AN ALGEBRAIC APPROACH TO META-LEVEL PROGRAMMING IN PROLOG

by G. LOUIS and M. VAUCLAIR

Philips Research Laboratory Brussels, Av. van Becelaere, 2 Box 8, B-1170 Brussels, Belgium

Abstract

Meta-level programming is used in Prolog when the standard semantics are not suitable for the task at hand: meta-interpreters are meta-programs that direct the execution of other programs and give them 'non-standard' semantics.

Meta-interpretation is inefficient, and program transformation is often preferred: from the initial program meant to be meta-interpreted, a new program is produced (compiled), whose execution produces the same result as meta-interpretation.

Partial evaluation of the meta-interpreter is often proposed as a technique for program transformation. The meta-interpreter is partially evaluated by fixing the program on which it would act. The resulting specialized program performs at object level the tasks the meta-interpreter would have produced at the meta-level.

In this paper, we introduce a more direct approach to program transformation: the compiled version of a program is specified as a transformation of its syntactic tree. Further, this transformation is viewed as an application of the universal property of word algebras: each clause in the source program is considered as a word constructed on the set of atomic predicates taken as generators, and the basic Prolog operators taken as signature. This view yields well structured translation programs, and provides insight in the semantics of the translation itself.

Keywords: algebra, logic programming, meta-level programming, partial evaluation, program transformation, Prolog.

1. Introduction

Meta-level programs treat other programs – called object-level programs – as data, to analyse, transform, or simulate them. Meta-interpreters are meta-programs that direct the execution of object-level programs. Programs executed under control of a meta-interpreter are thus given ‘non-standard’ semantics. Meta-interpretation is used in Prolog when the standard seman-
tics are not suitable for the task at hand. Among the usual applications for meta-interpreters, one can mention debugging and explanation generation for expert systems, which rely both on the construction and/or recording of the ‘proof tree’ generated by the execution of the program – or sometimes on a more complete trace of the program’s execution including failures. Other applications of meta-interpretation include the execution of programs using a control strategy different from the ‘left-to-right depth-first’ strategy of Prolog.

When compared to direct execution, meta-interpretation entails a loss of efficiency. First, syntactic analysis of the object program is performed by the meta-interpreter. Second, the execution of each object program step is simulated by several steps of the meta-interpreter. Third, if optimisations are applied by the Prolog implementation (be it a compiler or an interpreter), such as indexing of clauses, detection of deterministic calls, or optimisation of tail recursive calls, these will now apply to the meta-interpreter and not to the object program itself, and will not take advantage of the peculiarities of the object program. Further, since the meta-interpreter is general and frequently rather small, the optimisations have little room for improving it.

To regain efficiency, program transformation appears as an attractive alternative to meta-interpretation: from a given object-level program, the source of the transformation, a new program, the target program is produced (compiled). Program transformation is then an alternative to meta-interpretation inasmuch as the execution of the transformed program produces – at the object-level – the results that the meta-interpreter would have produced at the meta-level.

To obtain through transformation a program equivalent to the meta-interpretation of a given object-level program, it has been proposed to apply partial evaluation to the meta-interpreter itself. Partial evaluation is a general program transformation technique whose benefits are well established. Partial evaluation consists in first specifying partially the input of a program, then producing a specialized version of the program by exploiting the partial knowledge of its input. When applied to input satisfying the specification taken as the basis for partial evaluation, the specialized program executes more efficiently than the initial program. A meta-interpreter takes as input both a program to be interpreted and its data. Partial evaluation is applied to a meta-interpreter by fixing the program – but not the data – on which it will act.

However attractive, this approach – meta-interpreters plus partial evaluation – inherits the limitations of partial evaluation. Partial evaluation is straightforward only as far as the program can be given a pure ‘reduction’
An algebraic approach to meta-level programming in Prolog

(replacement) semantics, and several general problems are still unsolved (e.g. the handling of cuts). For partial evaluation to deliver its benefits, the program, i.e. the meta-interpreter, must be carefully annotated. One also observes that the most specific efficiency gain of partial evaluation of meta-interpreters seems to be the removal of object program syntactic analysis, which is performed during the partial evaluation process and not any more during execution by the meta-interpreter.

In this paper, we introduce and study a more direct approach. The compiled version of a program is specified as a transformation of its syntactic tree. Further, this transformation is viewed as an application of the universal property of word algebras: the source program is considered as a word constructed on the set of atomic predicates are taken as generators, and the basic Prolog operators taken as signature. Object program analysis is done at transformation time and is produced automatically.

The benefit of the algebraic approach is twofold: viewing the transformation scheme in algebraic terms helps to construct an abstract view of the transformation process and proposes a useful structure for the program that implements it. Also, since the program transformation paradigm is adopted from the onset, the efficiency is rather naturally achieved – sometimes at the price of the dynamic character of the resulting program, see below.

The rest of this paper is organized as follows. Section 2 introduces the basic concepts of universal algebra that will be needed for the rest of the paper. Section 3 describes meta-level programming in the algebraic framework of sec. 2. Sections 4 to 6 present a number of examples to illustrate several aspects of algebraic meta-programming. More precisely, in sec. 4, we illustrate the basic differences which are generally observed between meta-interpreters and algebraic translation schemes; Section 5 discusses meta-level information and introduces lambda expressions to deal with it; Section 6 presents a technique to merge two or more algebraic translation schemes into one; Section 7 presents an example for which the semantic domain is not trivial and is meant to show how the algebraic framework provides help in mastering such complexity. Section 8 discusses efficiency aspects. Section 9 contains concluding remarks.

2. Word algebras as a basis for specifying program translation

The use of concepts of universal algebra to specify compilers is not new\(^1\). The application of these concepts to program structuring, although apparently little known, is presented in \(^2\). Algebraic concepts are also used to define language semantics\(^3\).
The basic concepts of universal algebra needed in the sequel will now be recalled. An algebra \(14-17\) is a set (the \textit{carrier} of the algebra) with operations (functions) defined on it. To compare different algebras, it is convenient to define a \textit{signature} or \textit{operator domain} \(\Omega\) as a set (also denoted \(\Omega\)), with a mapping \(a: \Omega \rightarrow \mathcal{N}\); the elements \(\Omega\) are called \textit{operators} (more precisely \textit{operator symbols}); for \(\omega \in \Omega\), \(a(\omega)\) is the \textit{arity} of \(\omega\). An \(\Omega\)-algebra, \(A_{\Omega}\) (\(A\) for short), is an algebra whose operations are put in correspondence with the operators of \(\Omega\): for each \(\omega \in \Omega\), there is an operation \(\omega_A\) of arity \(a(\omega)\). An \(\Omega\)-word algebra, \(A_{\Omega}(X)\) (also named the \(\Omega\)-algebra \textit{freely generated by} \(X\)), is the set of syntactically correct terms that can be constructed using generators in \(X\) and operators in \(\Omega\). Given an operator \(\omega\) of arity \(n\) in \(\Omega\), and \(n\) terms \(T_1, \ldots, T_n\) in \(A_{\Omega}(X)\), the term \(\omega(T_1, \ldots, T_n)\) is taken as the result of the application of \(\omega\) on the \(T_i\)'s. Usual precautions, such as the use of a fully parenthesised form, the adoption of suitable operator priorities, or the definition of terms as trees (as done in Prolog) guarantee that each term in \(A_{\Omega}(X)\) is uniquely analysable as the application of an operator (the principal operator or principal functor of the term) to sub-terms.

