A' \leftarrow Bs|ABs], ABs') \leftarrow \mathit{all\_matches}(A, ABs, ABs') \\
\mathit{term\_instances}(B, [\,], C, [\,]) \leftarrow \\
\mathit{term\_instances}(B, [B\theta|B\theta s], C, [C\theta|C\theta s]) \leftarrow \mathit{copy}(f(B, C), f(B\theta', C\theta)), \\
\qquad \mathit{unify}(B\theta, B\theta'), \\
\qquad \mathit{term\_instances}(B, B\theta s, C, C\theta s)
\end{array}
$$

Figure 1. *A deterministic, exhaustive-traversal metainterpreter for arbitrary definite programs. For readability, we use a string $X\theta$ as the name of a variable taking as value an instance of the value of X. Similarly, $X\theta s$ is the name of a variable taking as value a list of instances of the value of X.*

project had experienced such a gap suggests looking at the work done in connection with this project.

First Ueda (Ueda 1987), and then Tamaki (Tamaki 1987), published methods converting a nondeterministic logic program into a deterministic version. The motivation for developing such methods was that of allowing the execution of OR-parallel programs by committed-choice processors (which are AND parallel). Understanding how these methods work might lead to key ideas for obtaining other deterministic-evaluation methods. However, the considerable intricacy of these methods is an obstacle for giving a clear and concise explanation of their central mechanisms.

Trying to elucidate the principles on which these methods are based, we found an important common characteristic. Both tend to hide, as it were, certain occurrences of variables, in a list behaving like a stack. In particular, such occurrences are those of variables that (1) receive a substitution *before* a subgoal A is selected and (2) occur in a subgoal selected *after* A has succeeded. Variables having such occurrences are called "pass-on" variables in (Tamaki 1987). Hence, *if we consider programs lacking pass-on variables, a fortiori, these transformations get simplified.*

A kind of program lacking pass-on variables is that of "chain" programs, having clauses of the form:

$$p(X_0, X_n) \leftarrow q_1(X_0, X_1),\ q_2(X_1, X_2),\ \ldots,\ q_n(X_{n-1}, X_n)$$

(as well as other clauses; we formally define chain programs in Sect. 2). Apparently, concentrating on chain programs should simplify the task of devising deterministic-traversal methods in general, and defining deterministic metainterpreters in particular. We confirm this possibility by deriving in Sect. 2 a deterministic metainterpreter by reasoning in terms of relational union and composition.

This metainterpreter is more useful once we provide ways of transforming logic programs that do not have chain form into such a form. The methods of (Ueda 1987) and (Tamaki 1987) can handle "moded" programs, that do not necessarily have chain form. In these programs, each argument place of each predicate is used either as input (instantiated) or as output (uninstantiated). By comparing the original methods with their versions simplified to chain programs, we have uncovered a transformation converting moded programs into chain form. In Sect. 3 we give such a transformation, which is in fact implicit in (Ueda 1987) and (Tamaki 1987). As in the methods of Ueda and Tamaki, we *hide pass-on variables in a list behaving like a stack.* (We have previously used this transformation for adapting parsers for context-free languages obtaining inference systems for moded logic programs (Rosenblueth 1996, Rosenblueth and Peralta 1998).)

Next, in Sect. 4 we give another transformation, converting arbitrary definite programs into chain form, inspired by the previous one.

Once we have this more general transformation, we can easily extend, in Sect. 5, the existing methods for deterministic, exhaustive traversal (Ueda 1987, Tamaki 1987) to handle arbitrary definite programs.

Section 6 shows how to write deterministic metainterpreters for other applications. First we exhibit a metainterpreter having the same behaviour as that of Prolog systems. Next we give a bounded-resource metainterpreter.

We will assume some familiarity with logic program transformation through the unfold/fold rules (Pettorossi and Proietti 1994).

Code appearing throughout the sequel, as well as further examples of programs converted into chain form, may be found at:

`http://leibniz.iimas.unam.mx/~drosenbl/detmeta`.

2 A Deterministic, Exhaustive-Traversal Metainterpreter for Chain Programs

In this section, we will write a deterministic, exhaustive-traversal metainterpreter for chain programs by reasoning in terms of relational union and composition. A chain program can be viewed as defining a system of equations of relational expressions. Such systems of relational equations have been studied for example in (Engelfriet 1974, de Bakker and Meertens 1975, Blikle 1977). Our translation of chain programs into relational equations enables us to regard logical inference from such programs as the evaluation of relational expressions built from union and composition. Our chain programs are similar to, but different from, certain programs occurring in the deductive-database literature under the same name; a difference is that we allow function symbols other than constants.

For clarity, we will discuss now this metainterpreter assuming that only ground terms are constructed. However, in Sect. 4 we will observe that with the addition of variable renaming and full unification (as opposed to matching, i.e. one-way unification), this metainterpreter is also valid for constructing terms with variables.

2.1 Chain programs as systems of relational equations

We define a *chain program* as a logic program consisting only of clauses of the form:

$$p(X_0, X_n) \leftarrow q_1(X_0, X_1),\ q_2(X_1, X_2),\ \ldots,\ q_n(X_{n-1}, X_n) \quad n > 0 \tag{1}$$

$$\text{or } p(t, t') \leftarrow \tag{2}$$

where the X_i's are distinct variables, and t and t' are any term. The first argument place of a predicate will be called its *input*, and the second argument place its *output*. It will be useful to single out a kind of chain program where all answers for such a program and a goal with a leftmost ground input are ground, assuming a leftmost computation rule. If $\mathrm{var}(t') \subseteq \mathrm{var}(t)$ in every clause of the form (2), then the program is called a *G-chain program*. Here, and throughout the sequel, we use $\mathrm{var}(t)$ to denote the set of variables in expression t.

Clearly, the clause (1) denotes the inclusion:

$$P \supseteq (Q_1\ ;\ Q_2\ ;\ \ldots\ ;\ Q_n) \tag{3}$$

where P, Q_i name the relations denoted by p, q_i, respectively, and ";" denotes relational composition.[1]

Let us define

$$P_j \ \ddot{=}\ Q_1\ ;\ Q_2\ ;\ \ldots\ ;\ Q_n$$

if the j-th clause defining the predicate with symbol p has the form (1). If, on the other hand, the j-th clause defining the predicate with symbol p has the form (2), then

$$P_j \ \ddot{=}\ \{(\mathbf{x}, \mathbf{x}') : p(\mathbf{x}, \mathbf{x}') \leftarrow \text{ is a ground instance of } p(t, t') \leftarrow\}$$

[1] The composition of relations P and Q is defined as: $P\ ;\ Q \ \ddot{=}\ \{(x, z) : \exists y[(x, y) \in P\ \&\ (y, z) \in Q]\}$.

Hence, a chain program denotes a system of relational expressions having an inclusion of the form

$$P \supseteq (P_1 \cup P_2 \cup \ldots \cup P_m)$$

for each largest set of clauses defining a predicate with symbol p. Just as sometimes the meaning of a logic program is defined as its least Herbrand model, here we are interested in the least solution of this system. It can be shown that such a solution is equal to the unique solution of the system obtained by replacing the inclusions by equalities.

For notational convenience, we will extend relational composition to the case where the first argument is a set. Let $S \subseteq D$ and $R \subseteq D \times D$.

$$S \,;\, R \;\ddot{=}\; \{z : \exists y[y \in S \;\&\; (y, z) \in R]\}$$

i.e. $S \,;\, R$ denotes the image of S under R. Also, we will sometimes omit the curly braces in singletons. By I we will denote the identity relation and by $\emptyset$ the empty relation as well as the empty set.

Given an object chain program, we will use X as a metavariable taking as value a ground term, Xs a set of ground terms, Q a relation denoted by an (object-level) predicate, Qs compositions $Q_1 \,;\, \ldots \,;\, Q_n$ of relations denoted by predicates, Pj a relation denoted by a single clause, and Pjs unions $P_1 \cup \ldots \cup P_m$ of relations denoted each by a single clause.

Computing all answers for a chain program and a goal $\leftarrow q(\mathbf{x}, Z)$, where $\mathbf{x}$ is a ground term, translates to evaluating the expression $\mathbf{x} \,;\, Q$, where Q names the relation denoted by q. During this evaluation, we will have to evaluate relational expressions of the form:

$$Xs \,;\, Qs \tag{4}$$

which represent AND branches of the SLD tree.

2.2 A metainterpreter as a relational-expression evaluator

Let us now establish an object-program representation. For simplicity, we use the ambivalent syntax of (Jiang 1994). If the j-th clause defining a predicate with symbol p is of the form (1) we will have the following clause at the metalevel:

$$\mathit{nonunit}(p_j, (q_1 \,;\, q_2 \,;\, \ldots \,;\, q_n \,;\, I)) \leftarrow$$

where ";" can be interpreted as a right-associative infix list constructor and I as a constant. If the j-th clause defining a predicate with symbol p is of the form (2) we will have:

$$\mathit{is\_unit}(p_j) \leftarrow \qquad \text{and}$$
$$\mathit{unit}(p_j, t, t') \leftarrow$$

(The *is_unit* predicate is clearly unnecessary, as could be defined using the *unit* predicate, but we will use it for readability.)

Also, for each largest set of clauses defining a predicate with symbol p we will have:

$$defn(p, (p_1 \cup p_2 \cup \ldots \cup p_m \cup \emptyset)) \leftarrow$$

where $\cup$ can be interpreted as a right-associative infix list constructor denoting set union and $\emptyset$ as a constant.

In general, we could have two main evaluation strategies for (4): "termwise" (i.e. decomposing Xs) and "relationwise" (i.e. decomposing Qs). Here we will concentrate on a termwise definition (but see (Rosenblueth 1998) for an application of a metainterpreter using a relationwise definition).

Consider (4). The predicate $a(Xs, Qs, Zs)$ is intended to hold if Zs is $Xs; Qs$. Then the next two clauses, which constitute a termwise definition of this predicate, follow from the distributivity of composition over union:

$$\begin{aligned}
&a(\emptyset, Qs, \emptyset) \leftarrow \\
&a(\{X\} \cup Xs, Qs, YsZs) \leftarrow a'(X, Qs, Ys),\ a(Xs \setminus \{X\}, Qs, Zs), \\
&\qquad\qquad\qquad\qquad\qquad\quad union(Ys, Zs, YsZs)
\end{aligned}$$

where $a'(X, Qs, Ys)$ is meant to hold when Ys is $X\ ;\ Qs$ and $union(Ys, Zs, YsZs)$ is meant to hold when $YsZs$ is $Ys \cup Zs$.