\(\Omega\)-word algebras enjoy the following \textit{universal} property: for each \(\Omega\)-algebra \(A_{\Omega}\), any mapping \(f: X \rightarrow A_{\Omega}\) has a unique homomorphic extension \(f^*: W_{\Omega}(X) \rightarrow A_{\Omega}\). Function \(f^*\) is defined by induction on the structure of terms: if term \(t\) is a generator in \(X\), then \(f^*(t) = f(t)^*\), otherwise, \(t\) is of the form \(\omega(T_1, \ldots, T_n)\) for some \(\omega\) of arity \(n\) and \(f^*(t) = \omega_A(f^*(T_1), \ldots, f^*(T_n))\).

Thus, to specify a computation on finite trees with leaves in some set \(X\) (i.e. a computation on \(W_{\Omega}(X)\)), it is sufficient to specify the suitable \(\Omega\)-algebra together with a function evaluating the generators into its carrier. The universal property defines a unique evaluation function for each tree. For example, take \(\Omega\) to contain only operator \(+\) of arity 2, \(X\) to be the set of integer numerals, \(A_{\Omega}\) to be the integers with addition corresponding to \(+\),

\[\textit{To be formal, one should not identify } t \text{ as an element of } X \text{ with } t \text{ as an element of } W_{\Omega}(X), \text{ and use } \eta: X \rightarrow W_{\Omega}(X) \text{ as an injection: } f^*(\eta(t)) = f(t). \text{ This remark is not pure pedantry: it is sometimes convenient in a program to represent } x \text{ in } X \text{ differently from } x \text{ in } W_{\Omega}(X), \text{ and function } \eta \text{ is then needed. In this paper, we attempt to keep such formal details to a minimum, sometimes at the price of full rigour. See the references on universal algebra for a formal treatment.}\]
and \( f \) to be the function yielding the value of a numeral. \( W_\Omega(X) \) will then be the set of additive expressions, and \( f^* \) the function yielding the value of such expressions.

The basic scheme for applying these concepts to language semantics or meta-programming is now clear: the language sentences will be viewed as words in \( W_\Omega(X) \) with a suitable choice of \( X \) and \( \Omega \). Some basic language constructs (e.g. atomic goals in Prolog) will be selected to form \( X \), while constructors (e.g. conjunction and disjunction in Prolog) will form \( \Omega \). The specification of semantics reduces to the definition of a semantic domain which is also an \( \Omega \)-algebra, and of an injection from \( X \) into it. The universality of the word algebra will guarantee that each sentence has a unique meaning in the selected domain.

If the language syntax is specified by a context-free grammar, the simplest approach is to define operators in the signature for syntactic rules in the grammar, although it is often preferred to take an abstract syntactic form as the basis of the definition. Note that some syntactic rules will correspond to operators, while the other ones will be used to define the set of generators. An algebraic translation scheme is thus a 4-tuple \((\Omega, X, A_\Omega, f : X \rightarrow A_\Omega)\).

However, in general, the language to be given semantics together with the range of applications determine the choice of the signature and of the set of generators, and in the sequel an (algebraic) translation scheme is simply a pair \((f, A_\Omega)\) when \( n \) and \( X \) are understood.

2.1. Many-sorted algebras

It is often convenient to generalize these concepts to many-sorted algebras, to express some context-dependencies in the syntax itself, and to give structure to the semantic domain. In Prolog, for example, clause heads are often translated differently from simple goals in bodies.

A many-sorted signature \( \Omega \) consists of a collection of sorts, \( S = \{s_1, s_2, \ldots\} \), and an operator domain (also denoted \( \Omega \)), whose mapping defines for each operator its (generalised) arity in \( S^* \times S \). The arity of an operator indicates the sorts of its arguments and the sort of its result. For a many-sorted algebra \( A \), the carrier is an \( S \)-indexed family of sets (denoted \( A_s \), with \( s \) in \( S \)), and the collection of functions, as in the one-sorted case, has a function, \( \omega_\omega \) for each operator \( \omega \) in \( \Omega \). If \( \omega \) has arity \( ([s_1, s_2, \ldots], s) \), \( \omega_\omega \) maps \( A_{s_1} \times A_{s_2} \times \ldots \) to \( A_s \). The construction of a many-sorted \( \Omega \)-algebra \( W_\Omega(X) \) requires the generators in \( X \) to be given a sort in \( S \). The construction of terms must of course respect sorts: if \( a(\omega) = ([s_1, s_2, \ldots], s) \), \( \omega(T_1, T_2, \ldots) \) is in \( W_\Omega(X) \) if and only if the \( T_i \)'s are of sort \( s_i \) respectively; the sort of the whole term will be \( s \). It
is again required that each term is uniquely analysable into its principal functor and component subterms. This precludes operator overloading, which must, when desired, be removed by prior syntactic analysis.

Many-sorted algebras again enjoy the universal property: for each $\Omega$-algebra $A_\Omega$, any $S$-indexed family of sets of generators, $X = \{X_s | s \in S\}$, any $S$-indexed family of functions $f = \{f_s | X_s \to A_s, s \in S\}$, there is a unique homomorphic extension $f^*$ of $f$, mapping words in $W_{f^*}(X)$ to $A_\Omega$.

3. Algebraic meta-level programming in Prolog

In this section, the framework of sec. 2 will be applied to meta-programming in Prolog. There are two aspects to it: the implementation of algebraic translation schemes in Prolog itself, which will be discussed first, and the definition of translation schemes that apply to object-level programs written in Prolog.

Once a signature and a set of generators has been selected, an algebraic translation scheme can be implemented by a Prolog program by defining:
- $'w_A'(+V_1, +V_2, \ldots, +V_n, -V)^*$) to implement $V = \omega_A(V_1, V_2, \ldots, V_n)$, for each $\omega$ in the signature.
- $f(+G, -V)$ to implement $V = f(G)$.

A final predicate, $\text{val}(+W, -V)$ is needed to implement $V = f^*(W)$, i.e. the semantic function proper. The clauses defining val are constructed from the signature only: there is one clause for each operator $\omega$ in $\Omega$, which defines the value of terms whose principal functor is $\omega$, e.g. if the signature contains $+/2$, there will be a clause:

$$\text{val}(W_1 + W_2, V) :- !, \text{val}(W_1, V_1), \text{val}(W_2, V_2), '+'A'(V_1, V_2, V).$$

There is a last clause to deal with the generators themselves:

$$\text{val}(G, V) :- f(G, V).$$

The construction of the clauses for val from $\Omega$ can be implemented by a straightforward Prolog program (a signature compiler). Thus, to implement a translation scheme, it is only necessary to provide code for predicates $'w_A'$ of arity $a(\omega) + 1$ for each $\omega$ in $\Omega$, and $f$ of arity 2 for $f$. The predicates $'w_A'$, $\omega \in \Omega$ implement the operations of $A_\Omega$, whose carrier does not have an explicit representation in the meta-program itself. Similarly, there is no explicit representation for $X$, but predicate $f$, should succeed when its first argument is a generator.