We can decompose Qs in the definition of a', which follows from the definition of composition:

$$\begin{aligned}
&a'(X, I, \{X\}) \leftarrow \\
&a'(X, (Q\ ;\ Qs), Zs) \leftarrow b'(X, Q, Ys),\ a(Ys, Qs, Zs)
\end{aligned}$$

where $b'(X, Q, Ys)$ represents the composition Ys of a single term X with a single relation Q.

The following clause translates such a composition into the composition of X with a union Pjs of relations:

$$b'(X, Q, Ys) \leftarrow defn(Q, Pjs),\ c'(X, Pjs, Ys)$$

where $c'(X, Pjs, Ys)$ is intended to hold if Ys is $X\ ;\ Pjs$.

Now we inductively define c' by decomposing Pjs. This definition of c', as that of a, follows from the distributivity of composition over union:

$$\begin{aligned}
&c'(X, \emptyset, \emptyset) \leftarrow \\
&c'(X, \{Pj\} \cup Pjs, YsZs) \leftarrow d'(X, Pj, Ys),\ c'(X, Pjs \setminus \{Pj\}, Zs), \\
&\qquad\qquad\qquad\qquad\qquad\qquad union(Ys, Zs, YsZs)
\end{aligned}$$

where the predicate $d'(X, Pj, Ys)$ is assumed to hold when Ys is $X\ ;\ Pj$, and the predicate $union(Ys, Zs, YsZs)$ is assumed to hold when $YsZs$ is $Ys \cup Zs$.

Next we write a definition of $d'(X, Pj, Ys)$. This definition uses an auxiliary predicate $e'(X, Pj, Ys)$ in case Pj represents a unit clause. If, on the other hand, Pj represents a nonunit clause, then the d' predicate uses the (object-level) definition of Pj to translate $X \; ; \, Pj$ into $X \; ; \, Qs$, so that the previously defined predicate $a'(X, Qs, Zs)$ can be used.

$$d'(X, Pj, Ys) \leftarrow \mathit{is\_unit}(Pj),\ \ e'(X, Pj, Ys)$$
$$d'(X, Pj, Zs) \leftarrow \mathit{nonunit}(Pj, Qs),\ \ a'(X, Qs, Zs)$$

It only remains to define $e'(X, Pj, Ys)$, intended to hold if Ys is $X \; ; \, Pj$ and Pj represents a unit clause.

$$e'(X, Pj, \{Y\}) \leftarrow \mathit{unit}(Pj, X, Y) \tag{5}$$
$$e'(X, Pj, \emptyset) \leftarrow \mathit{not}(\mathit{unit}(Pj, X, \_)) \tag{6}$$

The clause (5) covers the case where X matches the input of the unit clause named Pj, whereas the clause (6) covers the case where X does not match the input of the unit clause named Pj.

Finally, we give the complete metainterpreter in a more standard Prolog notation. In addition, we have approximated set union with list concatenation. (A more efficiently executable metainterpreter would use difference lists, but for clarity we prefer ordinary lists.) We will call this the *abcde* metainterpreter.

$$\begin{aligned} &a([\,], Qs, [\,]) \leftarrow \\ &a([X|Xs], Qs, YsZs) \leftarrow a'(X, Qs, Ys),\ \ a(Xs, Qs, Zs), \\ &\qquad\qquad\qquad\qquad\quad \mathit{append}(Ys, Zs, YsZs) \end{aligned} \tag{7}$$

$$\begin{aligned} &a'(X, [\,], [X]) \leftarrow \\ &a'(X, [Q|Qs], Zs) \leftarrow b'(X, Q, Ys),\ \ a(Ys, Qs, Zs) \end{aligned} \tag{8}$$

$$b'(X, Q, Ys) \leftarrow \mathit{defn}(Q, Pjs),\ \ c'(X, Pjs, Ys)$$

$$\begin{aligned} &c'(X, [\,], [\,]) \leftarrow \\ &c'(X, [Pj|Pjs], YsZs) \leftarrow d'(X, Pj, Ys),\ \ c'(X, Pjs, Zs), \\ &\qquad\qquad\qquad\qquad\quad \mathit{append}(Ys, Zs, YsZs) \end{aligned} \tag{9}$$

$$\begin{aligned} &d'(X, Pj, Ys) \leftarrow \mathit{is\_unit}(Pj),\ \ e'(X, Pj, Ys) \\ &d'(X, Pj, Zs) \leftarrow \mathit{nonunit}(Pj, Qs),\ \ a'(X, Qs, Zs) \end{aligned} \tag{10}$$

$$\begin{aligned} &e'(X, Pj, [Y]) \leftarrow \mathit{unit}(Pj, X, Y) \\ &e'(X, Pj, [\,]) \leftarrow \mathit{not}(\mathit{unit}(Pj, X, \_)) \end{aligned}$$

3 Conversion of Moded Programs into Chain Form

Having written a deterministic metainterpreter for chain programs, our aim now is to develop a transformation converting "moded" programs (Apt 1997) into chain form. This transformation is in fact implicit in two existing methods for deterministic, exhaustive traversal: the continuation-based (Ueda 1987) and the stream-based (Tamaki 1987) methods. We have previously used such a transformation for adapting parsers for context-free grammars obtaining inference systems for moded logic programs (Rosenblueth 1996, Rosenblueth and Peralta 1998).

A clause

$$p(t_0, t'_n) \leftarrow q_1(t'_0, t_1),\ q_2(t'_1, t_2),\ \ldots,\ q_n(t'_{n-1}, t_n) \qquad n \geq 0 \tag{11}$$

is called *moded* if:

1. $\mathrm{var}(t'_i) \subseteq \mathrm{var}(t_0) \cup \cdots \cup \mathrm{var}(t_i)$, for $i = 0, \ldots, n$; and
2. $\mathrm{var}(t_i) \cap \mathrm{var}(t_j) = \emptyset$, for $i, j = 0, \ldots, n$ and $i \neq j$.

A program is called *moded* if it consists only of moded clauses.

Condition 1 causes every input to be ground if the input of the initial goal is also ground and we use a leftmost computation rule. When a subgoal succeeds, condition 2 causes the constructed term to have an effect only on the input of other subgoals, thus avoiding speculative bindings. In (Rosenblueth 1996, Rosenblueth and Peralta 1998) we had a third condition (that each variable occurring in t'_i occurs only once in t'_i, for $i = 0, \ldots, n$, if $n > 0$) meant only for simplifying the transformation. However, as we observe below, it is possible to eliminate such a condition without excessively elaborating the transformation.

We will use the *standard equality theory*. Given a program P, this theory consists of the following axioms:

$$\begin{aligned}
&X = X \leftarrow \\
&X = Y \leftarrow Y = X \\
&X = Z \leftarrow X = Y,\ Y = Z \\
&\{f(X_1, \ldots, X_{n_f}) = f(Y_1, \ldots, Y_{n_f}) \leftarrow X_1 = Y_1,\ \ldots,\ X_{n_f} = Y_{n_f} : \\
&\qquad\qquad\qquad\qquad\qquad\qquad f \text{ is a function symbol occurring in } P\} \\
&\{p(X_1, \ldots, X_{n_p}) \leftarrow X_1 = Y_1,\ \ldots,\ X_{n_p} = Y_{n_p},\ p(Y_1, \ldots, Y_{n_p}) : \\
&\qquad\qquad\qquad\qquad\qquad\qquad p \text{ is a predicate symbol occurring in } P\}
\end{aligned}$$

which are called, reflexivity, symmetry, transitivity, function substitutivity, and predicate substitutivity, respectively.

One way of obtaining a chain clause from a moded clause would be to resolve first the moded clause with predicate substitutivity so as to replace each argument of a subgoal by a variable and then to fold the resulting clause using some new predicates so as to remove the introduced equations. We will see, however, that such a folding operation may not always be sound (Gardner and Shepherdson 1992, Tamaki and Sato 1984).

Consider the following *append* program used for splitting lists.

$$s(\langle L \rangle, \langle [\,], L \rangle) \leftarrow \tag{12}$$

$$\underbrace{s}_{p}\,(\underbrace{\langle[A|N]\rangle}_{t_0},\underbrace{\langle[A|L],M\rangle}_{t_1'}) \leftarrow \underbrace{s}_{q_1}\,(\underbrace{\langle N\rangle}_{t_0'},\underbrace{\langle L,M\rangle}_{t_1}) \tag{13}$$

We employ angled brackets $\langle\ \rangle$ instead of ordinary brackets $[\ ]$ for grouping input and output arguments. We do this for clarity.

By predicate substitutivity and symmetry, it is possible to derive from (13):

$$\begin{aligned} s(X,Y) \leftarrow\ & \boxed{X = \langle[A|N]\rangle},\ Y = \langle[A|L],M\rangle,\ \boxed{X' = \langle N\rangle},\ Y' = \langle L,M\rangle,\\ & s(X',Y') \end{aligned} \tag{14}$$

Next, we could fold (14) using the following definitions:

$$\begin{aligned} &\mathit{naive\_h}_0(X,X') \leftrightarrow \exists A\,\exists N\,(X = \langle[A|N]\rangle\ \&\ X' = \langle N\rangle)\\ &\mathit{naive\_h}_1(Y',Y) \leftrightarrow \exists A\,\exists L\,\exists M\,(Y' = \langle L,M\rangle\ \&\ Y = \langle[A|L],M\rangle) \end{aligned}$$

In the case of $\mathit{naive\_h}_0$, this folding operation would replace the subgoals enclosed in rectangles by the definiendum $\mathit{naive\_h}_0(X,X')$. However, as observed in (Tamaki and Sato 1984), in general it is incorrect to fold a clause such as (14) using a definition such as that of $\mathit{naive\_h}_0$, where a variable like A: (a) appears in atoms replaced by the definiendum (i.e. $X = \langle[A|N]\rangle$), (b) appears in atoms not replaced by the definiendum (i.e. $Y = \langle[A|L],M\rangle$), and (c) does not unify with any variable appearing in the definiendum. To see this incorrectness, fold and subsequently unfold (14) with $\mathit{naive\_h}_0$, and a generalization of (14) is obtained.

Note that unlike A in (13), N (which does not cause the incorrectness of Tamaki and Sato) occurs only in t_0 and t_0'. This suggests that a minimal strengthening of the syntactic conditions defining moded clauses so as to avoid the situation above would be requiring that all variables of t_i' occur in t_i (for $i = 0,\ldots,n$). Hence, we define a clause in *prechain form* as:

$$\hat{p}(s_0,s_n') \leftarrow \hat{q}_1(s_0',s_1),\ \hat{q}_2(s_1',s_2),\ \ldots,\ \hat{q}_n(s_{n-1}',s_n) \qquad n \geq 0$$

where:

1. $\mathrm{var}(s_i') \subseteq \mathrm{var}(s_i)$ for $i = 0,\ldots,n$; and
2. $\mathrm{var}(s_i) \cap \mathrm{var}(s_j) = \emptyset$, for $i,j = 0,\ldots,n$ and $i \neq j$.

From a clause in prechain form it is possible to arrive at chain form by first applying predicate substitutivity and then folding with the completed definitions of predicates of the form:

$$h_i(s_i,s_i') \leftarrow$$

Such foldings satisfy also Gardner and Shepherdson's stronger condition (Gardner and Shepherdson 1992) for folding.

We face now the problem of converting a moded clause into prechain form. Recall first the pass-on variables of the deterministic, exhaustive-traversal methods (Ueda 1987, Tamaki 1987), i.e. variables that receive a substitution before a subgoal B is selected and occur in a subgoal selected after B has succeeded. Observe next that prechain form clauses lack pass-on variables. Finally, note that the fact that these methods use a stack to hide pass-on variables, suggests that *we also use a stack to achieve prechain form.*

Since both chain form and prechain form lack pass-on variables, we could have chosen prechain form for the object programs of our deterministic metainterpreter. The resulting metainterpreter, however, would not have been as concise.

In converting a moded clause into prechain form, we will often apply the following sequence of unfolding steps, which we group in a lemma.

Lemma 1 (*Equation introduction*)
Let C be a definite clause having an occurrence of a variable X. Then the clause obtained by replacing that occurrence of X by X' and adding the equation $X = X'$ is logically implied by C and the standard equality theory for C.

Proof
First, we resolve C with predicate substitutivity as follows. If the occurrence of X is in the head of C then we select the subgoal of predicate substitutivity which is not an equation. Next, we apply symmetry to all equations of the resolvent. The resulting clause has the form:

$$p(Y_1, \ldots, Y_d) \leftarrow t_1 = Y_1,\ \ldots,\ t_d = Y_d,\ A_1,\ \ldots,\ A_n$$

If, on the other hand, the occurrence of X is in a subgoal A_i of C, then we select such a subgoal. The resulting clause has the form:

$$A_0 \leftarrow A_1,\ \ldots,\ A_{i-1},\ t_1 = Y_1,\ \ldots,\ t_d = Y_d,\ p(Y_1, \ldots, Y_d),\ A_{i+1},\ \ldots,\ A_n$$

In either case, the occurrence of X is in some equation $t_j = Y_j$.

Next, we apply function substitutivity to $t_j = Y_j$ as many times as it is necessary to make such an occurrence appear at the top level of an equation $X = X'$.

Finally, we apply reflexivity to all equations except $X = X'$, thus disposing of all unwanted equations. The resulting clause has the claimed form. Hence the lemma holds. $\square$

Often, we will use equation introduction followed by symmetry, to which we will also refer as equation introduction.

Example 1
Let us convert (13) first into prechain form and then into chain form. To achieve prechain form, we will use an auxiliary stack, which we introduce in the following predicate:

$$\hat{s}(\langle St, X\rangle, \langle St, Y, Z\rangle) \leftarrow s(\langle X\rangle, \langle Y, Z\rangle)$$

We first apply equation introduction to the definition of $\hat{s}$:

$$\hat{s}(\langle St, X\rangle, \langle St', Y, Z\rangle) \leftarrow St = St',\ \underline{s(\langle X\rangle, \langle Y, Z\rangle)} \tag{15}$$

Next, we apply equation introduction to (13).

$$\underline{s(\langle [A|N]\rangle, \langle [A'|L], M\rangle)} \leftarrow A = A',\ s(\langle N\rangle, \langle L, M\rangle) \tag{16}$$

Subsequently, we resolve (15) and (16) unifying the two underlined atoms.

$$\hat{s}(\langle St,[A|N]\rangle,\langle St',[A'|L],M\rangle) \leftarrow \boxed{St = St'},\ \boxed{A = A'},\ s(\langle N\rangle,\langle L,M\rangle)$$

Next, we fold using function substitutivity.

$$\hat{s}(\langle St,[A|N]\rangle,\langle St',[A'|L],M\rangle) \leftarrow \boxed{[A|St] = [A'|St']},\ \boxed{s(\langle N\rangle,\langle L,M\rangle)}$$

Finally, we fold using (15) and systematically rename variables.

$$\hat{s}(\langle St_0,[A_0|N_0]\rangle,\langle St_1,[A_1|L_1],M_1\rangle) \leftarrow \hat{s}(\langle [A_0|St_0],N_0\rangle,\langle [A_1|St_1],L_1,M_1\rangle)$$

Once we have prechain form, we can apply predicate substitutivity and symmetry, and then fold w.r.t. the completed definitions of:

$$h_0(\langle St_0,[A_0|N_0]\rangle,\langle [A_0|St_0],N_0\rangle) \leftarrow$$
$$h_1(\langle [A_1|St_1],L_1,M_1\rangle,\langle St_1,[A_1|L_1],M_1\rangle) \leftarrow$$

arriving at:

$$\hat{s}(X_0,X_3) \leftarrow h_0(X_0,X_1),\ \hat{s}(X_1,X_2),\ h_1(X_2,X_3) \tag{17}$$

End of example

In (Rosenblueth 1996, Rosenblueth and Peralta 1998) we required that each variable occurring in t_i' occurs only once in t_i'. Note that if there is more than one occurrence of a variable in t_1', like in:

$$r(\langle [A|N]\rangle,\langle [A|L],M,\boxed{A}\rangle) \leftarrow s(\langle N\rangle,\langle L,M\rangle)$$

thus violating such a condition, we would have an extra equation after applying equation introduction twice:

$$r(\langle [A|N]\rangle,\langle [A'|L],M,A''\rangle) \leftarrow A = A',\ \boxed{A = A''},\ s(\langle N\rangle,\langle L,M\rangle)$$

Note that we do not wish to eliminate $A = A''$ with reflexivity, since we would obtain a clause with a variable occurring both in the input and the output of the head, preventing us from folding the h's. However, equations such as this one can be eliminated by *factoring* all equations with the same left-hand side. Hence we have withdrawn such a condition.

This derivation suggests a proof of a theorem relating a moded clause with its chain form.

Theorem 1
Let C be a moded clause:

$$p(t_0,t_n') \leftarrow q_1(t_0',t_1),\ q_2(t_1',t_2),\ \ldots,\ q_n(t_{n-1}',t_n) \qquad n \geq 0$$

and let

$$\Pi_j = \big(\mathrm{var}(t_0) \cup \cdots \cup \mathrm{var}(t_{j-1})\big) \cap \big(\mathrm{var}(t_j') \cup \cdots \cup \mathrm{var}(t_n')\big) \qquad j = 0,\ldots,n+1$$

Then the clause $\hat{C}$:

$$\hat{p}(U_0, U'_n) \leftarrow h_0(U_0, U'_0),\ \hat{q}_1(U'_0, U_1),\ h_1(U_1, U'_1),\ \hat{q}_2(U'_1, U_2),\ \ldots,\\ \hat{q}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

is logically implied by C, the standard equality theory for C, the "iff" version of the function substitutivity axiom for the list-constructor function symbol:

$$[X|Y] = [X'|Y'] \leftrightarrow X = X' \ \&\ Y = Y'$$

and the completed definitions of:

$$\begin{array}{ll} \hat{p}(\langle St|X\rangle, \langle St|Y\rangle) \leftarrow p(X, Y) & \\ \hat{q}_j(\langle St|X\rangle, \langle St|Y\rangle) \leftarrow q_j(X, Y) & j = 1, \ldots, n \\ h_j(\langle \Sigma_j|t_j\rangle, \langle \Sigma_{j+1}|t'_j\rangle) \leftarrow & j = 0, \ldots, n \end{array}$$

where Σ_j is any list of the form $[X_{1,j}, \ldots, X_{d_j,j}|St]$, such that $\{X_{1,j}, \ldots, X_{d_j,j}\} = \Pi_j$, if $\Pi_j \neq \emptyset$, and Σ_j is St if $\Pi_j = \emptyset$, for $j = 0, \ldots, n+1$, and the h_j's are predicate symbols not occurring in C.

Proof
First, we apply equation introduction to the definitions of $\hat{p}$ and the $\hat{q}_i$'s.

$$\hat{p}(\langle St|X\rangle, \langle St'|Y\rangle) \leftarrow St = St',\ p(X, Y) \tag{18}$$

$$\hat{q}_j(\langle St|X\rangle, \langle St'|Y\rangle) \leftarrow St = St',\ q_j(X, Y) \qquad j = 1, \ldots, n \tag{19}$$

Let σ_i, $i = 0, \ldots, n$, be renaming substitutions (Lloyd 1987, p. 22) for C such that:

$$\sigma_i \ \ddot{=}\ \{X/Z : X \in \mathrm{var}(C)\ \&\ Z \notin \big(\mathrm{var}(C) \cup \bigcup_{j \neq i} \mathrm{rhs}(\sigma_j)\big)\}$$

where $\mathrm{rhs}(\sigma_j) = \{Z_1, \ldots, Z_k\}$ if $\sigma_j = \{X_1/Z_1, \ldots, X_k/Z_k\}$.

Since the variables in $\mathrm{rhs}(\sigma_i)$ do not occur in C or in any other σ_j $(i \neq j)$, the application of σ_i to a variable X renames X uniquely. Hence, we can think of such an application as the addition of the subscript i to X.