*) Where the notation $p(+P, -O)$ indicates that upon call, the argument for $P$ is known (not an uninstantiated variable), i.e. $P$ is an input parameter. Conversely $O$ is an output parameter, i.e. its argument is an uninstantiated variable.
An algebraic approach to meta-level programming in Prolog

For the example of additive expressions introduced in sec. 2 above, (the implementation of) the translation scheme is simply:

\[ f(V, V). \]
\[ '+_A'(V1, V2, V) :- V \text{ is } V1 + V2. \]

We now discuss how translation schemes apply to Prolog programs. For most meta-level programs – all of the examples in this paper anyway – the atomic goals in Prolog, together with clause heads which are syntactically identical to them, will constitute the set \( X \) of generators.

Since Prolog program phrases are terms (i.e. the grammar is an operator grammar) and the syntactic sugar is reduced to a minimum (all operators are meaningful), the correspondence with the theoretical framework is straightforward and the signature will consist of operators used in the syntax.

The translation is most often performed clause by clause, as the examples in the next sections will show. Observe that facts as syntactic sugar for clauses whose body reduces to \texttt{true} will not be handled correctly by a straightforward translation scheme, and should be submitted to \texttt{val} in their extended form, i.e. as clauses with a \texttt{true} body.

When the problem at hand requires the translation to apply to procedures, the input syntax must include an explicit operator to bind clauses in procedures. For example, symbol \& could be chosen, and a procedure would be written:

\[ \text{concat}([], L, L) : - \text{true} \& \text{concat}(\text{[H|T]}, L, \text{[H|TL]}) :- \text{concat}(T, L, TL). \]

In the rare occasions when the translation to be implemented is defined for complete programs, yet another operator must be introduced to bind procedures to one another, e.g. \#.

Queries, which are clause bodies, must also be translated by \texttt{val} prior to their submission to the translated program.

The many-sorted formalism is most useful when the translation of heads and bodies should differ. Sorts for heads and clause bodies are then introduced, atomic goals being generators of sort \( b \) and heads being generators of sort \( h \). Of course, \( A_h \) is limited to these generators. A third sort, \( r \), for clauses, will also be introduced, leading to the many-sorted signature with sorts \( S \) and operator domain \( \Omega \) where:

\[ S = \{ h, b, r \} \]
\[ \Omega = \{':-'/(\{h, b\}, r), ','/(\{b, b\}, b), ...\} \]
Sorts must be assigned to generators. This cannot be done without modifying the syntax or performing some preliminary syntactic analysis. However, if it is assumed that programs to be translated are correct, the left-hand side operand of :- is known to be a head. Thus, assignment of sorts can be provided by a slight modification to predicate val: the clauses

\[
\text{val}(\text{H} :- \text{Body}), \text{V}) \leftarrow !, \text{val}((\text{H}, \text{V})), \text{val}((\text{Body}, \text{V})), ':-_\text{A}'(\text{VH}, \text{VB}, \text{V}).\\
\text{val}(\text{G}, \text{V}) \leftarrow \text{f}(\text{G}, \text{V}).
\]

defining the translation of object-level clauses are easily modified into:

\[
\text{val}(\text{H} :- \text{Body}), \text{V}) \leftarrow !, \text{f}_\text{head}(\text{H}, \text{V}), \text{val}((\text{Body}, \text{V})), ':-_\text{A}'(\text{VH}, \text{VB}, \text{V}).\\
\text{val}(\text{G}, \text{V}) \leftarrow \text{f}_\text{body}(\text{G}, \text{V}).
\]

In the sequel of this paper, we will assume that this approach has been followed, and, when required, predicates \( f_\text{head} \) and \( f_\text{body} \) will implement \( f_h \) and \( f_b \) respectively.

The translation scheme should of course produce Prolog programs. This means, for example, that when the translation applies to clauses, the value of a clause should be a clause (or maybe a list of clauses), so that the collection of the translations of all the clauses in the source program produce the target program. Similarly, if the translation applies to procedures, the respective translations of the procedures of the source program will constitute the target program. As intermediate results for the elaboration of the target program, any convenient set of terms can be taken as the semantics of e.g. clause bodies. Such terms will in general contain part of the target program under construction. In many-sorted terms, this means that carrier \( A_r \) for clauses should consist of clauses, while the carrier \( A_b \) can be arbitrarily selected. To summarize, a translation scheme is a meta-program that manipulates two object-level programs: the source and the target of the translation. The intermediate semantic carriers used to construct the target program are not restricted to valid program constructs, and are simply Prolog terms.

Further sections will present specific examples. As usual, we select simplified examples, for the sake of clarity and brevity. For example, we mostly restrict ourselves to programs which consist of pure Horn clauses. Due to their simplicity, the examples we present could sometimes be better handled by other techniques. Recall, however, that they were selected as a representative of a larger class of problems, for illustration purpose.

A number of notations are adopted to denote Prolog constructs: \( p \) (pos-
possibly indexed) ranges over predicate symbols, $T$ over terms, $X$ over variables, $G$ over atomic goals, $H$ over clause heads, $B$ over clause bodies, $P$ over programs, and $Q$ over queries.

4. Non-standard unification

The first example implements a variant of Prolog with modified unification: we assume that predicate unify succeeds if its two arguments unify according to some non-standard specification of unification. Non-standard unification is known to be useful in a number of cases. To mention a few: unification with occurrence checks\(^{18,19}\) used, for example, when difference lists are manipulated\(^{19-21}\); functional extensions to Prolog\(^{22-23}\); introduction of associativity, commutativity and second-order unification\(^{24}\); unification failure analysis\(^{25}\).

For comparison with the algebraic approach, the meta-interpreter is:

```prolog
demo(true).
demo((Goal1,Goal2)) :-
    demo(Goal1), demo(Goal2).
demo(Goal) :-
    functor(Goal,Name,Arity), functor(Template,Name,Arity),
    clause(Template,Body),
    unify(Goal,Template),
    demo(Body).
```

Observe that the code

```prolog
functor(Goal,Name,Arity), functor(Template,Name,Arity)
```

creates term Template with the same principal functor (i.e. the predicate symbol) as term Goal and with fresh variables as arguments, to access clauses in such a way that standard unification is not performed between terms in the clause head and terms in the goal under interpretation. Non-standard unification, defined by predicate unify can then be applied instead.

For the transformational approach, each program clause of the form

$$\rho(T_1,T_2,\ldots):=B$$

is translated into

$$\rho(X_1,X_2,\ldots):=\text{unify}(\rho(X_1,X_2,\ldots),\rho(T_1,T_2,\ldots)),B$$
where the \(x_i\)'s are new variables not occurring in the original clause. Thus, the clause defining the concatenation of difference lists:

\[
\text{concat}(X-Y, Y-Z, X-Z) :- \text{true}.
\]

would be translated as:

\[
\text{concat}(V1, V2, V3) :- \text{unify(\text{concat}(V1, V2, V3), \text{concat}(X-Y, Y-Z, X-Z))}, \text{true}.
\]

This transformation is implemented by a translation scheme based on signature \([',/',2, ':/-2]\). Thus, the code for the translation reduces to the definition of predicates \(':-.NSU' (NSU for 'non-standard unification' is the name of the semantic algebra), \'::-.NSU', and \(f\):

\[
\begin{align*}
\&':-.NSU'(\text{Goal1, Goal2, (Goal1, Goal2))}. \\
\&':-.NSU'(\text{Head, Body, (Template :- unify(Template, Head), Body))} :- \\
\&\quad \text{functor(Head, Name, Arity), functor(Template, Name, Arity).} \\
\&f(\text{Goal, Goal}).
\end{align*}
\]

For example:

\[
\begin{align*}
\text{reverse}([I, I]) & :- \text{!}. \\
\text{reverse}([X|\text{Tail}, \text{Liat}X]) & :- \\
\text{reverse}(\text{Tail}, \text{Liat}), \text{append}(\text{Liat, [X], LiatX}). \\
\text{append}([], X, X) & :- \text{true}. \\
\text{append}([X|\text{Tail}]Y, [X|\text{Tail}Y]) & :- \\
\text{append}(\text{Tail, Y, Tail}Y).
\end{align*}
\]

is translated into:

\[
\begin{align*}
\text{reverse}(V1, V2) & :- \\
\text{unify}(&\text{reverse}(V1, V2), \text{reverse}([I, I]), \text{!}). \\
\text{reverse}(V1, V2) & :- \\
\text{unify}(&\text{reverse}(V1, V2), \text{reverse}([X|\text{Tail}, \text{Liat}X]), \\
\text{reverse}(\text{Tail, Liat}), \\
\text{append}(\text{Liat, [X], LiatX}).
\end{align*}
\]

*) Strictly speaking, since the clause bodies are their own translation, \([::/-2]\) could be taken as signature. But further extensions will require changes inside clause bodies.
An algebraic approach to meta-level programming in Prolog

append(V1,V2,V3) :-
    unify(append(V1,V2,V3), append([],X,X)), true.
append(V1,V2,V3) :-
    unify(append(V1,V2,V3), append([X|Tail],Y,[X|TailY])),
    append(Tail,Y,TailY).

As they stand, the meta-interpreter and the translation scheme are roughly equivalent in complexity. Comparison of the meta-interpreter with the translation reveals differences which are generally observed between the two approaches:

- In both cases, there is a slight limitation on the unifiers which can be used, as it is implicitly assumed that two goals with different predicate symbols never unify. This restriction can be lifted for the meta-interpreter (at the price of efficiency): remove the construction of Template, to access all clauses of the program. The modification is deeper for the translation approach: in the translation scheme, program clauses are manipulated at translation-time, and are thus fixed at run-time; to access clauses at runtime, each goal of the form \( p(T_1, T_2, \ldots) \) should be translated into

\[
\text{clause}(X_H, X_B), \text{unify}(X_H, p(T_1, T_2, \ldots)), \text{call}(X_B)
\]

where \( X_H \) and \( X_B \) are new variables.

This translation scheme applies to atomic goals in bodies and not to heads, so the many-sorted technique must be used: the clause defining \( f \) in the previous translation scheme, should be replaced by:

\[
f_{\text{head}}(\text{Head}, \text{Head}).
\]

\[
f_{\text{body}}(\text{Goal}, (\text{clause}(\text{Head}, \text{Body}), \text{unify}(\text{Head}, \text{Goal}), \text{call}(\text{body}))).
\]

and ‘:-_NSU’ should just reconstruct a clause:

‘:-_NSU’(Head, Body, (Head:-Body)).

The general observation is that simple meta-interpreters handle dynamic programs, i.e. programs that are partially constructed at run-time, e.g. by asserting new clauses or constructing new goals dynamically and calling them. This is possible at the price of some efficiency which can be recovered at the price of some complexity by introducing filter code to benefit from static cases. Conversely, simple translation schemes produce rea-
sonably efficient programs but must be made more complex to yield programs with a more dynamic character. This is not surprising, since it reflects the usual trade-off between interpreters and compilers.

- The algebraic translation is capable of handling cuts quite easily, since (and when) the translation process does not alter the general structure of the program, hence does not alter the locality of cuts. It is sufficient to translate cuts by cuts, which entails in some cases – and in the extended version of NSU in particular – inserting

\[ f\text{\_body}('!', '!'):\text{-}! \]

as the first clause for \( f\text{\_body} \).

On the other hand, a meta-interpreter handling cuts would be fairly complex.

5. Counting logical inferences

The second example is taken as a simple instance of a program that records information on its proof tree. Recording the proof tree is useful e.g. for explanation generation in expert systems or for debugging; simply counting the number of inferences in the proof is useful for tuning heuristic programs or for loop prevention.

Start from a program, \( P \), to which a query \( Q = p(T_1, T_2, \ldots) \) is submitted. The intention is to arrive at a new program, call it \( P_{pl} \), which is equivalent to \( P \) (it delivers the same answers), but will compute the length of each proof necessary to arrive at each answer. The queries for \( P_{pl} \) will accomodate a new variable for the proof length. A first possibility is to modify \( Q \) as follows:

\[ Q_{pl} = p(<X_1, X_2, \ldots) ? C_{pl} \]

with a suitable operator definition for ‘?’.

Each atomic goal in \( P \) will be modified to construct \( P_{pl} \): \( f(Q) = Q ? P_L \). function \( P_L \) (\( P_L \) for ‘proof length’ is the name of the semantic algebra for this example) should combine two atomic goals into a new goal ensuring that the correct proof length is computed:

\[ C_1 ? \mathcal{X}_{pl,1} , P_L C_2 ? \mathcal{X}_{pl,2} = C_1 ? \mathcal{X}_{pl,1}, C_2 ? \mathcal{X}_{pl,2}, \mathcal{X}_{pl} \is \mathcal{X}_{pl,1} + \mathcal{X}_{pl,2} \]

When two non-atomic goals are conjunct, variable \( \mathcal{X}_{pl} \) plays an important role in the translation process, and who should be made visible and easy to retrieve at the meta-level. It is therefore convenient to define the semantic domain for clause bodies as Prolog terms of the form

\[ \text{lambda}([\mathcal{X}_{pl}], R) \]
An algebraic approach to meta-level programming in Prolog

the choice of lambda as functor name is justified since this construct indeed resembles \( \lambda \)-expressions. \( \beta \)-reduction can be defined as follows:

\[
\text{beta}(\text{Lambda}, \text{Argument}, \text{Reduct}) : - \\
\text{copy_term}(\text{Lambda}, \lambda(\text{Argument}, \text{Reduct})).
\]

The copy is necessary if the given lambda expression is subject to several \( \beta \)-reductions: otherwise, the first reduction would instantiate its variables and it would become impossible to apply other reductions to it.

Observe how predicate beta generalizes \( \beta \)-reduction: whereas the \( \beta \)-reduct is obtained by substitution \((\lambda P.E)(A) = \lambda \beta E[P \leftarrow A]\), beta specifies the unification of parameters with arguments. The \( \lambda \)-expressions here behave in that respect as Prolog predicates, and the unification can instantiate arguments. It is also possible to invoke \( \lambda \)-expressions as (anonymous) Prolog predicates:

\[
\text{call}(\text{Lambda}, \text{Arguments}) : - \\
\text{beta}(\text{Lambda}, \text{Arguments}, \text{Goal}), \text{call}(\text{Goal}).
\]

In a translation scheme, the semantic domain often consists of object-level Prolog constructs (e.g. atomic goals are translated as goals). These object-level constructs are then combined into more complex constructs by further operators. Such a combination generally depends on meta-level information which should annotate the component construct, and \( \lambda \)-expressions constitute a general mechanism to carry such meta-level information about object-level constructs in their body. Uninstantiated variables are frequently needed as meta-level information to express dependencies among the components used to form a new construct.