Next, we apply equation introduction to C and rename variables, obtaining:

$$p(t_0\sigma_0, t'_n\sigma_n) \leftarrow E,\ q_1(t'_0\sigma_0, t_1\sigma_1),\ q_2(t'_1\sigma_1, t_2\sigma_2),\ \ldots,\ q_n(t'_{n-1}\sigma_{n-1}, t_n\sigma_n)$$

where

$$E = \{X\sigma_i = X\sigma_k : X \in \big(\mathrm{var}(t_i) \cap \mathrm{var}(t'_k)\big)\ \&\ i < k\}$$

Subsequently, we resolve with (18) and rename St and St'.

$$\hat{p}(\langle St_0|t_0\sigma_0\rangle, \langle St_n|t'_n\sigma_n\rangle) \leftarrow St_0 = St_n,\ E,\\ q_1(t'_0\sigma_0, t_1\sigma_1),\ q_2(t'_1\sigma_1, t_2\sigma_2),\ \ldots,\ q_n(t'_{n-1}\sigma_{n-1}, t_n\sigma_n)$$

Let $\Pi \ \ddot{=}\ \bigcup_j \Pi_j$ and $X \in \Pi$ (i.e. X occurs in some (and only one) t_i and some t'_k, for $i < k$). We now define

$$\begin{aligned} \phi(X) &= i, \text{ where } X \in \mathrm{var}(t_i) \\ \gamma(X) &= \max\{k : X \in \mathrm{var}(t'_k)\} \end{aligned}$$

and apply transitivity and factoring, replacing the equations by (up to variable renaming):

$$\{St_0 = St_1,\ St_1 = St_2,\ \ldots,\ St_{n-1} = St_n\} \cup \bigcup_{X \in \Pi} E_X$$

where

$$E_X = \{X\sigma_{\phi(X)} = X\sigma_{\phi(X)+1},\ X\sigma_{\phi(X)+1} = X\sigma_{\phi(X)+2},\ \ldots,\ X\sigma_{\gamma(X)-1} = X\sigma_{\gamma(X)}\}$$

Note that for all j and all X:

$$X \in \big(\mathrm{var}(t_i) \cap \mathrm{var}(t'_k)\big) \text{ for some } i \leq j-1 \text{ and some } k \geq j$$

iff

$$(X\sigma_{j-1} = X\sigma_j) \in E_X$$

Equivalently, for all j and all X:

$$X \in \bigcup_{i \leq (j-1),\ k \geq j} \big(\mathrm{var}(t_i) \cap \mathrm{var}(t'_k)\big) \text{ iff } (X\sigma_{j-1} = X\sigma_j) \in E_X$$

Hence,

$$\Pi_j = \{X : (X\sigma_{j-1} = X\sigma_j) \in E_X\}$$

for $j = 0, \ldots, n+1$, so that we have exactly all equations for constructing the Σ_j's.

Next, we repetitively fold using function substitutivity, and arrive at:

$$\begin{aligned}\hat{p}(\langle St_0|t_0\theta_0\rangle, \langle St_n|t'_n\theta_n\rangle) \leftarrow{}& \Sigma_1\theta_0 = \Sigma_1\theta_1,\ \Sigma_2\theta_1 = \Sigma_2\theta_2,\ \ldots,\ \Sigma_n\theta_{n-1} = \Sigma_n\theta_n,\\ & q_1(t'_0\theta_0, t_1\theta_1),\ q_2(t'_1\theta_1, t_2\theta_2),\ \ldots,\ q_n(t'_{n-1}\theta_{n-1}, t_n\theta_n)\end{aligned}$$

Prechain form is now obtained by folding with (19):

$$\begin{aligned}\hat{p}(\langle St_0|t_0\theta_0\rangle, \langle St_n|t'_n\theta_n\rangle) \leftarrow{}& \hat{q}_1(\langle \Sigma_1\theta_0|t'_0\theta_0\rangle, \langle \Sigma_1\theta_1|t_1\theta_1\rangle),\\ & \hat{q}_2(\langle \Sigma_2\theta_1|t'_1\theta_1\rangle, \langle \Sigma_2\theta_2|t_2\theta_2\rangle),\ \ldots,\\ & \hat{q}_n(\langle \Sigma_n\theta_{n-1}|t'_{n-1}\theta_{n-1}\rangle, \langle \Sigma_n\theta_n|t_n\theta_n\rangle)\end{aligned}$$

Finally, we apply predicate substitutivity and symmetry, and then fold with the completed definitions of the h_i predicates. The resulting clause has the desired form; hence we conclude that the theorem holds. □

Example 2
Let us apply Theorem 1 to the clause (13), of Example 1.

$$\underbrace{s}_{p}\,(\underbrace{\langle [A|N]\rangle}_{t_0}, \underbrace{\langle [A|L], M\rangle}_{t'_1}) \leftarrow \underbrace{s}_{q_1}\,(\underbrace{\langle N\rangle}_{t'_0}, \underbrace{\langle L, M\rangle}_{t_1})$$

$$\begin{array}{lcccl}\Pi_0 & = & \emptyset \ \cap \ \big(\mathrm{var}(t'_0) \cup \mathrm{var}(t'_1)\big) & = & \emptyset\\ \Pi_1 & = & \mathrm{var}(t_0) \ \cap \ \mathrm{var}(t'_1) & = & \{A\}\\ \Pi_2 & = & \big(\mathrm{var}(t_0) \cup \mathrm{var}(t_1)\big) \ \cap \ \emptyset & = & \emptyset\end{array}$$

So, $\Sigma_0 = St$, $\Sigma_1 = [A|St]$, and $\Sigma_2 = St$. The resulting clause in chain form is (17). The definitions of the h_i's are as before, up to variable renaming.
End of example

Let P be a moded program. We define $\hat{P}$ as the program resulting from applying Theorem 1 to every clause in P in such a way that the h_i predicate symbols of a clause do not occur in any other clause of $\hat{P}$.

Theorem 1 associates a chain program $\hat{P}$ with a moded program P in such a way that $\hat{P}$ is a logical consequence of a conservative extension of P. The implication in the other direction also holds. That P is logically implied by a conservative extension of $\hat{P}$ can be seen by first resolving each clause in $\hat{P}$ having a head with predicate symbol $\hat{p}$ with the only-if part of the definition of $\hat{p}$ and unfolding the definitions of the h_i's and the $\hat{q}_i$'s.

Finally, note that the chain program $\hat{P}$ of a moded program P is a G-chain program (i.e. a program such that in every unit clause $p(t, t') \leftarrow$, $\mathrm{var}(t') \subseteq \mathrm{var}(t)$, so that all answers for a subgoal with a ground input are ground using a leftmost computation rule (cf. Sect. 2)).

Example 3
As an example explicitly linking this transformation with the abcde metainterpreter, we give the object-program representation of the chain form of the clauses (12) and (13).

$$\begin{array}{l}
\mathit{defn}(\hat{s}, [\hat{s}_1, \hat{s}_2]) \leftarrow \\
\mathit{defn}(h_0, [h_0']) \leftarrow \\
\mathit{defn}(h_1, [h_1']) \leftarrow \\
\mathit{nonunit}(\hat{s}_2, [h_0, \hat{s}, h_1]) \leftarrow \\
\mathit{unit}(\hat{s}_1, \langle St, L\rangle, \langle St, [\,], L\rangle) \leftarrow \\
\mathit{unit}(h_0', \langle St, [A|N]\rangle, \langle [A|St], N\rangle) \leftarrow \\
\mathit{unit}(h_1', \langle [A|St], L, M\rangle, \langle St, [A|L], M\rangle) \leftarrow \\
\mathit{is_unit}(\hat{s}_1) \leftarrow \qquad \mathit{is_unit}(h_0') \leftarrow \qquad \mathit{is_unit}(h_1') \leftarrow
\end{array}$$

End of example

4 Conversion of Definite Programs into Chain Form

In this section, we will first give a transformation inspired by the previous one, converting an arbitrary definite program into chain form. Next, we will explain how to couple the abcde metainterpreter to this "unmoded" transformation.

4.1 Transformation

Roughly, the *moded* transformation takes a clause with predicates having one input argument place and one output argument place, disposes of the pass-on variables (the Π_j's) by adding a stack, and replaces both arguments of each subgoal by variables.

Hence, the first apparent obstacle we find in trying to convert an arbitrary, *unmoded* clause into chain form, is that arguments of predicates in such a clause do not have predetermined input/output roles. The fact that any argument of a predicate may play the role of either input or output suggests treating all arguments uniformly. One way of doing so and yet have binary predicates could be to replicate the arguments of each predicate so as to have two copies of each set of arguments: one copy behaving as a single input (possibly having variables when the subgoal is selected), and the other copy behaving as a single output. Naturally, we now have to give up the groundness property of runtime terms, which translates to having to use full unification instead of matching.

Thus, we can associate with each predicate $p(X_1, \ldots, X_n)$, another predicate, defined as: $\hat{p}(\langle X_1, \ldots, X_n\rangle, \langle X_1, \ldots, X_n\rangle) \leftarrow p(X_1, \ldots, X_n)$, which denotes a *subset* of the identity relation of the Herbrand universe. But we also have to add the stack, so that we have: $\hat{p}(\langle St, X_1, \ldots, X_n\rangle, \langle St, X_1, \ldots, X_n\rangle) \leftarrow p(X_1, \ldots, X_n)$.

Another decision we have to make is which variables to push onto the stack. In fact, we could push all variables of the clause, but we will be more economical by pushing only the variables not occurring in all atoms of the clause.

Example 4
Consider the usual *append* program:

$$a([\,], L, L) \leftarrow \tag{20}$$

$$\underline{a([A|L], M, [A|N])} \leftarrow a(L, M, N) \tag{21}$$

First we write the if-part of the definition of $\hat{a}$:

$$\hat{a}(\langle St, X, Y, Z\rangle, \langle St, X, Y, Z\rangle) \leftarrow a(X, Y, Z) \tag{22}$$

We now apply equation introduction to (22), getting:

$$\begin{aligned} &\hat{a}(\langle St, X, Y, Z\rangle, \langle St', X', Y', Z'\rangle) \leftarrow \\ &\qquad St = St',\ X = X',\ Y = Y',\ Z = Z', \\ &\qquad a(X', Y', Z') \end{aligned} \tag{23}$$

which we will need for a later folding application.

Next, we obtain an instance of (22), to which we apply equation introduction:

$$\begin{aligned} &\hat{a}(\langle St, [A|L], M, [A|N]\rangle, \langle St', [A'|L'], M', [A'|N']\rangle) \leftarrow \\ &\qquad St = St',\ A = A',\ L = L',\ M = M',\ N = N', \\ &\qquad \underline{a([A'|L'], M', [A'|N'])} \end{aligned} \tag{24}$$

Now we resolve (21) with (24) unifying the two underlined atoms, and get:

$$\begin{aligned} &\hat{a}(\langle St, [A|L], M, [A|N]\rangle, \langle St', [A'|L'], M', [A'|N']\rangle) \leftarrow \\ &\qquad \boxed{St = St'},\ \boxed{A = A'},\ L = L',\ M = M',\ N = N', \\ &\qquad a(L', M', N') \end{aligned}$$

Subsequently, we fold using function substitutivity.

$$\begin{aligned} &\hat{a}(\langle St, [A|L], M, [A|N]\rangle, \langle St', [A'|L'], M', [A'|N']\rangle) \leftarrow \\ &\qquad \boxed{[A|St] = [A'|St']},\ \boxed{L = L'},\ \boxed{M = M'},\ \boxed{N = N'}, \\ &\qquad \boxed{a(L', M', N')} \end{aligned}$$

Finally, we fold using (23) and systematically rename variables.

$$\hat{a}(\langle St_0, [A_0|L_0], M_0, [A_0|N_0]\rangle, \langle St_1, [A_1|L_1], M_1, [A_1|N_1]\rangle) \leftarrow \\ \hat{a}(\langle [A_0|St_0], L_0, M_0, N_0\rangle, \langle [A_1|St_1], L_1, M_1, N_1\rangle)$$

We can now apply predicate substitutivity and then fold w.r.t. the completed definitions of the predicates manipulating the stack, as in the moded transformation, thus arriving at chain form.
End of example

An arbitrary definite program can be converted into chain form with the following theorem.

Theorem 2
Let C be a definite clause:

$$p(\tilde{t}_0) \leftarrow q_1(\tilde{t}_1),\ q_2(\tilde{t}_2),\ \ldots,\ q_n(\tilde{t}_n) \qquad\qquad n \geq 0$$

and let:

$$\Pi = \big(\mathrm{var}(\tilde{t}_0) \cup \cdots \cup \mathrm{var}(\tilde{t}_n)\big) \setminus \big(\mathrm{var}(\tilde{t}_0) \cap \cdots \cap \mathrm{var}(\tilde{t}_n)\big)$$

Then the clause C':

$$\hat{p}(U_0, U_n') \leftarrow h_0(U_0, U_0'),\ \hat{q}_1(U_0', U_1),\ h_1(U_1, U_1'),\ \hat{q}_2(U_1', U_2),\ \ldots, \\ \hat{q}_n(U_{n-1}', U_n),\ h_n(U_n, U_n')$$

is logically implied by C, the standard equality theory for C, the “iff” version of the function substitutivity axiom for the list-constructor function symbol:

$$[X\,|\,Y] = [X'|\,Y'] \leftrightarrow X = X' \ \&\ Y = Y'$$

and the completed definitions of:

$$\begin{array}{ll}
\hat{p}(\langle St, X_{1,0}, \ldots, X_{r_0,0}\rangle, \langle St, X_{1,0}, \ldots, X_{r_0,0}\rangle) \leftarrow p(X_{1,0}, \ldots, X_{r_0,0}) & \\
\hat{q}_i(\langle St, X_{1,i}, \ldots, X_{r_i,i}\rangle, \langle St, X_{1,i}, \ldots, X_{r_i,i}\rangle) \leftarrow q_i(X_{1,i}, \ldots, X_{r_i,i}) & i = 1, \ldots, n \\
h_0(\langle St|\tilde{t}_0\rangle, \langle \Sigma|\tilde{t}_1\rangle) \leftarrow \qquad (i = 0) & \\
h_i(\langle \Sigma|\tilde{t}_i\rangle, \langle \Sigma|\tilde{t}_{i+1}\rangle) \leftarrow \qquad i = 1, \ldots, n-1 & \\
h_n(\langle \Sigma|\tilde{t}_n\rangle, \langle St|\tilde{t}_0\rangle) \leftarrow \qquad (i = n) &
\end{array}$$

where Σ is any list of the form $[X_1, \ldots, X_d|St]$, such that $\{X_1, \ldots, X_d\} = \Pi$, if $\Pi \neq \emptyset$, and Σ is St if $\Pi = \emptyset$.

Proof
First, we use equation introduction on the definitions of the $\hat{q}_i$'s.

$$\hat{q}_i(\langle St, X_{1,i}, \ldots, X_{r_i,i}\rangle, \langle St', X_{1,i}', \ldots, X_{r_i,i}'\rangle) \leftarrow St = St', \\ X_{1,i} = X_{1,i}', \ldots, X_{r_i,i} = X_{r_i,i}', \\ q_i(X_{1,i}', \ldots, X_{r_i,i}') \\ i = 1, \ldots, n \qquad (25)$$

Next, we apply equation introduction to C in such a way that no two $\tilde{t}_i$'s have variables in common, except for $\tilde{t}_0$ and $\tilde{t}_n$. We obtain:

$$p(\tilde{t}_0\sigma_n) \leftarrow E,\ q_1(\tilde{t}_1\sigma_1),\ q_2(\tilde{t}_2\sigma_2),\ \ldots,\ q_n(\tilde{t}_n\sigma_n) \tag{26}$$

where the σ_j's are as in Theorem 1 and

$$\begin{aligned} E = \{X\sigma_j = X\sigma_k : X \in \big(\mathrm{var}(\tilde{t}_j) \cap \mathrm{var}(\tilde{t}_k)\big)\ \&\ 0 < j < k < n\} \\ \cup \{X\sigma_j = X\sigma_n : X \in \Big(\mathrm{var}(\tilde{t}_j) \cap \big(\mathrm{var}(\tilde{t}_0) \cup \mathrm{var}(\tilde{t}_n)\big)\Big)\ \&\ 0 < j < n\} \end{aligned}$$

Now, we apply θ to the if-part of the definition of $\hat{p}$, where θ is the mgu of $p(X_{1,0}, \ldots, X_{r_0,0})$ and $p(\tilde{t}_0)$ and use equation introduction in the resulting instance, getting (up to variable renaming):

$$\hat{p}(\langle St_0|\tilde{t}_0\sigma_0\rangle, \langle St_n|\tilde{t}_0\sigma_n\rangle) \leftarrow St_0 = St_n,\ F,\ p(\tilde{t}_0\sigma_n) \tag{27}$$

where

$$F = \{X\sigma_0 = X\sigma_n : X \in \mathrm{var}(\tilde{t}_0)\}$$

Subsequently, we resolve (26) with (27):

$$\begin{aligned} \hat{p}(\langle St_0|\tilde{t}_0\sigma_0\rangle, \langle St_n|\tilde{t}_0\sigma_n\rangle) \leftarrow\ & St_0 = St_n,\ F,\ E, \\ & q_1(\tilde{t}_1\sigma_1),\ q_2(\tilde{t}_2\sigma_2),\ \ldots,\ q_n(\tilde{t}_n\sigma_n) \end{aligned}$$

Before folding, we add the following equations, recalling that subgoal addition preserves soundness:

$$\{X\sigma_0 = X\sigma_n : X \in \big(\Pi \setminus \mathrm{var}(\tilde{t}_0)\big)\}$$

(Such equations, after applying transitivity, will enable us to have the same Σ in each subgoal of the resulting clause in prechain form.)

Now we add the equations:

$$\{X\sigma_{i-1} = X\sigma_i : X \in \mathrm{var}(\tilde{t}_i)\} \qquad 0 < i \leq n$$

(Such equations will enable us to fold w.r.t. (25).)

We now apply transitivity and factoring in such a way that the equations are replaced by (up to variable renaming):

$$\{St_0 = St_1,\ St_1 = St_2,\ \ldots,\ St_{n-1} = St_n\} \cup \bigcup_{0<i\leq n} G_i \cup \bigcup_{0<i\leq n} H_i$$

where

$$\begin{aligned} G_i &= \{X\sigma_{i-1} = X\sigma_i : X \in \Pi\} \\ H_i &= \{X\sigma_{i-1} = X\sigma_i : X \in \mathrm{var}(\tilde{t}_i)\} \end{aligned}$$

Next, we repetitively fold using function substitutivity and arrive at:

$$\begin{aligned} \hat{p}(\langle St_0|\tilde{t}_0\sigma_0\rangle, \langle St_n|\tilde{t}_0\sigma_n\rangle) \leftarrow\ & \Sigma\sigma_0 = \Sigma\sigma_1,\ \Sigma\sigma_1 = \Sigma\sigma_2,\ \ldots,\ \Sigma\sigma_{n-1} = \Sigma\sigma_n, \\ & \tilde{t}_1\sigma_0 \cong \tilde{t}_1\sigma_1,\ \tilde{t}_2\sigma_1 \cong \tilde{t}_2\sigma_2,\ \ldots,\ \tilde{t}_n\sigma_{n-1} \cong \tilde{t}_n\sigma_n, \\ & q_1(\tilde{t}_1\sigma_1),\quad q_2(\tilde{t}_2\sigma_2),\quad \ldots,\ q_n(\tilde{t}_n\sigma_n) \end{aligned}$$

where

$$\tilde{r} \cong \tilde{s} = \{r^i = s^i : r^i \text{ is the } i^{\text{th}} \text{ term of } \tilde{r} \text{ and } s^i \text{ is the } i^{\text{th}} \text{ term of } \tilde{s}\}$$

Prechain form is now obtained by folding with (25):

$$\begin{aligned}\hat{p}(\langle St_0|\tilde{t}_0\sigma_0\rangle, \langle St_n|\tilde{t}_0\sigma_n\rangle) \leftarrow\ & \hat{q}_1(\langle \Sigma\sigma_0|\tilde{t}_1\sigma_0\rangle, \langle \Sigma\sigma_1|\tilde{t}_1\sigma_1\rangle),\\ & \hat{q}_2(\langle \Sigma\sigma_1|\tilde{t}_2\sigma_1\rangle, \langle \Sigma\sigma_2|\tilde{t}_2\sigma_2\rangle),\ \ldots,\\ & \hat{q}_n(\langle \Sigma\sigma_{n-1}|\tilde{t}_n\sigma_{n-1}\rangle, \langle \Sigma\sigma_n|\tilde{t}_n\sigma_n\rangle)\end{aligned}$$

Finally, we apply predicate substitutivity and fold with the completed definitions of the h_i predicates. The resulting clause has the desired form; hence we conclude that the theorem holds. □

As in the moded transformation, that P is logically implied by a conservative extension of $\hat{P}$ can be seen by unfolding the definitions of the h_i's and the $\hat{q}_i$'s.

4.2 A deterministic metainterpreter for arbitrary chain programs

In Sect. 2 we wrote a deterministic metainterpreter assuming that the leftmost input in every goal of the LD tree (Apt 1997) (i.e. an SLD tree with a leftmost computation rule) was ground. We will now modify such a metainterpreter so that it is also correct for LD trees that do not necessarily have this groundness property.