In some cases, it is necessary to postpone part of the computation of meta-level goals themselves, until more context for their application is known, and \( \lambda \)-expressions can be used as anonymous Prolog predicates for that purpose.

To apply this technique in full rigour, predicate '$_{PL}' in our example would be

\[
'_{PL'}(\text{Lambda}_1, \\
\quad \text{Lambda}_2, \\
\quad \lambda\text{body}(\text{PL}), (\text{Body}_1, \text{Body}_2, \text{PL} \text{ is } \text{PL}_1 + \text{PL}_2)) \\
: - \text{beta}(\text{Lambda}_1, [\text{PL}_1], \text{Body}_1), \\
\quad \text{beta}(\text{Lambda}_2, [\text{PL}_2], \text{Body}_2).
\]
However, a simplified form of \( \beta \)-reduction without copy is sufficient, since \( \lambda \)-expressions are only \( \beta \)-reduced once. The translation scheme thus reads:

\[
'._{PL}'(\lambda([\pi_1], \text{Body}_1), \\
\quad \lambda([\pi_2], \text{Body}_2), \\
\quad \lambda([\pi], (\text{Body}_1, \text{Body}_2, \pi \text{ is } \pi_1 + \pi_2))).
\]

Clauses are constructed to count an inference for their own application:

\[
' :-_{PL}'(\lambda([\pi_h], \text{Head}), \\
\quad \lambda([\pi_b], \text{Body}), \\
\quad (\text{Head} :- \text{Body}, \pi_h \text{ is } \pi_b + 1)).
\]

f(Goal, \lambda([\pi], \text{Goal} ? \pi)).

There is a slight difficulty with built-in predicates, of course. A simple — admittedly rather cumbersome — solution is to redefine all of them under the new form, e.g. for predicates \text{atomic} and \text{true}

\[
\text{atomic}(X) \ ? 1 :- \text{atomic}(X) \quad \text{% The number of logical inferences} \\
\text{% for each built-in can be} \\
\text{% specified at will}
\]

\[
\text{true} ? 0 .
\]

This idea can be implemented in a less cumbersome manner, if the built-ins count for one logical inference:

\[
\text{true} ? 0 :- !. \\
\text{Built}_i \ ? 1 :- \text{built}_i(\text{Built}_i), !, \text{call}(\text{Built}_i).
\]

Another solution is to deal with built-ins explicitly, in a separate clause for \( f \):

\[
f(\text{true, } \lambda([0], \text{true})):! . \\
f(\text{Built}_i, \lambda([1], \text{Built}_i)) :- \text{built}_i(\text{Built}_i), !.
\]

Adopting this solution for built-ins, the following program:

\[
\text{reverse}([],[]): \text{true}. \\
\text{reverse}([X|\text{Tail}], \text{LiatX}):= \\
\quad \text{reverse}(\text{Tail}, \text{Liat}), \text{append}(\text{Liat}, [X], \text{LiatX}).
\]
An algebraic approach to meta-level programming in Prolog

is translated into:

reverse([], [] ? PI :-
  true, PI is 0 + 1.

reverse([X|Tail], LiatX) ? PI :-
  reverse(Tail, Liat) ? PI_1,
  append(Liat, [X], LiatX) ? PI_2,
  PI_3 is PI_1 + PI_2,
  PI is PI_3 + 1.

There is still room for improvement in the translation, e.g. by applying to it a simple partial evaluator, whose role would be similar to peephole optimisation in a classical compiler. However, peephole optimisation can be easily introduced in the translation itself, by defining the semantic domain $A_b$ as consisting of $\lambda$-expressions of the form $\lambda([\text{Count}], \text{Body})$ where $\text{Count}$ is an arithmetic expression yielding the length of the proof of $\text{Body}$. This requires only a change of predicate $\text{'_PL'}$:

\[
\text{'_PL'}(\lambda([\text{PI}_1], \text{Body}_1),
\lambda([\text{PI}_2], \text{Body}_2),
\lambda([\text{PI}_1 + \text{PI}_2], (\text{Body}_1, \text{Body}_2))).
\]

On the example above, the new translation scheme produces clause (1):

\[
\text{reverse}([X|Tail], LiatX) ? PI :-
  reverse(Tail, Liat) ? PI_1,
  append(Liat, [X], LiatX) ? PI_2,
  PI is PI_1 + PI_2 + 1, \quad (1)
\]

The translation produced by this scheme – call it the naive approach – suffers, however, from a major drawback: a single predicate symbol remains in the target program: ? (apart from the built-ins, of course). This implies that clause indexing will be rather ineffective and that efficiency will suffer. A similar observation has been made in the case of partial evaluation of meta-interpreters\footnote{after partial evaluation, the meta-interpreter still consists only of predicate demo, whose first argument is the goal to be proved, and other arguments carry meta-level information. The solution proposed is to ‘push’ the meta-level arguments as new arguments to the goal itself. A similar solution applies here: in an atomic goal of the form}: we can push the meta-level arguments as new arguments to the goal itself. A similar solution applies here: in an atomic goal of the form.
\[ \rho(\mathcal{T}_1, \mathcal{T}_2 \ldots) \ ? \ x_{pl}, \]

\(x_{pl}\) is a meta-level variable, and should be 'pushed' as a new argument of predicate \(\rho(\mathcal{T}_1, \mathcal{T}_2, \ldots, x_{pl})\). This idea translates simply in our case: it is sufficient to redefine the last clause for \(f\) so that meta-arguments are pushed:

\[
\begin{align*}
&f(\text{Goal}, \text{lambda}([\text{PI}], \text{NewGoal})) :- \\
&\quad \text{push_args}(\text{Goal}, [\text{PI}], \text{NewGoal}). \\
&\text{push_args}(\text{Goal}, \text{Args}, \text{NewGoal}) :- \\
&\quad \text{Goal} = .. \text{List}, \\
&\quad \text{append} (\text{List}, \text{Args}, \text{NewList}), \\
&\quad \text{NewGoal} = .. \text{NewList}.
\end{align*}
\]

Meta-arguments are pushed at the end of the object-argument list, because most Prolog implementations index program clauses on the predicate name and the first argument (or first few arguments). This decision might well be critical for the efficiency of the compiled program, and should be considered carefully as it may sometimes be more efficient to push meta-level arguments before object-level ones.

Using the technique of meta-argument pushing, the second clause of reverse becomes:

\[
\begin{align*}
&\text{reverse}([X|\text{Tail}], \text{LiatX}, \text{PI}) :- \\
&\quad \text{reverse}(\text{Tail}, \text{Liat}, \text{PI}_{-1}), \\
&\quad \text{append}(\text{Liat}, [X], \text{LiatX}, \text{PI}_{-2}), \\
&\quad \text{PI} \text{ is } \text{PI}_{-1} + \text{PI}_{-2} + 1.
\end{align*}
\]

Once again, the built-in predicates need special attention, as meta-arguments should not be pushed for them.

It is worth noting that meta-argument pushing shifts the program towards a more static character: since all goals defined in the program get new arguments, programs that manipulate such goals (if only to assert new facts in the database) would have to be modified rather deeply, in a way that is beyond the expressiveness of a translation scheme, at least in general. This observation applies to meta-interpreters as well.