It is possible to handle terms with variables by generalizing matching to full unification. A well-known metainterpreter explicitly using unification is Bowen and Kowalski's *demo* metainterpreter (Bowen and Kowalski 1982). In our case, unification for clauses of the form (1) reduces to argument passing so that we need only incorporate it to the clauses of the form (2). Consider for instance our previous definition of e', which included the clause:

$$e'(X, Pj, [Y]) \leftarrow \mathit{unit}(Pj, X, Y)$$

Following the *demo* metainterpreter, we would replace this clause by:

$$\begin{aligned}e'(X, Pj, [Y]) \leftarrow\ & \mathit{unit}(Pj, X', Y'),\\ & \mathit{rename}(f(X', Y'), X, f(X'', Y'')),\\ & \mathit{match}(X, X'', \mathit{Sub}),\\ & \mathit{apply}(Y'', \mathit{Sub}, Y)\end{aligned} \tag{28}$$

where

1. $\mathit{rename}(Z, X, Z')$ holds when Z' is the result of renaming the variables in Z so that they are distinct from the variables in X,
2. $\mathit{match}(X, X'', \mathit{Sub})$ holds when Sub is the mgu of X and X'', and
3. $\mathit{apply}(Y'', \mathit{Sub}, Y)$ holds when Y is the result of applying Sub to Y''.

Prolog provides a way to approximate this effect through the extra-logical *copy* "predicate":

$$e'(X, Pj, [Y]) \leftarrow \mathit{copy}(X, X\theta),\ \mathit{unit}(Pj, X\theta, Y) \tag{29}$$

Note that our addition of unification to the abcde metainterpreter occurs at a single point, unlike the unifiers of the metainterpreter in Fig. 1, which are pervasive (cf. the variables with a θ in their name).

5 Extending Existing Committed-Choice Traversal Methods to Arbitrary Definite Programs

This section deals first with a reconstruction and then with an extension of the existing stream-based (Tamaki 1987) and continuation-based (Ueda 1987) deterministic, exhaustive-traversal methods. An objective of these methods is that of executing OR-parallel programs in committed-choice processors (which are AND parallel).

The existing versions of such methods are restricted to moded programs. Hence, our reconstruction uses our transformation of moded programs into G-chain form. The extension modifies such methods so as to make them applicable to arbitrary definite programs essentially by replacing the moded transformation of Sect. 3 by the definite transformation of Sect. 4.

We fall short of proposing practical methods because we do not eliminate the layer of interpretation. One way of eliminating such a layer would be to feed the metainterpreter and the object program to a general-purpose partial evaluator such as Mixtus (Sahlin 1993). However, the resulting residual program may be enormous. Another possibility would be to compile away the layer of interpretation "by hand," but we have not done so in the present work.

5.1 Reconstruction

The derivations of both methods start from the abcde metainterpreter. For brevity, we will omit detailed derivations, and will only indicate how such derivations could be obtained. Also, instead of using difference lists as the original methods do, we will employ ordinary lists for clarity.

5.1.1 A chain-program reconstruction of the stream-based method

As observed by Tamaki, programs originally having some degree of AND parallelism may lose such a parallelism if we only capture their OR parallelism. He thus treats clauses with AND parallelism in a special way. For simplicity we will not be concerned with such a special treatment here, and will concentrate on the main component of this method, that converts OR parallelism into AND parallelism.

We can obtain programs produced by the stream-based method if we unfold (7) using (8):

$$\begin{aligned} a([X|Xs],[Q|Qs],YsZs) \leftarrow\ & b'(X,Q,Ys),\ a(Ys,Qs,Ys'), \\ & a(Xs,[Q|Qs],Zs),\ \mathit{append}(Ys',Zs,YsZs) \end{aligned} \tag{30}$$

Let us consider first how a chain program is transformed by the stream-based method. Perhaps the most interesting clause in the program produced by this method is a clause associated with each subgoal $q_{i+1}(X_i, X_{i+1})$, which is of the form:

$$k_i([X|Xs], YsZs) \leftarrow all\_q_{i+1}(X, Ys),\ k_{i+1}(Ys, Ys'),\\ k_i(Xs, Zs),\ append(Ys', Zs, YsZs) \tag{31}$$

where $all\_q_{i+1}(\mathbf{x}, \mathbf{ys})$ is intended to hold when $\mathbf{ys}$ is the set of answers to $\leftarrow q_{i+1}(\mathbf{x}, Y)$. Observe first that $all\_q_{i+1}(\mathbf{x}, \mathbf{ys})$ has the same intended meaning as $b'(\mathbf{x}, q_{i+1}, \mathbf{ys})$. Next, it is easy to obtain (31) from (30) by identifying the k_i predicate with the a predicate. The rest of the clauses resulting from transforming a chain program are readily obtainable from the abcde metainterpreter.

Consider now an arbitrary moded program. The clause corresponding to (31) in this case is:

$$k_i(\pi_i, [t_i|Xs], YsZs) \leftarrow all\_q_{i+1}(t_i', Ys),\ k_{i+1}(\pi_{i+1}, Ys, Ys'),\\ k_i(\pi_i, Xs, Zs),\ append(Ys', Zs, YsZs) \tag{32}$$

where π_j is any term such that $\mathrm{var}(\pi_j) = \Pi_j$, and Π_j is as defined in Theorem 1.

Recall now (30) and an object clause transformed by our moded transformation. A subgoal with an h_i predicate symbol, when partially evaluated with the abcde metainterpreter, results in a subgoal of the form $b'(X, h_i, Ys)$, which can be easily unfolded away producing the following clause, similar to (32):

$$a([\langle\Sigma_i|t_i\rangle|Xs], [Q|Qs], YsZs) \leftarrow b'(\langle\Sigma_{i+1}|t_i'\rangle, Q, Ys),\ a(Ys, Qs, Ys'),\\ a(Xs, [Q|Qs], Zs),\ append(Ys', Zs, YsZs) \tag{33}$$

A difference between (32) and (33) is that the stream-based method uses a parameter π_i in the k_i predicates for recording the values of the Π_i variables, whereas (33) keeps the Π_i variables as part of each term $\langle\Sigma_i|t_i\rangle$. Even with this difference, the stream-based compiled program and the abcde metainterpreter follow the same search strategy. By using a separate parameter π_i, however, the stream-based method is more economical because of exploiting the fact that only the h_i predicates may modify the stack: all other predicates hold for relations with an output stack equal to the input stack. (To see this, observe the stack in the definitions of $\hat{p}$, $\hat{q}_i$ in Theorem 1.) Hence, we can compute all answers to a goal $\leftarrow b'(\langle\Sigma_{i+1}|t_i'\rangle, Q, Ys)$, by first computing all answers to $\leftarrow b'(t_i', Q, Ws)$ and then affixing Σ_{i+1} in front of each such answer, if Q is not an h_i predicate:

$$\langle St|t\rangle\ ;\ Q = St \diamond (t\ ;\ Q) \tag{34}$$

where $X \diamond Xs$ is the list obtained by concatenating all lists having X affixed in front of every list in Xs:

$$X \diamond [\,] = [\,]$$
$$X \diamond [Y|Ys] = [[X|Y]|X \diamond Ys]$$

Let us first rewrite the stream-based clause (32) as:

$$a_{stream}(\Sigma_i, [t_i|Xs], [Q|Qs], YsZs) \leftarrow b'(t_i', Q, Ys),\ a_{stream}(\Sigma_{i+1}, Ys, Qs, Ys'),\\ a_{stream}(\Sigma_i, Xs, [Q|Qs], Zs),\\ append(Ys', Zs, YsZs) \tag{35}$$

where the predicate $a_{stream}(St, Xs, Qs, YsZs)$ is intended to hold iff

$$(St \diamond Xs)\,;\,Qs = St \diamond YsZs$$

Hence, (35) asserts that:

$$\Sigma_i \diamond [t_i|Xs]\,;\,(Q\,;\,Qs) \supseteq \big(\Sigma_{i+1} \diamond (t_i'\,;\,Q)\,;\,Qs\big) +\!\!+ \big(\Sigma_i \diamond Xs\,;\,(Q\,;\,Qs)\big) \tag{36}$$

Here, and throughout this section, $\diamond$ binds stronger than ";", and $+\!\!+$ denotes list concatenation.

Let us now consider the abcde metainterpreter. For clarity, we rename variables in (33):

$$\begin{aligned} a([\langle\Sigma_i|t_i\rangle|StXs], [Q|Qs], StYsZs) \leftarrow\ & b'(\langle\Sigma_{i+1}|t_i'\rangle, Q, StYs),\ a(StYs, Qs, StYs'),\\ & a(StXs, [Q|Qs], StZs),\\ & append(StYs', StZs, StYsZs) \end{aligned} \tag{37}$$

where $a(StXs, Qs, StYsZs)$ holds when $StXs\,;\,Qs = StYsZs$ and the same stack Σ_i occurs in front of every element of $StXs$ and $StYsZs$. Hence, (37) asserts that:

$$[\langle\Sigma_i|t_i\rangle|\Sigma_i \diamond Xs]\,;\,(Q\,;\,Qs) \supseteq \big((\langle\Sigma_{i+1}|t_i'\rangle\,;\,Q)\,;\,Qs\big) +\!\!+ \big(\Sigma_i \diamond Xs\,;\,(Q\,;\,Qs)\big) \tag{38}$$

By the definition of $\diamond$ and (34), we obtain (36) from (38). (Specifically, by the recursive equation in the definition of $\diamond$ we obtain the left-hand side and by (34) we obtain the right-hand side.)

5.1.2 A chain-program reconstruction of the continuation-based method

Let us now turn our attention to the continuation-based method. Starting also from the abcde metainterpreter, we use the completed definitions of:

$$c'\_a(X, Pjs, Qs, M) \leftarrow \underline{c'(X, Pjs, L)},\ a(L, Qs, M) \tag{39}$$

$$d'\_a(X, Pj, Qs, M) \leftarrow \underline{d'(X, Pj, L)},\ a(L, Qs, M) \tag{40}$$

where Qs acts like a list of "continuations," and the underlined subgoals indicate a forthcoming unfolding application.

First we unfold (39) using (9):

$$\begin{aligned} c'\_a(X, [Pj|Pjs], Qs, M) \leftarrow\ & d'(X, Pj, L_1),\ c'(X, Pjs, L_2),\\ & append(L_1, L_2, L),\ a(L, Qs, M) \end{aligned} \tag{41}$$

Using now the identity

$$\big((X\,;\,P_1) \cup \ldots \cup (X\,;\,P_m)\big)\,;\,Qs = (X\,;\,P_1\,;\,Qs) \cup \ldots \cup (X\,;\,P_m\,;\,Qs)$$

which follows from the right distributivity of composition over union, we rewrite (41) as:

$$\begin{aligned} c'\_a(X, [Pj|Pjs], Qs, M) \leftarrow\ & \boxed{d'(X, Pj, L_1),\ a(L_1, Qs, M_1)},\\ & \boxed{c'(X, Pjs, L_2),\ a(L_2, Qs, M_2)},\\ & append(M_1, M_2, M) \end{aligned} \tag{42}$$

where each rectangle indicates a forthcoming folding application. We can now fold (42) using the definitions of the d'_a and c'_a predicates:

$$\begin{aligned} c'\_a(X, [Pj|Pjs], Qs, M) \leftarrow\ & d'\_a(X, Pj, Qs, M_1), \\ & c'\_a(X, Pjs, Qs, M_2), \\ & append(M_1, M_2, M) \end{aligned} \tag{43}$$

arriving at a clause of the continuation-based metainterpreter.