In the case of meta-interpreters, it has been proposed to perform meta-argument pushing via a program to be invoked after partial evaluation\(^2\). Such a separate transformation, an argument pusher can also be defined here,
An algebraic approach to meta-level programming in Prolog

if convenient: here is an argument pusher to go from the translated program of clause (1) to that of clause (2)

'._AP'(Goal1,Goal2,(Goal1,Goal2)).
'._-AP'(Head,Body,(Head :- Body)).

f(MetaGoal,NewMetaGoal)
   :- MetaGoal = ObjGoal ? MetaArgs
      -> push_args(ObjGoal,MetaArgs,NewMetaGoal)
         ; NewMetaGoal = MetaGoal.

We are thus faced with two approaches to meta-argument pushing:
- the direct pushing approach, (clause (2)),
- the naive approach followed by the meta-argument pusher.

Usually, it is convenient to express a complex transformation scheme as a combination of simpler transformations. Once again a similar observation applies to meta-interpreters, for which it has been proposed\(^2\) to obtain a complex meta-interpreter as a combination of simpler ones called *flavours*.

6. Flavour mixing

The only way to combine two meta-interpreters presently described in the literature is to apply them after the other, i.e. to use a second meta-interpreter to interpret the first one (or the partial evaluation of the first one w.r.t. a given object program). This is exactly what has been done above: the direct pushing translation scheme has been obtained as the successive application of the naive translation scheme and the argument pusher. However, there are other possibilities. We present a new flavour mixing technique below for the algebraic translation schemes, but similar techniques could be devised for meta-interpreters.

First let's consider the above example in more abstract terms: consider two translation schemes \((f_1, A_1)\) and \((f_2, A_2)\). The combination of the example (proof length followed by argument pushing) amounts to the computation of \(f_1 \circ i \circ f_2\) where function \(i\) trivially injects terms of \(A_1\) into \(W(X)\) (\(X\) being the set of Prolog atomic goals). Note that \(i\) is not an homomorphism, thus \(f_1 \circ i \circ f_2 \neq (f_1 \circ i \circ f_2)^*\); indeed, such general algebraic properties seldom hold for the translation schemes encountered in practice. As another (counter) example of a general property of little use, recall that the Cartesian product of \(\Omega\)-algebras is itself a \(\Omega\)-algebra. This property can only help us to combine two unrelated translation schemes into one, to produce two
independent translations at once, while the obvious need is for a single translation combining the information obtained from two translation schemes.

To get this result, the structure of the translation itself must be analysed to arrive at useful combinations of translation schemes.

Suppose that a translation scheme specification is of the form

$$\lambda(M_1, G_1) \rightarrow A \lambda(M_2, G_2)$$

$$= \lambda(M, (B_1[M_1, M_2, M], G_1, G_2, B_2[M_1, M_2, M]))$$

$$\lambda(M, L) : A \lambda(M_3, B)$$

$$= L : (B_3[M_3, M_3], B, B_4[M_3, M_3])$$

$$f_A(G) = \lambda(M, \text{push}(G, M))$$

where \( M \) ranges over lists of meta-level terms (terms in which the only variables are meta-level ones), \( M_p \) denotes a subset of \( M \), \( \text{push}(G, M) \) denotes the pushing of terms in \( M \) as new arguments to goal \( G, M \parallel M_2 \) denotes the concatenation of two such lists, and \( B[M_1, M_2, ...] \) indicates that the compound goal \( B \) instantiates only variables occurring in \( M_1, M_2, ... \)

Two schemes of this form with independent meta-arguments will easily combine into a single one:

$$\lambda(M_1 \parallel M_1, G_1) \rightarrow A + A' \lambda(M_2 \parallel M_2, G_2)$$

$$= \lambda(M \parallel M', (B_1[M_1, M_2, M], B_1[M_1, M_2, M'], G_1, G_2, B_2[M_1, M_2, M], B_2[M_1, M_2, M']))$$

$$\lambda(M, L_3) : A + A' \lambda(M_3, B)$$

$$= L : (B_3[M_3, M_3], B_3[M_3, M_3], B, B_4[M_3, M_3], B_4[M_3, M_3])$$

$$f_{A+A'}(G) = \lambda(M \parallel M', \text{push}(G, M_p \parallel M_p))$$

As an example of such a combination, consider inference counting and proof tree construction. The proof length translation has the form:

$$f_{pl,b} (\text{true}) = \lambda([0], \text{push(true,[]}))$$

$$f_{pl,b}(G) = \lambda([P]l, \text{push}(G, [P]l))$$

$$f_{pl,h}(G) = \lambda([P]l, \text{push}(G, [P]l))$$
An algebraic approach to meta-level programming in Prolog

\[
\text{lambda}([\Pi_1,\gamma_1], \gamma_1) \cdot p_l \text{ lambda}([\Pi_2,\gamma_2])
\]
\[
= \lambda([\Pi_1 + \Pi_2], (\text{true}, \gamma_1, \gamma_2, \text{true}))
\]

\[
\text{lambda}([\Pi_h,\Pi], \Pi) \cdot p_l \text{ lambda}([\Pi_b,\Pi])
\]
\[
= H : - \text{true}, \Pi_h = \Pi_b + 1
\]

The proof collector is as follows:

\[
\text{fproof}_b(\text{true}) = \lambda([\text{true}], \text{push}(\text{true},[]))
\]
\[
\text{fproof}_b(\gamma) = \lambda([\Pi, \text{Proof}], \text{push}(\gamma, [\Pi, \text{Proof}]))
\]
\[
\text{fproof}_h(\gamma) = \lambda([\Pi, \text{Proof}, \gamma], \text{push}(\gamma, [\Pi, \text{Proof}]))
\]

\[
\lambda([\Pi, \text{Proof}, \text{Code}_h], \Pi) \cdot p_l \lambda([\Pi, \text{Proof}, \text{Code}_h], \Pi)
\]
\[
= \lambda([\Pi, \text{Proof}, \text{Code}_h], \Pi)
\]
\[
= H : - \text{true}, \text{true}, \Pi_h = \Pi_b + 1,
\]
\[
\text{Proof}_h = \text{rule}([\Pi, \text{Code}_h], \text{Proof}_b)
\]

Both translation schemes having the requested form, they can be combined as indicated. This form of combination solves the problems encountered with the successive application of meta-interpreters or of translation schemes, which produce erroneous results. If the proof length counter is applied after the proof constructor, the steps of the proof construction will be counted, and the length computed will be too large; if the proof constructor is applied after the proof length counter, the steps of the counter will appear as part of the proof of the initial program. The combined translation scheme reads:

\[
\text{fproof}_b(\text{true}) = \lambda([\text{true}], \text{push}(\text{true},[]))
\]
\[
\text{fproof}_b(\gamma) = \lambda([\Pi, \text{Proof}], \text{push}(\gamma, [\Pi, \text{Proof}]))
\]
\[
\text{fproof}_h(\gamma) = \lambda([\Pi, \text{Proof}, \gamma], \text{push}(\gamma, [\Pi, \text{Proof}]))
\]

\[
\lambda([\Pi, \text{Proof}, \text{Code}_h], \Pi) \cdot p_l \lambda([\Pi, \text{Proof}, \text{Code}_h], \Pi)
\]
\[
= \lambda([\Pi, \text{Proof}, \text{Code}_h], \Pi)
\]
\[
= H : - \text{true}, \text{true}, \Pi_h = \Pi_b + 1,
\]
\[
\text{Proof}_h = \text{rule}([\Pi, \text{Code}_h], \text{Proof}_b)
\]
The resulting translation for the example is:

\[\text{reverse}([], [], \text{Pl}_h, \text{Proof}_h) :-
\begin{align*}
\text{true}, \\
\text{true}, \\
\text{true}, \\
\text{Pl}_h \text{ is } 0 + 1, \\
\text{Proof}_h = \text{rule}(\text{reverse}([], []), \text{true}).
\end{align*}\]

\[\text{reverse}([X|\text{Tail}], \text{Liat}_X, \text{Pl}_h, \text{Proof}_h) :-
\begin{align*}
\text{true}, \\
\text{true}, \\
\text{true}, \\
\text{true}, \\
\text{reverse}(\text{Tail}, \text{Liat}, \text{Pl}_1, \text{Proof}_1), \\
\text{append}(\text{Liat}, [X], \text{Liat}_X, \text{Pl}_2, \text{Proof}_2), \\
\text{true}, \\
\text{Proof} = \text{and}(\text{Proof}_1, \text{Proof}_2), \\
\text{Pl}_h \text{ is } \text{Pl}_1 + \text{Pl}_2 + 1, \\
\text{Proof}_h \text{ } X = X \text{ rule}(\text{reverse}([X|\text{Tail}], \text{Liat}_X), \text{Proof}_b).
\end{align*}\]

7. Normal form

It is sometimes convenient to transform a Prolog program into an equivalent one with some specific properties, a normal form program. In this section, such a transformation is presented as an example of a translation scheme acting on procedures, and not simply on clauses, as in the examples before. This is also an example of meta-programming for which meta-interpreters do not apply. The construction of the semantic domain is not trivial, but the example shows how the algebraic view induces some useful structure.

The normal form presented here can be characterized as follows:

1. The clauses reduce to one of the following forms:

\[\mathcal{H} :- \text{true} \quad (4)\]
\[\mathcal{H} :- \mathcal{G} \quad (5)\]
\[\mathcal{H} :- \mathcal{G}_1, \mathcal{G}_2 \quad (6)\]
\[\mathcal{H} :- \mathcal{G}_1; \mathcal{G}_2 \quad (7)\]

2. The atomic goal in clause (5) is the only point where there can be a direct failure by lack of a clause to unify with. Hence, the conjuncts and disjuncts in clause (6) and (7) will never fail directly.
3. Each procedure of the program consists of a single clause. As a consequence, the choice points of the program are localized at the semicolons. Programs in such a form are simpler to handle than programs in the general form, e.g. for compilation.

First, the normalization will be described for programs where all atomic goals refer to predicates for which there is a procedure in the program itself. This excludes the invocations of built-in predicates.

As an auxiliary operation, the simplification of goals is defined as follows:

- simplification applies to atomic goals, to conjunctions of atomic goals and disjunctions of atomic goals, which constitute the class of simplifiable goals. $\mathcal{S}$ will range over simplifiable goals.
- the simplification of atomic goal $\mathcal{G}$ is $\mathcal{G}$, and no clauses are produced.
- the simplification of $(\mathcal{G}_1, \mathcal{G}_2)$ is its replacement by $p(\mathcal{X}_1, \ldots)$ with $p$ a new predicate symbol and the $\mathcal{X}$ all the variables occurring in $\mathcal{G}_1$ and $\mathcal{G}_2$. A new clause is produced:

$$p(\mathcal{X}_1, \ldots) :\mathcal{G}_1, \mathcal{G}_2$$

- the simplification of disjunctions is similar: $(\mathcal{G}_1 ; \mathcal{G}_2)$ is replaced by $p(\mathcal{X}_1, \ldots)$, and the new clause is

$$p(\mathcal{X}_1, \ldots) :\mathcal{G}_1 ; \mathcal{G}_2.$$ 

The normalization of clause bodies amounts to the normalization of disjunctions and conjunctions. A normalized body will always be a simplifiable goal. Hence, it is only necessary to define the normalization of conjunctions and disjunctions of simplifiable goals.

Again, this will imply both goal replacement and the production of new clauses: the normalization of the conjunction of simplifiable goals is the conjunction of their respective simplifications, with the new clauses produced by these simplifications. Algebraically, the domain $A_b$ for the semantics of clause bodies will consist of pairs $(\mathcal{S}, \mathcal{C})$, where $\mathcal{S}$ is a simplifiable goal and $\mathcal{C}$ a set of clauses. Normalization can be described as follows:
where the simplification of $\mathcal{S}_i$ is $\mathcal{S}'_i$ with the production of clauses $\mathcal{C}'_i(i = 1,2)$.

The normalization of a clause $p(\mathcal{F}_1, \ldots) :- \mathcal{S}$ with a normalized (hence simplifiable) body, is a list of clauses of the form:

$$p(\mathcal{F}_1, \ldots) :-_{nf} (\mathcal{S}, \mathcal{C}) = [(p(\mathcal{X}_1, \ldots) :- p'(\mathcal{X}_1, \ldots)), (p'(\mathcal{F}_1, \ldots) :- \mathcal{S}) | \mathcal{C}]$$

where $p'$ is a new predicate symbol and $\mathcal{C}$ the set of clauses produced by the normalization of $\mathcal{S}$.

Finally, the semantics for $\&$ is as follows:

$$[(p(\mathcal{X}_1, \ldots) :- \mathcal{S} | \mathcal{C}) \& nf [(p(\mathcal{X}_1, \ldots) :- p') | \mathcal{C}'])$$

$$= [(p(\mathcal{X}_1, \ldots) :- \mathcal{S}_0; \mathcal{S}'_0) | \mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}']$$

where $\mathcal{S}_0$ and $\mathcal{S}'_0$ are the respective simplifications of $\mathcal{S}$ and $\mathcal{S}'$, producing (together) the list of clauses $\mathcal{C}'$. Notice how the variable in the heads of first clauses (hence in bodies) are made identical by unification. Observe that the semantics of clauses indeed produce lists of clauses whose first clause has a simplifiable body, as requested by the definition of $\&$.

The translation above critically depends on the hypothesis that all atomic clauses of the initial program invoke predicates for which there is a definition in the program. Every program-defined predicate $p$ is defined by a clause $p(\mathcal{X}_1, \ldots) :- \mathcal{S}$ which will unify with any goal invoking $p$. In this way, characterization (2) for the normal form is satisfied. To lift this restriction it is sufficient to modify the specification for function $f$ as follows:

$$f_b(\text{true}) = (\text{true}, [])$$
$$f_b(p(\mathcal{F}_1, \ldots)) = (p'(\mathcal{F}_1, \ldots), [p'(\mathcal{X}_1, \ldots) :- p(\mathcal{X}_1, \ldots)])$$

where $p'$ is a new predicate symbol.

8. Efficiency comparisons

For Prolog programs, efficiency comparisons are difficult in general. This difficulty stems from several factors particular to Prolog. Those factors con-
cur to render time measurements highly imprecise. Among others, no proper instrumentation is readily available to measure Prolog programs in terms of time and space usage; Prolog implementations are heavy consumers of virtual memory management; scheduling algorithms allow only a discrete sampling of clocks whose resolution is commonly rather poor.