Another interesting clause is obtained by unfolding (40) using (10):

$$d'\_a(X, Pj, Qs', M) \leftarrow nonunit(Pj, Qs),\ \ a'(X, Qs, L),\ \ a(L, Qs', M)$$

which we rewrite as:

$$\begin{aligned} d'\_a(X, Pj, Qs', M) \leftarrow\ & nonunit(Pj, Qs), \\ & append(Qs, Qs', QsQs'),\ \ a'(X, QsQs', M) \end{aligned}$$

This step can be justified using the associativity of composition:

$$(X\ ;\ Qs)\ ;\ Qs' = X\ ;\ (Qs\ ;\ Qs')$$

Figure 2 shows the resulting metainterpreter, where we have applied an unfolding step using the definition of e'.

$$\begin{aligned}
& a'(X, [\,], [X]) \leftarrow \\
& a'(X, [Q|Qs], Zs) \leftarrow defn(Q, Pjs),\ \ c'_a(X, Pjs, Qs, Zs) \\
& c'_a(X, [\,], Qs, [\,]) \leftarrow \\
& c'_a(X, [Pj|Pjs], Qs, YsZs) \leftarrow d'_a(X, Pj, Qs, Ys),\ \ c'_a(X, Pjs, Qs, Zs), \\
& \qquad\qquad\qquad\qquad\qquad\qquad append(Ys, Zs, YsZs) \\
& d'_a(X, Pj, Qs, Zs) \leftarrow is_unit(Pj),\ \ unit(Pj, X, Y),\ \ a'(Y, Qs, Zs) \\
& d'_a(X, Pj, Qs, [\,]) \leftarrow is_unit(Pj),\ \ not(unit(Pj, X, Y)) \\
& d'_a(X, Pj, Qs', Zs) \leftarrow nonunit(Pj, Qs),\ \ append(Qs, Qs', QsQs'),\ \ a'(X, QsQs', Zs)
\end{aligned}$$

Figure 2. *A continuation-based, deterministic, exhaustive-traversal metainterpreter.*

5.2 Unmoded versions of the stream-based and the continuation-based methods

Having reconstructed the stream- and continuation-based methods through chain programs, we can now replace the moded transformation by the definite transformation. However, as in Sect. 4, we must also rename variables when using the *unit* predicate with either (28) or (29).

6 Other Applications

6.1 Prolog as the continuation-based metainterpreter together with the definite transformation

So far we have designed metainterpreters for performing traversals in committed-choice processors. Observe that just as committed-choice processors have deterministic bindings, so do standard (deterministic) imperative languages. This suggests the possibility of using our metainterpreters for describing search strategies in one such imperative language. In particular, we will see how to obtain an imperative implementation of Prolog's search strategy by slightly modifying the continuation-based metainterpreter of Sect. 5 (Fig. 2).

A difference between our previous metainterpreters and the standard implementations of Prolog is that whereas we perform exhaustive traversals, Prolog systems may or may not do so. However, we can easily modify the continuation-based metainterpreter so as to ask the user whether or not more answers are requested.

Also, note that Prolog systems do not usually remember the answers to a query, allowing us to eliminate the answer list.

$$a'(X, [\,], halt) \leftarrow write(X),\ \ write('\ more?\ '),\ \ read(n)$$
$$a'(X, [\,], cont) \leftarrow$$
$$a'(X, [Q|Qs], HaltCont) \leftarrow defn(Q, Pjs),\ \ c'\_a(X, Pjs, Qs, HaltCont)$$

$$c'\_a(X, [\,], Qs, cont) \leftarrow$$
$$c'\_a(X, [Pj|Pjs], Qs, HaltCont') \leftarrow d'\_a(X, Pj, Qs, HaltCont),$$
$$halt\_cont(X, Pjs, Qs, HaltCont, HaltCont')$$

$$d'\_a(X, Pj, Qs, HaltCont) \leftarrow is\_unit(Pj),\ \ unit(Pj, X, Y),\ \ a'(Y, Qs, HaltCont)$$
$$d'\_a(X, Pj, Qs, cont) \leftarrow is\_unit(Pj),\ \ not(unit(Pj, X, Y))$$
$$d'\_a(X, Pj, Qs', HaltCont) \leftarrow nonunit(Pj, Qs),$$
$$append(Qs, Qs', QsQs'),\ \ a'(X, QsQs', HaltCont)$$

$$halt\_cont(X, Pjs, Qs, halt, halt) \leftarrow$$
$$halt\_cont(X, Pjs, Qs, cont, HaltCont) \leftarrow c'\_a(X, Pjs, Qs, HaltCont)$$

This metainterpreter is meant for constructing ground terms. To obtain a true (pure) Prolog system, handling terms with variables, we would have to include variable renaming and unification in a manner similar to that of either (28) or (29).

Deterministic metainterpreters for arbitrary definite programs can be written directly, without using our transformations, into chain form. (An example is the metainterpreter in Fig. 1.) The programmer, however, has to be aware of the set of unifiers of the children of a node in the derivation tree. By contrast, in our approach the programmer can write a metainterpreter without considering such unifiers, except when using the *unit* predicate, in which case the object clause has no body, simplifying the treatment of unifiers.

6.2 A bounded-resource metainterpreter

As a final application, we exhibit a bounded-resource metainterpreter. As usual, we will give a metainterpreter for chain programs. Our transformations of moded and definite programs into chain form make this metainterpreter applicable to programs that do not necessarily have chain form.

The next variant of the abcde metainterpreter constructs at most one proof, and has an extra argument to indicate the amount of resources needed to construct such a proof.

$$a'(X, [\,], ans(X), 0) \leftarrow$$
$$a'(X, [Q|Qs], Z, R+1) \leftarrow defn(Q, Pjs),\ \ c'\_a(X, Pjs, Qs, Z, R)$$

$$c'\_a(X, [\,], Qs, no\_ans, 0) \leftarrow$$
$$\begin{aligned} c'\_a(X, [Pj|Pjs], Qs, Z, R+S) \leftarrow\ & d'\_a(X, Pj, Qs, Y, R), \\ & halt\_cont(X, Pjs, Qs, Y, Z, S) \end{aligned}$$

$$d'\_a(X, Pj, Qs, Z, R) \leftarrow is\_unit(Pj),\ \ unit(Pj, X, Y),\ \ a'(Y, Qs, Z, R)$$
$$d'\_a(X, Pj, Qs, no\_ans, 0) \leftarrow is\_unit(Pj),\ \ not(unit(Pj, X, Y))$$
$$\begin{aligned} d'\_a(X, Pj, Qs', Z, R) \leftarrow\ & nonunit(Pj, Qs), \\ & append(Qs, Qs', QsQs'),\ \ a'(X, QsQs', Z, R) \end{aligned}$$

$$halt\_cont(X, Pjs, Qs, ans(Y), ans(Y), 0) \leftarrow$$
$$halt\_cont(X, Pjs, Qs, no\_ans, Z, S) \leftarrow c'\_a(X, Pjs, Qs, Z, S)$$

This style of writing bounded-resource metainterpreters may be viewed as an alternative to that appearing in (Kowalski 1995, Kowalski and Sadri 1996).

7 Concluding Remarks

7.1 Contributions

Some applications of metainterpreters have been neglected, perhaps because of being based on convoluted definitions. By comparison with the demo predicate, deterministic metainterpreters, for example, result especially elaborate, since the programmer must consider the set of unifiers of the children of a node in the derivation tree. Thus, metainterpreters for (1) converting OR parallelism into AND parallelism (Ueda 1987, Tamaki 1987), (2) describing search-strategies in logic-based, state-oriented languages (Clark and van Emden 1981), and (3) simulating bounded-resource reasoning (Kowalski 1995, Kowalski and Sadri 1996), have not received due attention.

Compilation methods converting OR parallelism into AND parallelism have been developed first by Ueda (Ueda 1987) and then by Tamaki (Tamaki 1987). By studying these methods, we have identified *chain* programs as important for exhibiting the essence of such techniques. If we use chain programs as a stepping stone, then the methods of (Ueda 1987, Tamaki 1987) can be viewed as comprising two parts:

a. conversion of a *moded* program into chain form, and
b. application of partial deduction to a deterministic metainterpreter for chain programs.

Our contribution to part (a) consisted first in having extracted from (Ueda 1987, Tamaki 1987) the implicit transformation that converts a moded program into an equivalent chain form. Next, by using a generalization of this transformation, we have given another, "unmoded" transformation, that converts *arbitrary* definite programs into chain form.

To part (b) we contributed by showing how to write deterministic metainterpreters for chain programs. One such metainterpreter served us first to reconstruct and then to extend to arbitrary (unmoded) definite programs the existing methods of (Ueda 1987, Tamaki 1987).

Finally, we observed that deterministic metainterpreters have applications other than exhaustive traversals. We gave a metainterpreter that follows Prolog's search strategy and another one that counts the number of steps in the search for a refutation (as opposed to the number of steps in the refutation).

Our methodology for designing deterministic-traversal methods is then as follows:

1. Write a deterministic metainterpreter for chain programs ignoring unification.
2. Incorporate to the metainterpreter one of the transformations converting either moded or unmoded programs into chain form.
3. In case the unmoded transformation was selected, add renaming and unification to the metaclauses dealing with the object *unit* clauses.

We observed that even after adding unification, we need only be concerned about substitutions at a single point of the chain-program metainterpreter, whereas in a metainterpreter written directly the unifiers are pervasive (cf. Fig. 1).

7.2 Performance study

We have made a study illustrating how the performance of some programs is degraded as a result of transforming such programs into chain form. For this study, we used SICStus Prolog version 3.7.1, which we ran under RedHat Linux version 6.0. In this table and the next, the columns labeled A show the data for the source program and the columns labeled B show the data for the corresponding transformed program.

First we exhibit the number of clauses and the program size (measured in bytes for compiled code).

program	*num. clauses* A	*num. clauses* B	*program size* A	*program size* B
split (moded *append*)	2	4	445	1,051
append (for splitting)	2	4	440	1,168
quicksort, ord. lists	7	22	1,405	5,833
quicksort, diff. lists	5	17	1,066	5,871

Now we give the relative execution times (according to SICStus' `profile_data/4`) and the memory requirements for the local and global stacks (according to SICStus' `statistics/0`) for splitting a 100-element list into all its prefixes and suffixes and sorting the reverse of a sorted 100-element list with quicksort programs.

program	*execution time* A	*execution time* B	*global+local stacks* A	*global+local stacks* B
split (moded *append*)	26	571	32,760	32,760
append (for splitting)	26	636	32,760	32,760
quicksort, ord. lists	2,360	7,539	147,720	1,310,400
quicksort, diff. lists	2,007	5,788	81,900	1,179,360

7.3 Related work

Our work stemmed from the continuation-based and the stream-based exhaustive-traversal methods. There are, however, various other publications studying deterministic traversals of search spaces within logic programming (Hirakawa, Chikayama and Furukawa 1984, Bansal and Sterling 1987, Codish and Shapiro 1987, Lichtenstein, Codish and Shapiro 1987, Shapiro 1987, Sato and Tamaki 1989b, Mariën and Demoen 1993). Of these contributions, (Mariën and Demoen 1993) has perhaps the closest motivation to ours: The authors sketch a reconstruction of the continuation-based method and give a metainterpreter of their exhaustive-search method (based on recomputation).

Similarly, there are a number of transformations converting logic programs into a syntactically restricted form (Sato and Tamaki 1989a, Tarau and Boyer 1990, Tarau 1991). The one in (Sato and Tamaki 1989a) has in common with our work a connection with the continuation-based exhaustive-traversal method. These transformations differ from ours, however, in producing programs in which every clause is binary (i.e. has only one atom in the body).

7.4 Future work

We have argued that writing deterministic metainterpreters is advantageous with our approach because this task amounts to that of describing an evaluation strategy for relational expressions of the form:

$$P = P_1 \cup P_2 \cup \ldots \cup P_m$$

where each P_i is defined as:

$$P_i = Q_1 \,;\, Q_2 \,;\, \ldots \,;\, Q_n$$

Evaluation strategies for expressions have a close connection with the implementation of functional programming languages (Peyton Jones 1987) and rewrite systems (Dershowitz and Jouannaud 1990). Investigating how different evaluation strategies for these relational expressions lead to different search strategies for spaces determined by chain programs would be one way of extending our contributions.

During the presentation of our results, we came across the need for eliminating the layer of interpretation. Our work would have a greater practical impact if it were combined with an algorithmic elimination of such a layer, without producing an excessively large residual program.

Chain programs have also proved to be useful in devising (Rosenblueth 1996, Rosenblueth and Peralta 1998) inference systems derived from context-free parsers, because we need only consider unification in the treatment of unit clauses and hence we need only modify the treatment of terminals. Studying other applications of chain programs where it might be helpful to relegate the role played by unifiers would be another avenue of research.

Acknowledgments

This work owes much to Carlos Velarde. He contributed with motivating discussions, he carefully read previous versions of this paper, and he spotted errors in the theorem proofs. We also thank the referees, whose comments substantially improved the presentation of these results. We gratefully acknowledge the facilities provided by IIMAS, UNAM.

References

Abramson, H. and Rogers, M. (eds) (1988). *Meta-Programming in Logic Programming*, MIT Press.

Apt, K. and Turini, F. (eds) (1995). *Meta-Logics and Logic Programming*, MIT Press.

Apt, K. R. (1997). *From Logic Programming to Prolog*, Prentice Hall.

Bansal, A. K. and Sterling, L. (1987). Compiling generate-and-test programs to committed-choice AND-parallelism, *Technical Report CES-87-13*, Case Western Reserve University, Cleveland, Ohio, U.S.A.

Blikle, A. (1977). A comparative review of some program verification methods, *in* J. Gruska (ed.), *Mathematical Foundations of Computer Science 1977*, Springer-Verlag, pp. 17–33. Lecture Notes in Computer Science No. 53.

Bowen, K. A. and Kowalski, R. A. (1982). Amalgamating language and metalanguage in logic programming, *in* K. Clark and S.-A. Tärnlund (eds), *Logic Programming*, Academic Press, pp. 153–172.

Clark, K. and Gregory, S. (1985). Notes on the implementation of Parlog, *The Journal of Logic Programming* **1**: 17–42.

Clark, K. L. and van Emden, M. (1981). Consequence verification of flowcharts, *IEEE Transactions on Software Engineering* **SE-7**(1): 52–60.

Codish, M. and Shapiro, E. (1987). Compiling or-parallelism into and-parallelism, *in* E. Shapiro (ed.), *Concurrent Prolog. Collected Papers*, Vol. 2, MIT Press, pp. 351–382.

de Bakker, J. and Meertens, L. (1975). On the completeness of the inductive assertion method, *Journal of Computer and System Sciences* **11**: 323–357.

Dershowitz, N. and Jouannaud, J.-P. (1990). Rewrite systems, *in* J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. B, Elsevier Science Publishers B.V., pp. 243–320.

Engelfriet, J. (1974). *Simple Program Schemes and Formal Languages*, Springer-Verlag. Lecture Notes in Computer Science No. 20.

Gardner, P. and Shepherdson, J. (1992). Unfold/fold transformations of logic programs, *in* J.-L. Lassez and G. Plotkin (eds), *Computational Logic. Essays in Honor of Alan Robinson*, The MIT Press, pp. 565–583.

Hirakawa, H., Chikayama, T. and Furukawa, K. (1984). Eager and lazy enumerations in Concurrent Prolog, *in* S.-Å. Tärnlund (ed.), *Proc. Second International Logic Programming Conference*, Uppsala, Sweden, pp. 89–100.

Jiang, Y. (1994). Ambivalent logic as the semantic basis of metalogic programming: I, *in* P. V. Hentenryck (ed.), *Proc. Eleventh International Conference on Logic Programming*, pp. 387–401.

Kowalski, R. A. (1990). Problems and promises of computational logic, *in* J. Lloyd (ed.), *Computational Logic. Symposium Proceedings*, Springer-Verlag, pp. 1–36.

Kowalski, R. A. (1993). A springboard for information processing in the 21st century, *ICOT Journal* **38**: 17–41. Pannel Discussion at the Fifth Generation Computer Systems Conference 1992.

Kowalski, R. A. (1995). Using meta-logic to reconcile reactive with rational agents, *in* K. Apt and F. Turini (eds), *Meta-Logics and Logic Programming*, MIT Press, pp. 227–242.

Kowalski, R. and Sadri, F. (1996). Towards a unified agent architecture that combines rationality with reactivity, *Proc. International Workshop on Logic in Databases*, Springer-Verlag, San Miniato, Italy. Lecture Notes in Computer Science No. 1154.

Lichtenstein, Y., Codish, M. and Shapiro, E. (1987). Representation and enumeration of Flat Concurrent Prolog computations, *in* E. Shapiro (ed.), *Concurrent Prolog. Collected Papers*, Vol. 2, MIT Press, pp. 197–210.

Lloyd, J. (1987). *Foundations of Logic Programming*, 2nd edn, Springer-Verlag.

Mariën, A. and Demoen, B. (1993). Findall without findall/3, *in* D. S. Warren (ed.), *Proceedings of the Tenth International Conference on Logic Programming*, MIT Press, pp. 408–423.

Pettorossi, A. and Proietti, M. (1994). Transformation of logic programs: Foundations and techniques, *The Journal of Logic Programming* **19, 20**: 261–320.

Peyton Jones, S. L. (1987). *The Implementation of Functional Languages*, Prentice Hall.

Rosenblueth, D. A. (1996). Chart parsers as inference systems for fixed-mode logic programs, *New Generation Computing* **14**(4): 429–458.

Rosenblueth, D. A. (1998). An exhaustive-search method using layered streams obtained through a meta-interpreter for chain programs, *Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transforma-*

tion, 15–19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html. A summary appeared in: Lecture Notes in Computer Science No. 1559, pp 322–324, Springer-Verlag, 1999.

Rosenblueth, D. A. and Peralta, J. C. (1998). SLR inference: An inference system for fixed-mode logic programs, based on SLR parsing, *The Journal of Logic Programming* **34**(3): 227–260.

Safra, S. and Shapiro, E. (1986). Meta interpreters for real, *in* H. Kugler (ed.), *Information Processing 86 IFIP*, Elsevier Science Publishers B.V., North-Holland, pp. 532–557.

Sahlin, D. (1993). Mixtus: An automatic partial evaluator for full Prolog, *New Generation Computing* **12**: 7–51.

Sato, T. and Tamaki, H. (1989a). Existential continuation, *New Generation Computing* **6**(4): 421–438.

Sato, T. and Tamaki, H. (1989b). First order compiler: A deterministic logic program synthesis algorithm, *Symbolic Computation* **8**: 605–627.

Sergot, M. (1982). A query-the-user facility for logic programming, *in* P. Degano and E. Sandewall (eds), *Proceedings European Conference on Integrated Interactive Computing Systems*, North Holland.

Shapiro, E. (1987). Or-parallel Prolog in Flat Concurrent Prolog, *in* E. Shapiro (ed.), *Concurrent Prolog. Collected Papers*, Vol. 2, MIT Press, pp. 415–441.

Shapiro, E. Y. (1982). *Algorithmic Program Debugging*, MIT Press.

Tamaki, H. (1987). Stream-based compilation of ground I/O Prolog into committed-choice languages, *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia, pp. 376–393.

Tamaki, H. and Sato, T. (1984). Unfold/fold transformation of logic programs, *Proceedings of the Second International Logic Programming Conference*, pp. 127–138.

Tarau, P. (1991). Program transformations and WAM-support for the compilation of definite metaprograms, *in* A. Voronkov (ed.), *Proc. Russian Conference on Logic Programming*, Springer-Verlag, pp. 462–473. Lecture Notes in Artificial Intelligence No. 592.

Tarau, P. and Boyer, M. (1990). Elementary logic programs, *in* P. Deransart and J. Małuszyński (eds), *Proc. Programming Language Implementation and Logic Programming*, Springer-Verlag, Linköping, Sweden, pp. 159–173. Lecture Notes in Computer Science No. 456.

Ueda, K. (1987). Making exhaustive search programs deterministic, *New Generation Computing* **5**: 29–44.