Nevertheless, a number of tests have been performed on a dedicated SUN 3/75 (with 8 Mbytes of central memory and 32 Mbytes of swap space) running Quintus Prolog Release 2.0. The only parameter measured was the execution time (CPU time) for finding all solutions of a goal with no output apart from the timings. All tests have been run hundreds or thousands of times*) in order to minimize statistical errors and to ease the determination of overhead costs of the benchmarking itself.

The gain of the algebraic approach over the meta-interpreters approach has been measured as the ratio between the execution time of a query by a meta-interpreter along with an object program and the execution time of the same query executed by the program output by the corresponding algebraic translation scheme. Both the meta-interpreter and the translated object program have been compiled with the Quintus Prolog compiler**). If a sufficiently powerful partial evaluator was available, the ratio between the target program and the partially evaluated meta-interpreter could become close to 1.

These tests have been conducted on a lot of different applications (unification with occur check (gains between 1 and 3), unification with term rewriting (gains > 100), proof tree length evaluation (gains between 1 and 6), positive and negative proof trees extraction (gains ≈ 5), extensions with freeze/2 predicate (gains ≈ 1), ...) applied to some of the Prolog programs of the Quintus Prolog benchmark suite and other programs of our own. Some results are given in table 1. The ratios obtained are highly variable and may range from 1 to more than 1000 apparently depending on several factors, such as the specific application, the sophistication of the meta-interpreter, the choice of the library functions for the auxiliary predicates, the optimisations made by the Prolog compiler. For example, on most of the test data for the 'occur check' flavour the ratio was between 1 and 2; this is due to the fact that most CPU time is spent by unification itself. In some rare cases the ratio was lower than 1, but in these cases, the reason for such deficiency has been traced back to some peculiarities in Prolog implementa-

*) The number of runs has also been determined in such a way that no garbage collection occurs during execution.

**) Some authors²,2⁰) seem to define the efficiency ratio by comparing an interpreted meta-interpreter with a compiled version of its partial evaluation.
G. Louis and M. Vauclair

### TABLE I

<table>
<thead>
<tr>
<th>rev</th>
<th>vanilla unify</th>
<th>occur check unify</th>
<th>proof* length</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rev</td>
<td>1.54</td>
<td>1.03</td>
<td>1.60 (x2.12)</td>
<td>Naive reverse of 30 elements</td>
</tr>
<tr>
<td>lips</td>
<td>5.50</td>
<td>1.83</td>
<td>2.38 (x1.11)</td>
<td>200 deterministic propositional calls</td>
</tr>
<tr>
<td>lipsconj</td>
<td>11.00</td>
<td>2.65</td>
<td>4.66 (x1.18)</td>
<td>Lots of propositional conjunctions</td>
</tr>
<tr>
<td>lipsback</td>
<td>4.50</td>
<td>1.79</td>
<td>1.12 (x1.03)</td>
<td>Heavy backtracking</td>
</tr>
<tr>
<td>succ</td>
<td>1.72</td>
<td>1.07</td>
<td>1.64 (x2.07)</td>
<td>Factorial of s(s(s(s(0))))</td>
</tr>
</tbody>
</table>

Concluding remarks

An algebraic framework for meta-level programming has been presented. Its main aspects have been illustrated by a number of examples. Comparisons have been made with meta-interpretation.

Since algebraic meta-programs are program transformation schemes, an efficiency gain w.r.t. meta-interpretation should be expected and is indeed observed in most cases. Thus, from an efficiency point of view, algebraic meta-level programming appears as a viable alternative to the partial evaluation of meta-interpreters.

Partial evaluation is a promising technique whose benefits are well established. However, it is not yet well understood, and its application remains complex. This renders the algebraic approach of this paper attractive at least on a temporary basis. Even with powerful partial evaluation strategies, the natural limits of undecidability and complexity will impose the exploitation of programmer's knowledge about the program under partial evaluation, e.g. via its annotation prior to partial evaluation. There might well remain cases where this knowledge will be better expressed explicitly inside the algebraic framework.

Algebraic translation schemes are structured specifications which yield structured implementations. The inherent complexity of thinking at two levels of abstraction is of course not overcome by such a structure, but it is felt that the discipline imposed by the algebraic view is an incentive towards clearer expression.

The structuring of algebraic translation schemes is helped by the following features:

*) Ratios in parentheses indicates the additional gain of argument pushing.
An algebraic approach to meta-level programming in Prolog

- Syntactic analysis of the object-level program is implicit in the signature.
- Lambda expressions provide means to convey meta-level information annotating the components of an object-level construct.
- There is no efficiency penalty for the final program if the global translation task is decomposed into a number of translation schemes applied in succession. The intermediate results need not be executable Prolog programs, and can thus consist of annotated Prolog text. One interesting case of such a multi-staged transformation is to have a final peephole optimization phase on the final result of an algebraic translation scheme.

One important difference between algebraic meta-programming and meta-interpretation is in the handling of dynamic programs. Clearly, algebraic translation schemes are not well adapted to dynamic programs, e.g., programs that modify themselves. On the other hand, it is often possible to translate programs containing cuts, a notably difficult task for meta-interpretation.

Finally, algebraic translation schemes inherit the limitations of program transformation in general: when the task at hand becomes really complex, the size of the generated program reaches the threshold of complexity that the underlying Prolog implementation can handle. Such a case has been met in practice, and overcome by the prior normalisation of the program.

A number of tools have been written to automate the implementation of translation schemes. Among them, a full fledged signature compiler which produces a complete translation program, and a benchmarking tool which helps to overcome the difficulties induced by the lack of predictability of today's Prolog implementation, which handle seemingly equivalent constructs quite differently from one another, and whose program optimization strategies are rather difficult to control. Using these tools, a number of non-trivial applications have been done, like positive and negative proof tree extraction. Further applications will be tackled, such as an algebraic version of a partial evaluator.

REFERENCES

G. Louis and M. Vauclair


Authors

Georges Louis; Ir. degree, Catholic University of Louvain, Belgium, 1970; University of Grenoble, France, 1970–1973 (DEA degree, 1971); Philips Research Laboratory, Brussels, Belgium, 1974–. At Philips, his initial interests were in programming languages and their formal description, and he participated in the definition of CHILL, the CCITT programming language. His interests include A.I. techniques, specifically knowledge representation and logical programming. He was responsible for the PRLB team in ESPRIT project P316 (ESTEAM). Presently, he is the head of a group working on algorithmic methods for digital design, and his interests are in architectural mapping and logic synthesis. He is a member of IFIP Working Group 2.4 (System Implementation languages). He is the author or the co-author of publications on these subjects. He is the co-author of the books: An Analytical Description of CHILL, the CCITT High Level Language, Springer-Verlag, 1982, and Approche Logique de l'Intelligence Artificielle, 1- de la logique classique à la programmation logique Dunod, 1988 (also in English by Wiley).

Marc Vauclair; Ir. degree (Electro-Mechanics), Brussels Free University, Belgium, 1983; Philips Research Laboratory Brussels, Belgium, 1983–. His research interest is in the field of logic programming languages, deductive databases and program transformation and specification techniques.